Convolutional ML Model for Image Location Prediction

Tyler Siwek
Department of Electrical and
Computer Engineerong
Colorado State University
Fort Collins CO
tylersi@colostate.edu

***Abstract*— This project is looking to use image data to determine the location an image was taken. This is done through training on image data from various rooms in an indoor environment and implemented on an embedded device through a convolutional neural network.**

Keywords—Embedded systems, Image detection, Tensorflow, TFlite

I. Introduction

The goal of my project is to connect an image with the location it was taken using a neural network. Being able to connect photos to a location has many practical applications in areas such as automatic geotagging of images posted to a website, a redundant check on navigation systems in autonomous vehicles to confirm and increase the accuracy of GPS and speeding up the logging of map data in cities. Additionally, this can be a helpful tool for certain indoor mapping problems, such as indoor navigation or a maintenance reporting system that automatically tags a reported location to help workers locate and optimize repairs.

For this project, I worked with both indoor and outdoor data but found better results with indoor data given the embedded system application and reduced variability of environments over time. From my collected images, I train a custom ML model that uses deep convolution to predict a class (location/room) that the image was taken in. Next, I compress the model and send it to a raspberry pi where the tflite model is used to predict the location that an image was taken with the attached camera. Model evaluation will use a set of images and finding the percentage of correct predictions.

II. Related Works

My project was inspired by the game GeoGuessr, where the objective is to predict the location of a Google street view image. People have made ML models to predict the location of Google street view images in the game. The model described by the source [1] is trained on a massive amount of street view data from many cities, requiring high compute power and access to a large database, which was not feasible for my application on an embedded system. The other part of my project, indoor mapping was inspired by research done at CSU on indoor navigation using Wi-Fi signals [2]. My project differs from the first related work by using an embedded system to run the model in application. This limits the scope of how many locations and generally how much data can be run through the model. This means I can't just download hundreds of thousands of images from Google to train the model. Another limitation is my access to street view data, I only had a certain amount of Google API credits and had to limit the area I was working with. My project differs from the indoor navigation related work because I am only using image data. I am also not predicting coordinates. I am instead using classes representing a location or room.

For model architecture, I used many techniques from the MIT paper, Places365 [3]. This paper shows how to improve scene recognition using their custom dataset and testing on various other models. After many underperforming custom ML models, I used a pretrained ResNet as was done in the paper to get much better results. The differences in complexity between scene recognition and image classification required a larger, more complex model and this paper helped me realize the need for a more complex model.

III. DATA

Outdoor location prediction: In order to get outdoor location images, I used the Google street view API to pull images for a specific location. The API allows requests to specify the angle of the panorama, the FOV and the tilt. Using this, I was able to generate many unique images at every location that had unique angles and FOVs.

Indoor location prediction: In order to generate data, I used my iPhone camera and took many photos of different angles of the space. My camera has wide angle, regular, and 2x lens options, which helped to prevent overfitting to a certain type of camera.

Data Augmentation: To reduce overfitting and increase generalization to other applications, I augmented the image data in a few ways for each image.

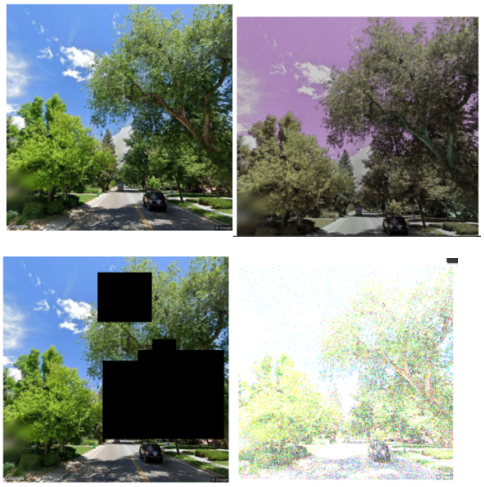
The first method, which I eventually decided was not helpful, was to separate the image into fourths to increase the amount of data. Instead of increasing accuracy, the split just decreased the spatial information available in every image.

The second method was to shift colors: I converted the images to HSV format from RGB, then shifted the hue and saturation values by a random amount for each image to account for differences in color between different cameras or the time of day/lighting at the time an image was taken.

The third type of augmentation was obscuring parts of the image using black boxes that were randomly generated. This was done to reduce dependance on any one specific feature in an image and learn more reliable and useful patterns.

The final type of data augmentation was adding noise, by adding a gaussian distribution of noise to the image, the model can learn patterns that will work on lower quality cameras and in different sub-optimal situations.

Exaggerated example images for each type of data augmentation:



Top left: original image. Top right: colors shifted. Bottom right: Black boxes randomly added. Bottom left: Noise Added

These 3 augmentations were layered on top of the original in less dramatic amounts than seen above and then shuffled and split in half to repeat training 2x, then split into training and validation data.

IV. Methods

General step by step workflow for this project:

1. Collect data:

Outdoor: Using the Google street view API, I pull hundreds of images with varying pitch, FOV and headings, to give many different perspectives of the same scene. This is done by using a formatted string as the API URL and running through a triple nested for loop with all these variables. Another constraint to my design was the fact that API requests were limited by Google's credit system. I was limited to $300 of credits before I would be charged, which was enough to map only a limited number of locations and motivated me to limit calls as much as possible. If the images at a location were already generated, I would save them in a folder in my Google drive and load them from there instead of generating them again. I also kept the code to request the API in a separate block of code to prevent myself from accidentally using it more than I needed to. The Google APIs tutorial [4] was a helpful resource for setting this up.

Indoor: After planning which rooms/spaces I would like to map out, I then take 250 images in each area using an iPhone camera, trying to capture different size and perspective images. These images are then uploaded to my google drive to be used in my Collab project.

2. Data Augmentation:

After the images have been generated, I apply the augmentation methods from above. I decide to limit the amount of noise and not split the images into smaller pieces and preserve their original size of 640x640. This did increase the size of the model but was able to capture more spatial information in every image and improve the accuracy in training statistics.

The images first have their colors augmented by applying a random scale of the H value by anywhere between 90% and 110% and a scaling of the S value by some random number between 90% and 110%. This will shift the colors a random amount for every image. The random scaling is slight but enough to account for variations in cameras.

Then, noise is added by simply creating a gaussian noise array in the shape of an image and tuning the standard deviation value to get a useful amount of noise. The array is added to the image to make it noisy.

Black boxes are added after this. They are created by generating a random width, height, x, and y coordinate for a specified number of boxes.

After these augmentations have been layered on one image, the images are ready to be labeled and split into testing, training and validation data.

3. Preparing for machine learning:

With an array of images generated for each location, these images need to be prepared and formatted correctly to be passed into the model. First, I create an array of the length of the number of images in a class containing a label for that class. Then, I combine labels and images into a combined labels and combined images array by concatenation. I then zip the labels and images together into an array of combined data before shuffling them. I use a method from sklearn to split my data into training and validation data, with a 30% validation split. Now, the data is labelled, shuffled, and split into training and validation sets, ready to be sent to the ML model.

Confirming that the labels are for the correct class:

For this test, class 1 is the study and class 0 is the family room.

4. Creating the ML model:

   My first attempt at making a model was done using a previous convolution model I had used when working with MNIST data. This was done for ECE 528 homework 1, where we learned about and used CNNs [5] to solve basic image recognition problems. This model was somewhat accurate in training, but I wanted to improve it further. I decided to try using VGG-16, a ML model that was built around a similar image detection problem. This surprisingly resulted in much worse performance. I then switched back to the original model and added more depth in the same pattern as before. Conv layer, then batch normalization and then dropout. This did not improve accuracy and I ended up returning to transfer learning, using ResNet50 with image net weights and the top removed. For classification, I used global average pooling, then a layer of dropout, followed by a dense layer and then the classifier. Although this model is larger than previous models, it saw much better results in training.

   ResNet50 above these layers

global_average_pooling2d… (GlobalAveragePooling2D)	(None, 2048)	0	conv5_block3_ou
dropout_6 (Dropout)	(None, 2048)	0	global_average_
dense_6 (Dense)	(None, 256)	524,544	dropout_6[0][0]
dense_7 (Dense)	(None, 3)	771	dense_6[0][0]

Total params: 24,113,027 (91.98 MB)
Trainable params: 525,315 (2.00 MB)
Non-trainable params: 23,587,712 (89.98 MB)

   Summary of model, ResNet base, then custom classification head

5. Training:
   For model training, I use the high RAM setting and the A100 GPU inside of Colab pro. This setup allowed me to use a batch size of 256 to speed up training. I use the adam optimizer and an exponential decay learning rate function.

   When training the model, even the high-end hardware available with Colab pro was being pushed to its limit. 1000 images loaded in different forms and arrays quickly overwhelmed system RAM. Because of this, I deleted old arrays as soon as they were not needed, split my images into two groups after shuffling them, and saved these off to drive. I then loaded them back as needed to not overwhelm the hardware, both the CPU in storing these and the GPU in training. By using a high RAM and A100 Colab environment, I have access to 83.5GB of system RAM and a further 40GB of GPU VRAM. By limiting the number of loaded images, I was able to optimize the batch size to train the model most efficiently.

6. Compression:
   Now that the model is trained, I save it as a TFLite file so that it can be run on an embedded system. I used the same process as ECE 528 assignment 3 [10] for model compression. I log into my google drive on the raspberry pi and download this to the same folder as the python program.

7. Raspberry pi setup:

   I chose to use a raspberry pi for this project because of the large amount of support and tutorials as well as past experience using a Pi with other projects. To get the Pi ready to apply the compressed model and predict locations, I had to first install many libraries to support machine learning and camera functionality.

   To use the raspberry pi without connecting directly to a monitor and power supply, I set up VNC viewer to gain direct access to the UI. I do this by running an ethernet cable from the pi to my MacBook and setting the pi's IP address to be static. I then connect my laptops VNC viewer to this IP address and can use the pi without a full desktop setup. This tutorial from bigl.es was helpful here [6]. The MacBook is also able to supply enough power to run the pi and camera without any issue.

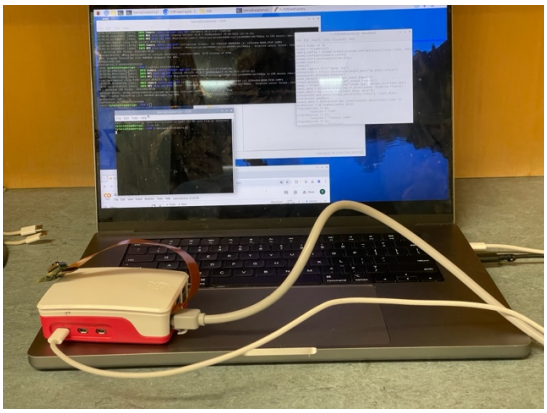


Image above: Full raspberry Pi 5 setup, using ethernet VNC to use my laptop as input and a display, pi camera module 3 and power supplied from my laptop.

The camera I am using is the pi camera module 3. To get this working, I needed to install a library to interact with the camera. I ran into trouble getting a working camera library that didn't interfere with dependencies for other libraries the project required. After trying pi cam 3 and libcam and having conflicting versions that were causing trouble, I tried picam 2 and was able to get this library working without conflict from tensorflow on NumPy versions. Then inside of a python virtual environment, I installed everything needed to run model inference, including tensorflow and tflite. The picam docs were a helpful resource in setting up and understanding the raspberry pi camera [7].

6. Applying the tflite model:

I download the tflite model from the training code in my Google drive and use the image taken as the input, I set the tensor and invoke the interpreter to get an array of confidence in each class. The class with the highest value is the prediction. This Google tflite on raspberry pi tutorial was helpful [8].

8. Evaluate results:

After gathering some predictions, I evaluate the percentage correct and the loss of the predicted classes.

V. Explaination of Early Results

My original plan was to use outdoor images pulled from Google street view, but this ended up being unsuccessful. The problem I was facing was most likely a lack of unique data and overfitting to the limited set. I was unable to get any kind of accuracy out of the model. With more extensive data collection and potentially a better suited model design, a model could be created that would perform well. There have been successful models as mentioned above, however with my limited amount of data in a limited space, I was too limited in my available data to get good results. If done right, there is a higher potential upside than with indoor location prediction, with more applications and greater amounts of relevant information captured in every image.

Because of this, I decided to pivot my evaluation testing to an indoor application, where I could more easily collect image data and test the model. The data and process from outdoor images still is helpful and I would like to come back to this part of the project given more time. Indoor images also have a few benefits for ease of training; lighting conditions are much more stable indoors, and weather and seasonal differences like leaves on trees and the color of the grass are all challenges that would require even more data augmentation and many repeated runs of image collection to fully capture the varying conditions.

Going forward, data will all be based on results from indoor testing, but I felt that the outdoor section was an important enough part of the process and could be built upon in the future with more data and time.

VI. Experiments and Results

Early Tests:

In the process of testing my project incrementally, I ran a few tests before collecting results to determine if my setup was capable of getting good results. The first test was ran using the street outside my house as one location and a randomly selected desert highway in Utah as the other location. This test had data sets of 144 images per location, all from the same street view panorama. The results confirmed to me that more data was needed for the model to effectively learn a location. Below are two example images:

Image 1: outside of my house, Image 2: US-6 in Utah

The contrast between the two locations was a way for me to test the model in an easier environment to confirm that the model was at the very least capable of accuracy in a simplified case. The model overfit the data and was not performing well, the prediction was consistently 99% confident it was outside of my house, even when given an image from the same area as location 2.

After determining that I should switch the focus of the project from outdoor to indoor, I tested this with two indoor locations. I used two rooms in my house to test, taking around 80 images at each location. I trained both a custom-built model that was described above and transferred the VGG 16 model to be used here. I used techniques learned in ECE 528 assignment 2 for this [9]. Again, accuracy was low, due to the low number of training images and lack of model complexity. The model was overfitting to the training data and unable to generalize. Below are sample images of the two classes used:



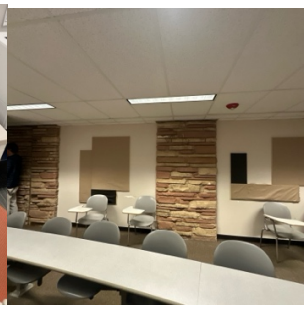
Image 1: family room Image 2: study

After seeing these results, I was realizing that my model may have been too simple for the task. I had assumed that a low number of classes and diverse enough features would result in a simple problem, easy enough for a standard CNN to solve, but because there are a variety of completely different objects in every scene, all belonging to the same class, the problem is more complex and requires deeper learning of complex relationships. This is when I decided to switch to a ResNet based model, with increased depth and residual learning.

Final Experimental Setup:

To test the model, I selected 3 locations in the CSU engineering building. The first location was the ECE lab next to the infill, the second was a hallway next to the physics labs on the second floor, and the third was the lecture room nearest to the physics room. Data collection involved taking around 250 images of different perspectives of each room using 2 different phones, both with a standard, wide, and macro lens. This gave a variety of different perspectives that should help to reduce overfitting.

Sample images from each of the 3 locations:

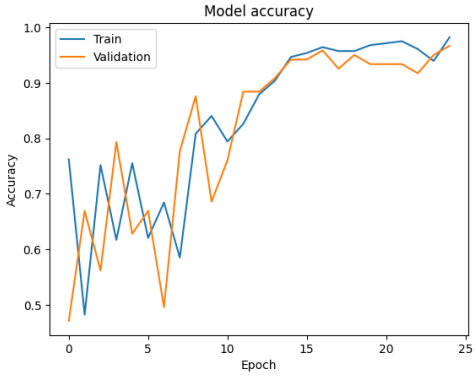


Location A, class 0: ECE lab

Location B, class 1: Lecture room


Location C, class 2: Physics hallway

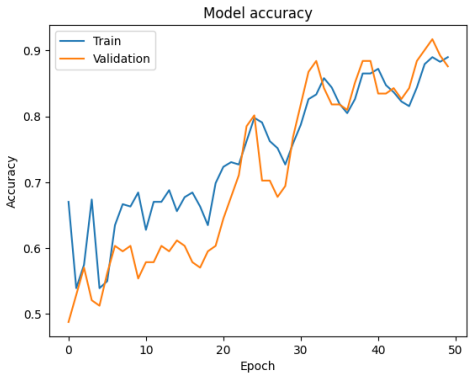
Training Results with differing setups:

Training the model for 25 epochs resulted in a 98% test accuracy and 96% validation accuracy. This was done with no augmentation.



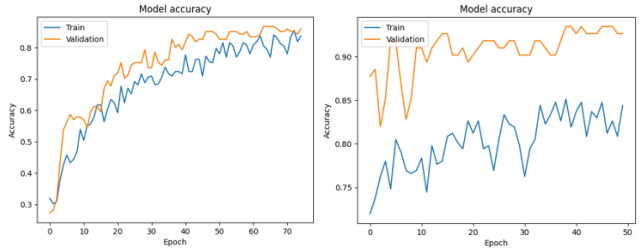
This was the best-case accuracy, there is no limiting image augmentation and this result of 96% validation accuracy confirmed that the model was working correctly and the setup to get to this point was all working.

By adding the black boxes to increase generalization, the accuracy was limited to 88% for training and 87% for validation over a training run of 50 epochs. This reduced accuracy is still high enough and is acceptable for the application.



My final training run used both sets of data, where previous runs only used the first half. This run also used all three augmentation techniques. To not crash out the RAM, I ran a 75-epoch training run with the first half of the data, deleted these images and then loaded the next half for a 50-epoch training run.

Testing accuracy was 85% and validation was 93%. This is lower than before for testing and is likely due to the larger data set forcing the model to generalize, which is a tradeoff that I was willing to make.



Left: first 75 epochs, where accuracy is steadily increasing
Right: next 50 epochs, where accuracy fluctuates and eventually increases slightly.

Compression: the model in its standard uncompressed form is 95.99MB and has 25M params. When compressing to tflite, I don't need any special quantization or other techniques. The standard compression to tflite reduces the size to 91.6 MB, which runs perfectly fine on the raspberry pi with an inference time of 2.4 seconds. Should this model be run in a real time application with a need for a high frame rate, more compression or a hardware accelerator would be needed.

Testing setup:

Now that the model is trained, I compress it into a tflite format and send it to the python code on the pi. Using the camera, I took 5 images at each location to get an accuracy and loss score for all 3 locations.

```
(env) tylersi@raspberrypi:~/528 $ python3 testCamera.py
[4:06:54.939950599] [5484]  INFO Camera camera_manager.cpp:325 libcame
[4:06:54.947612420] [5494]  INFO RPI pisp.cpp:695 libpisp version v1.0
[4:06:54.957648591] [5494]  INFO RPI pisp.cpp:1154 Registered camera /
vice /dev/media2 using PiSP variant BCM2712_C0
[4:06:54.961169635] [5484]  INFO Camera camera.cpp:1197 configuring s
[4:06:54.961302472] [5494]  INFO RPI pisp.cpp:1450 Sensor: /base/axi/p
0 - Selected CFE format: 1536x864-PC1B
QStandardPaths: wrong permissions on runtime directory /run/user/1000,
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
prediction: hallway
confidence: 0.9252356886863708
time to predict: 2.398768663406372
```

Example of running code on Pi, python program gives a preview for a few seconds to line up the shot, then captures the image and invokes the tflite interpreter on the captured image. The output is an array of predictions that I take the argmax of to classify, the confidence is the actual value of the argmax.

Loss metric: to evaluate the model, I will be using both a standard accuracy metric and categorical cross entropy, which I will calculate manually for testing data. This is the negation of the sum over all predictions of the score for the predicted class multiplied with the log of the highest predicted class. I want to calculate loss in addition to accuracy to get further insight how close the model is to being correct on failures and how confident it is on correct predictions.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Formula: yi is actual, y^i is predicted

Testing results:

After testing the model at 3 locations with 5 runs each, the overall accuracy was 86.7%. I calculated the loss at each location and found that the lowest performing location was the classroom. Both other locations were perfectly accurate and 2 were missed in the classroom with an overall lower confidence as well.

Note: if a value is missing, the loss formula did not need this value for its calculation. If there is just one value, the prediction was correct.

Table1

# Trials Lab c	class 0 confidence	class 1 confidence	class 2 confidence	Column 1
1	0.99			-0.0043
2	0.93			-0.029
3	0.99			-0.0043
4	0.98			-0.0085
5	0.58			-0.137
total loss				0.1831

Predictions for the ECE lab: 100% accuracy with only 1 lower confidence prediction.

Table2

# Trials class	class 0 confidence	class 1 confidence	class 2 confidence	Column 1
1		0.702		-0.107
2		0.1714	0.8047	-0.6139
3	0.687	0.0754		-0.0122
4		0.64		-0.124
5		0.79		-0.08
total loss				0.9371

Predictions for the classroom: 2 incorrect predictions that were high confidence. Correct predictions were lower confidence as well.

Table3

# Trials Hallw	class 0 confidence	class 1 confidence	class 2 confidence	Column 1
1			0.795	-0.0792
2			0.9999	-0.00043
3			0.993	-0.003
4			0.9563	-0.01855
5			0.84	-0.0636
total loss				0.16478

Total loss over all categories: 1.28499

Predictions for hallway: 100% accuracy and all high confidence predictions.

Total loss over all categories was 1.28499.

VII. Conclusion

In conclusion, I was able to successfully predict the location of images on an embedded system in an indoor environment with 86.7% accuracy. While I was unable to demonstrate this on outdoor data, I have an effective setup that can work with either indoor or outdoor applications. With more training data collected over multiple different conditions, the model I have

created should be able to be effective at location detection outdoors.

Along with learning how to create and prepare data, train and compress a ML model and deploy it on an embedded system, I also learned how to write adaptable code. My python functions can handle both indoor and outdoor image generation and processing, being able to either request from the API or load pre-existing images based on function parameters. This project can easily be improved upon by adding more locations. The performance and generalizability should increase as more, similar classes are added, because the most important and distinguishing features need to be learned. If enough data is gathered, a version of this project that outputs continuous coordinates is possible.

[1] S. Stelath, “Geoguessr AI,” GitHub repository, https://github.com/Stelath/geoguessr-ai

[2] S. Pasricha, “Indoor localization,” Colorado State University, https://www.engr.colostate.edu/~sudeep/indoor-localization/

[3] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using Places database," Massachusetts Institute of Technology (MIT), PAMI, [Online]. Available: http://places2.csail.mit.edu/PAMI_places.pdf

[4] Google, "Street View Static API overview," Google Developers, https://developers.google.com/maps/documentation/streetview/overview

[5] S. Pasricha, Lab assignment 1, [assignment], ECE 528, CSU

[6] L. Bigelow, "Directly connecting to a Raspberry Pi," [Online]. Available: https://bigl.es/directly-connecting-to-a-raspberry-pi/

[7] Raspberry Pi Foundation, "The official Raspberry Pi Camera Guide: PiCamera2 manual," [Online]. Available: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf

[8] Google, "TensorFlow Lite for Microcontrollers: Python support," Google AI, [Online]. Available: https://ai.google/dev/edge/litert/microcontrollers/python

[9] S. Pasricha, Lab assignment 2, [assignment], ECE 528, CSU

[10] S. Pasricha, Lab assignment 3, [assignment], ECE 528, CSU