Machine Learning Engineer Nanodegree
Capstone Project

David Ghan
February 17, 2018

I. Project Overview

When disaster strikes in developing countries, governments and humanitarian organizations depend on satellite imagery and maps to determine which communities or towns might be most vulnerable. This helps to maximize the impact of an aid mission's response to crises. Unfortunately, there is still a number of communities that have not been mapped.

To tackle this problem, international medical humanitarian organization Doctors Without Borders (MSF) launched crowd sourcing app MapSwipe. An example of the GUI can be found in figure 1. The app is part of the "Missing Maps" project, an open collaboration that aims to map vulnerable places in the developing world. The goal of MapSwipe is to enable users around the world to access and label satellite images of rural and developing areas as having features such as settlements or buildings, roads or trails, or rivers. The target labels change according to each project and area.

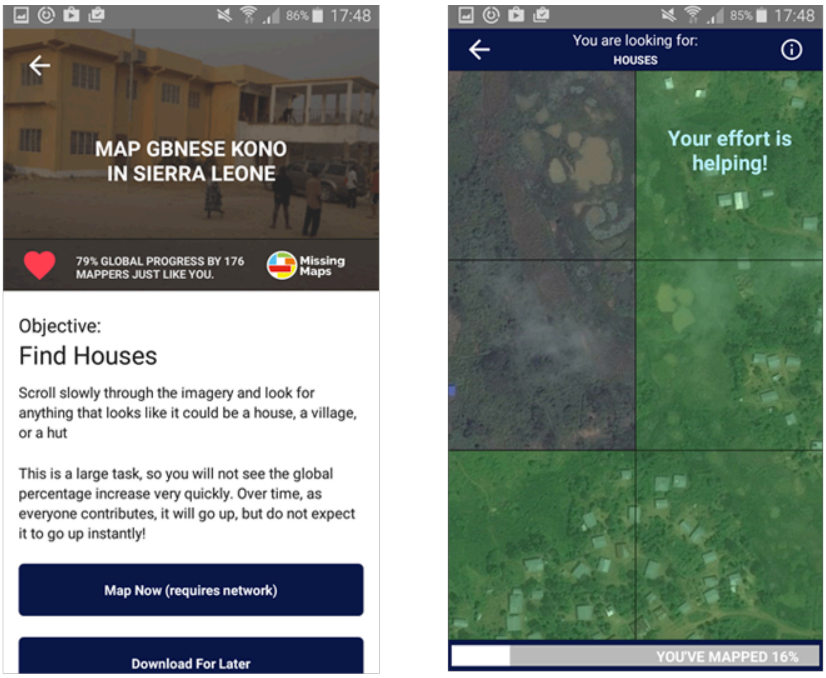


Fig. 1: Example of MapSwipe GUI for a Sierra Leone project (link)

By organizing the public in labeling large datasets of satellite imagery, MapSwipe saves hours and days for organizations like MSF from going through each satellite image, which is vital during humanitarian and disaster responses. As MSF's coordinator for the Missing Maps project Peter Masters said, "There are vast amounts of unmapped world out there and the human eye is still the most powerful tool we have to spot where people are living."

While I agree with Masters' statement, I believe deep learning can play an equally, if not bigger, role in quickly mapping communities around the globe. Machine learning has streamlined many classification tasks, including credit card fraud detection and image identification. I believe the task of mapping images can also benefit from the power of convolutional neural networks.

Problem Statement

The goal is to identify specific targets within satellite images. The targets vary according to MapSwipe's project requirements, but I plan to focus on identifying images with buildings, huts or settlements. This seems to be the most commonly required target amongst MapSwipe projects. This will be a binary classification problem, as in identifying whether or not a satellite image contains a building, hut or settlement.

Extracting the data

MapSwipe doesn't make the image data readily available, but they do release JSON files of coordinates, user labels, and project information of past projects that I will use to train my Neural Network. The link to MapSwipe's projects can be found here. Thanks to Github user philiptromans, I am able to access MapSwipe's project files and simultaneously extract the satellite images of that project from Bing Map's API. User philiptromans' repo can be found here. The python script "generate_dataset.py" is what is executed to extract the labeled imagery data from using MapSwipe's JSON file of projects as well as Bing Maps.

The readme file outlines how the "generate_dataset.py" code works and what inputs are required. The code I executed to extract the specific dataset is:

> *~/generate_dataset.py 2293 --bing-maps-key AtDPAFdfgOr5EYzHZeIb97DyzjlgTmygu1X6f5o3RVW1FCwEazOT74tgpyIWo31N -o Malawi -n 12000*"

Here, -n refers to the number of images, -o refers to the desired file name, and 2293 is the MapSwipe project ID for the images I am extracting. In this case since the project was Malawi satellite images, I named my file as such. The images were allocated as 9000 training images and 3000 testing images. However, I ran into some issues while attempting to work with the allocated text images, so I removed the test images and split the remaining images into train, test, and valid sets.

The resulting dataset contained images according to three labels:

1. 'built' – images that are classified as containing the target (a building for example
2. 'empty' – images that do not have the target
3. 'bad imagery' – images with cloud cover preventing users from classifying one way or the other.

Ultimately, I decided to remove images that were classified as 'bad imagery' for the sake of developing my model within a shorter time frame. I will go further into detail as to why I chose to remove these images from training and testing in the "Data Exploration" section of this paper.

Metrics

For this project I will use two primary metrics to gauge my model performance: accuracy and logarithmic loss. Accuracy is simply defined as the proportionate occurrences of correctly labeled test images by the model. This isn't an ideal metric in most cases, but I am able to rely on accuracy because the dataset is evenly balanced between both classes of images, "empty" and "built".

In addition to accuracy, I will use the log loss to track model performance, which takes into account the uncertainty of your prediction based on how much it varies from the actual label (fast.ai). This gives us a more nuanced view into the performance of our model. The log loss is calculated as so:

$$-(y\log(p) + (1-y)\log(1-p))$$

II. Analysis

Data Exploration

The dataset is produced via MapSwipe's API. The app API lists all projects and user classifications that have been conducted. With the help of Github user philiptroman's code, I am able to select from specific projects to download image data along with the user results. I can also limit the number of images to download, making memory management easier. Each image is 256pi x 256pi and has three channels (RGB).

For this project, I am focusing primarily on a project that focused on a rural area in the Southeast African country of Malawi. The project was to identify areas via satellite images that have evidence of buildings or structures, including for example homes, huts or commercial buildings.

As mentioned above, the images are labeled with three different classifications: bad_imagery, built, and empty. One issue that I had while going through this data was with the 'bad imagery' files (figure 2). The files represent tiles in the MapSwipe interface that cannot be classified due to cloud cover or other image distortions. Looking at a few examples of these images, it's difficult for me to determine with my own eyes characteristics that might set these images apart from other images that are classified as 'empty'.

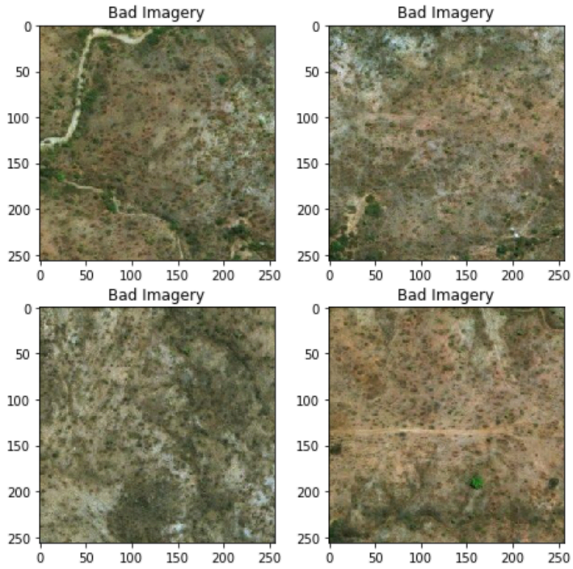


Fig. 2: Example of images classified as bad_imagery

My suspicion for this is that users labeled some of these images bad_imagery rather than labeling these as empty. For the sake of the purpose of this project, I decided to remove the bad_imagery images to ensure the model performance is not affected by mislabeled images. An alternative approach to this problem might mean I include these images with 'empty' labels, but I wanted to ensure with as much certainty as possible that my model trained on images that were labeled correctly.

After removing the bad_imagery images, the dataset consists of 9000 images total. They are partitioned as being 7400 labeled images for training and validation and 1600 unlabeled images for testing. I decided to set aside 1000 images for validation during training. The final organization of the dataset can be found in figure 3:

Total images:	9000
Training set:	6400
Validation set:	1000
Testing set:	1600

Fig. 3: Our dataset

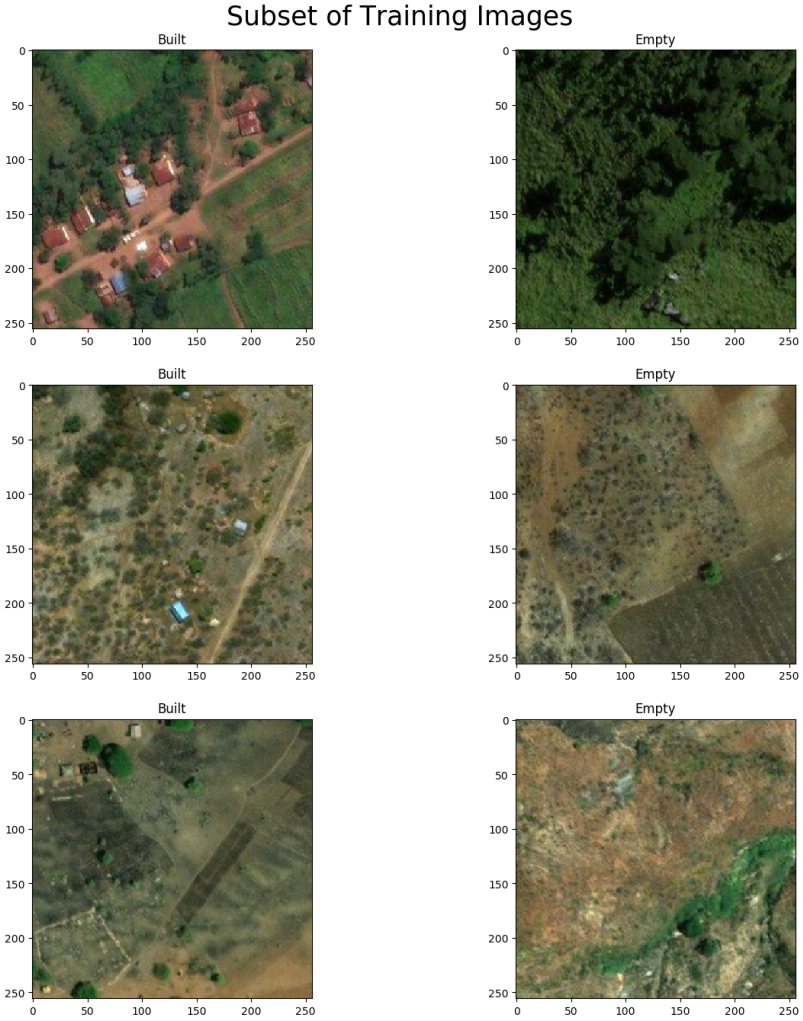


Fig. 4: Example of "built" and "empty" images from our dataset

Exploratory Visualization

In order to get a good idea of how both classes differ, we can see the distribution of RGB values for each image. The histograms of how the colors are distributed will tell us how much of each channel we can find in each image. I calculated the summary statistics for the RBG distribution after normalizing the data:

Built Images Color Distribution				
Channel	**Min**	**Mean**	**Std Deviation**	**Max**
red	*0.00*	*0.38*	*0.14*	*1.00*
green	*0.00*	*0.38*	*0.11*	*1.00*
blue	*0.00*	*0.27*	*0.10*	*1.00*

Fig. 5: RGB distribution for "built" images

Empty Images Color Distribution				
Channel	**Min**	**Mean**	**Std Deviation**	**Max**
red	*0.00*	*0.33*	*0.13*	*1.00*
green	*0.00*	*0.34*	*0.11*	*1.00*
blue	*0.00*	*0.24*	*0.10*	*1.00*

Fig. 6: RGB distribution for "empty" images

We see there's some difference in how the means of each channel, empty images being slightly less than built images. Histograms of the first six images in the dataset help to visualize how the channels are distributed.

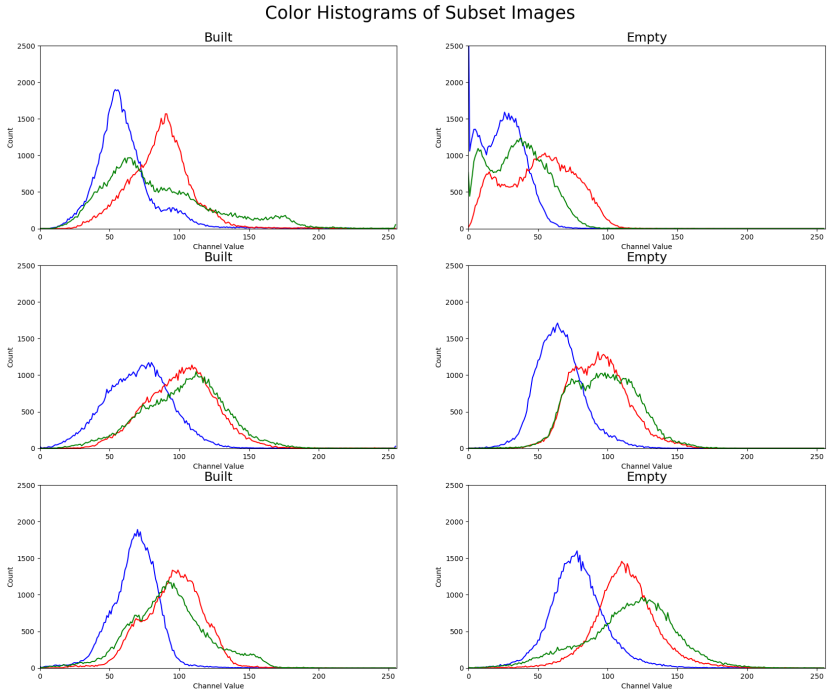


Fig. 7: Sample of how our RGB channels are distributed among "empty" and "built" images

Algorithms and Techniques

For this project, I will explore two convolutional neural networks for training our data to classify satellite images with buildings. The decision to use CNNs is reflective of this paper by Stanford University alum Daniel Gardner and David Nichols. The paper focuses specifically on satellite imagery of the Amazon Rainforest. Like Gardner and Nichols, I will be creating a scratch model in addition to transfer learning with pre-trained models.

The model from scratch will consist of:

1. *3x3 convolutional layers*: This is a 3pi x 3pi area of the image the model learns before moving onto the next 3x3 layer. This way our model can study the image in small sections rather than single pixels at a time.
2. *ReLU activation layers*: It is standard practice to use ReLU activation layers prior to using an activation layer on our output. The ReLU activation layer changes the negative activations of the convolutional layer to 0. This helps the model run exponentially faster without sacrificing performance.
3. *Flattening layer*: This flattens our data into an array.

4. *Sigmoid activation*: The sigmoid activation at the end of our model will make a determination for our labels as 0 or 1, rather than probability. This is preferable for binary classification. For multi-label classification a machine learning engineer would likely use the Softmax activation layer which produces probabilities.

I will also explore two different pre-trained models when applying transfer learning on the dataset: VGG-16 and InceptionV3. The VGG-16 pre-trained model is developed by a team of members of the Visual Geometry Group at University of Oxford. My reasoning for exploring VGG-16 is due to the model's simple architecture, along with its long-standing performance with other deep learning tasks. The model consists of 16 fully connected layers with 3x3 filters in the convolutional layers.

The second pre-trained model I will use is the InceptionV3 model developed by the GoogLeNet team. This model is robust in its architecture, but the train time on our bottleneck features is comparable to VGG-16. This model's strength is its complexity at minimal computational cost.

The primary reason for using transfer learning is the speed and accuracy is hard to beat. With a pre-trained model, we only have to subject our data to one-time training to extract the data's bottleneck features. Our model is then reduced to a final layer, with the input of our bottleneck features

Benchmark

My benchmark model will be entirely random choice. This will be a good baseline for my model to launch f rom. As this is a binary classification problem, I will aim to produce a model that is more than 50%. The ac curacy of the random choice model ended up producing a 50.75% accuracy against the Malawi test imag es, with a log loss of 1.01498.

III. Methodology

Data Preprocessing

Let's first look at the structure of our satellite dataset. The Malawi train, valid and test datasets are evenly distributed between the two classes:

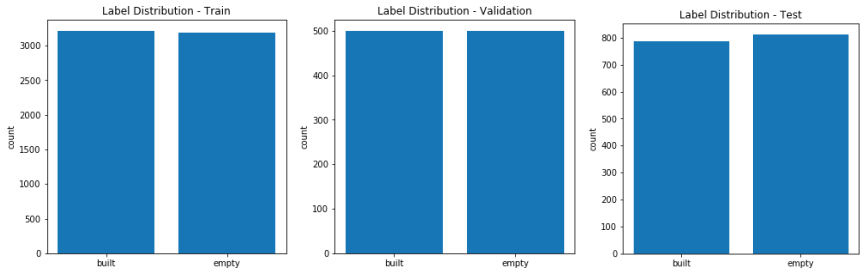


Fig. 8: Distribution of classes in each subset of our data

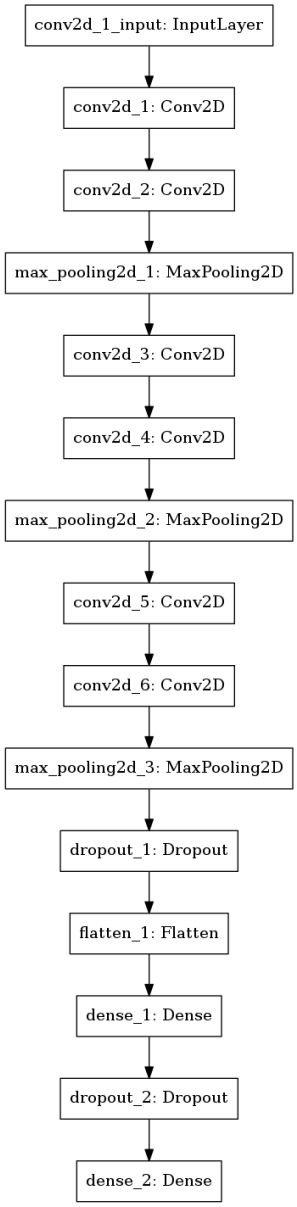
Initial preprocessing of my data for all models begins by converting the image dataset into 4D tensors compatible for 2D convolutional layers in Keras. This involves reading images and converting into arrays for us to compute and learn statistically. For example, the train dataset takes the shape of (6400, 256, 256, 3), which translates to 6400 images, a size for each image of 256pi x 256pi, with three channels to read color (RGB). I then normalize the datasets (range from 0 to 1) by dividing the tensors by the range of channel values (255) in order to help increase the performance speed of our models.

For the scratch model, I will apply image augmentation to my dataset in order to attempt to increase my model score. The augmentation will include flipping images and moving them within 20% of the image's width and height. I will refrain from image augmentation for my pre-trained models.

Implementation

Scratch CNN Model

The scratch model that is built in Keras consists of six Conv2D layers with 3x3 Max Pooling layers every two Conv2D layers. I flatten the image arrays and run the data through a Dense layer and Dropout function. The architecture from Keras can be found below:



Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 254, 254, 32)	896
conv2d_2 (Conv2D)	(None, 252, 252, 32)	9248
max_pooling2d_1 (MaxPooling2	(None, 126, 126, 32)	0
conv2d_3 (Conv2D)	(None, 124, 124, 64)	18496
conv2d_4 (Conv2D)	(None, 122, 122, 64)	36928
max_pooling2d_2 (MaxPooling2	(None, 61, 61, 64)	0
conv2d_5 (Conv2D)	(None, 59, 59, 128)	73856
conv2d_6 (Conv2D)	(None, 57, 57, 128)	147584
max_pooling2d_3 (MaxPooling2	(None, 28, 28, 128)	0
dropout_1 (Dropout)	(None, 28, 28, 128)	0
flatten_1 (Flatten)	(None, 100352)	0
dense_1 (Dense)	(None, 256)	25690368
dropout_2 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 2)	514

Total params: 25,977,890
Trainable params: 25,977,890
Non-trainable params: 0

Fig. 9: Scratch Model Architecture

Each Conv2D layer has a 3x3 filter and a ReLU activation. I apply three max pooling layers, each with a pool size of 2x2. The first dropout layer is set to 0.2 and the latter is 0.4, to help prevent overfitting on our training data. After flattening the data, I apply a dense layer before the second dropout layer, ending the training with another dense layer and a sigmoid activation.

Considering the high number of params or units the model will be processing (more than 25 million), the model was run using an AWS EC2 instance with 8GB of GPU memory. Even with this horsepower, the model was still running over the course of several hours. For my scratch model, my parameter setup is as follows:

1. Image preprocessing (with Keras ImageDataGenerator):
    a. width_shift_range=0.2,
    b. height_shift_range=0.2,
    c. horizontal_flip=True,
    d. vertical_flip=True)
2. Optimizer: Adam
    a. Learning rate=0.001
3. Epochs: 20
4. Batch size: 32
5. Loss: binary_crossentropy
6. Model metric: accuracy

Transfer Learning Architecture

A lot of the labor in our transfer learning models is extracting our bottleneck features. This is computationally expensive but only has to be done once. With our MapSwipe bottleneck features in hand, we can use a simpler model, which will allow us to run for more epochs. Here is what are simple architecture looks like:

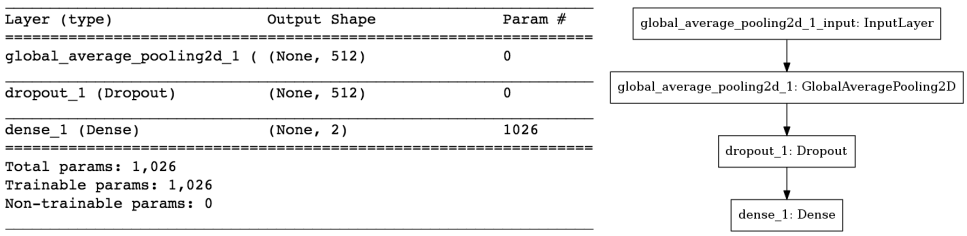


Fig. 10: Architecture using pre-trained weights

The first layer is a global average pooling layer which will help to prevent overfitting. I follow this with a 0.2 dropout layer, and close with a dense layer and sigmoid activation. This model will be applied to both InceptionV3 and VGG-16 models.

Transfer Learning: InceptionV3

For the InceptionV3 model, I conducted the same preprocessing as our scratch data, converting our images into 4D tensors. However, I will not be using image augmentation during the transfer learning process. In order to prepare the model, I will run the MapSwipe images through the InceptionV3 model to withdraw the weights. The architecture of the InceptionV3 model is deep and convoluted, as shown to the right. Fortunately, the heaving lifting is done for us and we only need to apply our weights to the above model and fine tune our parameters.

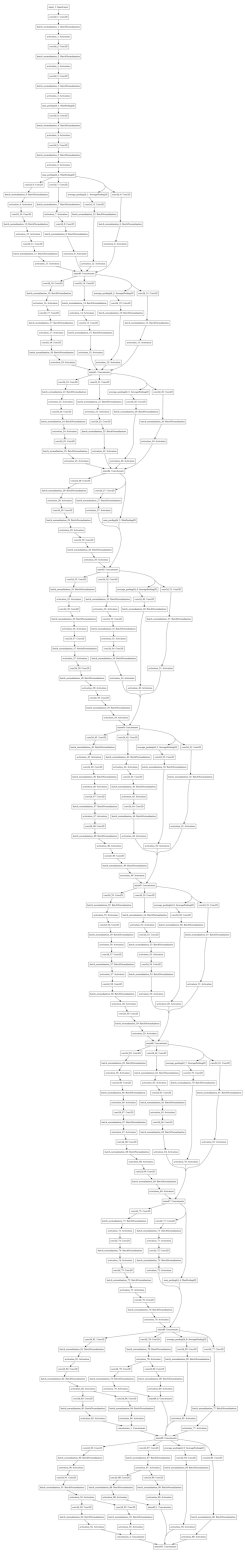


Fig. 11: InceptionV3 Architecture

For my InceptionV3 model, my parameter setup is as follows:

1. Optimizer: RMSProp
    a. Learning rate=0.001
1. Epochs: 200
2. Batch size: 126
3. Loss: binary_crossentropy
4. Model metric: accuracy

Transfer Learning: VGG-16

While I mentioned how InceptionV3 model architecture is complex, the VGG-16 architecture is on the other end of the spectrum with a simpler group of layers, shown to the right. After extracting the bottleneck features of the MapSwipe data with VGG-16's pre-trained model, the shape of our data becomes (n, 8, 8, 512), a significantly lower level of channels compared to the InceptionV3 bottleneck features.

For my VGG-16 model, my parameter setup is as follows:

1. Optimizer: RMSprop
    a. Learning rate=0.001
2. Epochs: 200
3. Batch size: 126
4. Loss: binary_crossentropy
5. Model metric: accuracy

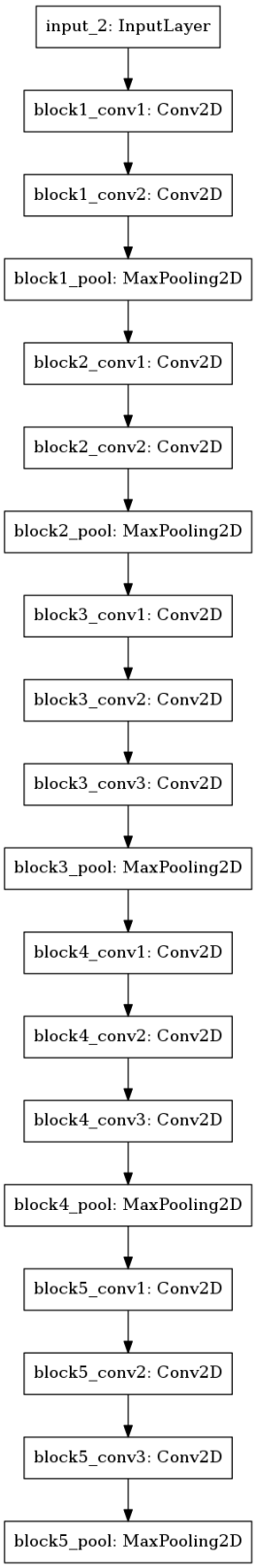


Fig. 13: VGG-16 architecture

Refinement

I began with running my models without image augmentation, which yielded a pretty low performance score. The problem I was facing was having to run the model overnight on an EC2 instance every time I wanted to alter the model or parameters. This kind of trial and error is computationally as well as monetarily expensive ($0.90/hour for the p2.xlarge instance on AWS). In the future, I plan to explore the possibility of using a grid search method to optimize my model parameters in a computationally cheap manner. This should help to make my process of model architecture development more efficient.

The one key parameter that seemed to alter my model performance was the chosen optimizer that is built into Keras. I started my InceptionV2 model with the Adagrad optimizer. However, I found that using either the Adam or RMSprop optimizers produced slightly better results. To better understand why this might be, I referred to information found here thanks to Stanford PhD student Andrej Karpathy, which explains how the Adagrad optimizer tends to stop learning earlier in the training process when compared to other optimizers like Adam or RMSprop.

For my InceptionV3 model, here is how the performances compared with the Adagrad and RMSprop optimizers:

Optimizer	Accuracy	Log Loss
Adagrad	74.07%	0.5145
RMSprop	75.13%	0.5035

Fig. 12: InceptionV3 performance

For my final scratch model, I utilized image augmentation to diversify my training images. This consisted of 20% shift in width and height, as well as a vertical and horizontal image flip. However, due to the computational restrictions, I was only able to run this model at 20 epochs. This is a stark difference when compared to the speed of using a pre-trained model with extracted bottleneck features.

For the InceptionV3 and VGG-16 models, I primarily focused on fine-tuning my parameters. I could adjust the architecture of the last layers that trained on the bottleneck features, but the real difference was with the adjusted learning rates and different optimizers.

IV. Results

Model Evaluation and Validation

Unsurprisingly, the pre-trained models outperformed the scratch model, according to both the accuracy and log loss metrics. The two pre-trained models performed somewhat equally:

Model	Accuracy	Log Loss
Benchmark Model	50.75%	1.0150
Scratch Model	65.00%	0.6275
InceptionV3	75.13%	0.5035
VGG-16	75.63%	0.5064

Fig. 14: Model Metrics

For my scratch model, the reported accuracy and binary cross entropy was only slightly better than my benchmark model of random choice:

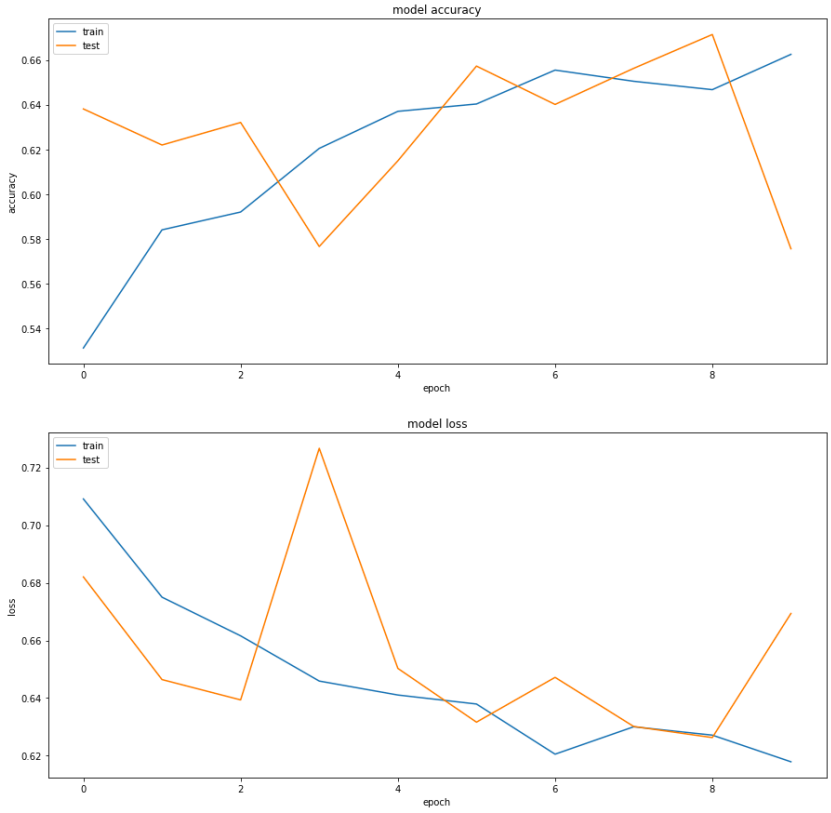


Fig. 15: Scratch Model performance plot

For the scratch model, we can see the validation accuracy and loss are a bit sporadic. The train performance seems to be at pace that is more to my liking, as it doesn't seem to make any significant leaps or dives. Running on a more powerful workstation would allow to introduce more epochs, which I suspect would yield better results than the 20 epochs this scratch model used.

The first of the two transfer learning models, InceptionV3, performance significantly better than the scratch model. The model was run for 200 epochs with batch sizes of 126. Here are the results:

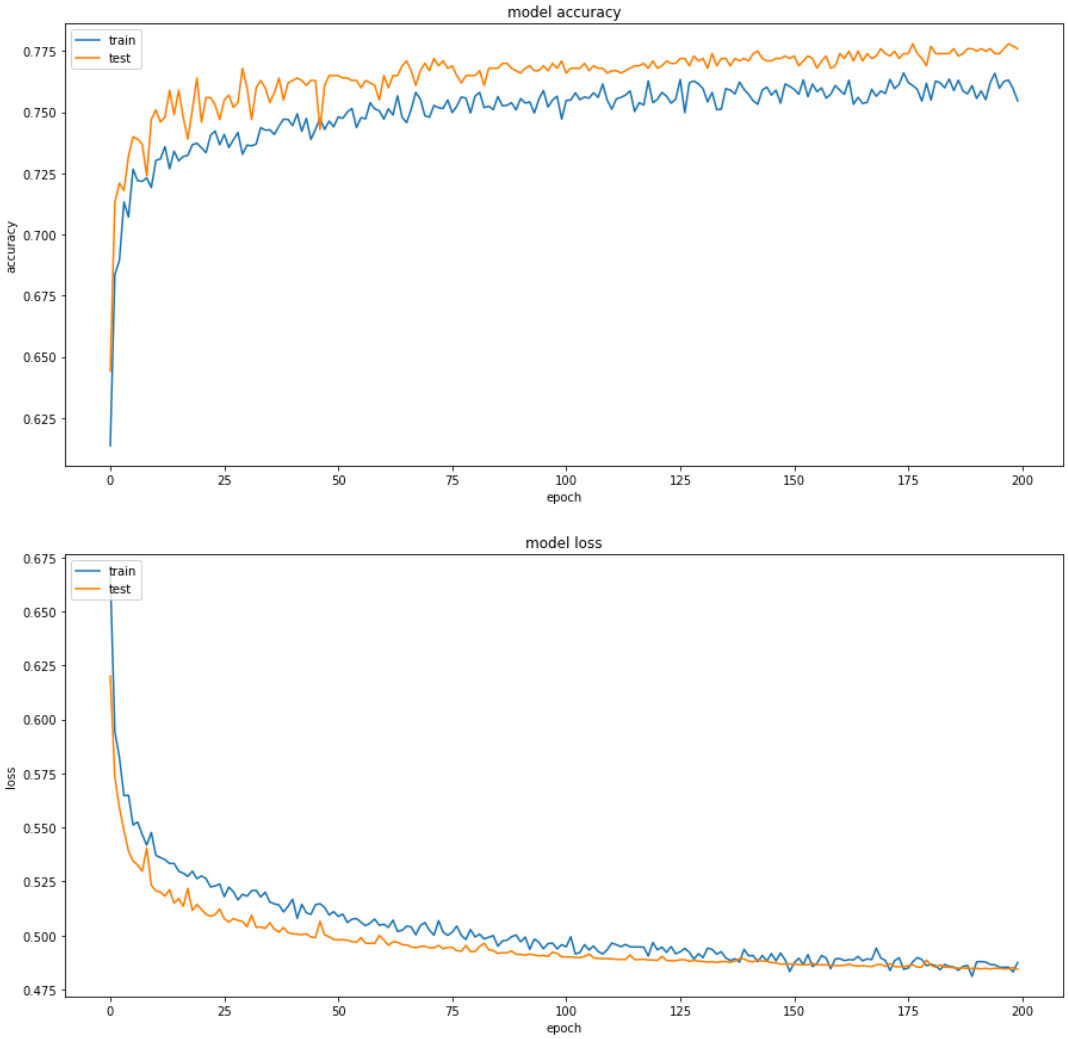


Fig. 16: InceptionV3 pre-trained model performance plot

The VGG-16 model only slightly outperformed the InceptionV3 model. I used a similar architecture for the final layer as I did with the InveptionV3 model. I wanted to use the same optimizer, RMSprop, as my InceptionV3 model. I kept the learning rate at default (0.001) and ran the model for 200 epochs with batch sizes of 126 with a sigmoid activation layer:

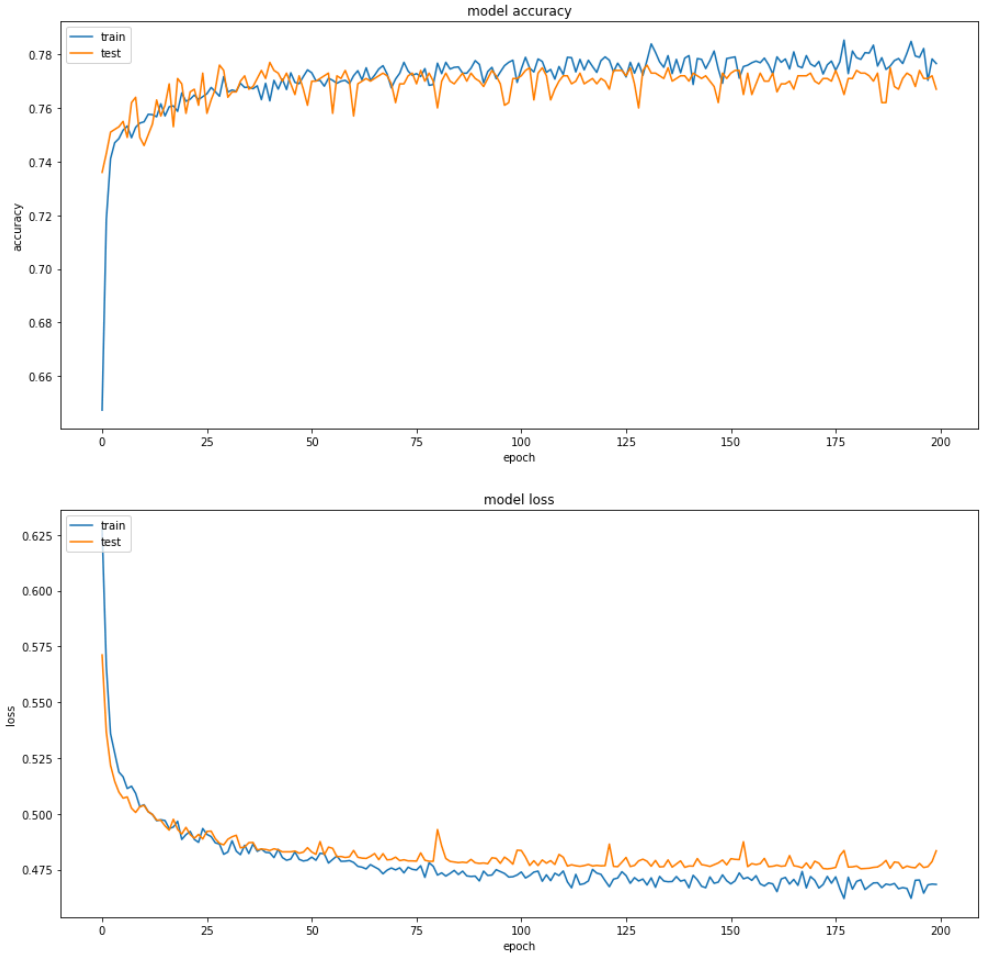


Fig. 17: VGG-16 pre-trained model performance plot

The plotted performance of the VGG-16 model seems to communicate that it is generalizing well on the validation set. When couple with the accuracy and log loss performance on the test set, it's clear the VGG-16 model outperforms the other models. Looking at the confusion matrix, we do see a little bit where the model does poorly, specifically with false positive classifications:

	Predicted=1	Predicted=0
Actual=1	524	126
Actual=0	264	686

Fig. 18: VGG-16 Confusion Matrix

In general, all models seemed to underperform to my expectations. I am pleased that the models outperformed the benchmark model of random choice, but I believe with more time and computing power, these models could be optimized further to perform better on a general dataset.

V. Conclusion

Free-Form Visualization

After looking at a few samples of the incorrectly labeled images, it seems there is some opportunity to further exploit the MapSwipe images during the feature engineering phase. As mentioned at the start of this report, MapSwipe's GUI supplies the user with six tiles of satellite images. These are tiles that are geospatially next to each other on the map, giving us an opportunity to perform a nearest neighbor analysis of the classified tiles around those that contain buildings (or the target variable). This might help us understand a way to see through terrain like boulders or rivers, reducing false positive classifications.

Fig. 19: Example of MapSwipe GUI

While looking at a sample of incorrectly labeled images, it's almost as if we can see why the model falsely labels some images as having buildings. In general, terrain is often homogeneous. This dataset is no exception, having gone through a number of images which encapsulate rolling green hills that blend into forested areas. I suspect that the model sees "abnormalities" or less common characteristics and assumes these to be buildings.

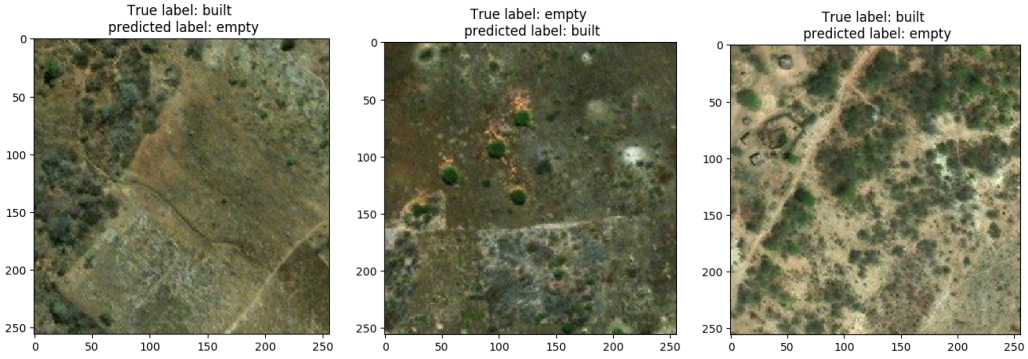


Fig. 20: Example of incorrectly labelled images from VGG-16 model

Reflection

Much of this project was learning as I coded. I addressed the problem with a familiarity of image classification for objects, but this being a geospatial problem carried unique obstacles that I familiarized with and addressed throughout the process. I suspected the images would be more easily classified by pre-trained models, so I had the plan to utilize transfer learning at the very start. I explored the possibility of pre-training a model using an alternative dataset to ImageNet, which is generally used with transfer learning. ImageNet is millions of images of objects, but the problem that I was addressing was identifying geographical targets like buildings. An alternative dataset that a model might have been pre-trained on is the DeepSat dataset. However, I moved away from this due to the computational and time restraints. Perhaps this can be pursued in the future, but I wanted to address the problem with tools that were available to me.

For the scratch model, I used image augmentation as this is a method that is likely to improve any image dataset by adding more data to the training data through slight image movements and rotations. I was unsure how I could take advantage of this using a pre-trained model which is trained on bottleneck features, so I did not concern myself with this step during the transfer learning process.

I was not surprised when the VGG-16 model outperformed the scratch model, but I was anticipating better results. I realized at this point how much more work could be done in order to improve and generalize my model. There is also an opportunity to take advantage of additional datasets from different MapSwipe projects that might help to build a more robust model that can better generalize to new data. This falls into the same previously stated restraint of working within modest computational capacity. More data can help the model, but it will slow the experimentation process.

This leads into my last point upon reflecting over the process. My method of optimizing my models was largely done manually. Fortunately, this was not as problematic with pre-trained models as it was with the scratch model, since training was signification faster for the former. The final model was produced through trial and error as well as additional research into optimizers and parameter tuning.

Improvement

One key way I would improve this model implementation is exploring the possibilities of further exploiting the image features. Perhaps more focus on shape and edge detection could improve our model accuracy. This will require a more robust understanding on my part to incorporate these features into the training phase of the model. I believe this model is not the best solution that exists, but it is a stepping stone to improving upon my current solution. I would consider this model to be my new benchmark when building an even more robust model in the future.

I will also return to the first step of extracting my dataset to see if there's a way to improve how I optimize how I use the images as they relate to each other. In other words, there is a unique opportunity to look at our images as they relate to nearest neighbors. Perhaps this will provide an opportunity to better train our model to handle varying terrain and handle random objects that might be mistaken for target variable.