# Machine Learning Engineering Nanodegree

# Landmark Image Recognition Report

May 30, 2018

By: Jacob Barrett

## Project Overview

The ability to identify an image as a particular object is at times a difficult skill for people, however with recent advances in machine learning and image recognition, this task has become much faster, simpler, and more accurate than ever before. Even with these recent breakthroughs, there is still a long way to go in perfecting image recognition towards new images.

One area of image recognition that could use improvement is recognition of landmarks, both famous and infamous. In fact, Google has recently posted an online competition on Kaggle[1] with the focus of identifying landmarks which is the data used for this capstone project. This projects goal was to take a wide variety of landmarks and identify key aspects of each image to help identify these images in the future. Faster and more accurate recognition of these images or others could help to improve a number of systems and applications used every day by people worldwide.

## Problem Statement

Machine learning has gotten very good at analyzing images from differing categories and put new unlabeled images into the appropriate category. However, in a domain as vast as landmarks, image recognition is still a major problem. I will use deep learning to train a complex convolutional neural network (CNN) to analyze the pixel values and patterns of each image, resulting in a model that can easily identify image characteristics and label new images as the most likely landmark. The model will use aspects of two proven CNNs and combine them to produce an improved solution. The output of my project is the individual images id followed by the appropriate landmark id if the probability is great enough.

## Metrics

My CNN was trained on a large dataset of images containing various landmarks, resulting in an output coefficient. When training the model, I evaluated each model by measuring the accuracy of images specific landmarks in the training dataset to that of the validation dataset. After compiling each model, the resulting model had the highest accuracy score and lowest loss, which was measured by subtracting actual and predicted values. I chose to use accuracy as the primary metric rather than F-1 score or F-beta because precision and recall are not as important. Meaning, it is not incredibly important if the model misidentifies a landmark, so accuracy is a perfectly acceptable metric. After training was completed, I began using testing images. To measure the resulting output, I aimed to minimize the total loss function and display the highest landmark probability.

## Data and Exploration

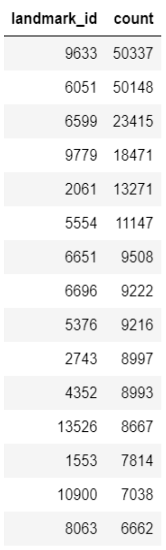
For this project, I downloaded the available training and testing datasets from the Kaggle competition[1]. The training data consists of 1,225,029 images, 14,951 landmark categories, and no uniform picture size. The images are all mostly in RGB format, however some images have a fourth transparency layer, and some images contain gifs that need to be converted into individual JPEG frames. Every image was labeled with its designated landmark_id.

Due to the large dataset size and images to run through the CNN, I decided it would not be cost effective to train the entire dataset, so for this project I only utilized the first 5000 images. From these images, the training data contains 2,286 unique landmarks. These images will be used to train the CNN to understand the individual characteristics of each landmark. The original testing data consists of 117,703 images, no uniform picture size and all images are in RGB format. Similar to the training data, I elected to only use the first 5000 images. Both the training and testing images will need to be normalized and preprocessed, prior to running them through the CNN. There are no NULL values in any of the datasets, training data only contains 'id', 'url', and 'landmark_id'; while testing data contains 'id' and 'url'.

Prior to training any CNN models, the training and testing images are loaded and scaled down to a size of 250 by 250 pixels. This scale does slightly distort many of the images due to a change in aspect ratio, but this makes it much easier for the different models to analyze the images and make more accurate calculations.

Figure 1: Sample of training images

To help visualize some of the major features in the data, I first looked at which landmarks appeared with the greatest frequency. Below are two figures, the first has a numerical breakdown of the fifteen most frequent landmarks, and figure two shows this information as a bar graph.



| landmark_id | count |
|---|---|
| 9633 | 50337 |
| 6051 | 50148 |
| 6599 | 23415 |
| 9779 | 18471 |
| 2061 | 13271 |
| 5554 | 11147 |
| 6651 | 9508 |
| 6696 | 9222 |
| 5376 | 9216 |
| 2743 | 8997 |
| 4352 | 8993 |
| 13526 | 8667 |
| 1553 | 7814 |
| 10900 | 7038 |
| 8063 | 6662 |

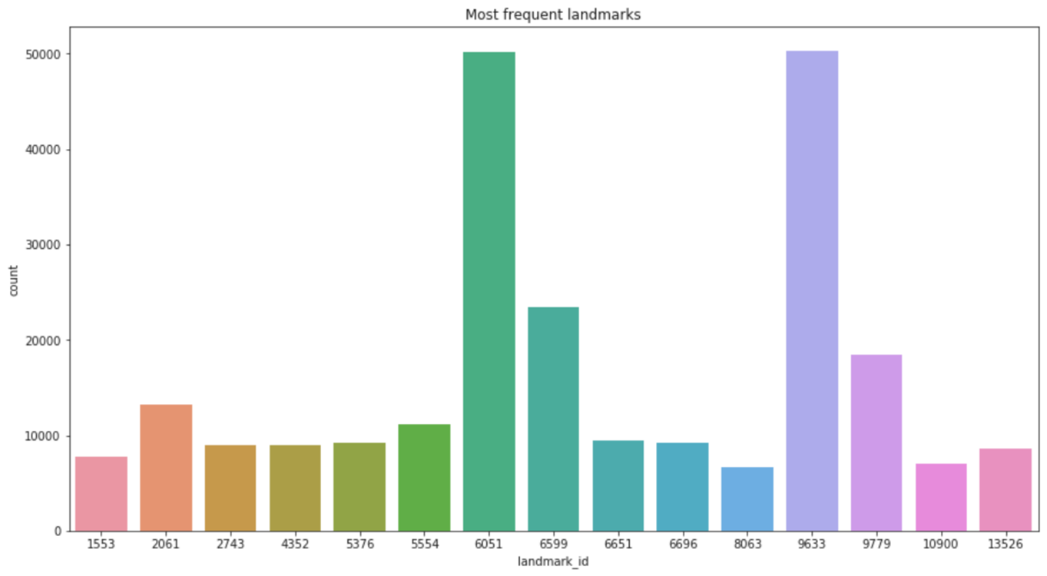


Figure 2 (left): Most frequent landmarks with occurrences summed together. Figure 3 (right): Most frequent landmarks visually plotted as a bar graph.

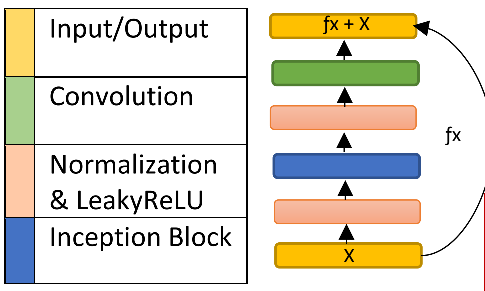
Upon further exploration of the landmarks, I searched to find the least frequently occurring landmarks. This produced a list of landmarks that only have one occurrence, meaning only one of the 5000 images used correspond to that landmark. Since the ability for a CNN to accurately train a model to identify images is directly affected by both the number of images in the dataset, and the number of images linked to each specific category, I thought it would be prudent to find how many landmarks had only one image. There are 159 landmarks that have a single image, which means that these images will have a hard time being accurately trained to identify similarities to new images.

## Algorithms and Techniques

Prior to any useful CNNs or preprocessing the images, the data needs to be converted from a url into an image. To do this I used the “urllib.requests” with Pillow functions to open the image before resizing to a 250 by 250 pixel image and converting all images into “RGB” format. Converting the image into “RGB” is necessary to allow the image to be run through the Keras CNNs, as well as aid in issues associated with URLs that are gifs rather than a basic image type. After converting the individual URLs to images, the images are saved into a folder that is labeled by the images corresponding landmark_id, and the image itself is saved as a JPEG file with the images ‘id’ as the filename. An example of this would be the saved image with a file path “C://home/GLRC/Training/10117/000bj4ccd1234.jpg”; where the “10117” would be the images corresponding landmark_id, and “000bj4ccd1234” would be the images id. In the event that an image could not be successfully loaded, or the URL did not have any image attached, the path should be deleted.

Once all the data have been successfully converted into an image format, I searched for landmark folders that did not contain any images. I found twelve landmark directories that were empty, to avoid future problems with these folders I deleted them as well. The next step was to preprocess the images before creating and feeding them into my CNN. To create my CNN, I took ideas from the pre-existing architectures of “ResNet 50” and “Inception V3” as the basis for my models’ design. A major issue facing many if not most deep neural networks is when networks “start converging, and a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated” [2]. To help with this issue, the ResNet 50 model uses the idea of residual learning where it takes a former input and maps future layers to that input. The belief is that “it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping” [2]. Figure 4 shows a basic block diagram of the residual network that I used in my architecture. As can be seen in the figure below, the CNN uses a convolutional layer, the most important layer in any image recognition model.

A convolutional layer works by scanning the original image with a “window” or frame of a specified size. An example would be an image that is scanned by a 5x5 frame. The frame reads each pixel value of the original image and combines them all into a single output. An image of this can be viewed in figure 6. Another very common layer that I use is a Max-Pooling layer, which is used to reduce the size of the image and the total number of parameters. This layer is used throughout my model to decrease the parameters of the model and to periodically enforce the residual learning code.



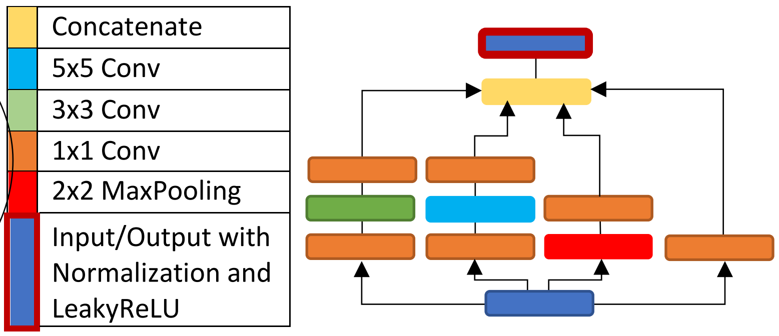
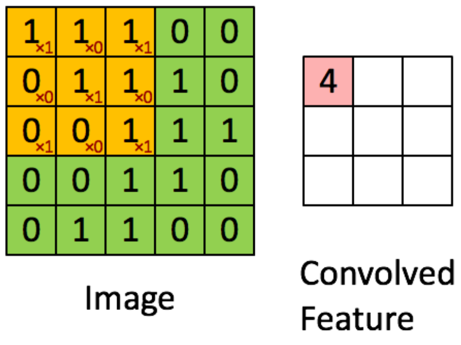


Figure 4 (left): Residual network block diagram with color key. Figure 5 (right): Inception block diagram with color key



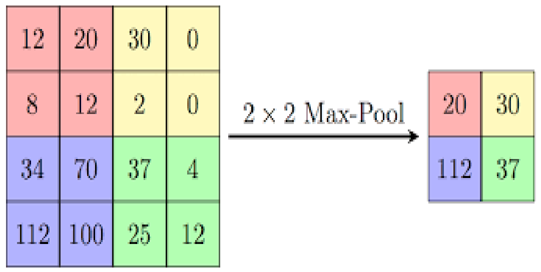


Figure 6 (left): Convolutional layer with a 3x3 filter outputting a single pixel value.
Figure 7 (right): Max-Pooling layer with a 2x2 filter.

Another aspect that I modeled my network off of is the Inception-V3 model, which uses parallel convolutions to increase the models' ability to train while decreasing the number of parameters[3]. I implemented an inception model with a more advanced configuration in the attempt to reduce overall dimensionality by implementing 1x1 convolutions. As can be seen in figure 5, the inception includes four parallel convolutional layers, with three of those layers beginning with a 1x1 convolution in attempts to "compute reductions before the expensive 3x3 and 5x5 convolutions" [3]. I found that applying another 1x1 convolution after the more expensive 3x3 and 5x5 layers, the parameters were further reduced. After the architecture has been built, the model will be compiled with stochastic gradient descent and RMSprop, equations are found in figure 8.

SGD: $\theta=\theta-\eta\cdot\nabla\theta J(\theta;x(i);y(i))$

RMSprop: $E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$

$$\Theta_{t+1} = \theta_t - \frac{n}{\sqrt{E[g2]t+\epsilon}}\, g_t$$

Figure 8: SGD and RMSprop equations[4].

## Benchmark

I decided to use the ResNet-50 image classification model to compare with the output of my CNN. I downloaded the ResNet-50 model from the Keras library, with the model weights set to "ImageNet". When the model was trained to ImageNet, it obtained a top-5 accuracy score of 92.9%, with 25,636,712 parameters and a model depth of 168 layers. With such a high accuracy score, and my model being designed around the idea of residual mapping, I thought it would be a good idea to use as a benchmark. To compare the models, I looked at training accuracy scores, image probabilities, and the time it took to run the model.

## Data Preprocessing

The original data needed to be converted from a URL into a usable image for the CNN models. As stated above, I implemented url.lib.requests to convert the url into an array that is saved to the specific file path. From here, I used the Pillow library to open the image, convert it to "RGB" format, resize the image to a 250x250 pixel square, and save the new image to the same file path. During this process, I ran into a number of issues with the raw data itself. There was no missing data in the files, however some of the URLs lead to sites with no images. To ensure that the CNN did not get an empty file to train, I deleted any URLs and file paths that could not be opened properly. Out of the 5000 images used for training, there were a total of 33 that were deleted due to an image issue. A final issue that I encountered was that a few of the URLs lead to gifs, which could be opened but not saved as a JPEG image. As stated above, by converting all the images that were opened using PIL to "RGB" format, this allowed these gifs to be saved as a single frame image.

After, the images have been converted into a JPEG image and saved, the images and folders containing them are then loaded into an array to quickly access the images as "train_files", and the folder names as the "train_targets". The targets are converted using "keras.utils.to_categorical" into binary digits. Due to the large number of targets in the dataset, the binary output is too long to display. Once the targets have been coded for the Keras models, the images need to be processed once more. Next, each image needs to be converted into a tensor. The original image shape was expanded into a tensor with a shape of (250, 250, 3). Next the images were converted into a 4D array before dividing each pixel by 255, this is done in order to convert the image pixels into a value between 0 and 1. This allows the model to read each image much faster than if the pixel values were left unaltered.

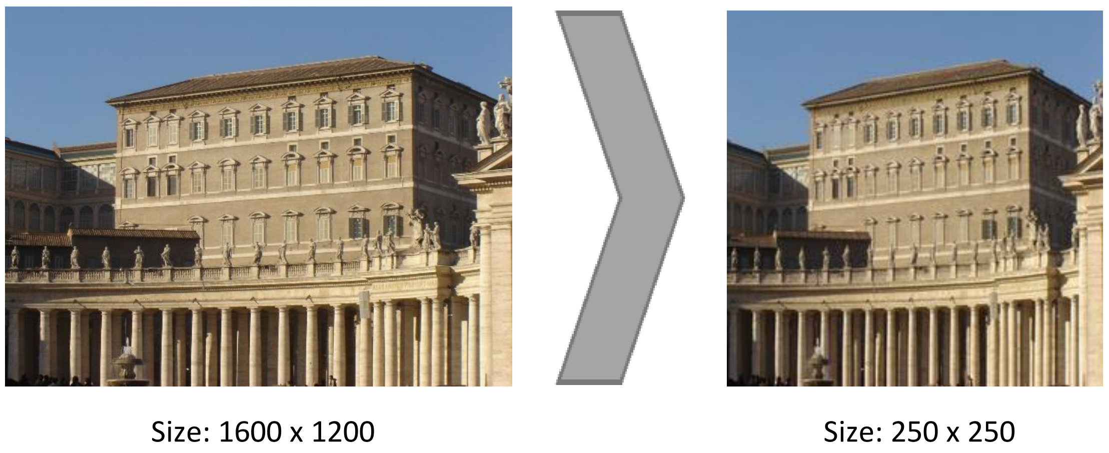


Figure 9: Resizing image

## Implementation

When building my CNN, I used the idea of residual mapping and parallel convolutional layers to help train the model with a great deal of accuracy but reduce the number of parameters used. I built the first model by creating a residual block function that implemented a basic convolution, batch normalization with a "Leaky ReLU" activation layer to normalize the data, an inception block function, and a final convolution that is only implemented when the stride changes from 1x1, altering the dimensions of the image. Next, I began to build my inception block function, which I decided to use a four-column block using RELU activations. The first convolution layer contains a 1x1 convolution layer, followed by a 3x3 layer, ending with a 1x1 layer. The 1x1 layers diminish the number of parameters. The second convolution layer is similar with a 1x1 layer followed by a 5x5, and a final 1x1 convolution. The third layer begins with a max-pooling layer to downsize the sample before being ran through a 1x1 convolutional layer. The fourth layer is a single 1x1 convolution. These four parallel convolutional layers are then concatenated together into a singular structure to make it more easily readable by the next layer in the CNN. After concatenation, the output is again normalized and sent through a final "Leaky ReLU" activation layer. This entire block can be seen in figure 10, the entire model is too large to display, but can be found here. I implemented a single convolutional layer prior to ten residual block operations with some normalization layers, and two dense layers with a dropout layer to reduce overfitting. I had set the number of filters for the residual blocks to 60, 120, 250, and 600, resulting in approximately 60 million parameters.

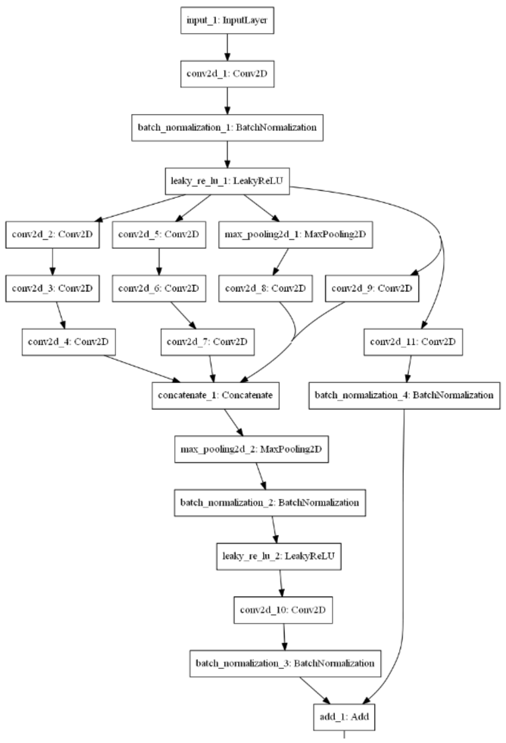


Figure 10: Single residual block plot of Res-Incep model

I attempted to train this model on my local machine, but the training was extremely cost effective and used all the memory. To counteract this issue, I decided to use an Amazon AWS instance. I chose to use the EC2 “p2.xlarge” instance which uses 1 GPU, 4 CPUs, and has 73 gigabytes of memory[5]. I encountered the same issue of running out of all the allocated memory space available. I tried to lower the batch size incrementally from 100 to 75, 50, 35, 25, 15, 10, 5, 2, and 1. With a batch size of 1, the training would take far too long for 5000 images and the accuracy would be highly ineffective. I then trained the model on a “p2.8xlarge” instance that utilizes 8 GPUs, 32 CPUs, and 584 gigabytes of memory[5]. I ran into the same problem of memory, so I had to lower the batch size to 8, and I decided to change the filters to 40, 60, 75, and 80. After changing the filters, the model had a total number of parameters around 2.2 million. I ran the training model on this instance and it took seven hours to train the model. The accuracy score was originally very low around 0.05% with the validation data. I increased the number of epochs from 10 to 20, finally finishing at 30 epochs that took ten hours to train. The new accuracy score resulted in a 5.89%. This does not appear to be a very accurate score, however if you take into account that many of the landmarks have very few images to train the model with, the likelihood of a high accuracy is improbable.

Next, I downloaded the ResNet-50 CNN model from Keras to use as my benchmark model. As stated in the benchmark section above, I downloaded the model using weights trained with the “ImageNet” dataset. To allow the model to be compatible to the new data I removed the top layers of the model and used a “Global Average Pooling” layer and a “Dense” layer with the appropriate number of outputs. The model was compiled using RMSprop as the opitimizer, 35 epochs, and a batch size of 55. The batch size for this model can be much greater than my original model because there is far less training necessary. It took approximately 30 minutes for the model to be trained, resulting in a validation testing accuracy of 4.02%.

After completion of training both CNN models I began running the model with the testing images to search for a landmark. I wanted to output the image id, and a landmark id if the probability score was greater than 4.5%, and time it took to run each image through the selected model. I elected to make the score low due to the low accuracy scores during training, and when I tested a number of images, the probability scores were fairly low for most images.

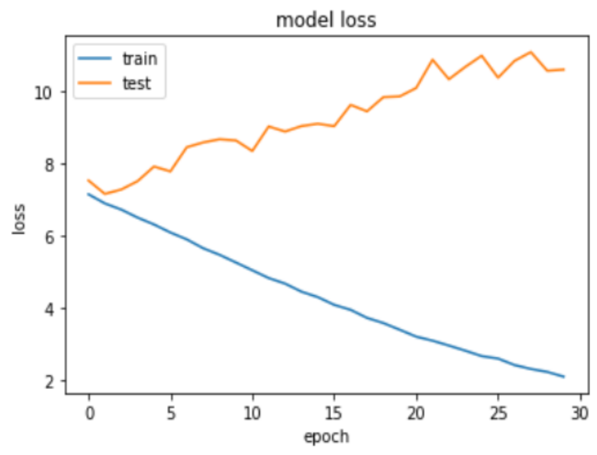
## Refinement

My initial model had a terrible accuracy score of around 0.009%, even for a classification problem this difficult that is bad. The first thing I changed about my model were some of the parameters, such as the optimizer, epochs, batch size, and number of filters in each layer of the network. I slightly increased batch size to the maximum that the memory could take, and changed the optimizer from RMSprop to SGD, increased the number of epochs slightly, and chose to decrease the number of filters in each inception layer to decrease the total parameters. After rerunning the model, the accuracy score increased slightly up to around 2.25%. I further decreased the number of filters in each inception layer, as to allow for the model to run more epochs in the same amount of time. I increased the epochs to 30, batch size maxed the memory load out at 6, and the resulting accuracy score was 5.89%.

The model looks to be overfitting the data, so in attempts to aid the model, I augmented the input images. The augmented images changed in size, rotation, scale, and flipping the image both vertically and horizontally. After running the model with the new augmented images added to the input data, the resulting accuracy score was lower than the previous model at 4.55%. Due to this decrease in accuracy, and notable increase in time spent training, I elected to continue with this project using the initial, unaugmented model. In an ideal world, I would increase the epochs drastically to allow for much more training, as well as play more with the hyperparameters of the model to maximize the accuracy. However, this model and EC2 instance is extremely costly in both time and money, and I cannot afford to run the model indefinitely.

## Model Evaluation

My final model is a ten-layer residual block as shown in figure 8, sandwiched between a convolutional layer and two dense layers with occasional dropout to prevent overfitting. As can be seen from figure 11 and 12, my model is still overfitting the data by quite a bit. Figure 9 displays the loss of the training and validation data; the stark separation shows that the model is overfitting the data. Figure 10 shows the accuracy of the training and validation data. The training data accuracy slowly arcs up in a rather smooth fashion, however the validation data is very shaky and only increases a minute amount.



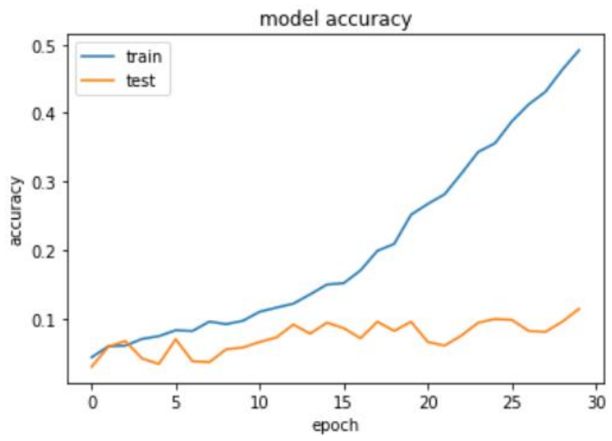


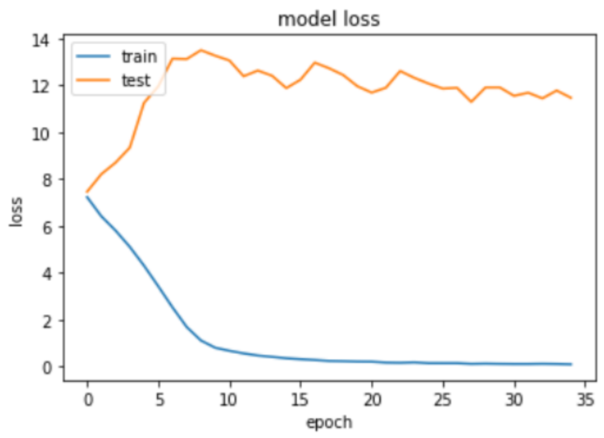
Figure 11 (left): Res-Incep model loss graph. Figure 12 (right): Res-Incep model accuracy graph.

A point of error that my model encountered was the large number of landmark categories, and the stark differences between the number of training images corresponding to each of the output categories. There is a large difference between the training, validation, and testing images in terms of the unique landmarks in each of them. I might be training the model for the training data accurately, but due to the differences in the validation data it is overfitting greatly. Regardless of the overfitting and low accuracy score, my score is not terrible in relation to the most accurate model submitted to Keras for the competition. I decided to continue forward with running my model on untrained images found in the "GLRC/TESTING" folder.

I sent a few testing images through my model, which resulted in two images outputting landmarks and the last not having a high enough probability score for a landmark identification. It took an average of two minutes to run each image through the model and produce an output of some kind. I checked other images in the training data that matched each outputted landmark to check if the landmark id was accurate or not. In both cases, the test images' perceived landmark was not accurate, however similarities can be found.

## Justification

I ran the same 10 testing images through both my model and the ResNet-50 model to compare the output of the two. When running the same ten images through the residual-inception model and ResNet-50 models, my Res-Incep model outputted a landmark for 60% of the images compared to the ResNet model that output none. The ResNet-50 model failed to reach a high enough probability score to output any landmarks with the ten images. Another area that my model surpassed the ResNet-50 model was in loss and accuracy scores, as can be seen in figures 13 and 14 below.



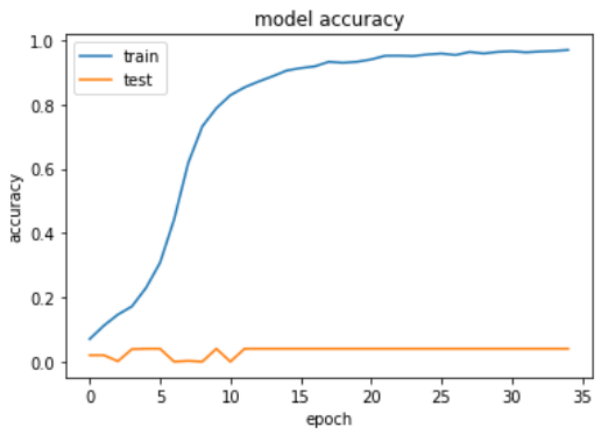


Figure 13 (left): ResNet-50 model loss graph. Figure 14 (right): ResNet-50 model accuracy graph.

As can be seen, the ResNet-50 model also overfits the data and has a much lower accuracy on the testing dataset. Finally, the average time for each image to run through my model took 33.14 seconds, while the average time for the ResNet-50 model took 24.37 seconds. This time difference might be important if I were to run the entire 5000 image testing image dataset through the models, however for just a handful of images this is of no great concern.

## Conclusion

### Visualization

Below are some of the many filters and patterns produced by my CNN model. The figure below contains a depiction of each filter in the first convolution layer and another convolution layer hidden in a residual learning block.

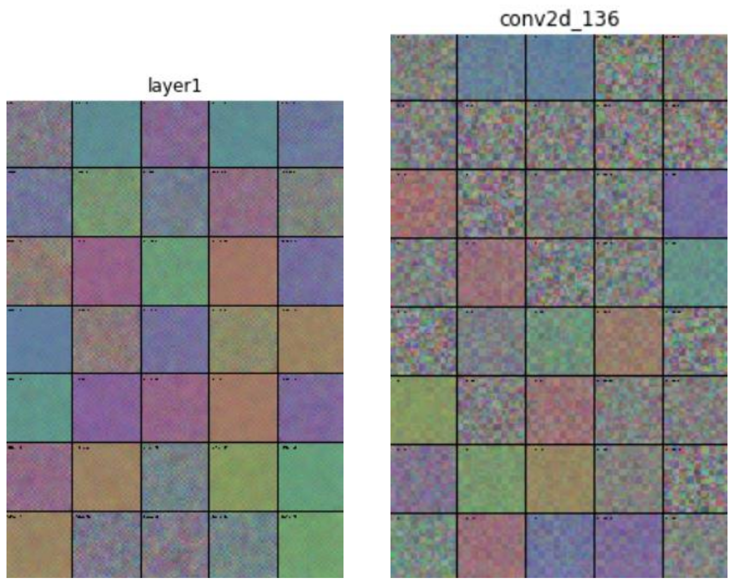
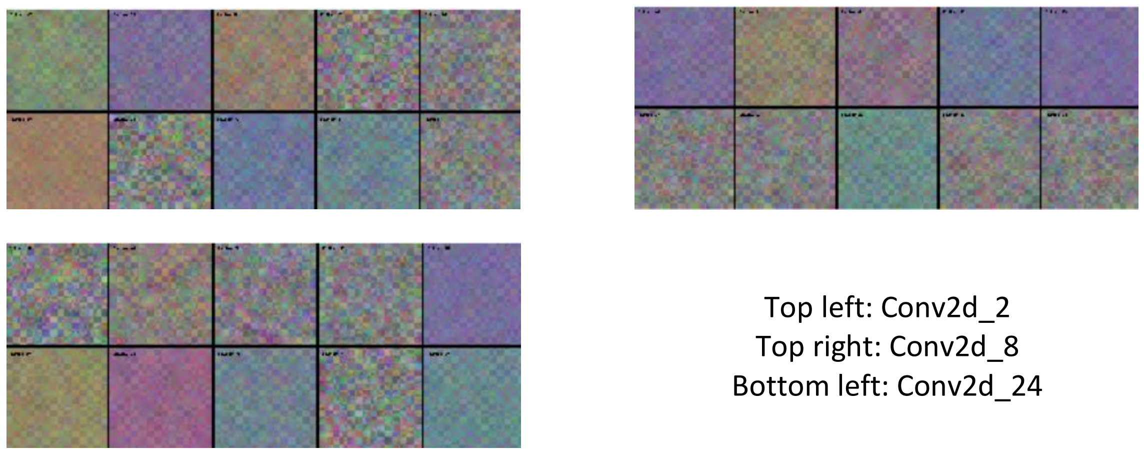


Figure 15: Conv-Layer1 with 35 filters and Conv-Laye136 with 40 filters.

Let’s take a closer look at the convolutional layers 2, 8, and 24 to see how the model changes what it is looking at and that patterns found in each.



Top left: Conv2d_2
Top right: Conv2d_8
Bottom left: Conv2d_24

Figure 16: Convolutional Layers and the first 10 filters in each layer

## Reflection

When I began this project, I didn’t expect many difficulties to arise, just a simple image classification model built using Keras. I knew that the amount of data available for this landmark classifier wouldn’t present any problems due to the large dataset size. However, due to the equally large number of landmarks with anywhere from 1-50,000 images corresponding to a particular landmark id, training an accurate model was going to be harder than I anticipated.

The biggest issues that I had continuous problems with was simply with the raw training data. Some of the URLs in the data had no pictures, some of the URL links lead to a gif rather than a normal image which would cause issues with viewing and saving the images. After loading the raw data appropriately, I had to preprocess the images and could begin building my CNN.

Building the model was the most fun and interesting part of this project, I began with building a very basic sequential CNN, training and checking accuracy. The accuracy score was unreadable, so I then implemented a residual block that allowed for residual learning of used weights. Again, the accuracy score was extremely low, but it did increase. Next, I implemented an inception block using four parallel convolution stacks. This model had a much higher accuracy, but still could be improved. I then spent a lot of time changing and tweaking hyperparameters to improve the model.

The model could be improved more than it was, but it would be costly in both time and money. Once the model was trained to an acceptable level, I imported the benchmark model and trained it to the dataset. After comparisons, my residual-inception model produced a better accuracy score. Although I would have wished that my CNN had a much higher accuracy score, the score obtained wasn't terrible given the variance in the data.

## Improvement

The model could be improved in several areas, removing all data with a low number of landmark ids, increasing the size of the dataset, using a larger EC2 instance, and train the model longer with increased epochs. A major issue affecting the models' accuracy was having landmarks with only one image to be trained for identification purposes. In the future, it might be smart to remove all the images that correspond to landmarks with a specific number of images or less (I.E. any landmarks with less than 5 images are deleted along with the corresponding images). This would make sure that the model would train data that has multiple images to analyze and recognize pixel patterns for each landmark, improving accuracy.

Another improvement could be increasing the dataset to be larger than the 5000 training images used. An issue with this would be a need for increased memory size and using a larger EC2 instance would help alleviate that issue. In addition to increased memory, it would be prudent to increase the number of epochs for the training of the model. This would train the model longer allowing for improved loss and increased accuracy in most cases.

A final improvement could be to implement a combination of CNN and recurrent neural networks (RNN). If the images were fed through the model in groupings of landmarks, using an RNN could allow the model to learn directly from the previous images in the landmark grouping. In combination with a traditional CNN, this could improve the recognition of the model in future attempts.

## Reference

1. “Google Landmark Recognition Challenge.” *Google Landmark Recognition Challenge | Kaggle*, Kaggle, www.kaggle.com/c/landmark-recognition-challenge/data.
2. He, Kaiming, et al. “Deep Residual Learning for Image Recognition.” *Microsoft*, ARXIV, 2015.
3. Szegedy, Christian et al. “Going Deeper with Convolutions.” ImageNet, ARXIV, 2017.
4. Ruder, Sebastian. “An Overview of Gradient Descent Optimization Algorithms.” *An Overview of Gradient Descent Optimization Algorithms*, Sebastian Ruder, 16 Apr. 2018, ruder.io/optimizing-gradient-descent/index.html.
5. “Amazon EC2 Instance Types.” *Amazon AWS*, Amazon, 2018, aws.amazon.com/ec2/instance-types/.