Capstone Project – Corvette Generation Recognition Project

Joel Haas

Machine Learning Engineer Nanodegree

October 2019

I. Definition

Project Overview

Object detection, classification and recognition enables companies like Apple to use face recognition to unlock a user's iPhone. The iPhone detects a face, classifies it as a human face, and then recognizes that the human face is the owner of the phone. Large areas of research include object detection, classification, and recognition. It's an exciting area of research that I would like to explore for my own professional growth.

This project is an object detection, classification and recognition project. Object detection is processing an image and detecting discreet objects within that image. Object classification is taking the objects detected and identifying what those objects are. And finally, object recognition is taking those classified objects and recognizing the differences between them. Object recognition is the most difficult of the three mentioned.

My project is an object detection, classification and recognition project. Given an image as input, I want to detect whether a car is in the image or not, classify the car as a corvette or not, and if it is a corvette, recognize the specific generation of corvette – C1 through C8. In this report, I mostly generalize these minute differences between classification, detection, and recognition and discuss the 'classification' of each image.

In my project proposal, I mention building three models: one for determining if there is a car in the image, one for determining if the car is a corvette, and one for determining what generation of corvette. After starting my project, I found that I could build a multi-classification model. I have never done this before and I thought it would be a good challenge and learning experience. So this is the path I chose.

In 2019, General Motors revealed the design of its eighth-generation corvette, the C8. Corvettes have long been an iconic American sports car. The first corvette debuted in 1953. The eighth-generation corvette is unique in the sense that it is the first corvette that is a mid-engine. Below are the years each generation were produced:[1]

[1] Chevrolet Corvette. (2019, September 5). Retrieved from https://en.wikipedia.org/wiki/Chevrolet_Corvette

• First generation, C1:	1953 – 1962
• Second generation, C2:	1963 – 1967
• Third generation, C3:	1968 – 1982
• Fourth generation, C4:	1984 – 1996
• Fifth generation, C5:	1997 – 2004
• Sixth generation, C6:	2005 – 2013
• Seventh generation, C7:	2014 – 2019
• Eighth generation, C8:	2020 –

In the picture below, from right to left are the C1 to C8 models.

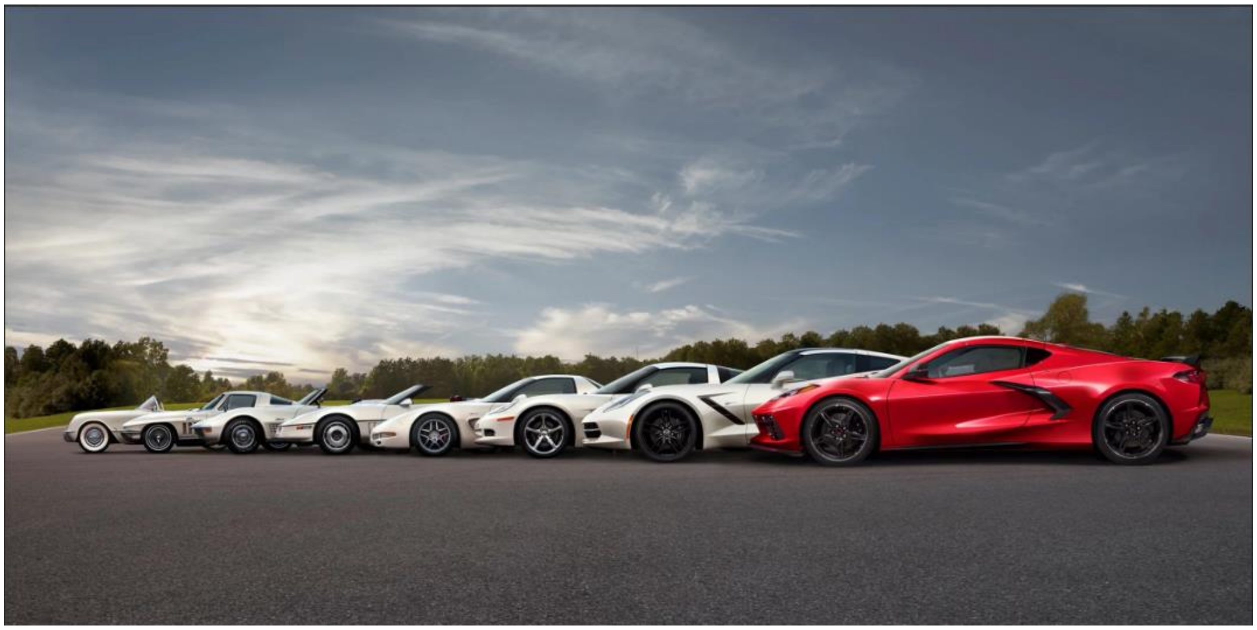

Fig 1: Corvette Generations, C1 – C8

Problem Statement

In this project, my goal is to detect if a car is in an image, classify the car as a corvette or not, and if it is a corvette, recognize what generation of corvette. I treated it as a multi-classification problem rather than an ensemble approach. I created a target value for each possible classification, described further in the following sections.

My workflow to solve this problem followed the steps depicted below. Note that the steps are not purely linear. Results from model evaluation led me to reformat how my data was organized for classification (back to step 2) and re-architecting the model (back to step 4).

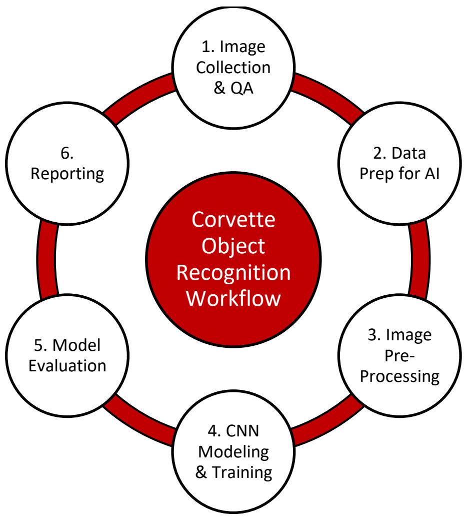

Fig 2: Project Workflow

I first collected and QA'd the data. I labeled the data. I pre-processed the images. I experimented with different ways to predict the classification of the images. I built a convolutional neural network (CNN) architecture. I trained and tested the model and measured its performance. And I leveraged transfer learning to see if I could improve performance.

The outcome of the project is a prediction of an image's classification. There are ten (10) possible classifications for each image:

1. not car, not corvette
2. car, not corvette
3. car, corvette, c1
4. car, corvette, c2
5. car, corvette, c3
6. car, corvette, c4
7. car, corvette, c5
8. car, corvette, c6
9. car, corvette, c7
10. car, corvette, c8

Only one target classification is selected for each image. This contrasts how I initially tackled this project for which I explain more in the data exploration section.

Metrics

As previously mentioned, only one target classification is predicted for each image analyzed. This prediction is compared with the true label. Each of the accurate predictions are summed and then divided by the total number of predictions. This gives an accuracy score for the model. Models are compared against each other base on each model's respective accuracy score against the hold-out test data set.

$$Model\ Accuracy = \frac{Correct\ Predictions}{Total\ Predictions} * 100$$

II. Analysis

Data Exploration

Image Collection & Quality Assurance

I needed to collect the data for this project. There is a Stanford dataset of cars that I downloaded that has images of over 8,000 car models.[2] I also needed to collect images of each generation of corvette and images with no cars. To collect these datasets, I wrote a python script to scrape the web using google_images_dowload[3] (see jupyter notebook titled "MLEND_Capstone_Image-Scraper_HaasJoel"). I scraped roughly 1,000 images for each generation of corvette and over 2,700 images without a car.

What I quickly discovered was that not all the images returned by Google aligned with my search terms. For example, if I scraped Google for 1953 corvettes (C1), I would get a few images of another generation corvette, such as a C3, in my search results. I would also get pictures of corvette toys and corvette paintings. I place each generation corvette into its own folder (e.g. 'corvette_C1') and manually reviewed each image in each folder.

In my quality assurance (QA) review, I ensured each of the 8 corvette generation folders only contained images of that respective generation. I deleted pictures of corvette toys and corvette paintings. I deleted pictures that had more than 1 generation corvette. I deleted pictures that had several cars. I deleted images where I could not tell what generation corvette it was. I deleted pictures that only showed the engine or interior of the corvette. I deleted most duplicates. And I deleted some of the pictures that did not give enough view of the corvette.

On the other hand, I kept pictures that had more than 1 car when the corvette was the prominent vehicle in the image. I kept images with backgrounds. I kept images with text. And I kept images showing all

[2] Stanford Cars Dataset. (2019, September 17). Retrieved from https://www.kaggle.com/jessicali9530/stanford-cars-dataset
[3] google_images_download 2.3.0. (2019, September 16). Retrieved from https://pypi.org/project/google_images_download/2.3.0/#examples

angles of a corvette's exterior (e.g. rear, front, side, etc.). Here are some examples of the images I analyzed in this project:

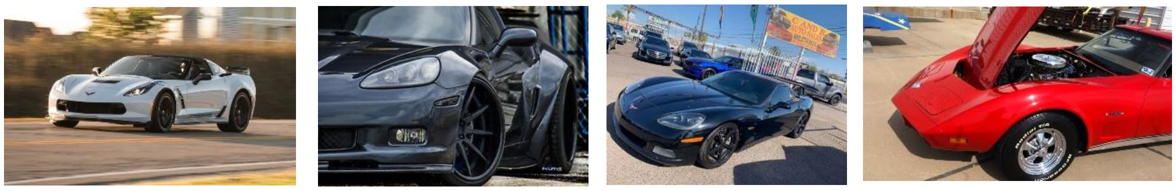

Fig 3: Examples of Images used for Project

Data Labeling

Now that I had collected the images I needed for the project, I had to label them. Each image had to be classified with the correct label. Otherwise, there would be no way to train the model and calculate model accuracy at scale.

Because I had organized the images into folders by class (recall the 10 classifications above), I was able to quickly rename all the images into a consistent format.

I used the following file naming convention: I organized all the images into their respective folder by class, e.g. 1st generation corvettes went into a 'corvette_C1' folder, pictures of beaches and mountains went into a 'no-cars' folder, etc. I highlighted all the images in one of the folders and renamed the first image. For the first generation C1 folder, I renamed the images as follows: 'corvette_1_1_1_ '.

My PC then renamed all the highlighted images in the C1 folder and iterated a number at the end of the name as follows:

- corvette_1_1_1 (1)
- corvette_1_1_1 (2)
- corvette_1_1_1 (3)
- …
- corvette_1_1_1 (n)

I used the same methodology with the images in the other folders but changed the naming convention for each category as depicted below. Note that the 'x' represents the number my PC automatically assigns to each image once hit enter to rename all the images in the folder.

Classification	**Naming Convention**
1. not car, not corvette	'notCars_0_0_0_ (x)'
2. car, not corvette	'cars_1_0_0_ (x)'
3. car, corvette, c1	'corvette_1_1_1_ (x)'
4. car, corvette, c2	'corvette_1_1_2_ (x)'
5. car, corvette, c3	'corvette_1_1_3_ (x)'
6. car, corvette, c4	'corvette_1_1_4_ (x)'
7. car, corvette, c5	'corvette_1_1_5_ (x)'

8. car, corvette, c6 ‘corvette_1_1_6_ (x)’
9. car, corvette, c7 ‘corvette_1_1_7_ (x)’
10. car, corvette, c8 ‘corvette_1_1_8_ (x)’

As I read in each image, I split the filename by the underscore and then analyzed the returned list and the value at each index to determine what the label should be. For example, splitting ‘corvette_1_1_8_ (x)’ returns [‘corvette’, ‘1’, ‘1’, ‘8’, ‘(x)’] and I can evaluate this list by looking at the string at index 0 and index 3 to determine this is a C8.

From this parsing, I generated an CSV with the filename as the ‘id’, and the additional columns: ‘car’, ‘corvette’, ‘c1’, ‘c2’, c3’, ‘c4’, ‘c5’, ‘c6’, ‘c7’, ‘c8’, ‘classification’. A C8 corvette would have a 1 in the ‘car’, ‘corvette’, and ‘c8’ columns, along with the classification value of [‘car’, ‘corvette’, ‘c8’]. As it turned out, I only needed the ‘id’ column and the ‘classification’ columns for my training and testing. See ‘labels’ folder for the labeled datasets and see the jupyter notebook “MLEND_Capstone_Create-Labeled-Data_HaasJoel” for the code I wrote to perform the labeling.

Exploratory Visualization

The total number of images collected for each categorization is depicted below.

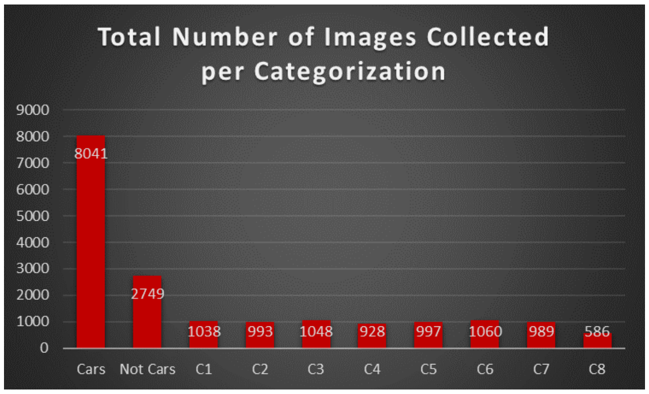

Fig 4: Total Number of Images Collected per Categorization

For training and testing, I did the following. I removed 50 of each classification and placed in a hold-out test set. This resulted in a test set of 500 images (50 for each of the 10 classifications). For the remaining images, I used 80% as my training set and 20% of the remaining as my validation set.

Dataset	Total	Training	Validation	Testing
Cars	8041	6393	1598	50
Not Cars	2749	2159	540	50
C1	1038	790	198	50
C2	993	754	189	50
C3	1048	798	200	50
C4	928	702	176	50
C5	997	758	189	50
C6	1060	808	202	50
C7	989	751	188	50
C8	586	429	107	50

Table 1: Number of Images used for Training, Validation, and Testing by Classification

Notice above the fewer number of C8 images. Because the C8 model was just recently revealed by GM in the last few months, there were not many unique pictures available. In order increase the number of C8's to analyze, I decided to keep several of the duplicate pictures, as well as several of the camouflaged C8 before it was officially revealed to the public.

I trained and built the model using the training and validation sets. I then fed the hold-out test dataset into this resulting model to calculate the model's accuracy.

Algorithms and Techniques

I built the models using a Convolutional Neural Network (CNN) (see jupyter notebook "MLEND_Capstone_Main_AWS_HaasJoel"). I built and tested 4 models: a CNN architecture I built from scratch and 3 models where I leveraged Transfer Learning. The two pre-trained architectures I used were VGG16 and ResNet50.[4] For the first iteration, I kept the pre-trained weights in the models and only trained the layers I added for new weights. In a final iteration, I re-trained the entire ResNet50 architecture along with the layers I added. The summary of my approach is captured below:

Model	Weights	Training	Validation	Epochs
Build from Scratch	Trained Entire Model	80%	20%	5
VGG16 +	Kept Pre-Trained Weights	80%	20%	5
ResNet50 +	Kept Pre-Trained Weights	80%	20%	5
ResNet50 +	Re-trained Entire Model	90%	10%	2

Table 2: Model Training Approach

Benchmark

I used two benchmarks for the performance of my model. The first benchmark is random guessing. With 10 different classifications and the same number of images for each classification (50) in my test set, random guessing would result in a 1 in 10 (10%) probability of guessing the correct category. Any model I built should be better than random guessing!

The next benchmark I used was the model I built from scratch. This model resulted in 20-25% accuracy (1 in 4 or 5). I would expect that transfer learning would result in improved performance. So, each attempt to leverage a pre-trained CNN architecture was compared against the more simple model I built from scratch.

To summarize, any CNN model I built should be better than random guessing (10%). And then any model leveraging Transfer Learning was expected to improve the model's performance above and beyond 25% accuracy.

[4] Transfer Learning using Keras and VGG. (2019, October 5). Retrieved from https://riptutorial.com/keras/example/32608/transfer-learning-using-keras-and-vgg

III. Methodology

Data Preprocessing

A CNN requires pre-processing of the images before the algorithm can be used for classification. These pre-processing steps are the following:

1) resize the image – I resized each image into (400,400,3)
2) convert the image into an array – the algorithm needs to evaluate numbers
3) divide each value in the array by 255 – pixel colors can have values between 0 and 255, so by dividing by 255, each pixel value is then between 0 and 1
4) all pre-processed image arrays were added to a list
5) the list of image arrays was converted from a list to an array

Implementation

The returned array (containing each pre-processed image array), which I stored as 'X', are the features the model trains on. And for the corresponding target variables, I tried two different methodologies.

This is a multi-label classification problem. Each image will have multiple labels, 'car' and 'corvette' and 'c8', or 'car' and 'not corvette'. The first methodology I used was to calculate the probability of each possible classification, and then return the top 3 highest probability classifications.[5] The results were confusing. For example, I may get a 15% that an image is a ['car', 'not corvette'], 13% probability that an image is a ['car', 'corvette', 'c4'], and an 11% probability that an image is a ['not car', 'not corvette'].

This approach works well if the model's goal is to predict different objects in an image, such as the probability that there is a dog, a cat, and a human in a picture. In such cases, you can establish a threshold value of say 50%, and only return the classification labels where the probability is greater than 50%. So if the probability of a dog in the picture is calculated as 83%, a human as 63%, and a cat as 41%, then you would return only the dog and human (based on your set threshold value) as labels for that image.

However, this approach does not work well when the labels returned contradict each other, such as the highest probability for objects in the picture to be ['not car'] and ['car', 'corvette', 'c6']. It also doesn't work well when the probabilities are really low where the threshold probability is not clear and has to be around 13%.

Since that methodology broke down when I implemented it, I went a different direction. Instead, I took each of the 10 classifications and one-hot encoded them. This then became my target value. After generating a prediction, I simply took the highest probability classification.

[5] Build your First Multi-Label Image Classification Model in Python. (2019, 22 October). Retrieved from https://www.analyticsvidhya.com/blog/2019/04/build-first-multi-label-image-classification-model-python/

With this new approach to my target value, I performed predictions using the 4 models described above: build from scratch, VGG16, ResNet50, and ResNet50 where I re-trained all the weights.

Refinement

In the model that I built from scratch, I tried several different architectures. The first model resulted in a performance of 9%, which is comparable to guessing. I changed the parameter values for strides; I added dropout layers; I modified the number of convolution layers and pooling. Iterating on the model architecture resulted in an increased performance that ranged from 20-20%.

Leveraging VGG16 and its pre-trained weights resulted in a performance of 47.9%. This is a dramatic improvement and not bad considering there are 10 classifications. Nevertheless, I wanted to improve the model further. By leveraging ResNet50 and its pre-trained weights, the model performance increased to 72.6%. I thought this was exceptional! But I did want to test one more hypothesis.

The ResNet50 architecture is more complicated than VGG16, so I'm not surprised it performs better. I do wonder if I could improve the model's performance further by making some changes.

The final hypothesis I tested is as follows (see notebook "MLEND_Capstone_Main_Mod1_HaasJoel" for code:

- Leverage the ResNet50 architecture, but this time, what if I retrained the entire neural network to generate new weights?
- Recall that in the current training data set, there are over 8,000 images of various car models and over 2,700 images without a car. What if my training dataset included roughly 900 images of each classification? Could the additional images of certain classifications be leading to an unwanted bias in the weights?
- What if my validation set was decreased from 0.20 to 0.10?
- What if I decreased the number of epochs from 5 to 2? (cost and time to process have become critical constraints, unfortunately)

The result of this test was a model performance of 30.2%. I wasn't expecting that low! The model took a long time to process, so if I had more time and money, I would have increased the number of epochs. The validation accuracy was 90%, so the model was overfitting on the training and validation data. I would also need to add more dropout and pooling layers.

These changes were also needed for VGG16 and ResNet50 where I kept the pre-trained weights. The validation accuracy for both were 95% and 96%, respectfully. Both models were overfitting and needed additional dropout and pooling layers. Below is a summary of each model's performance.

Model	Weights	Validation Accruacy	Prediction Accuracy
Build from Scratch	Trained Entire Model	92.4%	20.2%
VGG16 +	Kept Pre-Trained Weights	95.2%	47.9%
ResNet50 +	Kept Pre-Trained Weights	96.5%	72.6%
ResNet50 +	Re-trained Entire Model	90.5%	30.2%

Table 3: Model Performance

IV. Results

Model Evaluation and Validation

The final model chosen was the ResNet50 model with pre-trained weights. The only additional layers I added to this model was a GlobalAveragePooling2D() layer and a Dense(10, activation='sigmoid') layer to align with the number of classifications.

I compiled the model using an 'adam' optimizer and 'binary_crossentropy' for loss. And the metric I used was accuracy. I performed 5 epochs using a training set and a validation set. The training set was 80% and the validation set was 20%, with a batch size of 20.

Over the course of the 5 epochs, the validation loss decreased from 0.2111 to 0.1029 and validation accuracy increased from 0.9357 to 0.9646, as shown below.

```
Train on 14140 samples, validate on 3536 samples
Epoch 1/5
14140/14140 [==============================] - 4374s - loss: 0.1668 - acc: 0.9435 - val_loss: 0.2
111 - val_acc: 0.9357
Epoch 2/5
14140/14140 [==============================] - 4391s - loss: 0.1207 - acc: 0.9561 - val_loss: 0.1
143 - val_acc: 0.9585
Epoch 3/5
14140/14140 [==============================] - 4391s - loss: 0.1057 - acc: 0.9612 - val_loss: 0.1
099 - val_acc: 0.9611
Epoch 4/5
14140/14140 [==============================] - 4391s - loss: 0.0989 - acc: 0.9634 - val_loss: 0.1
066 - val_acc: 0.9634
Epoch 5/5
14140/14140 [==============================] - 4398s - loss: 0.0914 - acc: 0.9666 - val_loss: 0.1
029 - val_acc: 0.9646
```

Fig 4: Model Performance per Epoch

As mentioned above, if I had more compute time and resources, I would have experimented with adding some additional layers to the model to try and get the prediction accuracy of 72.6% closer to the validation accuracy of 96.5%.

Justification and Free-Form Visualization

The final model's performance is significantly better than guessing (10% accuracy) and the model that I built from scratch (20% accuracy).

To further measure the final model's performance, I generated a confusion matrix compared the predicted labels versus the true labels across the 10 classifications.[6] I wanted to see if the model was biased towards certain classifications or if the model was consistent for each classification.

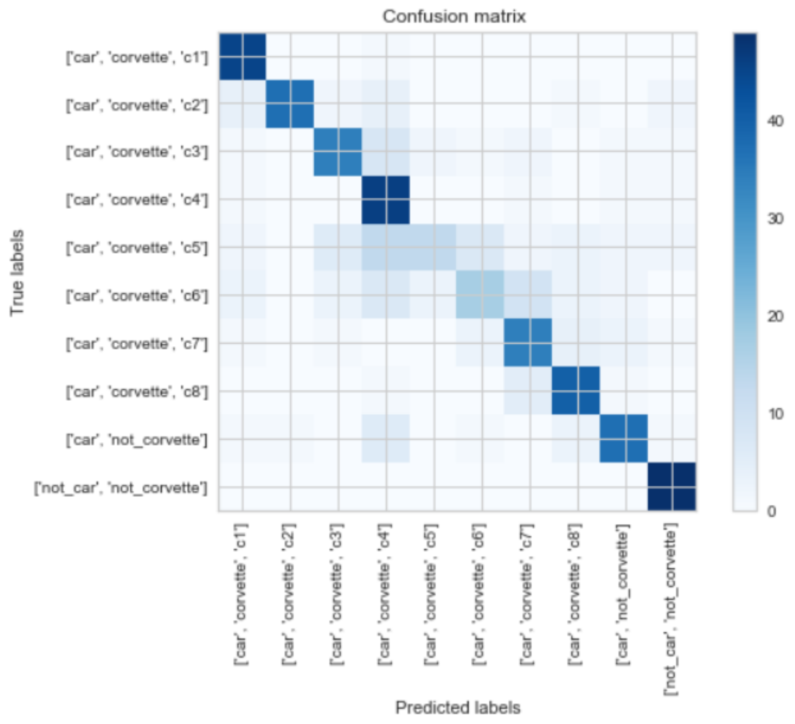

Fig 5: Confusion Matrix

You can in the confusion matrix that three classifications stand out as being predicted most accurately by the model – the C1, C4, and images without a vehicle. Five classifications were predicted fairly well and those are the C2, C3, C7, C8, and images with cars that are not corvettes. And finally, there are two classifications that were not predicted well by the model – the C5 and C6.

For the C5, the corvette generations predicted incorrectly include the C3, C4 and C6 mostly. And for the C6, the corvette generations predicted incorrectly include the C4 and C7.

In general, the model performs pretty well across each of classification. There is definitely some improvement that needs to be made hen predicting the C5 and C6 generation corvettes.

I am happy with the model's performance of 72.6% given this is a multi-classification problem with 10 unique classifications. I also think this is a good result because of the images I used, and that they were not all clean pictures. Instead of only using images of the side of a car with no background, I used

[6] plotting confusion matrix for an image classification model. (2019, 22 October). Retrieved from https://stackoverflow.com/questions/51282098/plotting-confusion-matrix-for-an-image-classification-model

pictures with other cars, images with backgrounds, with partial views of the corvettes, views of the rear of the corvettes only, etc. This made the learning more challenging, but it turned out pretty good!

V. Conclusion

Reflection

This was a great project. It allowed me to go through the entire data science pipeline of building a machine learning model (except for operational deployment). I collected images and performed QA on them to ensure they were good images and organized correctly. I generated labels for the images. I pre-processed the images. I prepared the dataset for the model. I built a CNN from scratch and leveraged transfer learning. I tuned the models and parameters. I used pre-trained weights as well as re-trained the entire model. I performed multiple methods of prediction. And I generated a confusion matrix to measure performance by classification.

I enjoyed the challenge of this project. It allowed me to do several things I have never done before. For example, I have never scraped for web for images. I have never labeled my own dataset. I have never done a multi-classification project. I have never performed transfer learning in the method that I chose. I have never had to massage the inputs to a confusion matrix for it to work.

The parts I found most interesting in this project were creating and labeling my own supervised learning dataset, tackling a multi-classification project, figuring out how to build a confusion matrix from scratch (not leveraging scikit-learn), and having to use an AWS EC2 GPU instance in order to process all the data.

There is always a lot more to learn, but I learned a lot in this project. Thank you for the opportunity and the challenge!

The final model fits my expectations. It is not ready for production; it requires more tuning before that makes sense. But I am happy with the final results.

Improvement

As mentioned above, the model requires tuning in order to lessen the gap between the validation accuracy and the prediction accuracy. The model also needs to improve its accuracy when trying to predict C5 and C6 generation corvettes. Using more images for training, more epochs, and generalizing the model to address overfitting would mitigate these issues.