Smith
SCHOOL OF BUSINESS
Queen's University

Master of Management
in Artificial Intelligence

MMAI 894
Deep Learning
Final Project

Table of Contents

Abstract

The aim of this project was to explore logo recognition through the use of deep learning techniques. More specifically, we aimed to not only accurately classify hundreds of logos but determine how similar an external logo was to one within our dataset. For the primary goal, we created a CNN-based pipeline to drive the multi-class classification – for the latter, we built a proof-of-concept through leveraging comparative metrics such as Mean Squared Error (MSE), Structural Similarity Index (SSIM), and Euclidean distance between pixel values.

The results of this paper will demonstrate that our novel architectural approach performed quite well in accurately classifying logo classes and sets the foundation for further investigation into image similarity. Throughout this paper, we will alternately address the methodology behind both the logo recognition and logo similarity portions of the project.

Business Context

One of the leading challenges new, and established, companies face when generating ideas for branding, and specifically in relation to their logo designs, is originality. Being unique can establish an organization as the leading disruptor or creator of a market. Consumers react positively to products, and even the brand, on how original they are overall due to the notion of exclusivity. To stay original, patents, trademarks and copyrights play a large role.

Logos are an example of intellectual property that is a company's most valued expression of their originality. Companies are notorious for protecting their logos under both copyright and trademark. Logos are ultimately how customers identify their company's brand and image in the marketplace and they spend millions of dollars protecting to protect it. To avoid copyright infringement, time and material needs to be spent to ensure to lawsuits cannot arise as the brand gains popularity.

Companies also spend thousands to hundreds of thousands of marketing dollars in creating logos for their company or even a specific product. This is to engrain their brands into consumers' mindset and, in the process, an extensive amount of research and development is completed in-house or by a marketing agency for multiple reasons. One of those reasons is to ensure that the logo will be unique, so that it is fully distinguishable by the customer and stand out amongst competitors.

With more and more companies opening their doors every day and new brands emerging into the market, there are, and will continue to be, more logos flooding the industry. There is not currently a centralized tool for all companies to cross-reference the uniqueness of their brand compared to others across the world – having such a resource would greatly reduce the need for all the time spent conducting this work manually and also having an international scope in a more global, connected landscape increases the breadth of coverage to determine uniqueness of a logo worldwide.

This tool would be leveraged by every marketing agency as a "Turnitin" for logos. It would help in the ideation process, as there is a lot wasted effort that goes behind determining a logo. Some of this effort includes an agile process of surveying logos and understanding why people like which logo and why. This process is typically driven by intuition, whereas the tool can help dig deeper and validate on why individuals liked certain logos by seeing which logos in the market it is similar to. Also, when marketing agencies create a set of draft logos, there is also a lot of time taken to refine them. This tool can help designers in the process by being the data-driven approach on which logos to pursue further. This will decrease the time required for design iteration cycles and enhance timelines to launch new logos to the marketplace. Ultimately, providing a lower cost for a higher quality logo which is backed by artificial intelligence (AI).

Whether it is copyright infringement avoidance or improving the process on creating an effective logo, cost is a key factor. To reduce these costs, automation using AI is the key driver to ensure processes are put in place. This will also allow companies to scale as less time is wasted on this effort, so more time can be spent on creative requirements. The logo uniqueness tool will provide exactly that and can become the standard on how companies evaluate logos to ensure their originality is preserved.

Pre-Processing Methodology

Logo Recognition Pre-Processing

The original Logo-2k+ dataset included 167,140 logos broken down into ten categories. The intent of the dataset is to serve as an avenue to create a scalable logo classification model across brands from different verticals. Within the various categories, there were 2,341 unique brands.

In more granularity, this was the original breakdown of the image dataset:

Category	Unique Logo	Images
Food	769	54,507
Clothes	286	20,413
Institution	238	17,103
Accessories	210	14,569
Transportation	203	14,719
Electronic	191	13,972
Necessities	182	13,205
Cosmetic	115	7,929
Leisure	99	7,338
Medical	48	3,385

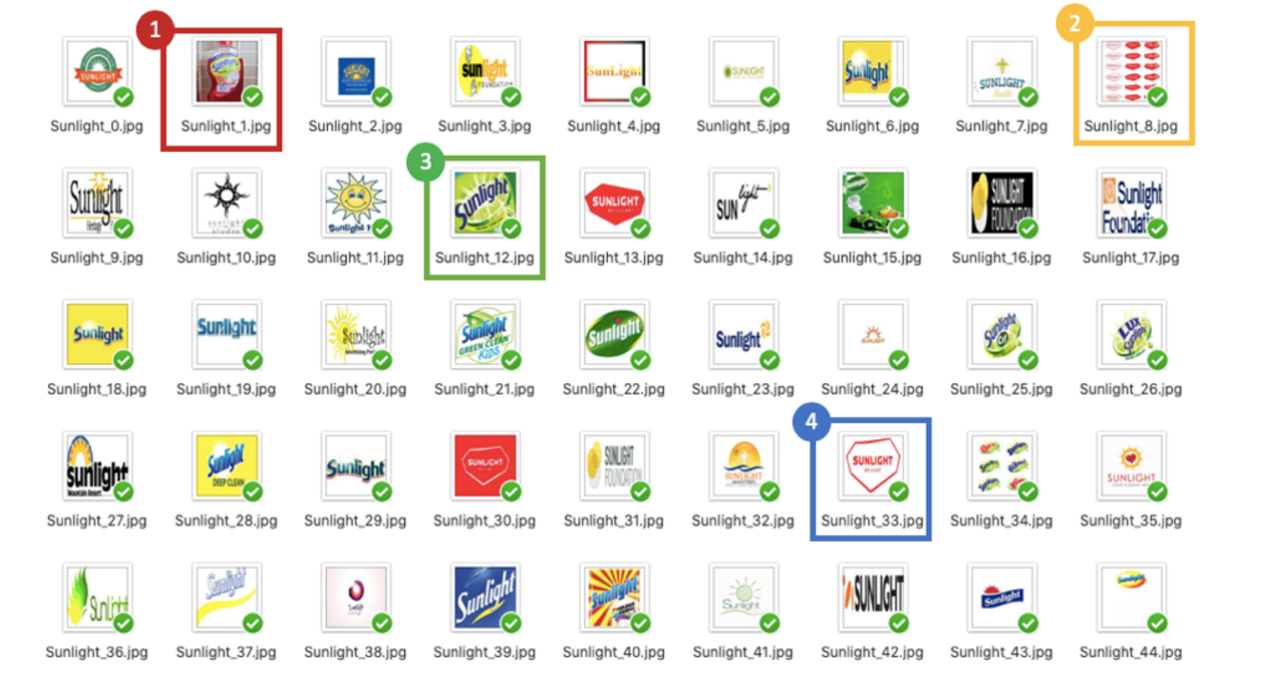
From looking at the image distribution, it was apparent that something would have to be done about the food category in particular, as it had almost three times more images than the second largest category (*Clothes*). This would surely bias our model to learning more food brands than all other categories combined.

Given the fact that the authors of this dataset collected these images through automated Google queries typing the names of the brands with the word “logo” at the end, we expected that exploratory analysis would require a lot of upfront time. Additionally, we anticipated that this dataset would require an extensive amount of computational power to process and train.

Initial Pre-Processing Findings

Upon initial inspection, our initial instincts were correct - the data was even messier than we had originally anticipated. First, the distribution of the images was not *just* imbalanced between categories, but between brands *within* those categories as well. For example, within the food category, one brand could have 85 images while another may have 40. Secondly, another major issue was that for brands where the name of the company appeared in multiple countries and industries, there were entirely different logos of identically named companies. Similarly, any brands that happened to have a name similar to a noun ended up containing images of that noun in addition to the brand.

To better illustrate these discoveries, if we take the washer brand **Sunlight**, we noticed the following four scenarios in that folder alone:



Scenario 1: The image contains the logo, but contains it on a real-life product. The type of CNN we'd be creating for this specific image would differ from the one that we're intending to base off standard logo images.

After further investigation into the source of the dataset, it was discovered that it was an amalgamated collection of images from the authors themselves combined with the FlickrLogos-32 'Logos in the Wild' dataset.

Scenario 2: The image is completely unrelated to the brand and would be a pure piece of noise.
Scenario 3: The image is actually of the brand that was intended.
Scenario 4: The image is of a different brand with the same name.

Of these different scenarios, we noticed that Scenario 4 was the most common, especially in cases where the brand name was not necessarily novel. We inspected a sample of 20 brands from the 10 categories and concluded that this problem was apparent in the entire dataset. We knew that this would make the process of building a convolutional neural network (CNN) from scratch significantly more difficult, as the network might only see the intended brand image a few times, and it would quickly get drowned out by the other noise in the folder.

Because we did not know whether we would leverage a typical function to load the images in, we wanted to keep our options open in terms of how we would import the labels of these logos. To address this, we wrote a program that went into each of the category folders and named all of the subfolder images with the name of that subfolder and an index as a suffix. Referring back above, whereas the image in each folder was originally named from one (1) to the i^{th} image, each image was now labeled in a distinguishable manner (e.g. Sunlight_11).

Through completing that process, we were able to inspect all the subfolders to gauge the imbalances in the dataset. To our surprise, this added form of exploration allowed us to discover three more issues with the categories in the data – some of the same brands appeared across different categories. For example, we noticed that a few brands appeared in multiple folders, and that one entire category did not have brands that really corresponded to its intended 'industry'. This category in question was **'Necessities'**, which included a medley of brands from all industries – some of the more notable ones were Montblanc (*a clothing brand*), Citizen (*a watch brand*), and Gorilla, a folder that comprised of 80 images of the actual animal. Given the complete mismatch of this folder's contents, we decided to drop the category altogether because we had sufficient coverage in all of the other ones.

To make the cleaning more efficient, we focused on taking one of the more reasonable-sized categories, **'Accessories'** (*with 210 images*), and scrub that folder. Some of the brands were hard to 'audit' because they included some that we were not familiar with; however, with that said, we had to research some of these companies to differentiate between the intended brand and the noise we alluded to earlier. As this began to consume more time, we decided to drop any brands that were not easily auditable, and continue the scrubbing from there. After going through the remaining brands, we tried to observe the average number of relevant images per folder – this was around 20. With that in mind, we went through each folder, and removed all the noisy irrelevant images. After completing this process, we decided to try and make the remaining folders as balanced as possible, so we decided on keeping 20 images per brand. If there were less than 20 images, we searched new images ourselves to bring them up to that number. Within the **'Accessories'** category, we ended this process with 30 brands with 20 images each (*600 images for each brand*); consequently, we decided to take this uniform approach across the other categories for consistency.

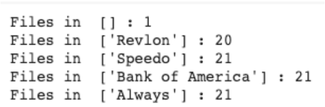
The resulting clean dataset that we used includes 270 unique brands across 9 folders:

Category	Unique Logo	Images
Food	30	600
Clothes	30	600
Institution	30	600
Accessories	30	600
Transportation	30	600

Electronic	30	600
Cosmetic	30	600
Leisure	30	600
Medical	30	600
Total	**270**	**5,400**

One last inconsistency we noted was that there were a combination of PNG and JPEG files in this iterated dataset – given that PNG and JPEG are sometimes interpreted differently in Python, we used our renaming program implemented earlier to blanket change the file format of the files to JPEG.

After this process, we decided to experiment with one of the finalized folders and loaded it into Python. We did some additional inspection by double-checking that we had a consistent number of folders per category and images per folder. Upon further examination, we noted that certain folders had <u>more</u> than 20 images showing up in the code:



```
Files in  [] : 1
Files in  ['Revlon'] : 20
Files in  ['Speedo'] : 21
Files in  ['Bank of America'] : 21
Files in  ['Always'] : 21
```

When we looked at the folders to verify this, we just saw 20 images. We tried to test running the data through a vanilla CNN framework, and kept receiving errors denoting images that did not even exist. For example, you would have the folder Adidas, with images labelled Adidas_01 – Adidas_20. The error would refer to Adidas_0, which we could not visibly see. Through investigation, the reason was that the program we wrote to label the photos created a few corrupt versions of the images in each folder. Additionally, the 'extra' images counted included a '._DS_store' file that was generated when unzipping the Logo Dataset folder which had to do with how Macintosh computers unzip files.

We deleted all corrupt images and the vanilla code was able to run, allowing us to move on.

Image Similarity Pre-Processing

For this portion of the project, much of the work done for the logo recognition pre-processing was repurposed. For our first level of experimentation using Mean Squared Error and the Structural Similarity Index, we were able to re-use all the images as-is. However, to test out the Euclidian distance using the cv2 and PIL libraries, we had to convert all the images to PNG format. The key difficulty with this related to a few obstacles encountered with the original dataset – due to the vast number of remaining images (5,400), we had to find a way to convert all the images in bulk. We attempted to use online conversion tools but, in the end, we wrote a program ourselves that converted the entire dataset into PNG, and then made sure to delete any files that became corrupted as a result of the process.

Upon trying to loop through the images with our image similarity experimentations, the session would continue to crash beyond 500 images of the 5,400. To remedy this, we decided on just taking one image per brand, and subjectively loading the most general version of the logo – given that this was well below 500 (*270 images*), the program was able to run as expected. The justification of this approach is that there is typically only one brand logo in the market at any given time and, as a result, we believe that only having one to compare against per brand would not sacrifice the ability to determine uniqueness than with having the full number of images in our dataset. Many images would simply have been duplicated or edge-case modifications of the original logos themselves and we didn't view it as necessary to have all images for this exercise.

Experimentation

Logo Recognition Experimentation

Since this data involved image recognition and object classification, the natural choice of general architecture is a CNN.

We initially started running our experiments on our local computers leveraging PyCharm – one thing we quickly noted was how computationally expensive this dataset was. We did not want to try loading the entire dataset all at once so we loaded the **'Accessories'** dataset and ran the vanilla CNN code. We noted that this took many hours just to run through once, so we anticipated that this would cause problems later on when we loaded the entire dataset (*containing 8 times more images*). In this specific project, our largest concern was our collective computers' inability to allow us to constantly experiment, so we made the executive decision to transition to Google Colab, which is able to run models in the cloud using a TPU engine and so we resorted to building and running our code there.

This ended up being a great decision as we were able to run 50 epochs on the **'Accessories'** dataset in half an hour compared to four hours on our local computers. The only caveat with making this decision was that we had to learn how to use certain Colab syntax to import the images.

Overall, when making the decision whether to pursue Colab or running PyCharm on our local computers, this was our thought-process:

Google Colab	
Pro	Con
- Can run models in the cloud, so has little to no bearing on performance of computer - Majority of packages are pre-installed and pre-updated to the latest versions - Runs extremely quickly, so many experiments can be run	- Requires Colab-specific syntax that can make it trickier to follow - Need to reinstall net new packages and dependencies with each run - Somewhat of a blackbox as to what dependencies are at play - Time-outs caused us to have to re-run certain models multiple times

Once we brought the data into Google Colab, we unzipped the folders, re-inspected the images, and proceeded to the evaluation portion of building this model.

Splitting the Data into Train/Validation/Test Sets

The next item that was addressed was finding a seamless way to split the remaining 270 folders into training, validation, and testing in an automated way. We could have done this by hand and dropped images into separate training, test and validation folders. Alternatively, there are a few functions that we could have written that could have loaded the images in as an array and then separately loaded the first portion of their individual names as the labels. However, we were looking for a clean approach to be able to do this in an elegant way. We leveraged an open source library called **split-folders** that was able to take all folders in the unzipped file and quickly split each one randomly into a train test split folder.

Once installed and integrated, we split the data by having 60% allocated to test and 20% allocated to both testing and validation.

Role of ImageDataGenerator

Rather than loading the data into a standard CNN model, we leveraged the Keras function ImageDataGenerator to allow us to conduct data augmentation. This would consequently help in minimizing the potential of overfitting our model.

ImageDataGenerator takes your training, testing, and validation data, and feeds it through a pipeline. In this pipeline, you can choose to make create a synthetic dataset on top of the existing one, providing your network a better ability to actually *understand* the features of the images instead of simply memorizing them.

With the ImageDataGenerator, you have the ability to make the following transformations:

Rescale	Rescales all the images between 0 and 1 for computational efficiency
featurewise_center	Sets input mean to 0 over the dataset
Shear_range	Shears the image within a given range
zoom_range	Zooms into the image within a given range
featurewise_center	set input mean to 0 over the dataset
samplewise_center	set each sample mean to 0
featurewise_std_normalization	divide inputs by std of the dataset
samplewise_std_normalization	divide each input by its std
zca_whitening	apply ZCA whitening
rotation_range	randomly rotate images in the range (degrees, 0 to 180)
width_shift_range	randomly shift images horizontally (fraction of total width)
height_shift_range	randomly shift images vertically (fraction of total height)
horizontal_flip	randomly flip images
vertical_flip	randomly flip images

By training using this synthetic data for the training validation sets, not only do we *minimize* the chance of overfitting, but we gain more confidence in the ability of the network to properly differentiate between the different logos. One additional advantage is that the **Flow_From_Directory** method allows for all images to become automatically labelled with the name of the folders they originate from.

After this point – our CNN model was setup and a few settings were optimized in the generator so that experimentation would be greatly facilitated moving forward.

ModelCheckpoints

Using the Keras.callbacks library, we integrated model checkpoints to log the weights of the epochs where the validation loss was successfully reduced. This was stored so that it could be downloaded and loaded for later use.

One of the primary reasons why this was needed was because although our models were now running on a TPU in the cloud, Google Colab sets a limit on how long run-times can extend. This means that during many of our runs, we would come back to realize that our connection timed-out halfway through the run, thus requiring us to kick off the entire process once more. The callback function allows us to start re-running the tests where the last epoch left off.

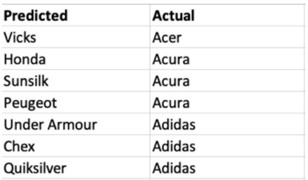
Early Stopping

We integrated early stopping in order to further avoid overfitting as well – it ended our runs early after a set of consecutive iterations without performance improvement to our validation accuracy. By default, we set our patience parameter from 5-10 depending on the model we were running and the number of epochs we were running it for.

Architecture to Assess the Model Performance

Building off of the previous data augmentation pipeline, there are two more elements that were implemented to monitor a model's results *after* running: **evaluate_generator** and **predict_generator**.

The evaluate_generator summarized the best validation accuracy noted during each run, and we leveraged the predict_generator to assess the model's performance on the test set. One major caveat with this was that it created an array of probabilities for each class rather than generating a single output. To maneuver around this and receive an overall level of the model's accuracy on the test set, we had to take the max probability of each class, replace the enumerated labels of the classes with their actual labels, and converted this into a dataframe. From here, all we had to do was isolate the Predicted versus Actual columns and calculated the portion of correctly predicted guessed over the total number of images in the test set (1,080). The output resembled the following:



Predicted	Actual
Vicks	Acer
Honda	Acura
Sunsilk	Acura
Peugeot	Acura
Under Armour	Adidas
Chex	Adidas
Quiksilver	Adidas

Reviewing the images that were predicted incorrectly allowed us to understand what hyperparameters may have been the major contributors – for example, we noted that certain images of Adidas featured the logo at the sides of the images rather than in the center. This finding eventually led us to experiment with the kernel sizing which greatly paid off in improving results.

Experimental Approach for Finding Our CNN Minimum Viable Product (MVP)

As learned throughout the course, the accuracy of a deep learning model can vary between ranges from 30-90% just based on the hyperparameters that are selected. With that being said, we attempted to tackle the creation of our CNN from scratch through a structured trial-and-error approach.

To begin, we started by attempting to create a model that could process one of the nine categories of logos well. In this case, again, the **'Accessories'** dataset was chosen.

As a starting point, we attempted to mimic the vanilla VGG16 model structure and scale-down from there based on observed performance. We started with processing through the whole network, and then began taking out more and more layers - we quickly noticed that this smaller subset of data responded better with less layers.

Very quick iteration was possible since this simple model was able to run with the subset of data for 50 epochs within a half hour. This provided us with the ability to understand the impacts of additional hidden layers and elements (*dropout, max pooling, filter sizes*), which was helpful when processing with the whole dataset.

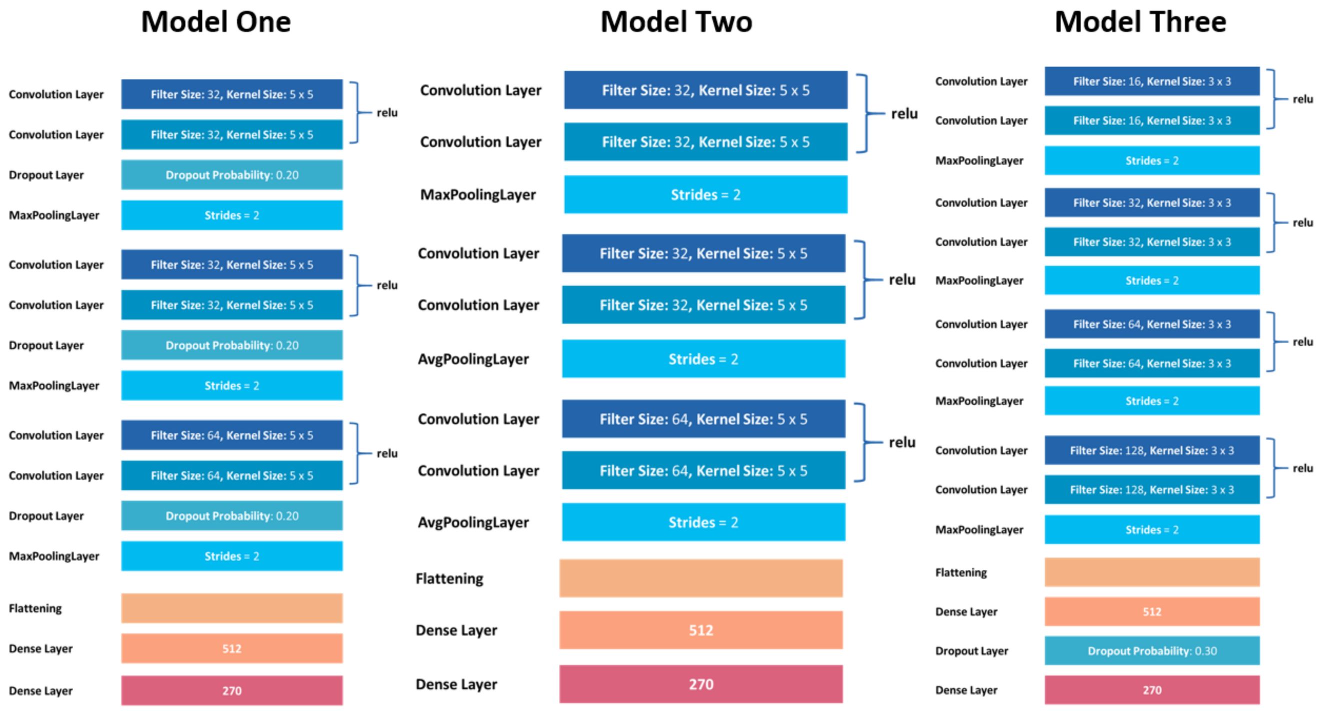
Using the same logic of scaling *down* to reach an MVP for the smaller network/dataset, we progressively scaled *up* with the full dataset.

We began with adding one additional layer to the MVP model used with 'Accessories' and added more layers from there. With each trial we did, we would try it with a consistent batch size to start (32), and toggle between 8/16/24/32/64 to gauge performance within 30 epochs. If we noticed that one variation of the model was performing particularly better, we re-ran that model with 50 epochs to gauge whether or not it would have any impact on performance.

After executing several initial trial runs with the larger dataset, we began looking for ways to increase efficiency and save time – even with the TPU engine running in the Google Colab environment, a single run could take from 3-6 hours with the entire dataset.

Model checkpoints were helpful especially when experiencing any timeouts in the Colab environment that would have otherwise prompted us to re-run the entire model from scratch. If a model that was converging well was interrupted or continued making progress that would be further enabled with more epochs, we just loaded the last set of weights from the previous run. This saved a lot of computational time and allowed the team to iterate more easily.

From our best performing runs, we then took the following three models specified below, and continued to experiment with some of the hyperparameters to help determine which one would culminate in our final version:



The **'Supporting Rational for Model Design'** section will demonstrate our selected model with further explanation on the rationale behind it.

Transfer Learning Experimentation

To stay consistent in our methodology, we integrated transfer learning with the use of generators as well. We introduced five key transfer learning methods which included the following:

- **VGG16**
  a. Of all the transfer learning structures we used in our study, VGG16 is the most vanilla version. The novelty in the VGG design is the increased depth and the use of very small (3x3) convolution filters compared to the state-of-the-art CNNs in 2014.

- **VGG19**
  a. The larger the network, the hungrier it gets. An ablation study showed that performance (e.g. Imagenet based error rates) started to converge as additional convolution layers were added to the base VGG-11 structure. VGG-16 and VGG-19, as shown in Figure 1, are most popular as that's where performance gain stops improving with additional layers.

- **Inception V3**
  a. No longer layer-based, but module-based (i.e. Network-In-Network). In regular VGG layers, a single convolution kernel is responsible for mapping cross-channel correlations

and spatial correlations. Basic idea of Inception's hypothesis assumes that cross-channel correlations and spatial correlations in images can be decoupled, and explicitly factors the convolution steps into a series of operations.

b. These series of operations are as follows:
   i. First, map the input cross-channel correlations via a set of 1x1 convolutions, factoring them into smaller 4 separate spaces
   ii. Second, map each smaller spaces' spatial correlations using the regular 3x3 convolution layers
   (see Appendix InceptionV3 Inception-A module)

c. Using this, Inception can have much more sparse structures when compared against a regular convolution/pooling structure such as VGG models. This sparse nature allows the network to grow deeper without overfitting the training data.

- **ResNet50**
  a. As the networks get deeper, vanishing gradient problem can degrade the model performance (e.g. VGG stopped improving after 19 weight layers). Key conceptual difference of ResNet is the existence of skip connection (a.k.a. residuals), which is used to mitigate the vanishing gradients during back-propagation while keeping the deep architecture to better approximate the mapping.
  b. ResNet uses Batch Normalization which supports easier optimization via normalized gradients, and which also allows a higher learning rate.

- **Xception**
  a. Xception takes it one step further from Inception. Instead of the regular Inception blocks, it replaces them with depth-wise separable convolutions. This allows more efficient use of weights during training. (i.e. better performance gain for the same amount of trainable weights). Note that the actual number of parameters to train in Xception (23M) are roughly the same as InceptionV1.
  b. It is worth noting that decoupled architecture and transfer learning methods are specifically designed for images that are structured: RGB channels on a spatial WxH grid.

For each of these models, the generator was once again re-introduced, and was re-run by fitting the transfer learning model into the vanilla CNN developed from scratch to see how it would function. The results of this implementation will be available in the experimentation section.

Image Similarity Experimentation

Given that we spent weeks trialing and testing our various model versions and transfer learning models to tune them as much as possible, there was less time than expected to develop a fully-functional image similarity framework; however a couple of frameworks were found through our research that compared images through two different means:

1) **Method One: Mean Squared (MSE) & Structural Similarity Index (SSIM)**

MSE was used to calculate the mean squared error between the pixels of the two images under comparison, element-wise.

The **lower** the MSE, the more similarly perceived the two images were.

SSIM compared the pixel density values between two images and created a form of ‘correlation’ index between -1 and 1.

The **higher** the SSIM, the more similarly perceived the two images are.

2) **Method Two: Euclidean Distance**

   This method involved calculating the Euclidean distance between element-wise pixels and determining the average pixel distance between them as a means to gauge similarity.

Again, the goal of employing these discovered approaches was to produce a proof-of-concept that could be further developed in the future.

Supporting Rationale for Model Design

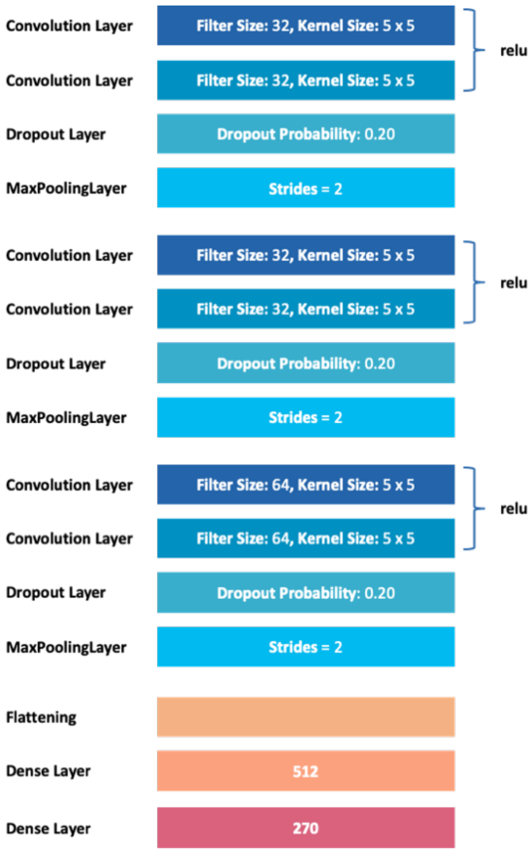
Note: Only the logo recognition portion of the project will be addressed in this section.

Logo Recognition CNN Design

The primary use case for leveraging deep learning for this experiment is the fact that it provides an efficient avenue to learn generic features rather than specialized ones – through conducting image classification, the multi-layer structure of a neural network allows it to extract unique features at every level which at the aggregate empowers it to more efficiently classify images.

Building on this, a CNN was chosen to carry out this project as it’s a specialized network that’s very capable of processing images – filters are naturally learned by the network rather than having to be manually laid out in other machine learning approaches. The closest network that is also efficient at image classification would be an RNN – because this is a multiclass problem with over 270 unique classes, the CNN is better-suited as the RNN’s use case is centred on more of a one-to-one mapping. For example, if our project prompt was to determine whether a logo had a black versus a white background, an RNN’s memory-charged network could be worth exploring.

After our extensive experimentations, the following is the architecture of the best performing network on the dataset:


Convolution Layer
Filter Size: 32, Kernel Size: 5 x 5
Convolution Layer
Filter Size: 32, Kernel Size: 5 x 5
relu
Dropout Layer
Dropout Probability: 0.20
MaxPoolingLayer
Strides = 2
Convolution Layer
Filter Size: 32, Kernel Size: 5 x 5
Convolution Layer
Filter Size: 32, Kernel Size: 5 x 5
relu
Dropout Layer
Dropout Probability: 0.20
MaxPoolingLayer
Strides = 2
Convolution Layer
Filter Size: 64, Kernel Size: 5 x 5
Convolution Layer
Filter Size: 64, Kernel Size: 5 x 5
relu
Dropout Layer
Dropout Probability: 0.20
MaxPoolingLayer
Strides = 2
Flattening
Dense Layer
512
Dense Layer
270

A smaller network was chosen as it not only produced better results, but it was exponentially less computationally expensive.

Here is a breakdown of the architecture seen above with some rationale where applicable:

- **Convolution Layer Pairs**: As noted in the experimentation portion of the paper above, we started by emulating a VGG16-like architecture, and stripping down layers one-by-one and gauging the network improvements/degradations without doing any initial hyperparameter tuning. Once we got to singular layers, we noticed a drop-off in performance from the previous iterations. So we added on some of the layers we took off one-by-one, and noted that the pairs of convolution layers seemed to perform well.

- **Dropout Layers**: With our penultimate version of our model, we didn't have any dropout layers, and observed a larger discrepancy between the training accuracy and the validation/testing accuracy, suggesting that our model was severely overfitting on the training data. One of the means we used to remedy this is by inputting a dropout layer after every pair of convolutions – after re-running this version of the model, we noted that the validation accuracy increased by 7% from around 55% to 62%. We experimented setting a dropout probability between 0.2 – 0.4, and noted that 0.2 version had the best validation accuracy – given that the size of the dataset was only 5,400 images, we discerned that the larger dropout rates began inhibiting the learning of the network rather than bettering its' intuition.

- **MaxPoolingLayer:** Regardless of how large or small the networks we tested were, the average and max pooling layers almost always had a positive effect on performance – this is principally due to the fact that it lowered the number of parameters that the network had to weight. With our final model, we noticed that average pooling layers performed slightly lower than max pooling (even when alternating between the two in the same model). As such, we chose with going with the MaxPoolingLayers with a stride of 2 throughout.

- **Filter/Kernel Sizes:** We experimented with several filters sizes, and eventually noted that filter sizes between 32 and 64 appeared to perform the best of all our iterations. Similarly, we spent many of our iterations with a kernel size of 3 x 3, with minimal improvement from model to model. Once we noted which logos the network was predicting incorrectly, we increased the filter size from 3 to 4, then 4 to 5, and noticed the overall accuracy doubled.

- **Dense Layer Size:** Given that this is a multi-class classification problem, the final dense layer was fixed since it needed to reflect the total number of classes (270) – for the layer before it, we experimented with a few values and noted that 512 appeared to yield the best results.

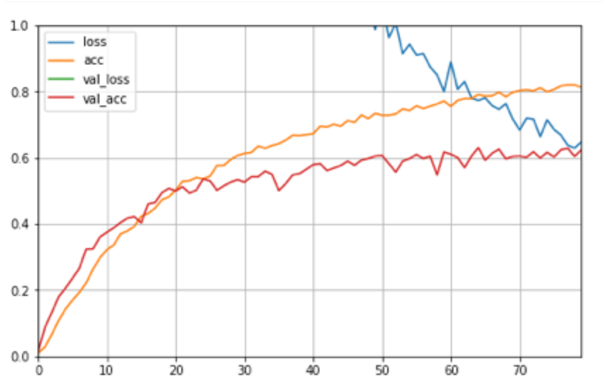
Results

Logo Recognition Results

For both our vanilla CNN model, as well as the transfer learning architectures we experimented with, we used accuracy as our primary performance metric of choice. Typically, this would be a suboptimal decision on a multi-class problem given the fact that its highly prone to being biased or deceiving for imbalanced datasets. However, given the fact that we confirmed that the dataset being used is exactly balanced across all classes, we were more confident in using accuracy to gauge performance.

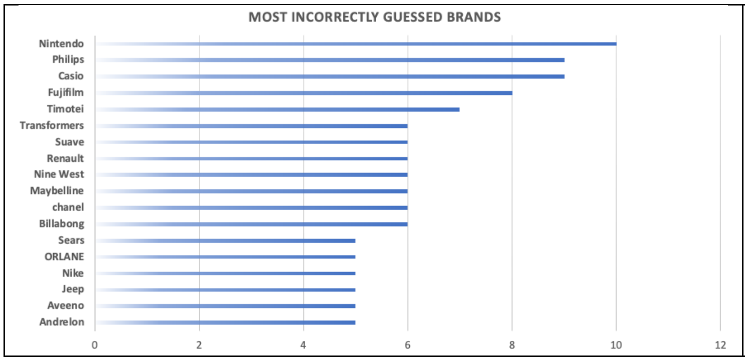
These were the results of our final model:

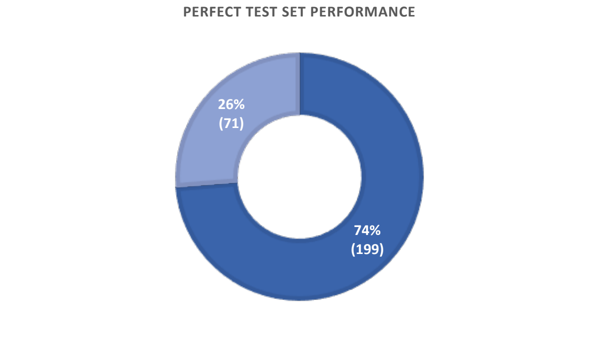
	Training Accuracy	Validation Accuracy	Testing Accuracy
Our CNN	82.5%	62.5%	62.5%



While our final training accuracy was very good, there was still a notable difference in the validation and testing accuracy. This is likely due to some overfitting on the training data.

Through the predict_generator, we exported the prediction results to gauge which logos the model kept confusing with other brands. The left figure on the chart below displays our findings:





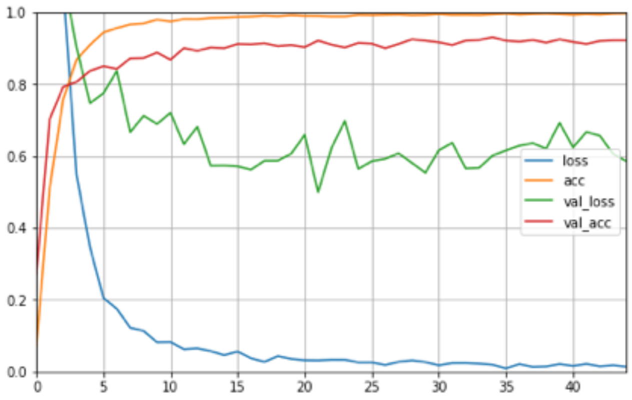
On the right side, we were curious to see how many logo classes the test set managed to get perfectly right (*guessing 4/4 images of that class correctly*). We think that with further hyperparameter tuning and more time, we could further reduce overfitting and align the training and validation/testing performance.

These are the results of our CNN vis-à-vis the transfer learning models that were integrated:

Transfer Learning Results

	Training Accuracy	Validation Accuracy	Testing Accuracy
Our CNN	82.5%	62.5%	62.5%
VGG16	87.1%	79.9%	78.7%
VGG19	94.2%	84.9%	85.9%
Inception V3	96.4%	86.8%	87.5%
ResNet	95.5%	81.5%	84.0%
Xception	99.7%	92.1%	91.9%

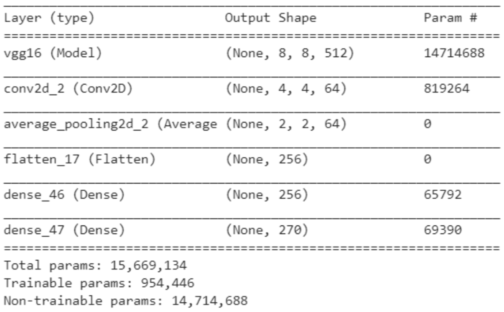
Best Performing Model: Xception



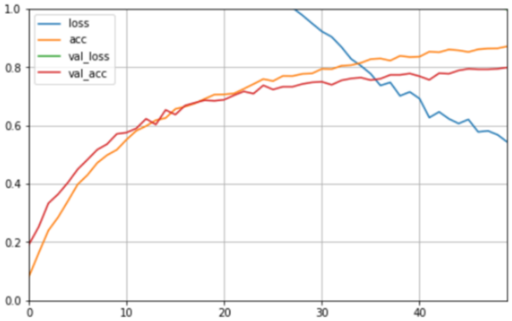
Extra Remarks on Transfer Learning

VGG16 Results

We assessed the VGG16 model summary which has roughly 14.7 million weights and froze all layers except the last one. We used VGG16 as the base network and added additional layers as shown below and reduced the trainable parameters from 14.7 million to 0.9 million. We trained the new model, using the same hyperparameters for comparing across different transfer learning base models (batch size = 32, number of epochs=50, optimizer = ‘RMSProp’, learning rate = 0.00001, metric= ‘accuracy’). We also kept track of its’ history.



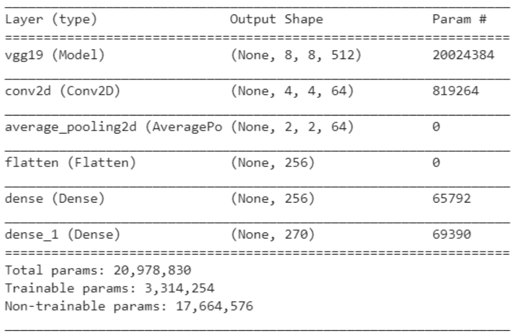
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 8, 8, 512)         14714688
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          819264
_________________________________________________________________
average_pooling2d_2 (Average (None, 2, 2, 64)          0
_________________________________________________________________
flatten_17 (Flatten)         (None, 256)               0
_________________________________________________________________
dense_46 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_47 (Dense)             (None, 270)               69390
=================================================================
Total params: 15,669,134
Trainable params: 954,446
Non-trainable params: 14,714,688
```



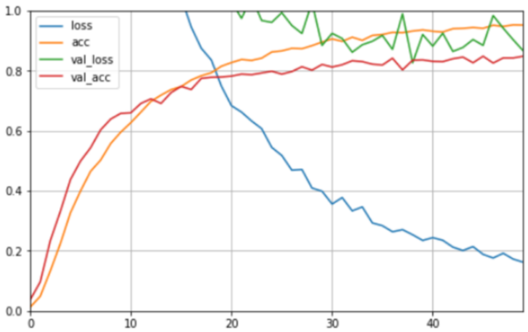
VGG19 Results

We used VGG19 as base network with frozen layers (except last 2) and added additional layers as shown. The last 2 layers of VGG19 were included in model training during back propagation.

We tuned the model with regularization using additional dropout layer this time which didn’t improve the validation or test predictions and hence no regularization was performed on these models. The number of trainable parameters dropped from 20 million to 3.3 million by freezing the VGG19 base model except last 2 layers



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg19 (Model)                (None, 8, 8, 512)         20024384
_________________________________________________________________
conv2d (Conv2D)              (None, 4, 4, 64)          819264
_________________________________________________________________
average_pooling2d (AveragePo (None, 2, 2, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 256)               65792
_________________________________________________________________
dense_1 (Dense)              (None, 270)               69390
=================================================================
Total params: 20,978,830
Trainable params: 3,314,254
Non-trainable params: 17,664,576
_________________________________________________________________
```



VGG-16 vs VGG19

The deeper VGG-19 model gave better prediction results for our Logo dataset, although performance was almost on par between the two on the training set. We infer that this could be due to inclusion of an extra layer in VGG19 model for trainable parameters.

InceptionV3 vs Xception

As explained in the Transfer Learning section, Xception replaces Inception modules with depthwise separable convolution, allowing a more efficient use of its weights during training. This is reflected in the result as shown in “Transfer Learning Results” table.

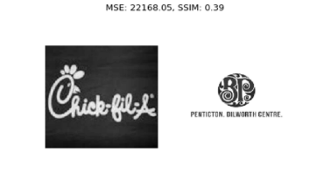
ResNet-50

Because of our limited time to explore transfer learning models, we only tested using ResNet-50, not the larger ResNet-101 or 152. The smaller version is generally used to transfer learnt weights but also to re-learn additional last few layers built on top of ResNet-50 for customized mapping. Trying out the deeper versions of ResNet may give us an even better result than Xception’s outstanding performance.

Based on the above observations, we can make a preliminary conclusion about our Logos-set image problem: It prefers deeper architectures for mapping out the features.

Image Similarity Preliminary Results

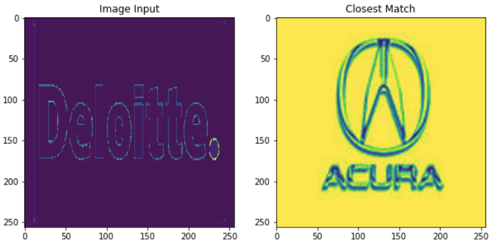
Image Similarity Via MSE and SSIM

Image Comparison 1 (*for testing purposes*)	Image Comparison 2	Image Comparison 3	Image Comparison 4
MSE: 0.00, SSIM: 1.00 CHANEL CHANEL	MSE: 37286.23, SSIM: 0.14 CHANEL	MSE: 22007.86, SSIM: 0.07 BILLABONG adidas	 MSE: 22168.05, SSIM: 0.39 Chick-fil-A PENTICTON. DILWORTH CENTRE.
MSE: 0.00 **SSIM:** 1.00	**MSE:** 37286.23 **SSIM:** 0.14	**MSE:** 22007.86 **SSIM:** 0.07	**MSE:** 22168.05 **SSIM:** 0.39

Looking at our proof-of-concept results, we tested out comparing multiple logos, and noted how the MSE and SSIM differed in each case. We compared two identical images as a base case to ensure that the metrics were performing as expected – of the remaining comparisons, we tried to using the wrong predictions our model made in logo recognition to see if there was any relation to these results. One of the model's most common errors was between Chick-fil-A and Boston Pizza – the SSIM of that image comparison seemed more correlated than the rest of the examples, so this may have some bearing on our model's confusion.

Computing Image Similarity through Euclidean Distance

For this experiment, we tried loading in a net new image (*Deloitte's logo*) and seeing what the function would output as the closest match based on the average pixel distance – the result demonstrated that further iterations would be needed to generate better matches:



Implications of Model Deployment

From the experience of conducting the pre-processing of the data for the models alone, it can be said with full confidence that a fully operational model in a production-like environment would require extensive back-end adaptability to perform at scale.

For both the image recognition networks and similarity models, many of the same obstacles encountered in scrubbing the original dataset would be amplified if there were to be millions of images funneling through the model – even though the production use case of our project would be for companies to be able to compare the uniqueness of logos to gauge any copyright infringement, if many companies were to deploy this model as-is, it would quickly display its inability to perform.

These are a few of the key areas that would present difficulties for operationalization:

1) **Input Data Variability**: Some of the key elements needed for initial images was that they had to be resized, converted to .jpeg format and checked for cleanliness – all of the logo images that were taken in the real-world or appeared on a product were manually removed. If this product were to be scaled, it would need to be able to not only handle images of different types but be able to detect logos within the backdrops of other images as well.

   As it stands right now, the current model is not optimized for real-life images, so we'd need to introduce those types of images and re-run our model to tune it effectively. Additionally, we haven't tested whether or not Tensorflow's ImageDataGenerator is compatible with certain types of images formats such as TIFF, BMP, or GIF for example. Not only would more images be need to fed into this model to make it robust, but more 'formats' of images as well.

2) **Inevitable Dataset Imbalances**: Categorical imbalances are bound to happen in reality. If we reflect on the initial dataset which included over 150,000 images, in preprocessing section we referred to the fact that many folders had a vastly larger number of images compared to others. In a production environment, the chances of the dataset being fed in being as clean as the one we spent hours manually putting together are extremely low – that being said, a proper oversampling framework of splitting the training/validation/testing distributions would have to be put into place to account for this.

3) **Data Acquisition**: As seen from the original dataset, the means with which the data was acquired was one of the principal reasons why the dataset was not clean. To be able to scale the number of categories and brands to make this model more meaningful, another means of data acquisition would be required. It is possible that web scraping Google could still be done as a means of automating the process, but there would have to be some form of validation process added to ensure the data integrity of the dataset.

4) **Library Dependencies**: As we built and ran this model in Google Colab, there are many Colab-specific libraries being leveraged. Additionally, there are many TensorFlow methods being leveraged as well, such as the flow_from_directory function (*that enables the ability to load image data without preset labels*). In a production environment, having these dependencies in play would be a huge liability and constraint to the ability to scale. With that said, some of the code would have to be re-written to be able to make it usable and ubiquitous to most stacks.

5) **Computational Cost**: We migrated our code to Colab as a result of the long processing times we experienced running through all 5,400 images – even with running our code on a TPU engine with almost 4x the computational speed and RAM as my desktop computer, it took around 8 hours to run our selected model until it converged. In a production environment where the size of the dataset would likely be in the hundreds of thousands of images, a much more robust infrastructure

would be needed to handle the consistent iteration that would be required to improve and maintain the model.

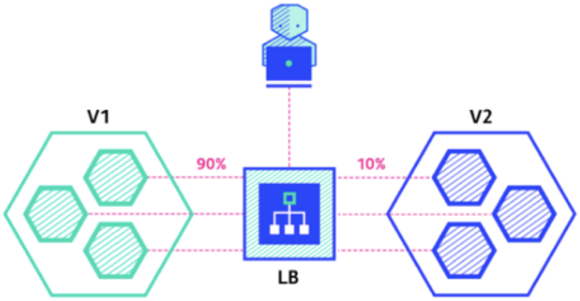
6) **Hyperparameter Retuning**: As alluded to in 1), with more volume and formats of data, our model will need to be re-calibrated, requiring consistent rounds of tuning and maintenance. With this added constraint, the ideal solution to this would be to implement a framework similar to hyperas to consistently adapt the model's grid search results to retune the model without having to go through the numerous iterations

7) **Explainability**: As with all deep learning models, one constant obstacle will be the ability to properly communicate what the model is doing, how its processing the data, and most importantly, how its 'learning' rather than memorizing the data its being fed.

Proposed Deployment Strategy:

Given that obstacles 1 – 7 are to be properly addressed, we recommend that if the model had to subsequently be deployed, that it be released using a canary deployment method (*deploying to small subset of users and then roll out change to rest of users once fully operationally ready*).

This would ensure that newer versions of the model would be released in a very gradual and progressive basis to ensure that it's behaving as expected and can quickly be rolled back if required.



Further Considerations and Final Thoughts

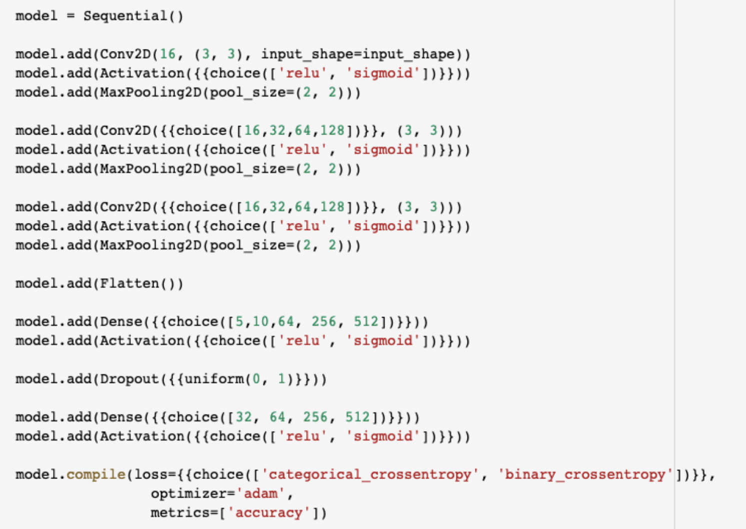
Looking back at the process that was conducted, we would have done quite a few things differently:

1) **EDA**: While we did go through samples of images from each category, we did it at a high level, and happened to initially observe the folders that were relatively clean. One of the natural biases that enabled this was choosing brands that were less innocuous to inspect (such Adidas, McDonalds and Acura). This bias naturally drew us to looking at images that would have been more likely to have been accurate with an automatic web scraper.
    a. Instead of this approach, we would have tried to use Python to pull and display a random three images from each folder. This would have been exponentially more efficient than manually going through the folders.

2) **Pre-processing Efficiency**: Close to 20 hours was spent progressively sifting through images and replacing/adding ones manually to create the resultant pre-processed dataset. The majority of this manual labour could have been avoided with some additional coding resource experience – using the split-folders library mentioned in the preprocessing section (hyperlink), we learned that we could have done the splitting of folders using oversampling where a folder didn't meet a certain threshold in terms of image count.

3) **Hyper-Parameter Tuning**: The process of conducting trial-and-error in search of the best performing vanilla CNN model would have been greatly expedited if we had the chance to properly integrate hyperopt - this module from the hyperas library would have allowed us to conduct a form of grid search which would have greatly expedited the hyperparameter tuning process. It allows the user to try multiple options for all the key hyperparameters, and even automatically experiment with adding different layers to gauge any potential performance improvements.

   Snippet of attempt to integrate hyperas:



```
model = Sequential()

model.add(Conv2D(16, (3, 3), input_shape=input_shape))
model.add(Activation({{choice(['relu', 'sigmoid'])}}))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D({{choice([16,32,64,128])}}, (3, 3)))
model.add(Activation({{choice(['relu', 'sigmoid'])}}))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D({{choice([16,32,64,128])}}, (3, 3)))
model.add(Activation({{choice(['relu', 'sigmoid'])}}))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense({{choice([5,10,64, 256, 512])}}))
model.add(Activation({{choice(['relu', 'sigmoid'])}}))

model.add(Dropout({{uniform(0, 1)}}))

model.add(Dense({{choice([32, 64, 256, 512])}}))
model.add(Activation({{choice(['relu', 'sigmoid'])}}))

model.compile(loss={{choice(['categorical_crossentropy', 'binary_crossentropy'])}},
                optimizer='adam',
                metrics=['accuracy'])
```

   All areas denoted with "choice" provided the proposed grid search for each hyperparameter. We endeavoured to integrate this with multiple iterations, but could not figure out how to properly mesh our image generators into the various functions.

4) **Image Similarity**: We gravely underestimated how difficult this task would be – had we had more time, we would have wanted to explore two additional methodologies that are more comprehensive and state-of-the-art in gauging image uniqueness:

   a) Deep Ranking – method used by Google to retrieve similar images to the inputted query image
   b) Locality Sensitive Hashing – method leveraging semantic similarly search through using embedding collections

As a result of this project, we developed an image recognition classifier using deep learning and a foundational similarity engine that is able to consume existing images and both identify their classes as well as gauge how similar they are to one another. The end-to-end solution would involve feeding a new logo into the engine and have it display the most similar logos that exist in the marketplace to be able to recognize and classify which company that logo represents. This information would allow the easy detection of copyright infringement as well enable companies to quickly identify whether or not they should iterate on their logos.

Overall, we have shown a method that would be able to perform this and retrieve back companies' logos with a reasonably high degree of accuracy which we would be able to be further refine with additional time and resources.

Appendix

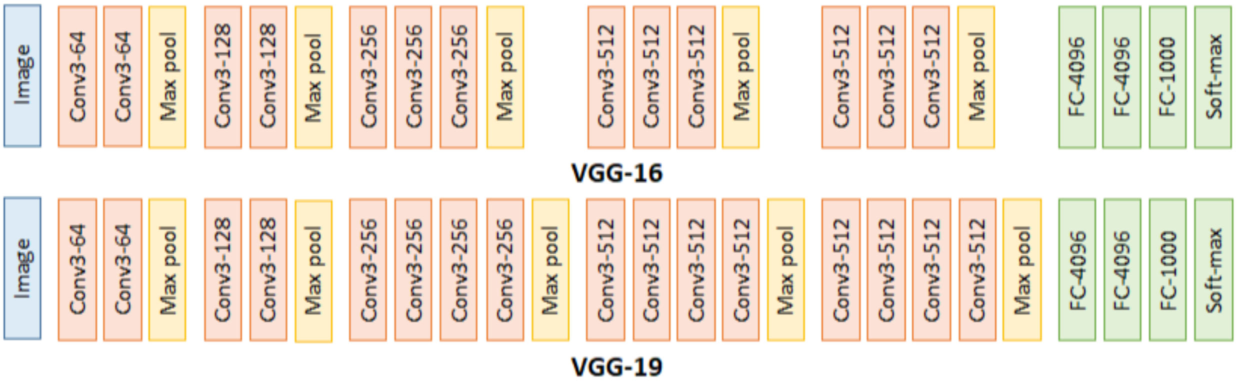


Figure 1   VGG-16 & VGG-19 description

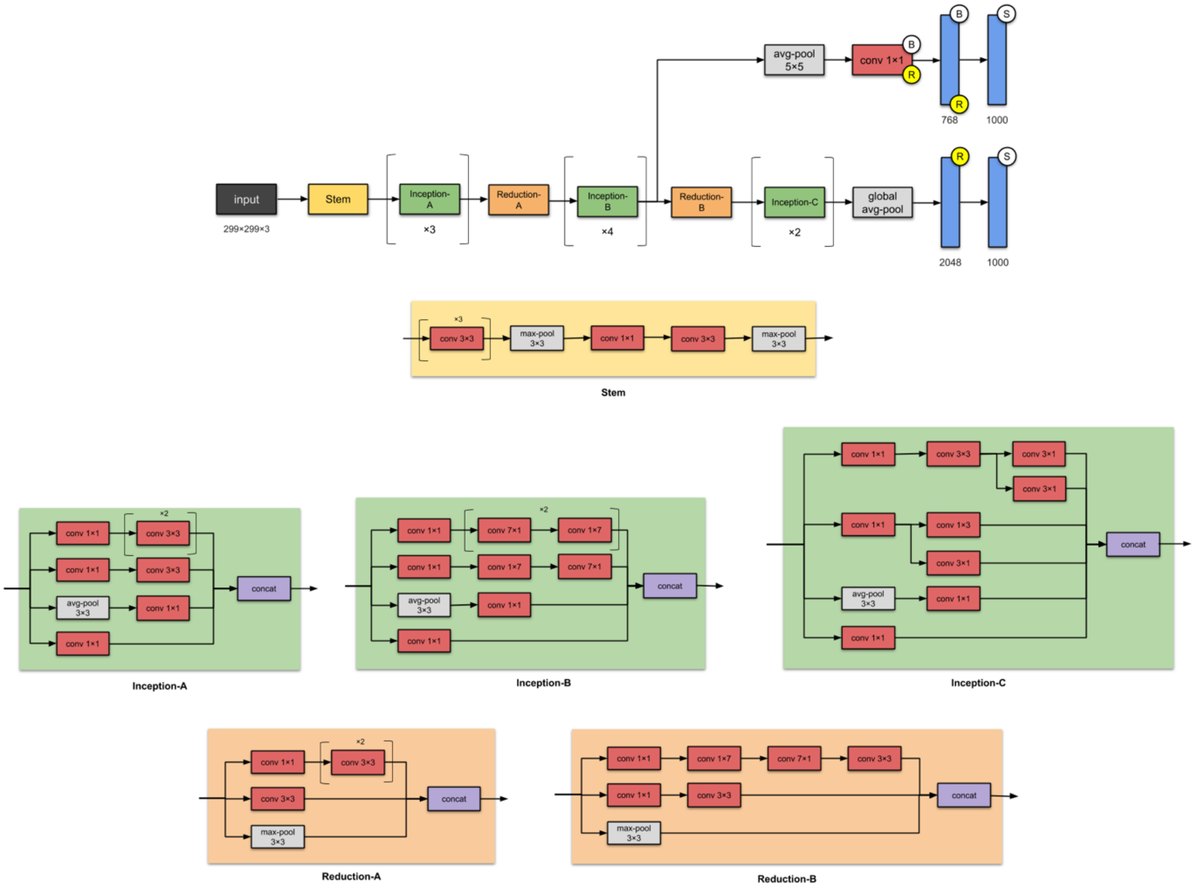


Figure 2   InceptionV3 Architecture

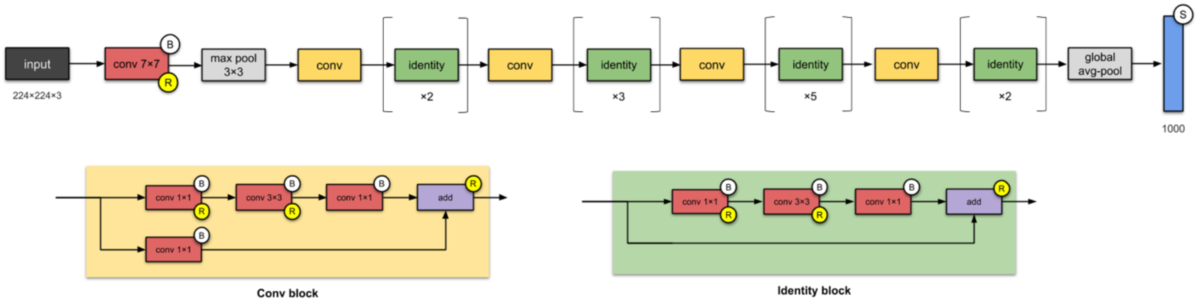


Figure 3 ResNet-50 Architecture

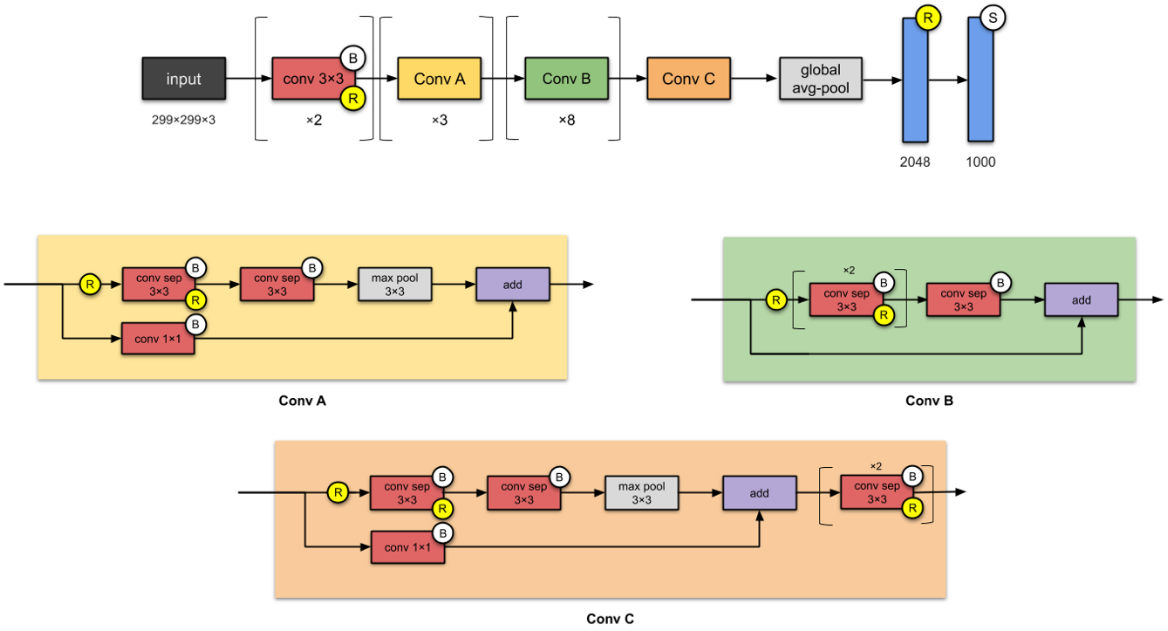


Figure 4 Xception Architecture

Works Cited

A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning. (2018). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a

A guide to an efficient way to build neural network architectures- Part I: Hyper-parameter…. (2018). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-i-hyper-parameter-8129009f131b

Brownlee, J. (2019). *How to Load Large Datasets From Directories for Deep Learning in Keras*. *Machine Learning Mastery*. Retrieved 1 March 2020, from https://machinelearningmastery.com/how-to-load-large-datasets-from-directories-for-deep-learning-with-keras/

Classifying Logos in Images with Convolutionary Neural Networks (CNNs) in Keras. (2019). *Medium*. Retrieved 1 March 2020, from https://medium.com/twodigits/classifying-logos-in-images-with-convolutionary-neural-networks-cnns-in-keras-21f02fcea5c2

Deep learning unbalanced training data?Solve it like this.. (2018). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/deep-learning-unbalanced-training-data-solve-it-like-this-6c528e9efea6

Final Project - Improving Brand Analytics with an Image Logo Detection Convolutional Neural Net in TensorFlow. (2016). *Maxmelnick.com*. Retrieved 1 March 2020, from http://maxmelnick.com/2016/08/31/final-project.html

Finding similar images using Deep learning and Locality Sensitive Hashing. (2019). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/finding-similar-images-using-deep-learning-and-locality-sensitive-hashing-9528afee02f5

Image Classification using SSIM. (2019). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/image-classification-using-ssim-34e549ec6e12

Image Similarity using Deep Ranking. (2018). *Medium*. Retrieved 1 March 2020, from https://medium.com/@akarshzingade/image-similarity-using-deep-ranking-c1bd83855978

Tensorflow 2.0—Create and Train a Vanilla CNN on Google Colab. (2019). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/tensorflow-2-0-create-and-train-a-vanilla-cnn-on-google-colab-c7a0ac86d61b

Tremel, E., Farmer, K., & Cassel, D. (2017). Six Strategies for Application Deployment - The New Stack. The New Stack. Retrieved 1 March 2020, from https://thenewstack.io/deployment-strategies/

Tutorial on using Keras flow_from_directory and generators. (2019). *Medium*. Retrieved 1 March 2020, from https://medium.com/@vijayabhaskar96/tutorial-image-classification-with-keras-flow-from-directory-and-generators-95f75ebe5720

Who's That Pokémon?. (2019). *Medium*. Retrieved 1 March 2020, from https://towardsdatascience.com/whos-that-pok%C3%A9mon-39d1150aedfe

Francois Chollet. (2017). *Xception: Deep Learning with Depthwise Separable Convolutions*. From 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)