November 4, 2018

Sidney’s Catalog: Comparison of CNNs for Highly Analogous Images

CUNY DATA698 – data Science Master Project

Chris G Martin

Contents

[Overview 3](#_Toc529015209)

[Discussion of the Approach and Tools 4](#_Toc529015210)

[Data Collection 4](#_Toc529015211)

[Comparison of Neural Networks for Image Recognition 5](#_Toc529015212)

[Multi-Layer Perceptron (MLP) 5](#_Toc529015213)

[Recurrent Neural Networks (RNN) 7](#_Toc529015214)

[Convolutional Neural Network (CNN) 8](#_Toc529015215)

[Generative Adversarial Networks (GAN) 9](#_Toc529015216)

[Neural Network Evaluation 9](#_Toc529015217)

[Transfer Learning and Evolution of Neural Network Architectures 9](#_Toc529015218)

[ImageNet & AlexNet 10](#_Toc529015219)

[VGG16 11](#_Toc529015220)

[GoogLeNet 12](#_Toc529015221)

[ResNet 12](#_Toc529015222)

[DenseNet 13](#_Toc529015223)

[Inception v3 14](#_Toc529015224)

[MobileNet 14](#_Toc529015225)

[NASNet 15](#_Toc529015226)

[PNASNet-5 15](#_Toc529015227)

[AmoebaNet 16](#_Toc529015228)

[Open-Source Deep Learning Libraries 16](#_Toc529015229)

[TensorFlow 16](#_Toc529015230)

[PyTorch 17](#_Toc529015231)

[Keras 17](#_Toc529015232)

[Library Evaluations 17](#_Toc529015233)

[Data Collection: In Practice 17](#_Toc529015234)

[Description of the Data Sets 18](#_Toc529015235)

[Convolutional Neural Network (CNN) Development: In Practice 18](#_Toc529015236)

[Self-Developed Networks 18](#_Toc529015237)

[Simple 2 Layer CNN with ReLU and Simple 7 Layer CNN with Softmax and ReLu 19](#_Toc529015238)

[Pre-Trained Networks using Tensorflow 19](#_Toc529015239)

[Evaluation of Models 20](#_Toc529015240)

[Simple 2 Layer CNN on TensorFlow with ReLU 20](#_Toc529015241)

[Simple 7 Layer CNN on TensorFlow with Softmax and ReLU 20](#_Toc529015242)

[Inception v3 21](#_Toc529015243)

[ResNet 21](#_Toc529015244)

[MobileNet 22](#_Toc529015245)

[PNASNet-5 22](#_Toc529015246)

[AmoebaNet 23](#_Toc529015247)

[Summary of Results 23](#_Toc529015248)

[References and Sources Used 24](#_Toc529015249)

[Prompt 25](#_Toc529015250)

# Overview

Art is incredibly subjective; it can be personal, abstract, or even non-existent. As such, putting a price on a piece of art can almost be considered art in itself. Usually a rare piece of art is valued at the last price it was sold for, with auctions or public or private sales driving the market. In fact, the common practice is for the owner to commission a third-party to reach a ‘fair’ valuation.

To further dive into the rabbit hole, there is no basis for what a ‘fair’ valuation even means as the motivations of the third-party are typically hidden; that is, the owner may not be aware of the motivations for the third-party who may be interested in personal gain by offering a lower valuation than its ‘true’ value, or who might be interested in the pedigree and offers a higher valuation. ‘Fair’ is also difficult to ascertain. When an item goes to auction, the audience is a small selection of those potentially interested in buying a limited piece of art. When an item is held by the owner for a long-period of time, the last selling price has been outdated long ago.

Additionally, the differences between art can be wide or incredibly minute. In some cases, a valuation would depend on obvious factors such as color, maker, year, vintage, pattern, or shape. In other cases, the differences in valuation can be for incredibly minor (and hard to spot) details such the appearance or visibility of a mark (the maker’s mark for example), the slightest of chips, or even the width of the object in relation to its peers. Understanding the subtleties takes years of experience and (often times) instinct that expensive brokers are able to charge for.

My long-term goal is to alleviate these long-standing pain-points that consumers are faced with. Sidney’s Catalog is intended to be an image recognition engine specifically directed at antique glassware. The objective of the engine is ambitious: properly classifying the type of glass and pattern based on a photograph and using that classification to forecast a price range for potential antique buyers or re-sellers. Unfortunately, the crux of this engine depends on an advanced prototype of a proper image recognition system; price forecasting is well outside of the scope of this project and requires a much different path for success.

There are several approaches both in market (used by companies for business opportunities) or in research for the development of an image recognition system. The most common approach is the use of a Convolutional Neural Network (CNN), which will be discussed in detail.

The scope of this paper is to evaluate the effectiveness of CNNs for image recognition as used in research and business, and to evaluate the variations of CNNs as they would be applied for this specific use case. Glassware, especially clear glassware, can be highly analogous making them difficult items for neural networks (or even humans) to differentiate. By evaluating the effectiveness of different CNN methodologies (including newly developed models and pre-trained models), the first (and most challenging) step towards cutting out brokers in art valuations can be achieved.

The requirements of this project also rely on the collection of a set of images to either (1) train a network on, or (2) test the effectiveness of a network with, or (3) train and test a network. Effectiveness of a network will be evaluated on three primary metrics: highest percentage of accuracy, runtime of the model, and cost of running the model. As the sole source of funding for this model is non-discretionary (subject to a strict personal budget), the evaluation criteria are not evenly distributed between each metric.

# Discussion of the Approach and Tools

The approach to this project can be laid out as follows:

I will first collect a sample of images for each class of glassware I want to identify. With limited tools at hand, data collection is achieved by simple web scraping across a variety of image search providers: Google, Yahoo, Yandex, and Bing. In addition, the imgdl package for Python is used in an attempt to simplify the data collection with varying results. The methods and tactics used for each provider (as well as the hands-on experience of using imgdl) is detailed.

After collecting the data, I will build a variety of models to determine the pros and cons of each type of model and methodology. The discussion for how these models were selected are discussed, as well as a discussion on the theoretical advantages and disadvantages of the approach of these models on my problem. The main goal of my project is to achieve the highest accuracy model, so understanding the advantages and disadvantages of each model as well as nuances of each model is important before diving in. Not only will it set expectations for my results, but it will also reduce the amount of tweaking I have to do. An evaluation of software and tools available for use is also discussed.

Finally, the experience of building these models is detailed as well as a comparison of the performance for each model.

## Data Collection

As with most challenges looking to be solved through the use of a deep learning, results can be highly dependent on the quality of the data it is given. For example, if the challenge was in distinguishing between different types of fauna as taken from a mobile phone camera then a data set which includes fauna in a variety of lighting or in a variety of angles, otherwise the likeliness of an accurate categorization would be low due to the dynamics of mobile phone cameras and unprofessional mobile phone photographers.

By using a variety of search engines, I hope to also gain a variety of images with a variety of quality standards. The ideal image would be a single piece of well-lit glassware against an all white background displaying the full glass. The search engines I expected to use were: Google Images, Yahoo Images, Yandex Images, and Bing Images.

From a high-level observation, the results of these search engines are remarkably different. Google Images tends to have images from online stores and retailers, so the quality is typically high though the result is fairly low. For example compared to our ideal image of a single glass, what we appear to get is a group of glasses or a cropping of the glass to make it more appealing for online shoppers. Yahoo Images, on the other hand, gives a range of quality images from professional pictures to personal images of the results. Yandex is a Russian website, and quality of the images were surprisingly good. Bing images are remarkably similar to Yahoo.

The python package imgdl can be used to download a collection of images from a set of URLs. The package appears to be an efficient means of quickly gathering a large set of data using multithreading and persistent caching. If this package can be used successfully, data gathering can become an extremely simple and effective process.

## Comparison of Neural Networks for Image Recognition

At its most basic, deep learning is simply the application of an artificial neural network for development, training, and use of a neural network. This section discusses the types of neural networks used in practice that are generally flexible and reliable on a range of problems and datasets:

* Multi-Layer Perceptron (MLP)
* Recurrent Neural Network (RNN)
  + Long-Short Term Memory (LSTM or STM)
* Convolutional Neural Network (CNN)
* Generative Adversarial Network (GAN)

In a sense, neural networks learn the mapping representation of training data and use this mapping to predict an output. The accuracy of a model is the number of correct outputs (guesses) given a test set of data.

### Multi-Layer Perceptron (MLP)

A multilayer perception is a class of artificial neural network comprised of one or more layers of neurons. This class of model can be considered the de-facto class; a perception is simply a single neuron model – the precursor to larger neural networks. A neuron is a computational unit of weighted inputs which produces an output using an activation function. A neural network is a combination of neurons. In a simple neural network there is an input later, one or many hidden layers, and an output layer. A “deep” neural network has a large number of hidden layers. Said differently a simple neural network consists of multiple layers of perceptions, or a Multi-Layer Perception (MLP) network.

By its definition, MLPs are very flexible and can be used in a general sense. MLPs can be used to learn a mapping from inputs to outputs. It is useful in a variety of circumstances, taking any kind of data as an input and returning a prediction output. For example, a word can be restructured as a single row vector and these values used as an input for mapping. An image can be mapped as a single row vector and used as an input for mapping.

MLP networks are also commonly known as feedforward neural networks, as the input data is fed as an input and generates an output. The data flows through each layer sequentially, moving forward and never backward. An enhancement of the feedforward network is the ability to back propagate errors and update network weights. Backpropagation can dramatically improve training in a neural network and shorten the time required.

The best use case for MLPs are on regression problems, classification problems, or tabular datasets (such as those in a CSV file or spreadsheet). MLPs can effectively be used on image data, text data, and time series data.

#### Activation Functions

Activation Functions are critical features of neural networks; they decide whether a neuron should be activated or not. Activation functions make backpropagation possible.

##### Binary Step Activation

The binary activation is straight forward: it simply says yes or no for a single class step function, either activating the neuron or leaving it to zero. In practice, this type of activation is not practical as there are likely to be multiple classes in the prediction problem than just a single class.

##### Linear Activation

Linear activation works similar to a linear model in which the activation is proportional to the input. In the case where the input is “x”, the linear activation input is simply “a\*x” as in a linear model. The derivative of the linear function is constant, it does not matter on the input value of x, and is simply “a”. In backpropagation, the gradient remains the same. The end result of this is that the network does not improve the error.

##### Sigmoid Activation

Sigmoid activation is widely used, and follows the form: f(x) = 1 / (1 + e^x). It is a smooth function and continuous, which means the advantage over the linear activation and binary activation is that it is non-linear. With multiple neurons, the output is non-linear. Essentially, small changes in “x” lead to large changes in the value of “Y”, pushing “Y” to the extremes, ideal for classifying images. The gradient is also smooth and dependent on “x” so that during backpropagation Sigmoid can easily be used and the error can be backpropagated and the weights can be updated accordingly.

Problems with the sigmoid function are that the gradients are very small at the extreme values of “x” so that the as gradient approaches zero the network does not learn as well. The second problem with sigmoid is that values only range from 0 to 1, so that values are all positive and not symmetric around the origin. An attempt to solve this problem is the tanh function (not described here).

##### ReLu Activation

ReLu activation (Rectified Linear Unit) is the most widely used activation function, defined as f(x) = max(0 , x). ReLu is non-linear allowing backpropagation to reduce the error rate and have multiple layers of neurons being activated by the function. The main advantage of ReLu over other activation functions is that it does not activate all of the neurons at the same time, so that if an input is negative then the function will convert it to zero and not be activated. It is thus more efficient and easier to use for computations.

The downside of ReLu is that the gradients tend to move towards zero, so that weights are not always updated during backpropagation. This creates “dead” neurons that are never activated. Attempts to solve this issue has lead to the Leaky ReLu activation function (not described here) and the Parameterized ReLu function (not described here). ReLu should only be used in the hidden layers of a neural network.

##### SoftMax Activation

SoftMax is a type of sigmoid function, but handy for classification problems. One of the major downsides of Sigmoid is that it could only handle two classes. SoftMax has the ability to handle multiple classes by effectively giving the probability of the input being in any particular class. Softmax is ideally used in the output later of the classifier neural network, where the goal is to attain the probability of the input as being in each class.

### Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are most effective at sequential prediction problems. There are three common examples of sequence problems: One-to-Many, Many-to-One, and Many-to-Many. One-to-Many problems are those which taken an input observation and map to a sequence with multiple steps as an output. Many-to-One problems are sequences with multiple steps as an input and a single class or quantity resulting as an output.

An example of a One-to-Many sequential problem is image captioning, where a single image is provided and a string (or sequence) of words are generated as an output. Image captioning is used in the simple case of a caption but also can be used to describe the characteristics of an image, which itself can be used as an input into a Many-to-Many sequential problem. A Many-to-One sequential problem uses a sequence of events or observations and predicts the next step.

Many-to-Many sequential problems are a sequence of multiple steps used as an input mapped to a sequence with multiple steps as an output. These problems are often referred to as sequence-to-sequence problems, or seq2seq. The number of sequence inputs do not have to match the number of sequence outputs. Many-to-Many sequential problems can be applied to video classification, which is commonly referred to Synced Many-to-Many sequential problems.

RNNs can be thought of as a neural network that leverages additional loops in the architecture. Unlike MLPs, for any given layer a neuron may pass laterally so that data can move forward in a layer or sideways in a layer, while the output of the network maybe be used as input back into the network as feedback for the next input vector. These recurrent connections allow the neural network to learn a broader abstraction from an input compared to the raw input vector data.

By its definition, backpropagation in RNNs do not work. As such, Backpropagation Through Time (BPTT) was established to address this issue. BPTT essentially restructures the network where a neuron has a connection to itself, so that a connection of A 🡪 A is then represented as a connection of A 🡪B with B having the exact same weight values as A. The effect of this modification is a representation of an RNN from a cyclic graph to an acyclic graph similar to a feedforward neural network so that backpropagation can be applied.

RNN models are traditionally difficult train, but most effective for text data, speech data, classification problems, regression problems, and generative models. They do not work well with tabular datasets or image data, and are surprisingly poor for time series forecasting and can be outperformed by simpler MLP applications.

#### Long-Short Term Memory (LSTM or STM)

When backpropagation is used in an RNN, gradients calculated in order to update weights tend to become unstable and become very large numbers – this is known as an exploding gradient problem – or very small numbers – known as a vanishing gradient problem. These very large or very small numbers are used to update weights in a network, causing the network to become unstable or even unreliable.

Long-Short Term Memory (LSTM) networks are simply RNNs trained using Backpropagation Through Time (BPTT) and overcomes the vanishing gradient problem. LSTMs use memory blocks connected into layers instead of neurons. The memory blocks are used to created stacked RNNs. These block also have a memory for recent sequences, and operate with the ability to make a change of state for information following through the unit based on conditions. These “gates” (Gate Restricted Units – GRUs) allow sigmoid activation functions to control whether or not the conditions are triggered, essentially deciding what information (1) gets discarded from the unit, (2) updates the input memory state, or (3) is output based on the input and the memory of the unit.

### Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNN) were primarily designed to map image data to an output variable. Their original use case was handwritten digit recognition and object recognition. As such, they are very common for use on any problem involving image data as an input, as well as computer vision and natural language processing (NLP) tasks.

A key difference between a CNN and a standard MLP model for image input data is that a traditional feedforward neural network is that a feedforward network requires a vectorized form of a matrix of pixels (an image) to be consistent (for example, an image size of 32x32 requires all other training images to be 32x32), whereas a CNN preserves the spatial relationships between pixels by learning the feature representations of smaller squares of the input data. Features are learned throughout the image in a square by square basis across the entire image, allowing objects in the images (or the image itself) to be rotated or represented differently to be still detectable in a network. By this definition, CNNs require few parameters (weights) than an MLP models and can generalize features from inputs.

There are three different types of layers in a Convolutional Neural Network: Convolutional Layers, Pooling Layers, and Fully-Connected Layers. A convolutional layer consists of filters and feature maps, with the filter acting as a neuron of the layer (with an input weight and output value) and the feature map acting as the output of one filter applied to the previous layer. A pooling layer follows a convolutional layer (or a sequence of one or more convolutional layers) and act as a compression layer or a layer to generalize feature representations to reduce overfitting of the training data by the model. The fully connected layer act as a typical feedforward neural network layer with a non-linear activation function or softmax activation function or ReLu activation function to output probabilities of class predictions. These layers are used at the end of the network – after feature extraction and consolidation from the Convolutional Layer and Pooling Layer.

Some notable features of CNNs are stride, filter, padding, pooling, and dropout. Stride is the number of pixels that the square feature mapping moves across images. The default stride is 1, but for larger images a stride of 2 or more could be used. Filters are feature detectors, generally fewer filters are used in the input layers and an increasing number of filters used in deeper layers of the CNN. Padding can be useful when image sizes need to be standardized (for example, some smaller images cannot be stretched as it would reduce the features in the mapping), but it is common to set padding to zero (called zero padding) for input data. Pooling (similar to the pooling layer) is used as a generalization process to reduce overfitting, almost always set to discard 75% of the activations from the output of the previous layer. Dropout should be used between fully connected layers to prevent overfitting, sometimes used after pooling layers.

CNNs are most effectively used on Image data, classification prediction problems, and regression prediction problems. They can be used on text data, time series data, sequence data, and (of course) image data. Any type of spatial relationship data is an effective problem to be solved for with a CNN.

### Generative Adversarial Networks (GAN)

Where CNNs are considered discriminative, a GAN is (by name and definition) generative. A discriminate algorithm is one which is used to classify input data – given features of an image, the algorithm attempts to predict a label or category for which the image belongs. Generative algorithms, on the other hand, attempt to model the distribution of the class – given the image, the algorithm attempts to figure out how the *get* the label or category.

A GAN is made up of two neural networks: a generator and a discriminator. The generator creates (or generates) new data while the discriminator evaluates the authenticity of the data. As the generator passes new images to the discriminator, the goal of the generator is to create new images that pass the authenticity barrier of the discriminator. This is performed as such: the generator takes random numbers and returns an image which is passed to the discriminator along with a stream of images from the actual dataset, the discriminator returns the probabilities of each image, and the generator then passes new images to the discriminator.

GANs are incredibly powerful tools, and have potential far greater than image replication. Especially in the art world, the power of GANs has already been proven with multiple cases of high value auction sales, not to mention an explosion in digital art. Unfortunately, GAN development has not yet become commonplace to solve my type of problem, though the possibilities for it are clear and obvious: a generator could be used to improve the training abilities of a discriminator, which would then be used as the classifier.

### Neural Network Evaluation

From the research discussed above, it is fairly clear that the best option is a Convoluted Neural Network. It is not only developed with my problem in mind, and there are several options available for experimenting with CNNs. Unfortunately, the GAN models will likely not readily solve my problem despite the potential.

## Transfer Learning and Evolution of Neural Network Architectures

From the perspective of practicing data scientist with limited resources, transfer learning is an incredible step towards making deep learning accessible and practical. When creating a deep learning model from scratch, everything the model “learns” comes from the data it is given. With poor data or poor modelling (due to any number of reasons, typically blameless), the expectations are likely to be as anticipated: poor.

Transfer learning has the ability to pull most of the weight in terms of creating an effective deep learning model. The concept is straight forward: a model is created to learn a task (typically an incredibly specific task), tuned for performance, and made available for others to use. The users (or consumers) do not use the new model directly, but use the “knowledge” learned by the model.

One of the fields most impacted by the development of transfer knowledge is that of computer vision. The benefits of transfer learning are obvious, training a model from scratch is time-consuming, resource intensive, and simply unnecessary. Rather than having multiple groups of individuals create multiple versions of a model to perform the same task with varying degrees of accuracy tuned for a variety of features, these models could be turned into a single highly optimized model with the abilities of each of these individual models. This new model can then be used by others hoping to achieve the same goal of distinguishing faces, identifying a variety of horticulture, or even the daunting task of distinguishing cats from dogs.

Transfer learning also gives the scientist the ability to modify models by adding layers, retraining, or hyper parameter tune. With thanks to a number of companies and individuals who have painstakingly put in their time, effort, and money to create optimized models, Sidney’s Catalog has the ability to use a number of pre-trained models for the purposes of image recognition and categorization.

Not every model described here will be tested.

### ImageNet & AlexNet

Originally published in 2009, the ImageNet dataset consisted of 3.2 million labelled images across 5,247 categories and 12 subtrees (i.e., “mammal”, “vehicle”, “geoform”). The premise of ImageNet was to highlight that data could be a catalyst for deep learning, just as important as algorithms. The simple nature of the dataset was to enable researchers and developers to use clean annotated image data with pre-set labels and bounding boxes for the development of better neural networks. The quality of this dataset is an essential ingredient.

After releasing the dataset, the ImageNet team quickly collaborated with PASCAL VOC, a well-known image recognition competition in Europe, on competition using ImageNet. The PASCAL challenge only represented 20 classes image images, far fewer than ImageNet’s 1,000 class dataset, which marked a significant shift in deep learning from algorithmic based advances to dataset quality and breadth.

The real power of ImageNet was shown in a demonstration in 2012 in which a team from the University of Toronto submitted a deep neural network architecture called AlexNet, beating the previous models and improving the performance of the next best model by 41%, with an error rate of 15.3% compared to 26.2% achieved by the second best entry. Today, the error rate in ImageNet is thought to be less than 2%.

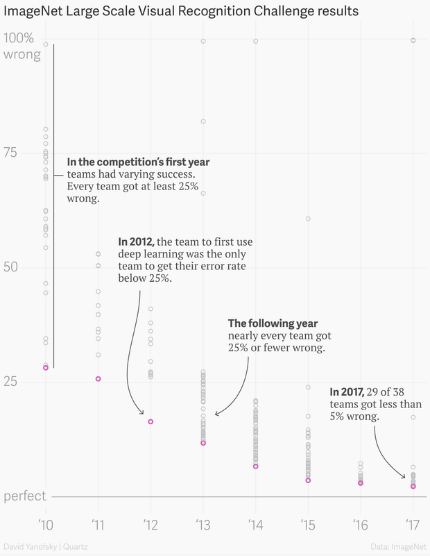


Figure 1 - Improvement of Neural Networks trained using ImageNet

The potential of ImageNet also enabled the concept of transfer learning to substantially change the field of image recognition. By using the weights of a state-of-the-art model trained on ImageNet, new models could be initialized and significantly improve performance of other image recognition problems. These pretrained networks have been used for a variety of tasks such as object recognition, semantic segmentation, human pose estimation, and video recognition, not only drastically improving the performance of models but also significantly reducing the cost of developing new ones.

AlexNet is a simple architecture with five consecutive Convolutional filters, max-pool layers, and three fully-connected layers.

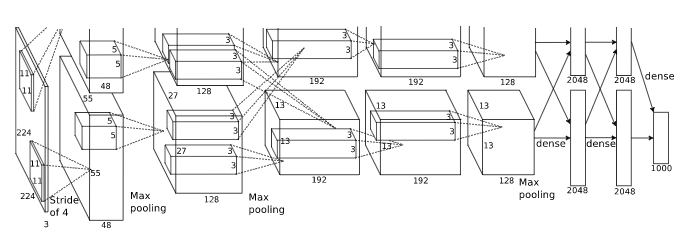


Figure 2 - AlexNet architecture

### VGG16

In 2014, VGG16 model was developed and composed of sixteen convolutional layers, multiple max-pool layers, and three fully-connected layers. One notable aspect of the VGG16 model is the use of ReLu activation on the convolutional layers, creating nonlinear transformations for more complex patterns. Another aspect is that the convolutional layers are only 3x3, much lower than the 11x11 used by AlexNet. The decreased sizes of the filter layers are able to recognize the same patters as the larger layers, but required far fewer parameters for training. The error rate of the VGG16 model reached 7.3% error rate in the 2014 ImageNet challenge, drastically beating the 2012 AlexNet model.



Figure 3 - VGG16 architecture

### GoogLeNet

Also in 2014 the concept of an Inception network was developed and used as part of the model known as GoogLeNet. While convolutional layers use linear transformations with nonlinear activation functions, Inception V1 proved that training multiple convolutional layers simultaneously and stacking their feature maps linked with a multi-layer perception could also produce a nonlinear transformation. Essentially, the design allows a model to increase in both depth and width while maintaining a constant computational power. GoogLeNet is a 22 layer network of inception “modules” – a total of 50 convolutional layers – with each module being comprised of a 1x1, 3x3, and 5v5 convolutional layer and 3x3 max-pool layer to obtain different types of patterns and increase sparsity. The feature maps are then concatenated and analyzed by the next inception module.

GoogLeNet achieved a 6.7% error rate, beating VGG16 and dramatically reducing the size of the model. VGG16 required 490 MB due to the large size of the three fully-connected layers, whereas GoogLeNet only required 55MB.

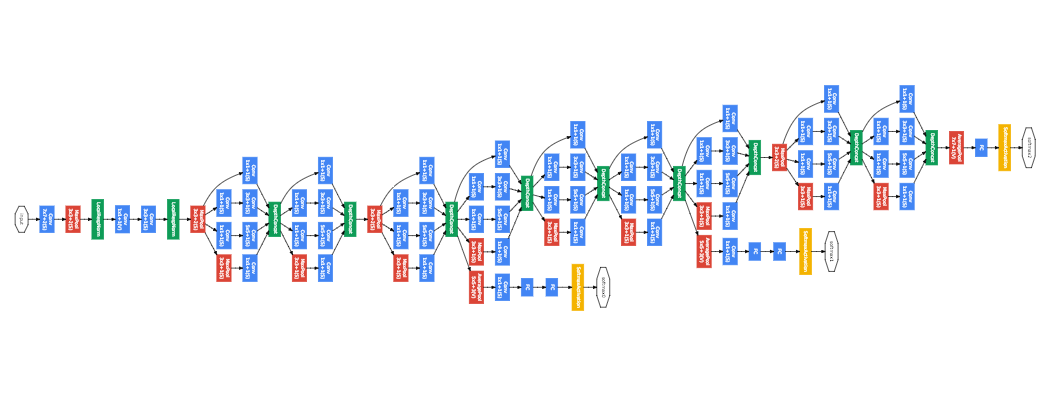


Figure 4 - GoogLeNet architecture

### ResNet

In 2015, another model yet again improved upon the learnings of the previous years’ entry and saw the introduction of a residual network, using what was then called Residual Learning. With increasing depth in CNNs, a common problem that arises is an increasing error rate due to the difficulties of training and optimizing on more and more layers in the model. Residual learning was created to connect the output of one or many convolutional layers and the original input with an identity map. The model then attempts to learn a residual function while keeping most of the original information intact, producing only slight changes. As a consequence, patterns from the original input image can be learned in deeper layers without lost most of the qualities due to further abstraction of having many deep layers.

The model for which Residual Learning was showcased in was known as ResNet, a 152 convolutional layer neural network with 3x3 filters using residual learning by block of two layers. Resnet won the 2015 ImageNet challenge with an error rate of 3.57%.

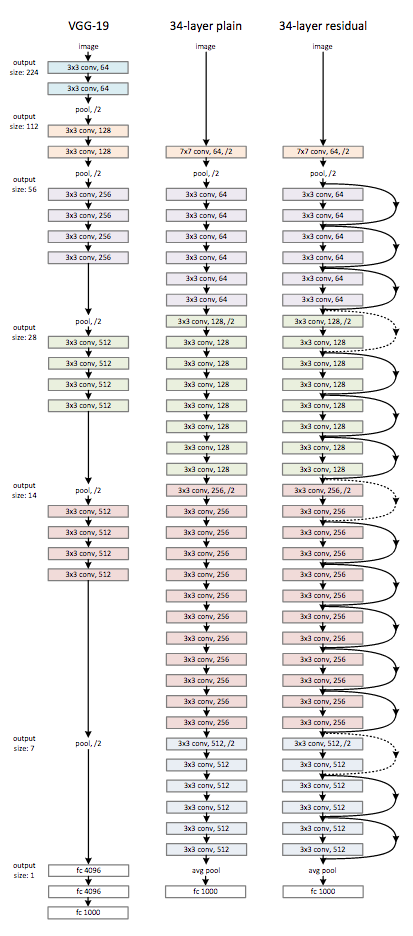


Figure 5 - ResNet Architecture compared to VGG19 and a 34-layer CNN

### DenseNet

Where ResNet uses a summation of outputs (the input for any layer is the summation of outputs from the previous layers), DenseNet was proposed to use a concatenation of outputs. This allows features of the original image to build upon one another with each layer inside of a dense block of layers. The dense layers are then passed through transitions layers containing convolution and pooling layers to reduce the height and width of the image while leaving the feature dimensions the same.

DenseNet achieved an error rate of 3.46% on ImageNet.

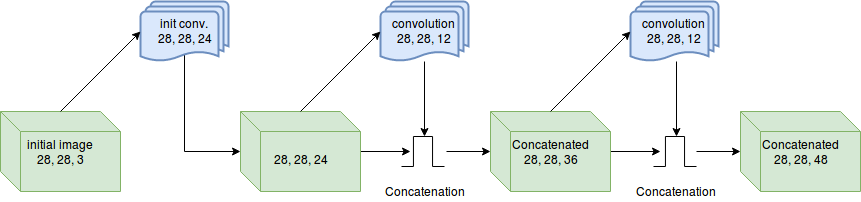


Figure 6 - A single Dense Block within DenseNet

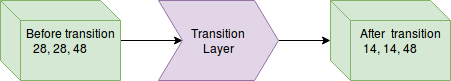


Figure 7 - A single transition layer in DenseNet

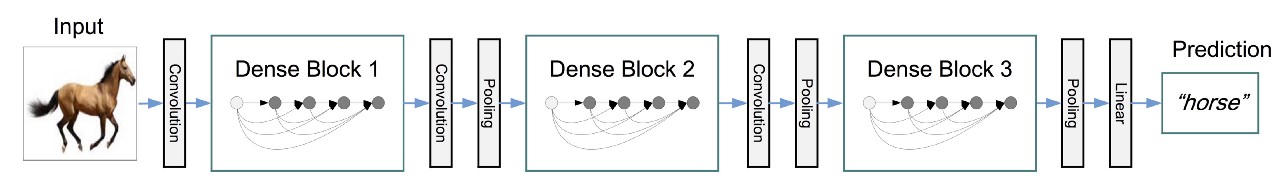


Figure 8 - DenseNet with three Dense Blocks

### Inception v3

At the end of 2015, leveraging the concept of the inception network, Inception v3 was introduced. It is still a very widely-used model. It is made up with a number of building blocks including convolutions, pooling, concatenations, dropouts, and fully connected layers. Inception v3 achieved 78.1% on the ImageNet dataset.

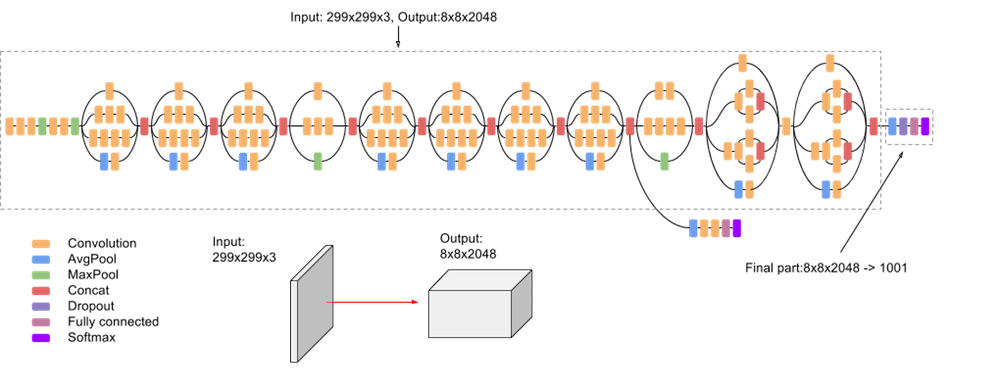


Figure 9 - Inception v3 architecture

### MobileNet

Released in 2017, MobileNet is a CNN optimized for TensorFlow in a manner that is “mobile-first”: designed to maximize accuracy using restricted resources for an on-device or embedded application. As described by the Google AI blogpost announcing the module, “MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of a variety of use cases.” MobileNet v2 achieved an accuracy of 70.4% (Top 5) on the ImageNet dataset.

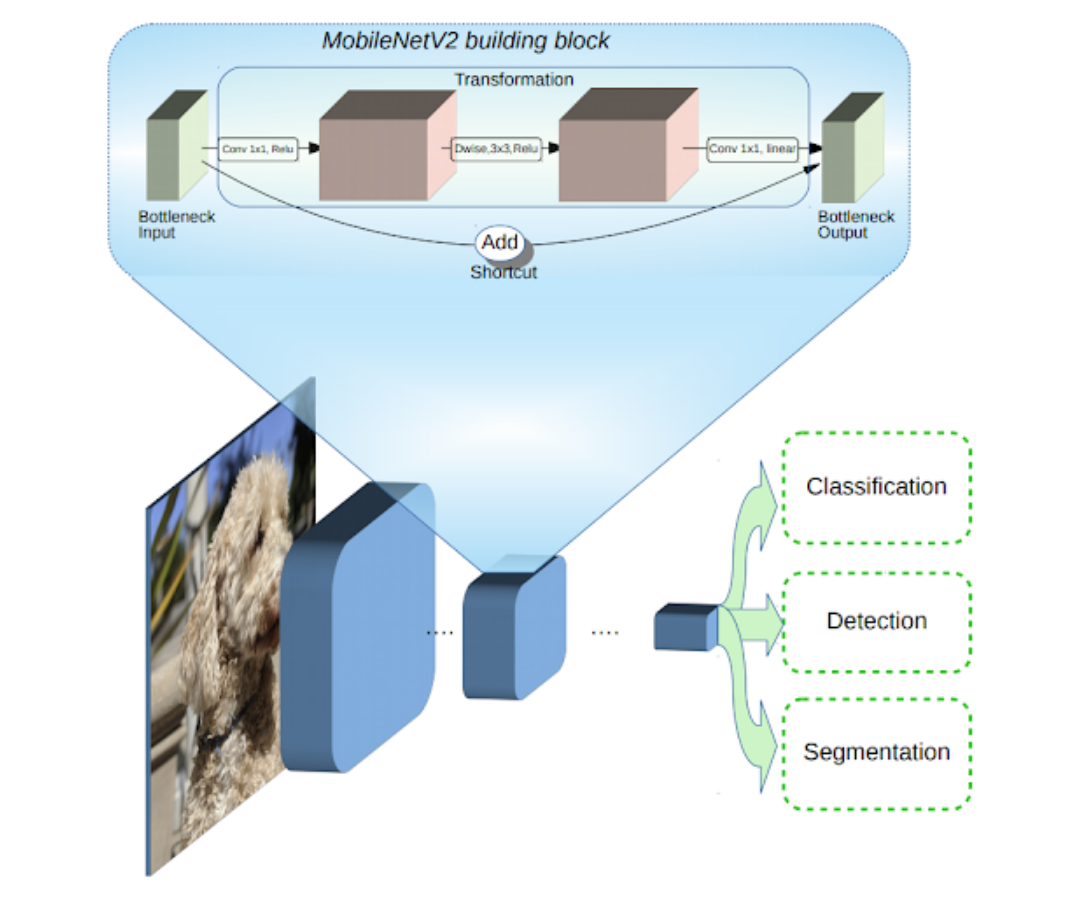


Figure 10 - MobileNet v2 architecture

### NASNet

NASNet was released by researchers at GoogleBrain in 2017, using an architecture concept known as Neural Architecture Search (NAS). Essentially, the NAS is used as a single cell in a Recurrent Neural Network (RNN), in which the objective of the cell is to learn the best sequence of operations to optimize its own architecture. A CNN trained with NAS achieved an error rate of 3.8% on the ImageNet datset.

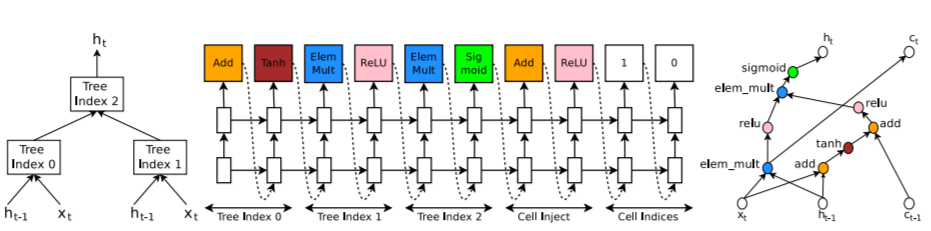


Figure 11 - Example of a NASNet CNN architecture

### PNASNet

PNASNet-5 was announced in 2017, reusing many of the techniques from NASNet. This model is known as a Progressive Neural Architecture Search (PNAS) model, reducing the search space of the NAS model while achieving competitive results with the best performing models. Essentially, PNASNet is equivalent to the NASNet without using reinforcement learning to create a progressive search. There are two variants of the PSANet: a mobile optimized CNN (image input size restricted to 224x224) and a ‘Large’ implementation (331x331 images). The mobile setting achieved an accuracy of 91.9% on the ImageNet dataset while the Large implementation achieved an accuracy of 96.2% accuracy. Of significant importance is that the PNASNet model reduced the training time of NASNet by a factor of 5.

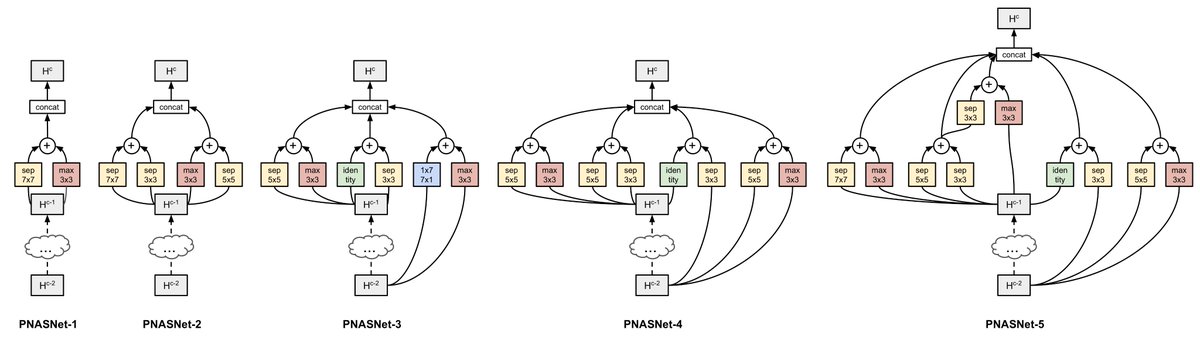


Figure 12 - PNASNet architecture evolution

### AmoebaNet

AmoebaNet is another step utilizing the NASNet search space and the current state of the art network architecture. The difference between AmoebaNet and NASNet is that AmeobaNet uses a genetic algorithm (referred to as ‘Evolution’) rather than the Reinforcement Learning algorithm. Using Evolution reduced the computational cost associated with NASNet and achieved an accuracy of 97.87% on the ImageNet dataset.

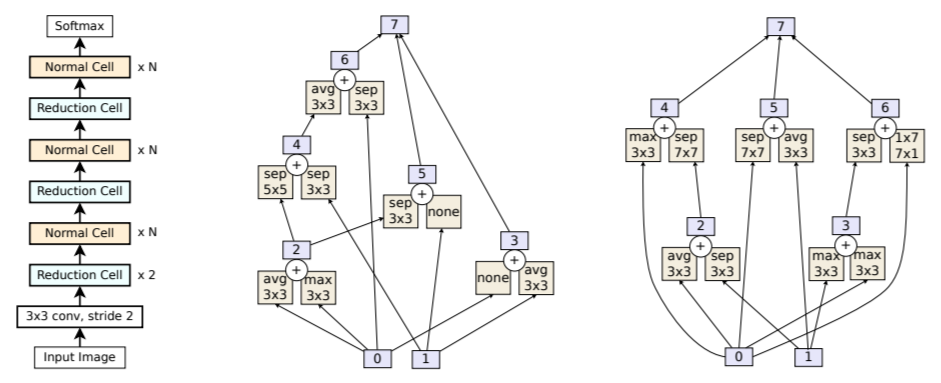


Figure 13 - AmoebaNet-A architecture

## Open-Source Deep Learning Libraries

Deep Learning libraries are essential to develop high quality neural networks, as the limitation of Python’s default libraries and scikit-learn are they that don’t support machine learning algorithms. Deep Learning libraries were developed specifically for this use.

### TensorFlow

TensorFlow is a open source software library for machine learning, which was released by Google in 2015 and has quickly become one of the most popular machine learning libraries being used by researchers and practitioners all over the world. TensorFlow as based on Theano, a Python library and optimizing compiler for manipulating and evaluating mathematical expressions involving multi-dimensional arrays. It is compiled to run efficiently on either CPUs or GPUs. TensorFlow creates static graphs, where the entire computation graph of the model is defined before running the neural network.

A key advantage that TensorFlow has over other machine learning libraries is that there is a large active community available, meaning that resources and troubleshooting is more plentiful. This can prove an essential resource to practicing developers, especially those of beginner or intermediary experience.

### PyTorch

Pytorch was initially developed by Facebook, based on Torch, but the number of companies and universities that contributed to its development are far and wide (from Stanford University to Twitter.) Because it was based on Torch, PyTorch is fast and efficient for GPUs rather than CPUs.

PyTorch operates using a dynamic graph, so that the developer can define and manipulate the graph as it is run. The main benefit of this type of graph is in an RNN using variable length inputs. Another key benefit is that networks are modular, so each part can be implemented separately and debugged individually. Compared with TensorFlow, PyTorch has a much smaller development community though it is widely regarded as easier to use for those familiar with Python than the high learning curve associated with TensorFlow. It is generally associated with building rapid prototypes.

### Keras

Keras, unlike TensorFlow and Pytorch is a front-end library; it is capable of running on top of other Machine Learning and Deep Learning libraries. Advantages of Keras is that it was written in Python, which is the language being used throughout this project, and it is designed for fast experimentation. A major disadvantage is the small size of Keras’ development community, and official documentation is known to be lacking.

### Library Evaluations

Overall, it seems that TensorFlow is the preferred method for me due to the size of its developer community, despite it’s presumed high learning curve. Another key advantage is the fact that is optimized (or can be optimized) for CPU usage. My personal laptop does not have a GPU, and the cost of GPU cores on a serverless service (such as Amazon Web Services or Google Cloud Platform) does not justify the use of other services. PyTorch is mainly used on Linux and MacOS, so it was automatically ruled out for my Windows 10 set-up. Keras does not solve any of the disadvantages of TensorFlow, and its lack of resources and documentation make it a barrier to use without much advantage. TensorFlow checks off more of my needs than the others, though Keras can also prove to be an effective tool in my deep learning classification problem.

# Data Collection: In Practice

Data collection was surprisingly difficult. Each webservice had its own methods of storing images, time-out limitations, and result handling. For example, Bing Images would only display 35 image results before requiring a click to move to the next page. Google Images displayed 100 images, then required a mouse scroll or click to view the next set of images on the same page.

The python package imgdl did not work on my Windows laptop. After several hours of tweaking deprecated packages, making changes for Windows to run properly, and installing packages required for the program to run, I was able to pull a long list of URLs for the images but with no means of extracting the images from the URLs.

The code for data collection can be found [here](https://github.com/chrisgmartin/DATA698/blob/master/Webscraping%20Images.ipynb).

## Description of the Data Sets

Images were resized for quicker processing and consistency to 299x299. They were then split into three sets: a training set, test set, and validation set. In total there were 1,936 images collected and resized into this format. Duplicate images are bound to exist, and the variation in quality of the images are extremely wide from poor quality to perfect quality. The breakdown of the classes and number of images per class are as follows:

* Champagne Flute: 358 images
* Martini Glass: 335 images
* Red Wine Glass: 328 images
* Rummer Glass: 238 images
* Sherry Glass: 222 images
* Snifters: 277 images
* White Wine Glass: 228 images

Duplicates are due to the timing of when images were pulled and the sources of the images; for example, a Yahoo Images search may show the same images as a Google Images search, and there is a possibility that images were pulled from the same source multiple times as the code was being tested. The quality of the images are random as they were directly pulled from the search results and not vetted for quality; some images show the full glass or multiple glasses or even a cropped portion of one or more glasses, shadows may exist, and images may be taken by professionals or amateurs. The number of images is also not consistent which may likely skew results, though the overall impact is likely to be minimal as I expect image quality to be a much more import factor.

The test set consisted of 20 randomly selected images per class from the total resized image set. The validation set consisted of 25 randomly selected images per class from the total resized image set. The training set consisted of the remainder of images per image class. The size of the test and validation set were determined with the knowledge that too many images would take away from the training set (which likely requires a higher number of images) and too few images would lead to unrealistic accuracy percentages. The test set is roughly 10% of the total per each class. The validation set was also selected randomly, with 25 images per class.

# Convolutional Neural Network (CNN) Development: In Practice

As discussed, the type of neural networks that I will test for this project will be Convolutional Neural Networks. I will use a variety of models from simple self-developed neural networks to those using transfer learning. For the purposes of illustration and as an important piece of the overall evaluation, the implementation of the neural networks are discussed, followed by an overall evaluation and summary of the results.

## Self-Developed Networks

As part of the process to evaluate the performance of various models, self-developed networks play a critical role. By developing some networks by hand, I can gain a deeper understanding of the various deep learning libraries and have a set of control models to see if there is any benefit to the pretrained networks.

### Simple 2 Layer CNN with ReLU and Simple 7 Layer CNN with Softmax and ReLu

The first model was a simple neural network devised using TensorFlow, a two layer CNN with ReLu activation. In practice, TensorFlow has a high learning curve, as there are nuances that are difficult to get an understanding of on first use. However I noticed that once the concepts were understood, implementing a CNN was fairly straight forward with some surprisingly difficult problems: using my own images as a dataset and knowing the options available.

The key difficulty was importing images; most of the official examples were based on pre-defined dataset, where the format was set and perfectly organized. Labels were pre-created, image sizes were standardized, and there was enough data to iterate over for high accuracy. In addition, TensorFlow requires specific formats and the resources available to figure out how to import a local dataset are limited and not very useful for beginner-to-intermediate users. Luckily there are some resources available, requiring substantial tweaking and a high prerequisite knowledge of python. After spending a substantial amount of time importing the data, the CNN was fairly straight forward to implement.

The second issue with TensorFlow in practice is knowing what options are available. This requires a deep understanding of TensorFlow and time to dive into the official documentation and tutorials. Optimization is certainly part of this challenge; as I am using a Windows 10 laptop without a GPU, many third party developers offer examples using MacOS or Linux or Windows with GPUs. Having to tweak the configuration to optimize for CPU processing is simple, but a task that is not often discussed.

## Pre-Trained Networks using Tensorflow

Resources for using pre-trained networks with TensorFlow is both plentiful and overwhelming, and at the same time confusing and difficult to use. Most of the training comes from tutorials using the MNIST or CIFAR-10 datasets (commonly used datasets for learning the intricacies of CNNs, and establishing a based-line for the quality of a neural network with a curated dataset). These tutorials are great in learning how to use TensorFlow, but they are not great when attempting to use own images within TensorFlow.

As seen in the previous two CNNs that were self-devised (Simple 2 Layer and Simple 7 Layer), there are several ways of importing data for use. Some methods are time consuming and possibly not worth the effort without pre-requisite knowledge of Python scripting or TensorFlow (e.g., creating a .txt or .csv listing the filenames and locations and labels, creating a .tfrecord of all images, etc.) Luckily, there happens to be an incredibly simple method, and it unfortunately took a very long time to find hidden within TensorFlow’s official documentation.

Using the TensorFlow retrain script (saved in this project’s GitHub folder), a model can be quickly retrained using any set of images and any class. Not only does this one single script allow the use of local images it also enables: an incredibly useful set of features such as graphs and logging, allows the selection of other models to train with, a large number of parameters to tweak such as input image size and the ability to use the retrained model in production.

# Evaluation of Models

## Simple 2 Layer CNN on TensorFlow with ReLU

The first model tested was a simple 2 layer CNN built on top of TensorFlow using ReLu activation. The model included the use of importing image data from my local disk, and feeding it into a training and evaluation node.

The model first imports the images from the local hard-drive. The import uses the local directory’s folder structure (names of the folders) to create the various labels for each classification. After importing the images into the test and training sets (evaluation was not used in this model – the evaluation image set was not included in either test or training set), the model is created using the ConvNet parameters and image reuse (images were not reused for training purposes but were able to be reused for testing purposes with a dropout rate of 75%, meaning 25% of the images would not be re-used in the test).

The model itself is two convolutional layers, each using ReLu activation and max pooling. After layering they’re flattened from a 2-D vector to a 1-D fully connected layer, where dropout is applied and the layer outputted for each step. The total number of steps were limited to 100 due to low computational bandwidth on my local machine.

The model was successful in that it ran successfully, but in terms of evaluation the results were poor. The model showed a successful test run with an accuracy of 10.7% and a loss of 0.0558. This is worse performance than the random selection probability of 14.3% accuracy.

The code for this CNN can be found [here](https://github.com/chrisgmartin/DATA698/blob/master/SimpleCNN_Tensorflow_2LayersReLu.ipynb).

## Simple 7 Layer CNN on TensorFlow with Softmax and ReLU

The second model tested was a 7 layer TensorFlow CNN with Softmax normalization and ReLu activation. The CNN is different from the previous (Simple 2 Layer) in many ways.

I used a different method of importing the dataset to TensorFlow, following a (an unfortunately) much more difficult to replicate method. The image importation method applied in this model also used a folder structure for labeling (where the folder name determined the class), but all test and training images were contained in a single folder whereas the previous method required physical separation. The code for importation spanned across multiple python files, and required the use of a terminal to run.

The first block of code is the build\_image\_data.py file, which takes the images (either JPG or JPEG) from their folders (using the folder name as labels) and shards the data into TFRecord files, which are TensorFlow record files used for importing data into TensorFlow. The output files may need to be manually appended with “.tfrecord” as there is a bug that does not append this file type automatically. The second block of code is the read\_tfrecord\_data. This code block reads the previously created .tfrecord files and resizes them to a fixed unit (in this case an image height and width of 299 – 299x299). The code will also drop the new images into the folder ‘./resized\_image/’ and append the names of the files to “class”+ class number + “Index” + index number. Finally, the resized images and the .tfrecord files are consumed by the CNN model for training and evaluation.

The model itself is similar to the first model (Simple 2 Layer) in that the convolutional layers are all ReLu activated, but the final layer is normalized by a Softmax function. Softmax essentially flattens the 7-D vector into a 1-D vector in the range of (0, 1), reducing the influence of extreme values and outliers (which is highly likely given our widely varying dataset). The CNN is also trained in a separate stage for evaluation in an attempt to reduce loss prior to evaluation. Unfortunately, this model did not perform so well when reducing the loss, maintaining a loss of 0.24000001 for all steps 0 to 100,000.

The final accuracy of the model was incredibly poor: 10.00%, less than the previous 2 layer model. There are a number of factors that lead to such a poor performance, and I expect hyperparameter tuning would substantially improve the performance, but I expect that the main driver for this is the quality of the dataset. Given that these two models were trained using the same poor quality data with different neural network architecture and achieved relatively similar results, the common factor was the dataset. My takeaway from these results is that a data quality is hugely important, and I expect that tuning these two models would only lead to a marginal gain in accuracy, far less than 50% (an educated guess) and far less than a usable model in practice.

The code for build\_image\_data can be found [here](https://github.com/chrisgmartin/DATA698/blob/master/build_image_data.py), the code for read\_tfrecord\_data can be found [here](https://github.com/chrisgmartin/DATA698/blob/master/read_tfrecord_data.py), and the code of this CNN can be found [here](https://github.com/chrisgmartin/DATA698/blob/master/SimpleCNN_Tensorflow_7LayersSoftmaxReLu.ipynb).

## Inception v3

The Inception v3 model was very quickly implmeneted using TensorFlow’s retrain.py. Using the parameters to simple call the local image directory and the module location, TensorFlow takes care of importing the images, training the model, and evaluating the model against a validation set. In the case of Inception v3, TensorFlow retrains the images by default so that the module address (<https://tfhub.dev/google/imagenet/inception_v3/feature_vector/1>) does not need to be included in the command line parameters. By default, TensorFlow uses 4,000 steps to retrain all models.

The final output of the accuracy for the Inception v3 model was 78.2% accuracy on the validation set, and 92% on training set. Clearly, the pretrained models perform significantly better than the self-developed models and it is no surprise.

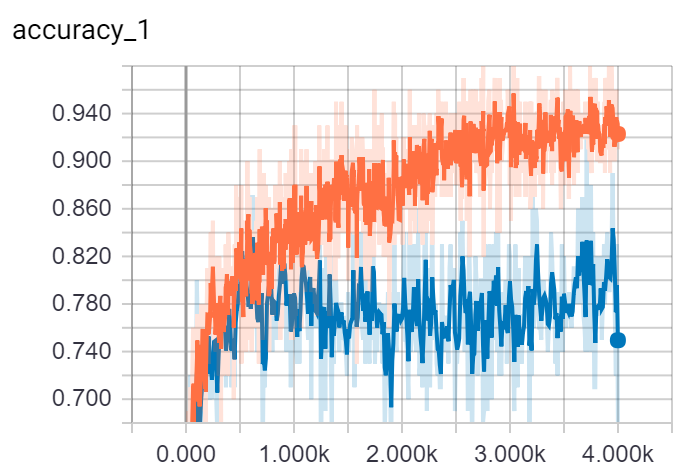


Figure 14 - Accuracy of Inception v3

## ResNet

Using TensorFlow’s retrain script, ResNet v2 was very quick to implement (module: <https://tfhub.dev/google/imagenet/resnet_v2_50/feature_vector/1>). The final output achieved an accuracy of 69% on the validation set and 99% on the training dataset.

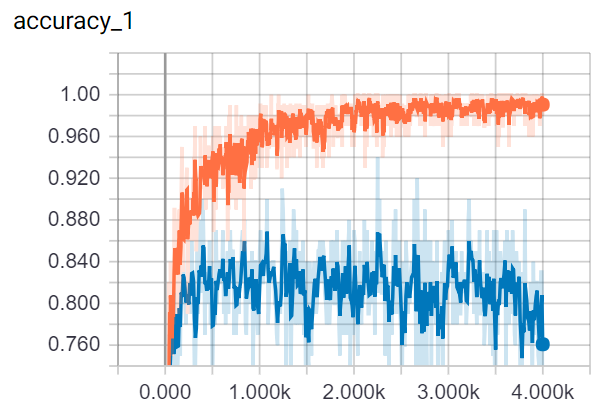


Figure 15 - Accuracy of ResNet v2

## MobileNet

MobileNet v2 followed suit, calling the module in the command line (<https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/feature_vector/2>) and achieving a final output accuracy of 84% on the validation set and 96% on the training dataset.

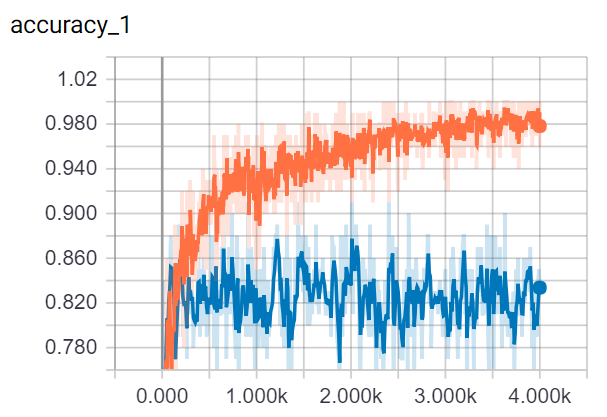


Figure 16 - Accuracy of MobileNet v2

## NASNet

Using the NASNet module (<https://tfhub.dev/google/imagenet/nasnet_large/classification/1>) a final output accuracy on the validation set of 84.21% was achieved, as well as an accuracy of 90.19% on the training dataset.

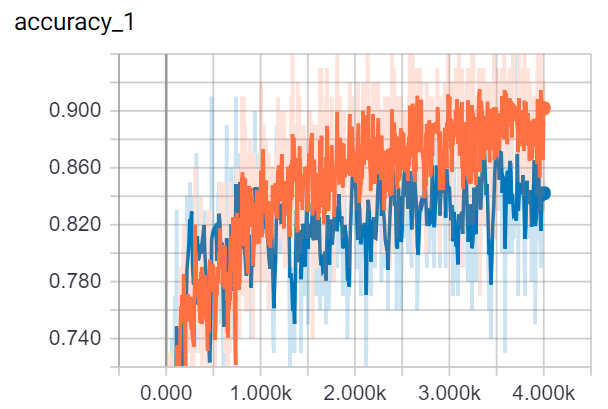


Figure 17 - Accuracy of NASNet

## PNASNet-5

PNASNet-5 achieved a final output accuracy on the validation set of 76% with an accuracy of 92% on the training dataset using the module <https://tfhub.dev/google/imagenet/pnasnet_large/feature_vector/2>.

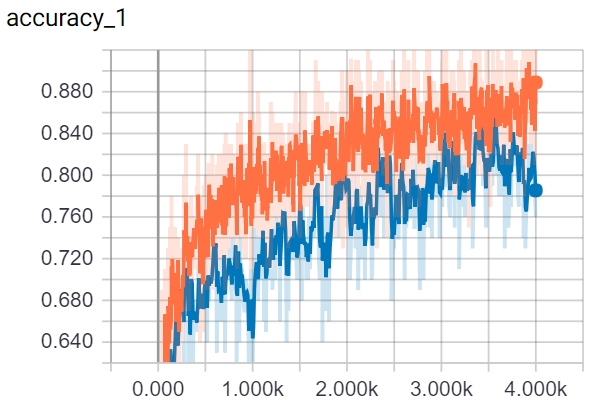


Figure 18 - Accuracy of PNASNet-5

## AmoebaNet

Unfortunately, AmoebaNet was not able to be implemented due to a bug within TensorFlow’s directory. It is included here for completeness.

Module: https://tfhub.dev/google/imagenet/amoebanet\_a\_n18\_f448/classification/1

# Summary of Results

TensorFlow has really blown me away by its simplicity. Its biggest downside is, as expected, its documentation but once the retrain module was discovered the ability to automatically import and train CNNs on any image dataset was incredibly simple: from the command line simply run the retrain.py script and declare the parameters. In addition the interactive graphs, plethora of models / optimization tools, and ability to save and reuse the newly trained models are absolutely value added on top.

Using TensorFlow, Sidney’s Catalog can quite easily be built with any model, and could even be improved on the fly. I foresee the ability to have several models being constantly retrained with new data, and the best performing model per retrain used in the consumer facing web interface. All of this can be done automatically with just a few lines of code.

The best performing model on the validation set v2 (a more useful metric than training set accuracy as it is a random ‘blind’ test – image classification are unknown at the time of testing) was the NASNet model. The second best performing model (by a very small margin) was MobileNet. In terms of practical model use, MobileNet has two significant advantages over the others: it is by far the quickest model (in processing time and compute requirements) to retrain and also has the lowest size requirement allowing the use of the model on web and mobile (if desired).

The model of choice for Sidney’s Catalog is clear: MobileNet v2. The reasoning is that this model not only achieved the second highest marks on the accuracy test (losing by a very small margin) it also ranked highest for the other criteria used for evaluation: shortest runtime and lowest cost (compute power, price – free, and processing requirements). Even though NASNet achieved a higher accuracy, it could not compete in terms of size or runtime.

## References and Sources Used

* (n.d.). CS231n Convolutional Neural Networks for Vision Recognition. Retrieved from http://cs231n.github.io/transfer-learning/
* (n.d.). Inception v3 In TensorFlow. Retrieved from https://tfhub.dev/google/imagenet/inception\_v3/feature\_vector/1
* (n.d.). PNASNet in TensorFlow. Retrieved from https://tfhub.dev/google/imagenet/pnasnet\_large/feature\_vector/2
* (n.d.). AmoebanNet in TensorFlow. Retrieved from https://tfhub.dev/google/imagenet/amoebanet\_a\_n18\_f448/feature\_vector/1
* Advanced Guide to Inception v3 on Cloud TPU  |  Cloud TPU  |  Google Cloud. (n.d.). Retrieved from https://cloud.google.com/tpu/docs/inception-v3-advanced
* Crash Course in Convolutional Neural Networks for Machine Learning. (2016, September 05). Retrieved from https://machinelearningmastery.com/crash-course-convolutional-neural-networks/
* Crash Course in Recurrent Neural Networks for Deep Learning. (2017, July 19). Retrieved from https://machinelearningmastery.com/crash-course-recurrent-neural-networks-deep-learning/
* Crash Course On Multi-Layer Perceptron Neural Networks. (2016, June 07). Retrieved from https://machinelearningmastery.com/neural-networks-crash-course/
* G., & S. (n.d.). Deep learning made easier with transfer learning. Retrieved from https://blog.fastforwardlabs.com/2018/09/17/deep-learning-is-easy-an-introduction-to-transfer-learning.html
* Z., R., S., & J. (2015, December 10). Deep Residual Learning for Image Recognition. Retrieved from https://arxiv.org/abs/1512.03385
* H., G., L., Maaten, V. D., L., W., & K. (2018, January 28). Densely Connected Convolutional Networks. Retrieved from https://arxiv.org/abs/1608.06993
* DIY Deep Learning for Vision: A Hands-On Tutorial with Caffe. (n.d.). Retrieved from https://docs.google.com/presentation/d/1UeKXVgRvvxg9OUdh\_UiC5G71UMscNPlvArsWER41PsU/edit#slide=id.gc2fcdcce7\_216\_257
* Gentle Introduction to Models for Sequence Prediction with Recurrent Neural Networks. (2017, July 19). Retrieved from https://machinelearningmastery.com/models-sequence-prediction-recurrent-neural-networks/
* Gershgorn, D. (2017, July 26). The data that transformed AI research-and possibly the world. Retrieved from https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/
* S., L., W., J., P., S., . . . Reed. (2014, September 17). Going Deeper with Convolutions. Retrieved from https://arxiv.org/abs/1409.4842
* Google's Entry to ImageNet 2014 Challenge. (n.d.). Retrieved from http://deeplearning.net/2014/09/19/googles-entry-to-imagenet-2014-challenge/
* Gupta, D., & D. (2017, October 23). Fundamentals of Deep Learning - Activation Functions and their use. Retrieved from https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/
* Hochreiter, S., & Schmidhuber, J. (1997, November 15). Long short-term memory. Retrieved from https://www.ncbi.nlm.nih.gov/pubmed/9377276
* How to Retrain an Image Classifier for New Categories  |  TensorFlow Hub  |  TensorFlow. (n.d.). Retrieved from https://www.tensorflow.org/hub/tutorials/image\_retraining
* How to Retrain an Image Classifier for New Categories  |  TensorFlow Hub  |  TensorFlow. (n.d.). Retrieved from https://www.tensorflow.org/hub/tutorials/image\_retraining
* Z., R., S., & J. (2016, July 25). Identity Mappings in Deep Residual Networks. Retrieved from https://arxiv.org/abs/1603.05027
* Imgdl. (n.d.). Retrieved from https://pypi.org/project/imgdl/
* B., V., V., J., & V., Q. (2018, April 11). Learning Transferable Architectures for Scalable Image Recognition. Retrieved from https://arxiv.org/abs/1707.07012
* B., V., V., J., & V., Q. (2018, April 11). Learning Transferable Architectures for Scalable Image Recognition. Retrieved from https://arxiv.org/abs/1707.07012
* MobileNets: Open-Source Models for Efficient On-Device Vision. (2017, June 14). Retrieved from https://ai.googleblog.com/2017/06/mobilenets-open-source-models-for.html
* Z., B., & V., Q. (2017, February 15). Neural Architecture Search with Reinforcement Learning. Retrieved from https://arxiv.org/abs/1611.01578
* R., M., T., B., & Y. (2013, February 16). On the difficulty of training Recurrent Neural Networks. Retrieved from https://arxiv.org/abs/1211.5063v2
* Ouaknine, A. (2018, January 16). Review of Deep Learning Algorithms for Image Classification. Retrieved from https://medium.com/comet-app/review-of-deep-learning-algorithms-for-image-classification-5fdbca4a05e2
* Pröve, P. (2018, July 25). History of Convolutional Blocks in simple Code – Towards Data Science. Retrieved from https://towardsdatascience.com/history-of-convolutional-blocks-in-simple-code-96a7ddceac0c
* L., Z., B., N., J., H., . . . K. (2018, July 26). Progressive Neural Architecture Search. Retrieved from https://arxiv.org/abs/1712.00559
* E., A., A., H., Y., & V, Q. (2018, October 26). Regularized Evolution for Image Classifier Architecture Search. Retrieved from https://arxiv.org/abs/1802.01548
* V., V., S., J., & Z. (2015, December 11). Rethinking the Inception Architecture for Computer Vision. Retrieved from https://arxiv.org/abs/1512.00567
* V., V., S., J., & Z. (2015, December 11). Rethinking the Inception Architecture for Computer Vision. Retrieved from https://arxiv.org/abs/1512.00567
* Santos, L. A. (n.d.). GoogleNet. Retrieved from https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/googlenet.html
* Sebastian Ruder. (2018, October 24). NLP's ImageNet moment has arrived. Retrieved from http://ruder.io/nlp-imagenet/
* T. (2018, September 11). Tensorflow/models. Retrieved from https://github.com/tensorflow/models/tree/master/research/inception
* T. (2018, September 25). Tensorflow/models. Retrieved from https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet
* S., K., Z., & A. (2015, April 10). Very Deep Convolutional Networks for Large-Scale Image Recognition. Retrieved from https://arxiv.org/abs/1409.1556
* When to Use MLP, CNN, and RNN Neural Networks. (2018, April 25). Retrieved from https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/
* Y. (2018, April 18). Yeephycho/tensorflow\_input\_image\_by\_tfrecord. Retrieved from https://github.com/yeephycho/tensorflow\_input\_image\_by\_tfrecord

List of Figures

[Figure 1 - Improvement of Neural Networks trained using ImageNet 10](#_Toc529015312)

[Figure 2 - AlexNet architecture 11](file:///C:\Users\chris\Documents\CUNY\DATA698%20-%20Final%20Masters%20Thesis\Sidneys%20Catalog%20-%20Master%20Project.docx#_Toc529015313)

[Figure 3 - VGG16 architecture 11](#_Toc529015314)

[Figure 4 - GoogLeNet architecture 12](#_Toc529015315)

[Figure 5 - ResNet Architecture compared to VGG19 and a 34-layer CNN 13](#_Toc529015316)

[Figure 6 - A single Dense Block within DenseNet 14](#_Toc529015317)

[Figure 7 - A single transition layer in DenseNet 14](#_Toc529015318)

[Figure 8 - DenseNet with three Dense Blocks 14](#_Toc529015319)

[Figure 9 - Inception v3 architecture 14](#_Toc529015320)

[Figure 10 - MobileNet v2 architecture 15](#_Toc529015321)

[Figure 11 - Example of a NASNet CNN architecture 15](#_Toc529015322)

[Figure 12 - PNASNet architecture evolution 16](#_Toc529015323)

[Figure 13 - AmoebaNet-A architecture 16](#_Toc529015324)

[Figure 14 - Accuracy of Inception v3 21](#_Toc529015325)

[Figure 15 - Accuracy of ResNet v2 22](#_Toc529015326)

[Figure 16 - Accuracy of MobileNet v2 22](#_Toc529015327)

[Figure 17 - Accuracy of PNASNet-5 22](#_Toc529015328)

## Prompt

The final version of the project is a medium length, 20-24 pages (10-12 pages, double-sided) paper describing the project, along with a runnable demo of any code. Code listings, interpreter output, and exploratory data visualization should not be included in the paper itself.