Stage 1: Context and Data Model Build

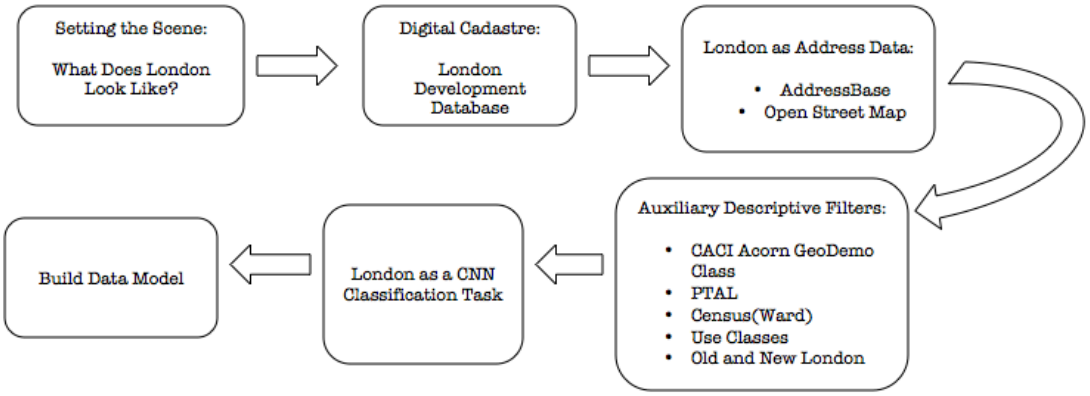


Fig 1.1 *–Workflow Stage Overview*

1.1 Stage Overview

- Case Study and Visual Descriptor Selection.
- Allows user to **explore**, **visualize** and quickly **train** a CNN Image Classification model on different data sets of Visual Descriptor anchored on London New Build Properties from 2004.
- Final Data Model Building Stage.

1.2 Minimum Viable Product: Quick Spin Tool

A minimum viable product (MVP) is a concept from the Agile Project Management Framework that stresses the impact of learning in new product development. Eric Ries, defined an MVP as that version of a new product which allows a team to collect maximum amount of validated learning about customers with the least effort. A team effectively uses MVP as the core piece of a strategy of experimentation.

For early stages of the data model exploration and creation stage, the quick spin tool was used for collecting an initial CNN Training precision metric to identify visual descriptors that were sufficiently machine learner friendly and geo-informative. This used the fast to run but also reasonably accurate Mobile Net version 1 model for images of 224 x 224 resolution. Visual descriptive features and benchmark datasets could then be trialed at this early stage and gauged as to whether to be included in the final model. As a heuristic, if the run obtained precision rates of 50% or more they were retained for further consideration.

Fig 1.2 *–QuickSpin CNN Trainer deploys mobilenet 224*

1.4 Visual Descriptor Exploration

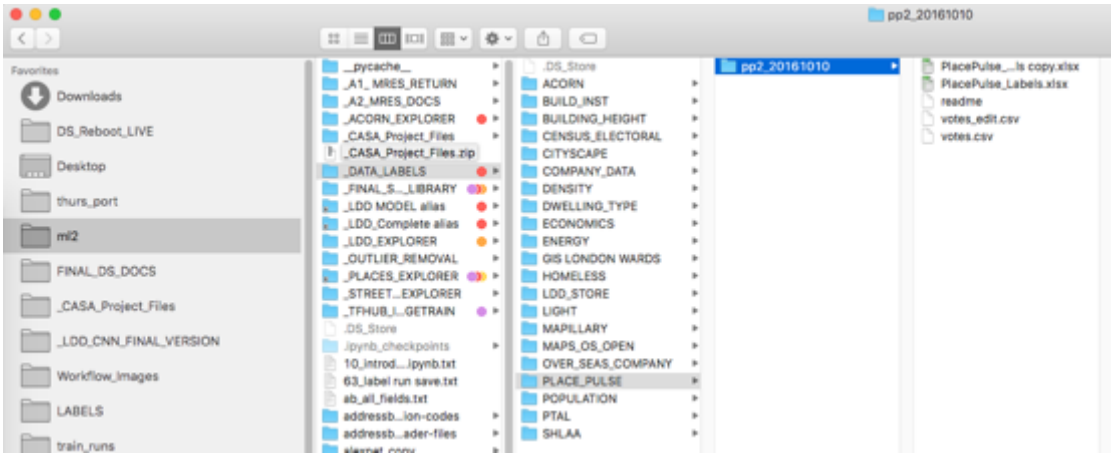


Fig 1.3 *–Auxilary Visual Descriptors*

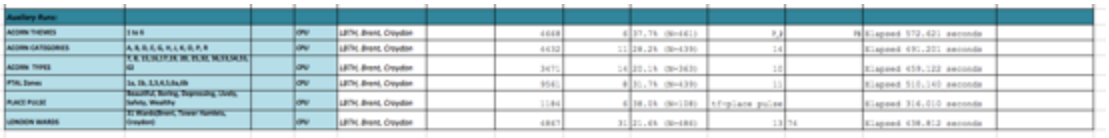


Fig 1.4 *–QuickSpin Visual Descriptor Metrics*

The rich set of visual descriptors optimistically plugged in to the skeleton model were found to produce low classification accuracy results ultimately leading us to discount the benefit of the untreated auxiliary descriptors in our final data model. However the overall process provided valuable ground truth context for our thematic exploration, pertinent pointers to the requirements of an image classification task and began to highlight the idea of the gap between perception and vision in interpreting urban scenes that forms part of this study's thematic exploration.

Of more use was the opportunity to reproduce the results produced by the Kang et Al Building instance study and to apply model weights trained on their released data set of Street Level Images from US cities, on the LDD anchored Data Model and to the question addressed in later sections: How do US Cities compare with London New Build in terms of their Visual and Physical form?

A small hand labeled set of new build high rise and large scale developments selected from the LDD repository (using completion date, unit delivery size,

affordability mix and payment in lieu of housing features) produced what appeared like satisfactory result in a one vs all image classification, and would be the base finding from which to explore further ways of sub dividing the development.

A full comparison of the Auxiliary Pilot stage training runs can be made with the full set of training runs in the Workflow Metric Tables, found in the CNN Training Section 4.265

Label Name	Description	Model	Platform	Location	Step Count	Number of Images	Number of Labels	Test Accuracy
Benchmark Categories:								
FLOWERS	Daisy, Dandelion, Roses, Sunflowers, Tulips	mobnet 0.5	CPU	n/p	4000	3677	5	90.5% (N=359)
US OSM PROPERTY TYPES	Apartment, Church, House, Industrial, Roof, Office, Retail, Garage	mobnet 0.5	CPU	US Cities (Montreal, New York, Denver)	4000	30000	8	51.6% (N=1782)
Trained US OSM PROPERTY TYPES Model applied to US OSM Test Set	Apartment, Church, House, Industrial, Roof, Office, Retail, Garage							
Trained US OSM PROPERTY TYPES Model applied to LDD	Apartment, Church, House, Industrial, Roof, Office, Retail, Garage	mobnet 0.5	CPU	US Cities (Montreal, New York, Denver)				
LDD HAND LABEL	Weak, Strong, Blank Candidate(London New Build Image, Noisy Street Scene Image, Google Map No Image Found)	mobnet 0.5	CPU	London Wide	4000	374	3	92.9% (N=42)

Fig 1.5 *–QuickSpin Auxiliary Datasets Metrics*

Stage 2: Bulk Data Downloading Street Level Image Data with the Google StreetView API

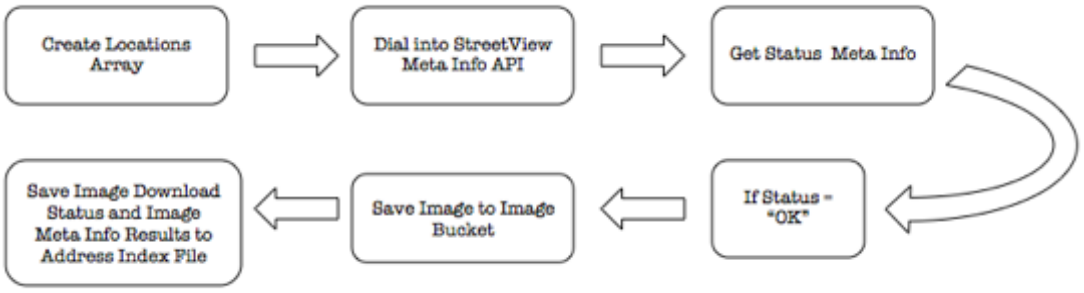


Fig 2.1 *–Workflow Stage Overview*

2.1 Stage Overview

- For building an urban street-level image model at scale.
- For downloading bulk Street Level Images and meta data using the Google Street View Static and Meta APIs.

Fig 2.2 *– Google Maps Platform*

2.2 Google Street View Static API

The Street View **Static** API lets you embed a static (non-interactive) Street View panorama or thumbnail into your web page, without the use of JavaScript. The viewport is defined with URL parameters sent through a standard HTTP request, and is returned as a static image.

The Street View **Meta** APIs is used to identify missing image data and to vary image size, panorama angles and other image settings.

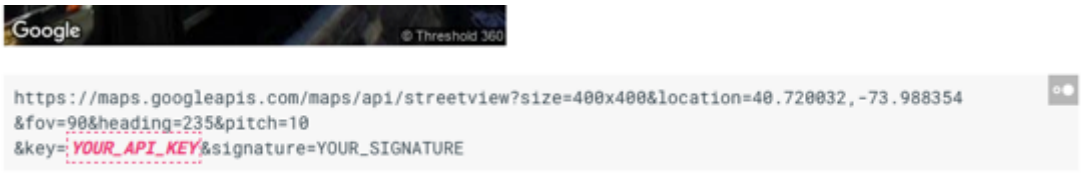


Fig 2.3 *– Static Street View API Request string*

Fig 2.4 *– The Resultant 400x400 StreetView Image*

Google StreetView Static API

- The Street View Static API lets you embed a static (non-interactive) Street View panorama or thumbnail into your web page, without the use of JavaScript. The viewport is defined with URL parameters sent through a standard HTTP request, and is returned as a static image

Required parameters

- location:
  - Can be either a text string (such as Chagrin Falls, OH) or a lat/lng value (40.457375,-80.009353). The Street View Static API will snap to the panorama photographed closest to this location. When an address text string is provided, the API may use a different camera location to better display the specified location. When a lat/lng is provided, the API searches a 50 meter radius for a photograph closest to this location. Because Street View imagery is periodically refreshed, and photographs may be taken from slightly different positions each time, it's possible that your location may snap to a different panorama when imagery is updated.
- pano:
  - Is a specific panorama ID. These are generally stable.
- size:
  - specifies the output size of the image in pixels. Size is specified as {width}x{height} - for example, size=600x400 returns an image 600 pixels wide, and 400 high.
- key:
  - allows you to monitor your application's API usage in the Google Cloud Platform Console, and ensures that Google can contact you about your application if necessary. For more information, see Get a Key and Signature.

https://developers.google.com/maps/documentation/streetview/intro

Optional parameters

- `signature` (*recommended*) is a digital signature used to verify that any site generating requests using your API key is authorized to do so. Requests that do not include a digital signature might fail. For more information, see Get a Key and Signature.

  Note: for Google Maps APIs Premium Plan customers, the **digital signature is required**. Get more information on authentication parameters for Premium Plan customers.
- `heading` indicates the compass heading of the camera. Accepted values are from `0` to `360` (both values indicating North, with `90` indicating East, and `180` South). If no heading is specified, a value will be calculated that directs the camera towards the specified `location`, from the point at which the closest photograph was taken.
- `fov` (*default is* `90`) determines the horizontal field of view of the image. The field of view is expressed in degrees, with a maximum allowed value of `120`. When dealing with a fixed-size viewport, as with a Street View image of a set size, field of view in essence represents zoom, with smaller numbers indicating a higher level of zoom.

Fig 2.5 *– Street View Static API parameters*

(*Left:* `fov=120`; *Right:* `fov=20`)

Fig 2.6 *– Optional Field of View parameter*

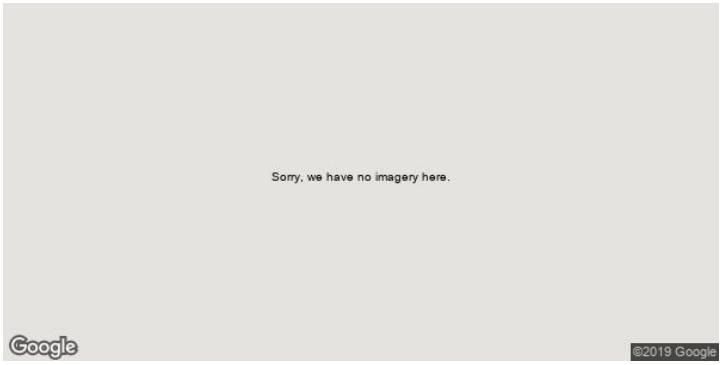


Fig 2.7 *– Zero Results Static API Response*

4.2.3 Street View Image Metadata

The Street View Static API metadata requests provide data about Street View panoramas. Using the metadata, you can find out if a Street View image is available at a given location, as well as getting programmatic access to the latitude and longitude, the panorama ID, the date the photo was taken, and the copyright information for the image. Accessing this metadata allows you to customize error behavior in your application, and we use it as part of the study's image pre processing and data model cleanse stage.

Fig 2.8 *– Meta Info API Request String*

Status	Description
`"OK"`	Indicates that no errors occurred; a panorama is found and metadata is returned.
`"ZERO_RESULTS"`	Indicates that no panorama could be found near the provided location. This may occur if a non-existent or invalid panorama ID is given.
`"NOT_FOUND"`	Indicates that the address string provided in the `location` parameter could not be found. This may occur if a non-existent address is given.
`"OVER_QUERY_LIMIT"`	Indicates that you have exceeded your daily quota or per-second quota for this API.
`"REQUEST_DENIED"`	Indicates that your request was denied. This may occur if you did not use an API key or client ID, or if the Street View Static API is not activated in the Google Cloud Platform Console project containing your API key.
`"INVALID_REQUEST"`	Generally indicates that the query parameters (address or latlng or components) are missing.
`"UNKNOWN_ERROR"`	Indicates that the request could not be processed due to a server error. This is often a temporary status. The request may succeed if you try again.

Fig 2.9 *– Meta Info API Response Types*

2.4 Workflow

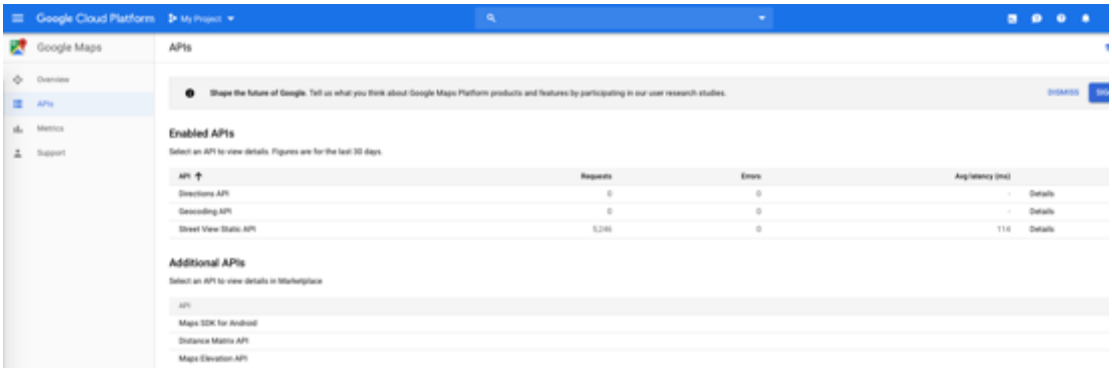


Fig 2.10 *– Google Cloud Console Enabled APIs*

Dial in and Get Image Meta Info

```
In [9]: import urllib.request
        from urllib.parse import quote
        #from urllib.parse import urlparse

        # Put your Google Street View API key here.
        #api_key = ""

        api_key = "AIzaSyD_O9mUG_8vdImaGWp-Iiwo72K95c3XkTY"

        #AIzaSyD_O9mUG_8vdImaGWp-Iiwo72K95c3XkTY

        #api_url = "https://maps.googleapis.com/maps/api/streetview?size=640x640&location={0},{1}&pitch=10&fov=120&source
        #api_url = "https://maps.googleapis.com/maps/api/streetview?size=640x640&location={0},{1}&source=outdoor&key={2}"
        api_url_meta = "https://maps.googleapis.com/maps/api/streetview/metadata?&location={0},{1}&key={2}"


        # Loop over every location, and for each location, loop over all the possible headings.

        for location in locations:
            #for direction in headings:
            # Create the URL for the request to Google.
            col, lat, lon, post = location
            #url = api_url.format(quote(lat), quote(lon), api_key)
            #url = api_url.format(lat, lon, api_key)
            #print(url)
            meta_url = api_url_meta.format(lat, lon, api_key)

            #filename = "/Users/anthonysutton/ml2/_ACORN_EXPLORER/London_Labels3/col_id_" + col + ".jpg"

            # Use the 'curl' command to actually make the request and save the file to disk.
            #urllib.request.urlretrieve(url, filename)

            with urllib.request.urlopen(meta_url) as response:
                    meta_resp = response.read()
                    #print(meta_resp)
                    #print(meta_resp.decode('utf-8'))
                    #responded = (col, lat, lon, post, meta_resp.decode('utf-8').replace("'", '"'))
                    responded = (col, lat, lon, post, meta_resp)
            #responded = (col, post, lat, lon)
                    responses.append(responded)

            print(meta_url)
            #print(url)
            print(responded)
            #print(filename)

        print ("Done")

        #31061
        #116
        #38647
        #105242
        #116810
```

```
https://maps.googleapis.com/maps/api/streetview/metadata?&location=51.47304631,0.201037913&key=AIzaSyD_O9mUG_
8vdImaGWp-Iiwo72K95c3XkTY
('4187', '51.47304631', '0.201037913', 'DA8 2JY', b'{\n   "copyright" : "\xc2\xa9 Google",\n   "date" : "2018
-05",\n   "location" : {\n      "lat" : 51.47334588910583,\n      "lng" : 0.2014602319549051\n   },\n   "pano
_id" : "5ZSCtIOtDLN51GkAyo0QxA",\n   "status" : "OK"\n}\n')
https://maps.googleapis.com/maps/api/streetview/metadata?&location=51.56127445,0.175170761&key=AIzaSyD_O9mUG_
8vdImaGWp-Iiwo72K95c3XkTY
('44', '51.56127445', '0.175170761', 'RM7 0TB', b'{\n   "copyright" : "\xc2\xa9 Google",\n   "date" : "2018-0
4",\n   "location" : {\n      "lat" : 51.56124369649879,\n      "lng" : 0.1754946290804363\n   },\n   "pano_i
d" : "J7CCslwKkL0jKsVIXV8dpQ",\n   "status" : "OK"\n}\n')
```

Fig 2.11 *– Accessing the Image Meta Data api*

Fig 2.12 *– Enabled API: Traffic and Error Metrics*

Stage 3: Image Preprocessing

Fig 3.1 *–Workflow Overview*

3.1 Aim

- This stage is designed for preparing the images and image dataset for the CNN Training Stage.

3.2 Street View Image Cleansing: Occlusions and Assemblage(Street Interfaces)

A potential problem for any real world CNN Task is erroneously labeled data. Using Google Street View and the Address Base data on a large scale significantly increases the likelihood of dealing with invalidated or anomalous data. These might be in the form of occluded images(tree foliage and buses) or misleading images(inadequate access to the building subject but the street view van).

Google Street View Photography is taken in real time, in the living and working urban environment via the Google Van at Street Level and from the vantage point of the public highway. This presents a challenge to the goal of building a comprehensive, fully complete image dataset of Street Level Scenes and a standardized categorical feature space. Local factors that prohibit the collation of a pristine and usable street level street scene image come in several guises and that will result in Occluded Images, Inaccessible Image Subjects and imprecise geo-referencing. Weather and related environmental factors (such as building works in progress or light occlusion effected by the local urban design of the street level in question)will also have an effect on lighting variation from image to image. Whilst a well labeled and put together CNN training set will often include wide varieties of locale of the

image subject to be labeled. The visually noisy and highly populated(with candidate visual objects) feature space of a street level image of an urban environment, as identified in the Kang et Al study, warrant a previous clean up workflow stage prior to the building instance classification stage.
Trees, Traffic and In accessible Building Facades presented by hard to reach assemblage can severely jeopardize the classification results.

Fortunately a large part of the CNN breakthroughs of recent years has been down to the production and usage of large scale conceptually semantic Image datasets, and key part of this has been work on Scene Recognition Model training datasets. We use Scene Recognition Visual Semantics to identify scenes which are likely to contain our set of building instance types

Fig 3.2 *–Typical Occlusions in the StreetView Repository: Trees, Traffic, Inaccessible Building Facades and awkward accessibilty afforded by restrictive public private building assemblage interface*

3.3 Pre Processing Workflow Tool

Processing Stage 1 - Using Pre Trained Places365 CNN to Remove Outlier Images

Fig 3.3 *–Pre Processing Stage Script*

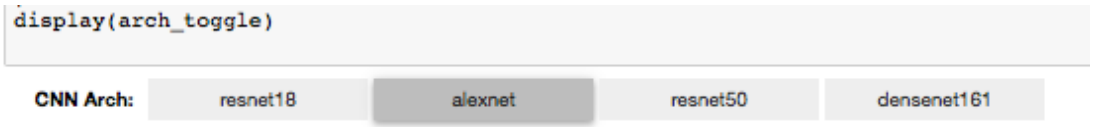


Fig 3.4 *–Python Widget for Model Selection*

Python Selection Widgets were incorporated into the beginning of the script along with a quick run single image tester, to allow for experimentation with different model architectures and a quick operational check of the code, during the exploratory stage and subsequent return iterations through this part of the workflow.

The fine grained and large number of categories deployed by the Places365 Classifiers limited the potential benefits of the pre processing stage. Image Classification accuracy was also found to be hit and miss. Fewer categories would have been of more benefit for our particular Building Classification Task and selecting just the classes chosen in the Kang et al study severely restricted sample sizes by omitting a large proportion of the valid imagery data. The main difference here is that the Kang et Al study worked with a supply of street level imagery from several large us cities and was not

restricted to LDD registered new development buildings for the London area only.

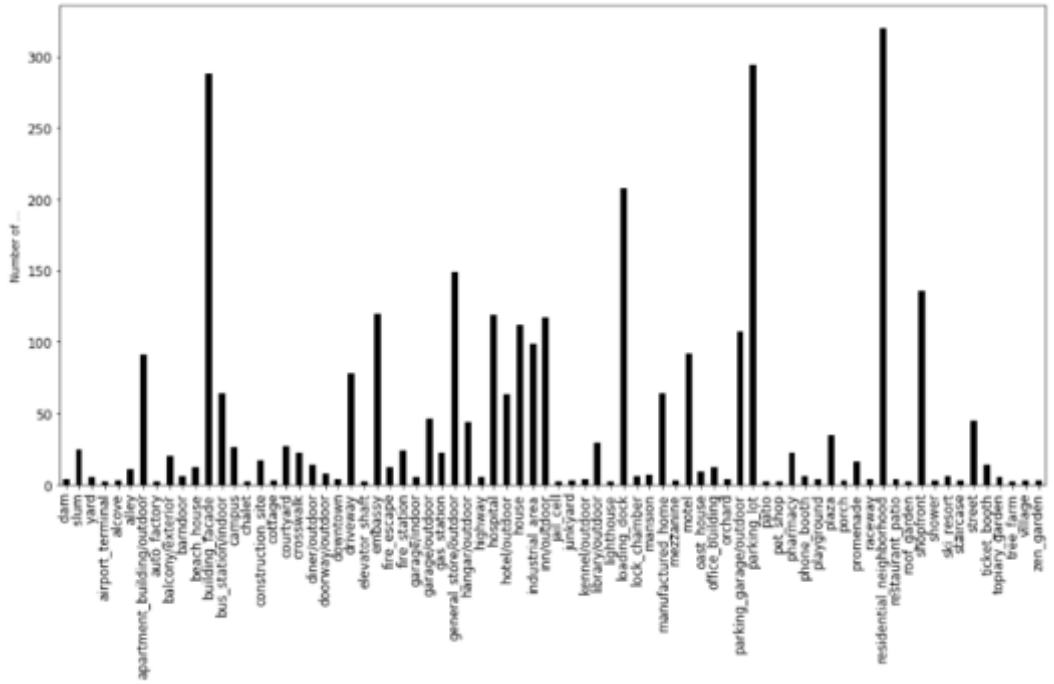


Fig 3.5 *–List of Places 365 Categories for House, Flat, Office, Retail and Industrial AddressBase Class Training Label Set*

On the other hand, the large number of overlapping categories was useful in identifying trees/bus occlusions (of which there were many examples in our street level image data and which could be picked up by selecting any category with a vegetative or vehicular reference in the description e.g. beer_garden, bamboo_garden or bus_station, fire_station) and wild outlier image instances(incorrect building objects idenitifed , for instance, erroneously picked up in the Industrial category due to assemblage issues associated with industrial estates with poor accessibility e.g. viaduct or junkyard). These anomalies could be filtered out with reasonable success. Also, the snowfield category was useful in identifying ‘no image found’ images from the streetview repository and which were missed in the meta api download stages.

Here we can see how the practical application of the workflow required iterative repeat progressions through already completed stages but which would improved data validations and conditions required for the next stage in any part of the sequence.

This limitation raises the question why not use the Places365 Model weights for our final building classification task? The Categories used by the Places365 model are too wide ranging to allocate Class membership within the confines of our specific and particular research question and the feature space that it inhabits: what type of building are we looking at?

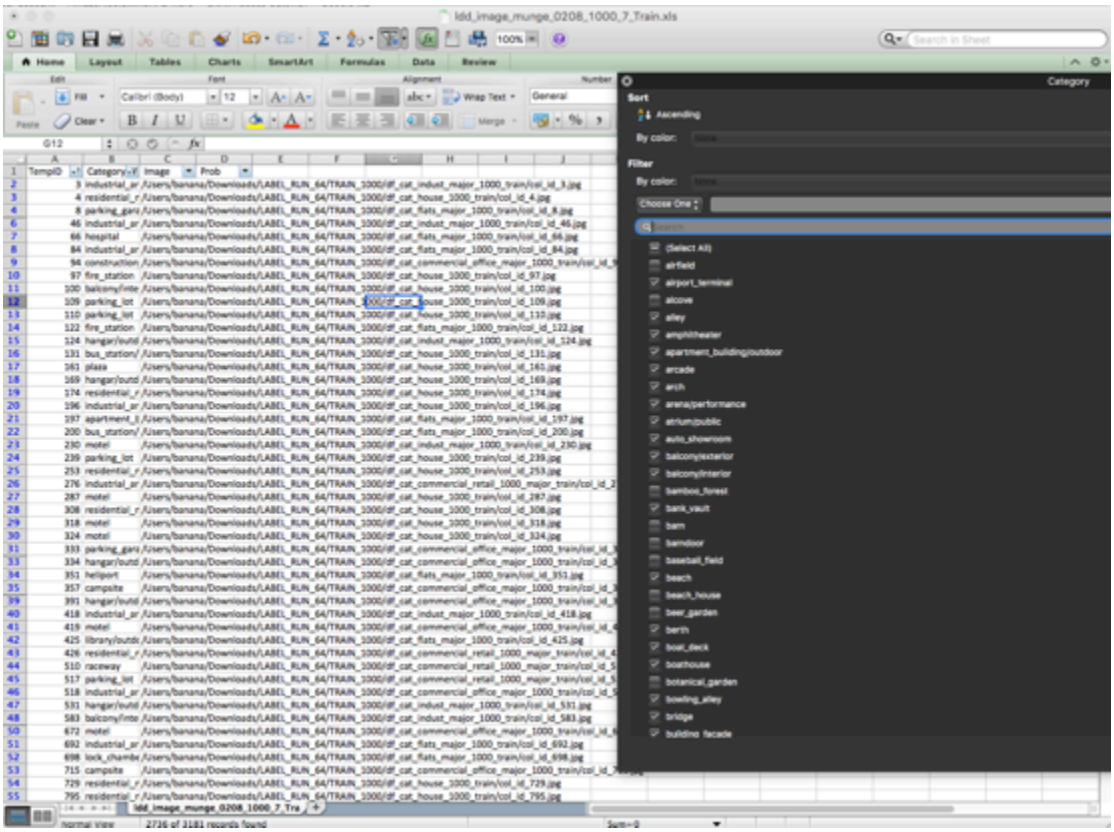


Fig 3.6 *–Manual Work around for Filtering Out unwanted Places365 Types*

After filtering on label types that would omit tree, traffic and similar occlusions, the inline Image Thumb Gallery was used to identify further candidate images for exclusion.

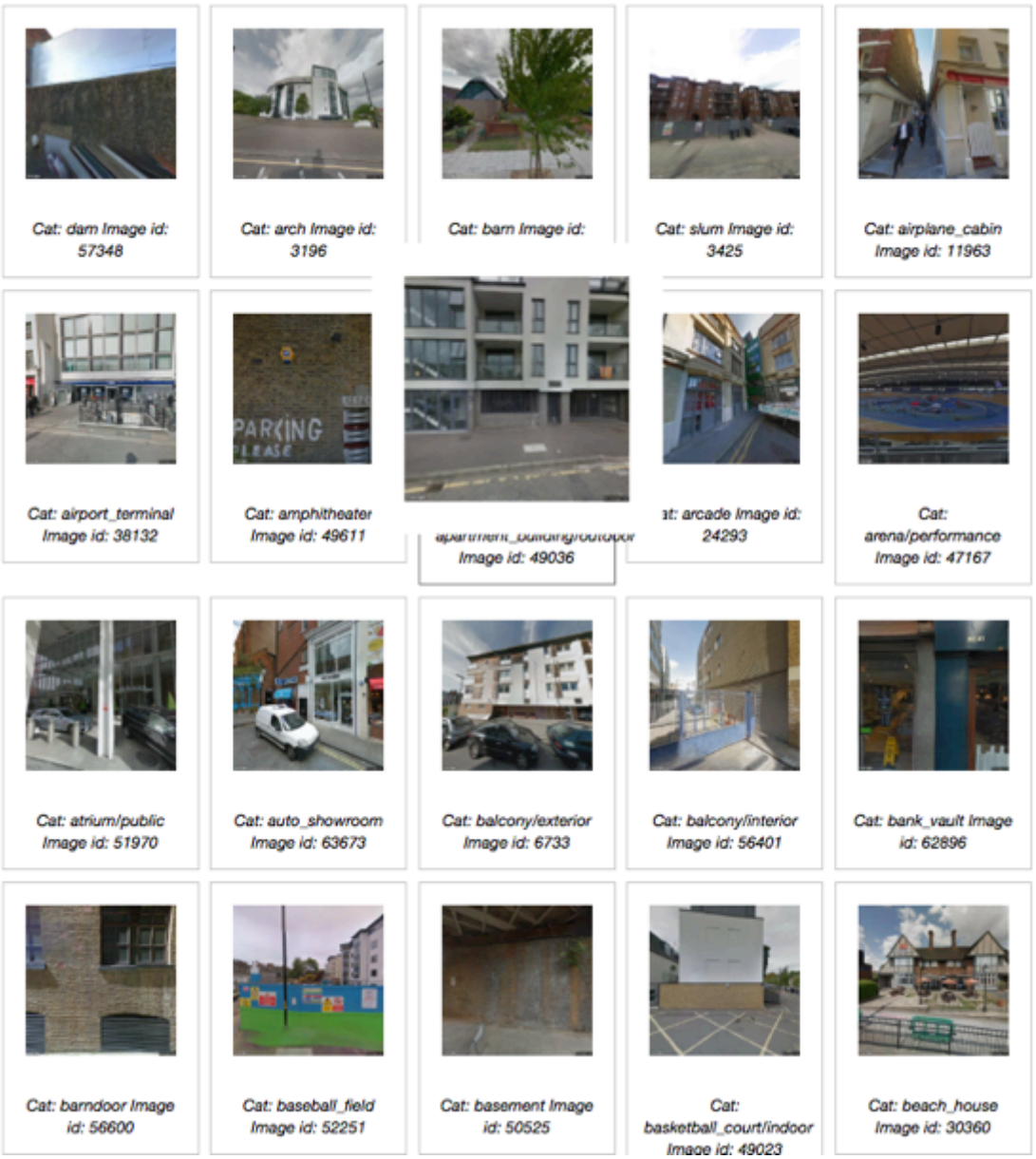


Fig 3.7 *–Thumb Gallery of Examples from the above Places365 Preprocessing Purge*

3.4 Scene Recognition with Places365 and Pytorch

CNN Architecture	Rank	Notes
resnet18		
alexnet		*Default Model and most commonly used in the workflow stage*
Resnet50		
Densenet161		

Network	Top-1 error	Top-5 error
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
Inception v3	22.55	6.44
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20

Fig 3.8 – *Places365 CaffeNet Implementation Models and ImageNet 1 crop error rates (224x224) for Pytorch Models available for use in the study*

Stage 4: Label Set Creation

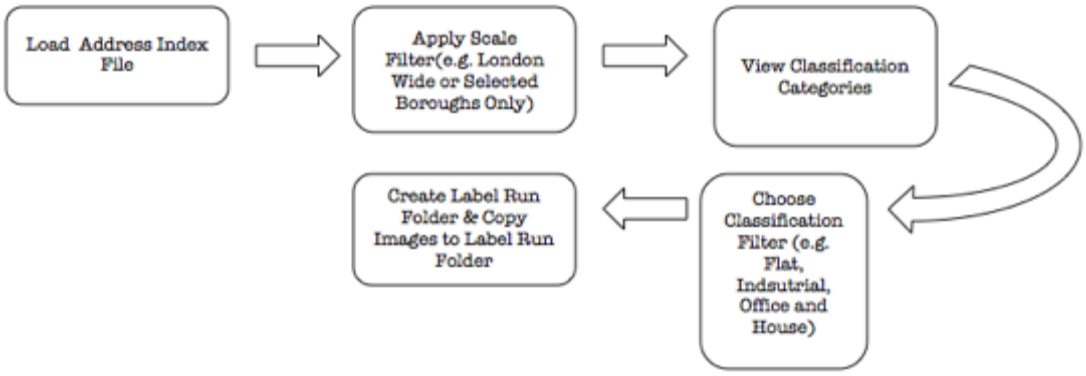


Fig 4.1 *– Label Set Creation Workflow Overview*

4.1 Stage Overview

- For creating taxonomic nomenclature variations of labeled training data from the master data model of London new development building types and instances.

4.2 Case Study Selection

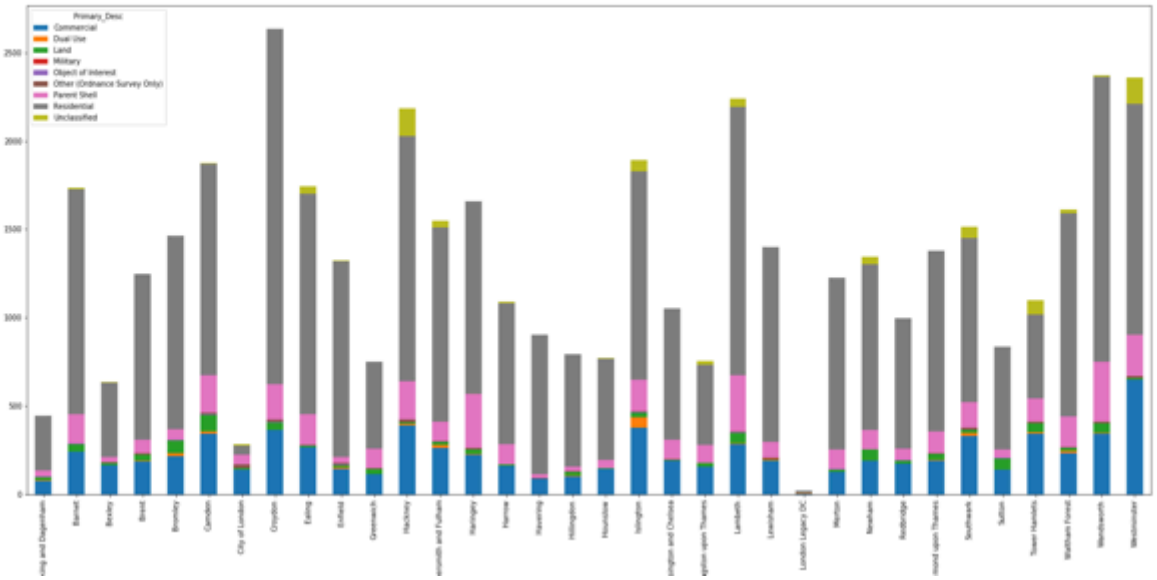


Fig 4.2 *– Master Data Model joined on AddressBase Primary Descriptor*

The aim of this workflow stage is to provide the CNN Training and Classification stages with data in sufficient quantity and quality to complete a successful Image Classification task. This is ultimately measured in terms of CNN accuracy and recall however, in line with Dorsh et al's research into using geographically weighted SVM for image classification, we are also concerned with building a representative model of London's new building development that is geo-informational and with which we can reach conclusions and a degree of insight into its aesthetic visual character.

For initial runs through the Project Workflow we limited label set creation and image classification attempts to a restricted, selected number of London Boroughs while we experimented with differing composite variations of Borough, Use Class/Building Type, Numbers of Records and so on. Throughout this process Ground Truthing and Dashboards approaches/tools were used to ensure that these objectives were being met and to make explicit and visual the data being sampled and parsed under the bonnet.

The above chart shows that it is apparent that boroughs such as Brent, Tower Hamlets and Westminster have large amounts of New Development Property Data and a comparatively favorable mix of the address types available. These 3 areas were also selected in the early stages as they represented areas with which we had employment, habitational and recreational experience.

4.4 Data Filtering

The raw data model(new development plots coupled with a street level image record and additional descriptor data joins) yields approx. 70000 records and at first sight offers us a rich seam of examples of London new build development. However upon further consideration it is apparent that several factors necessitate the removal of a large number of these records.

Figure 4.2 reveals familiar and, for the building type classification task at hand, pertinent categories such as the Commercial and Residential Primary AddressBase Descriptor classes but the chart also highlights numerous building type labels that we will discard(unclassified, parent shell, mixed use) as they do not provide us with a sufficiently distinct visual category of building form. Orphaned Use Class categories which do not belong to a discernable class of building(e.g. a point of interest could be a member of several address categories) will also be discarded

In addition, certain LDD Record types provide us with visually confusing and ambiguous street level image records: incomplete building permissions and demolitions.

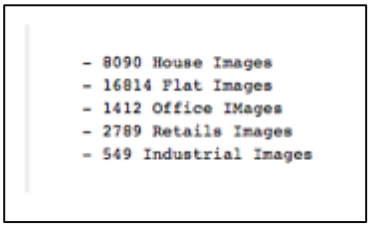


Fig 4.3 *– Population Data Size after Data Filtering*

After Data Filtering we appear to have an adequate amount of data in our population sample. However, we now need to proceed to looking at whether this translates into an even spread of property types and what the quality of this address and image data is like.

4.5 Use Class Distribution

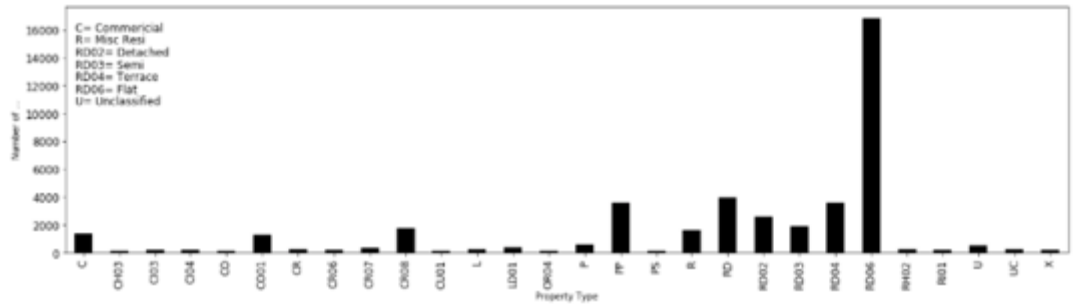


Fig 4.4 *– Breakdown of Building Types by Use Class Code*

As we might expect in a highly dense urban setting we see the largest amounts of data in the Residential categories, with RD06 (Self Contained Flats) eclipsing the Residential House Classes(RD02 to RD04, Detached, Semi-Detached and Terraces) and also over the smallest numbers to be found in the Commercial Industrial Class(CO01).

One point of interest is that there are large numbers of address in the generic Commercial and Residential classes and also several well populated miscellaneous categories that represent addresses that do not deal with Physical Buliding Types(L, PP, R, U). We will omit these from our study although it will be interstng to see what StreetView images are associated with these address types.

Figure 4.6 highlights the uneven representation of Primary Classes in our sample data and which will need be normalized to avoid skewing the Image Classification.

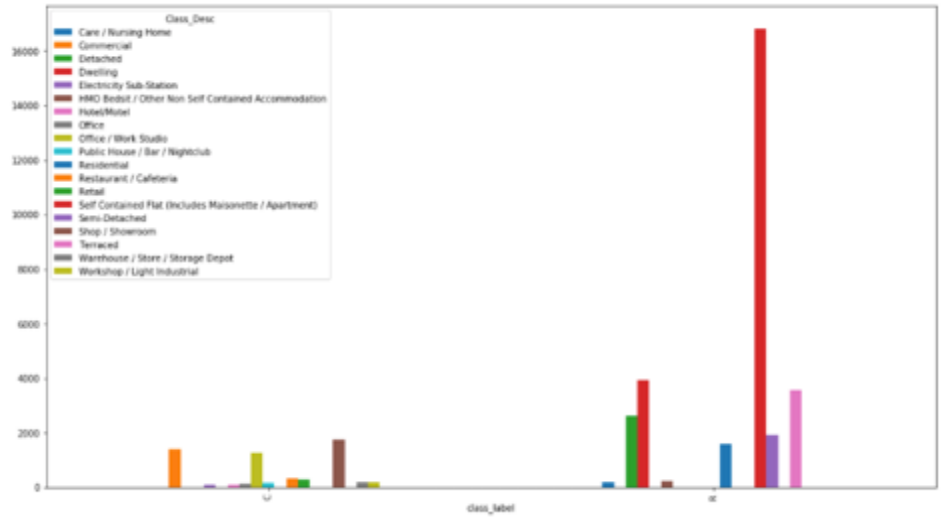
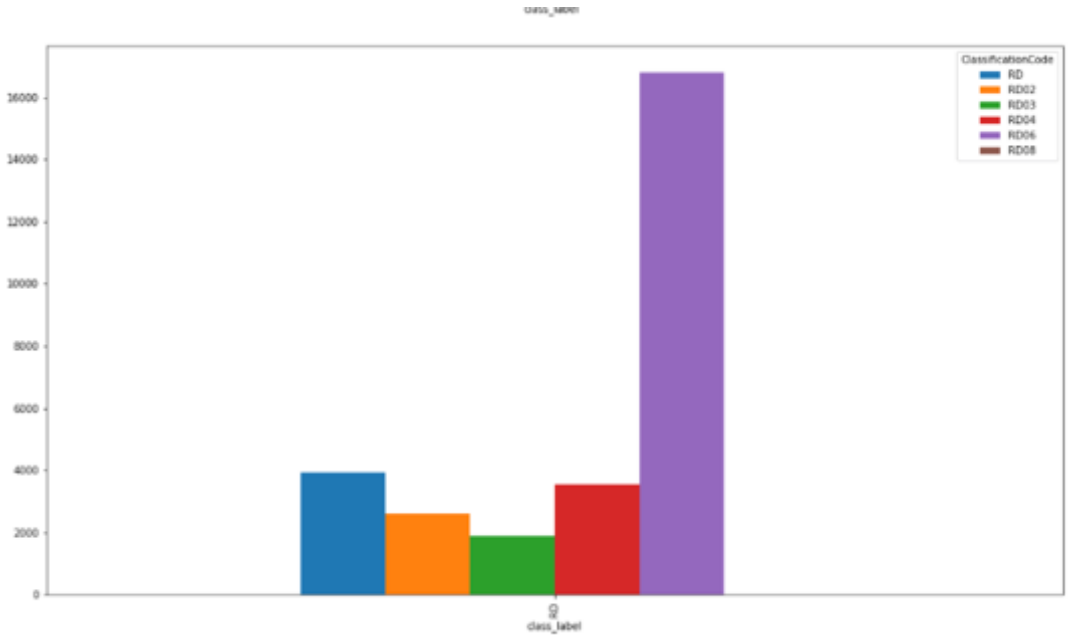


Fig 4.5 *- Disproportionate Numbers of Residential and Commercial Use Classes Data*

Figure 4.8 illustrates the subsequent sample sizes at our disposal once similar use classes have been grouped into a parent building type class. For instance, Detached, Semi-Detached, Terrace House Types are all Members of the 'House' Class.

Residential(Flats and Houses)



Commercial(Industrial, Office, Retail)

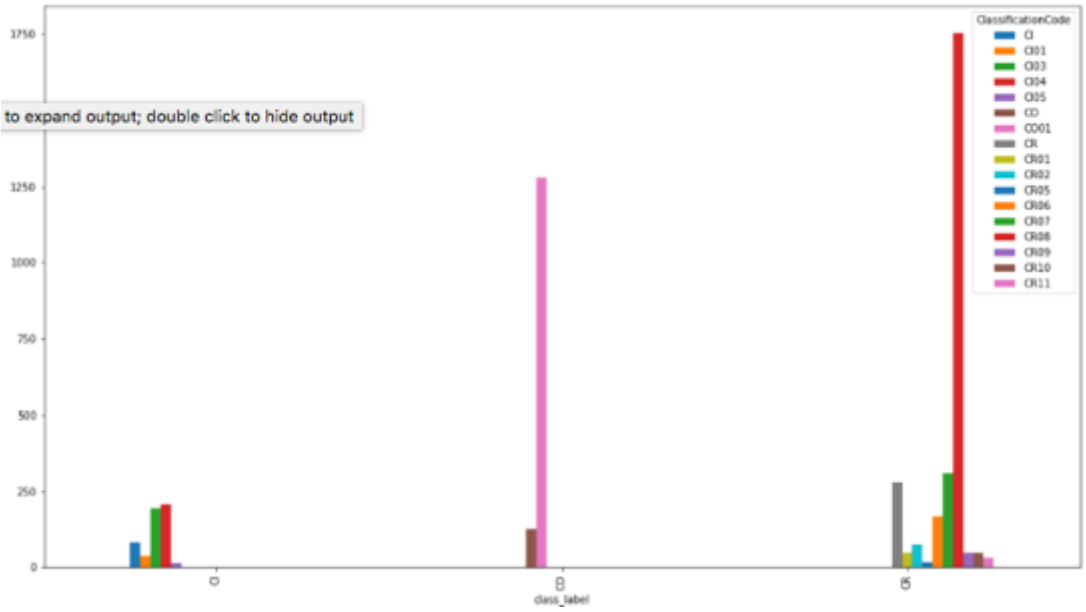


Fig 4.6 - *Combining Tertiary Classes into Primary Descriptors of Interest*

4.6 Adopted 200 and 1000 Sample Size approach

Kang et al and TensorFlow Benchmark and guidance

When choosing our candidate Boroughs and Use Class Categories, we are looking for an equal blend of sufficient amounts of data (e.g. 100s and 1000s of label candidates) coupled with data that evenly represents the full range of Building Types that can be found in the London architectural terrain.

The TensorFlow creators suggest that reasonable recall results can be obtained with a clean, clearly defined set of labeled image data samples numbering in the 100s. Meanwhile the Kang et al Building Instance Research Study achieved 70% accuracy rates with image training data of 2500 examples for each of the 8 clearly defined building type labels.

We therefore created 2 kinds of label set: one that groups secondary and tertiary address types into a more generalized grouping offering the benefit of a more numinous sample base and a secondary set that was designed to provide a wider representation of building class types on a more granular, sub category basis, but which as a consequence had a smaller population size.

Both Label Variations will have strengths and weakness Trade Offs in the Image Classification Task, which are measured and commented upon in the log tables and summary sections later on, but also highlights the advantage of setting up a workflow and workflow tools which are amenable to a heuristic exploration of the data model in the early trial stages of the research.

4.7 Machine Learning Considerations¶

Creating a Set of Training Images

The TensorFlow literature makes the following recommendations regarding creating a set of training images. Fig 4.7 outlines how these were taken into consideration in the design and implementation of our own particular label creation workflow stage.

	TF Label Creation Strategy	Project Mitigation
1	Garbage in-garbage out	Image Pre processing, Variations of Label Set Approach
2	More data = more accuracy	Experimentation with Several Variations of Label Set Creation Methods, yielding greater numbers but fewer classes or vice versa
3	Obtain Training data wide from a variety of situations	Increase Sample Size and Enlargement of Borough Selection
4	Split big categories into smaller categories	Variations of Label Set Approach
5	Try using an Unknown class to deal with large numbers of anomalous samples	Variations of Label Set Approach
6	Hand Weeding	Hand Labelled Label Sets created during Pilot Stages of the Research to verify CNN operational expectations

Fig 4.7 – *TF Framework Suggested Approaches*

Test/Train Split

The TensorFlow Script divides the image dataset into 3 sets. 80% of the images are put into the main training set, 10% are kept aside to run as validation frequently during training, and then a final 10% are used as a final Test Set.

With this in mind our approach was to split the label set images into approximately a 90-10 train test split. Keeping back a percentage for a more

widespread test of the CNN trained model, which could be used for aggregate confusion matrices examination and for use in the spatial analysis stage, (for drilling into building type and location characteristics for misclassified and indeed correctly classified labels) and for deriving insight into the character of London's New Build Development.

4.8 Creating Machine Learner Friendly CNN Classification Label Sets

We are now able to proceed with the process of creating sets of labels to feed into our label pipeline. We are seeking to strike a balance between settling on an informative enough representation of the class under scrutiny(London building types) but which also allows the selected CNN Architectures to achieve the highest precision and accuracy recall rates. As the above section demonstrates, the majority of our labeling data relates to Building Usage. For a CNN Friendly set of Labels we need to differentiate between Visual Building Typologies and Property Use Class Categories.

4.9 Mapping the OS Addressbase Premium Address (BS7666) Schema to OSM Building Types

Address Base Schema:

Classifications are sourced from the Local Land and Property Gazetteers and from Ordnance Survey large-scale data. Please see Figure 4.14 for detailed reference material on Statutory bodies for and Address Standards.

There are over 500 types contained in the schema used by Ordnance Survey. These describe characteristics that go beyond visual appearance and differentiate how a property is used. Have a look at the Schema Descriptions below. They drill down to Tertiary and Quaternary groupings, and provide a continuous and fine grained description of the property type.

This is an example of a how a classification can be structured in AddressBase Plus and Premium:

C	Commercial	**H**	Hotel/Motel/ Boarding/ Guest House	**01**	Boarding/Guest House/ Bed And Breakfast/Youth Hostel	Object defined by local government contributing authority, includes: Commercial Lodging.	**YH**	Youth Hostel

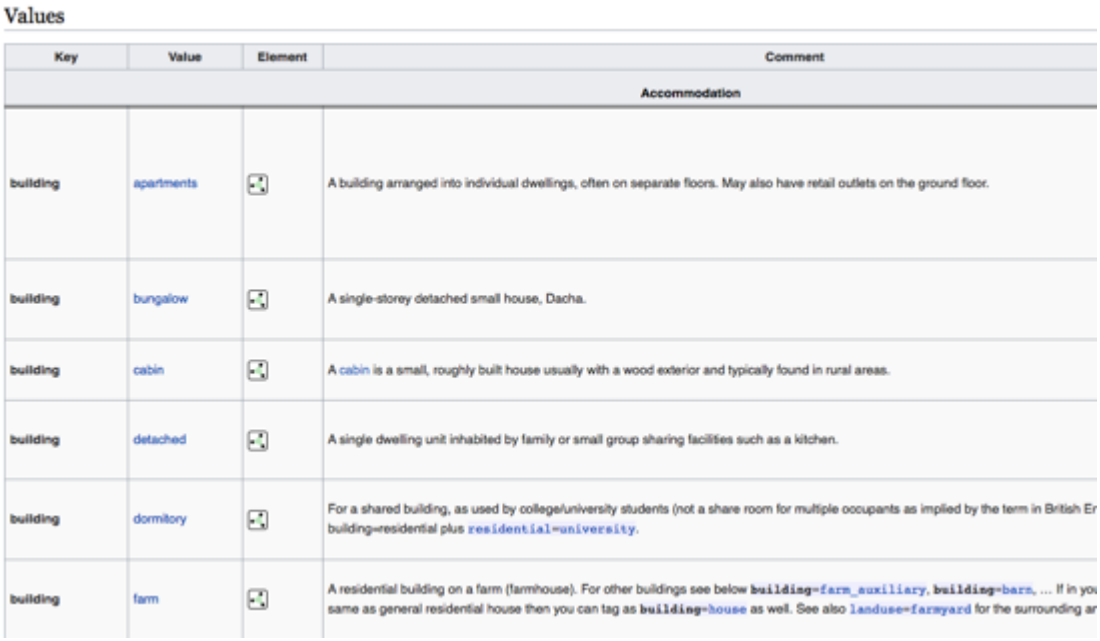
Fig 4.8 – *Ordnance Survey AddressBase Schema Breakdown*

Strategy 1 of the TF recommendations from Figure 4.10 suggest a more granular approach to labeling as one possible approach. We employed Label Set runs along these lines, but these were found to not prove successful due to the limited sample sizes. Also , early exploratory CNN Training and Classification runs were attempted with different variations of Address base Label, but proved consistently inaccurate. Please See notes on the Auxiliary Descriptor runs below.

As we might expect AddressBase classes do not necessarily translate well into Machine Learner friendly Classification Labels. For the purposes of a CNN Classification Task we are primarily concerned with finding and adopting a Visual Classification schema, one that can define and differentiate Building Features and Types based on Visual Appearance alone.

Open Street Map Schema:

Open Street Map provide a building description schema, which at its most basic is also concerned with a building's usage. However the value and schema also lend themselves to categorization by type and crucially of visual type that better discriminate between building objects with clearly delineated and individual visual characteristics, and so would prove more ML friendly.



Values

Key	Value	Element	Comment
			Accommodation
building	apartments		A building arranged into individual dwellings, often on separate floors. May also have retail outlets on the ground floor.
building	bungalow		A single-storey detached small house, Dacha.
building	cabin		A cabin is a small, roughly built house usually with a wood exterior and typically found in rural areas.
building	detached		A single dwelling unit inhabited by family or small group sharing facilities such as a kitchen.
building	dormitory		For a shared building, as used by college/university students (not a share room for multiple occupants as implied by the term in British Er building=residential plus residential=university.
building	farm		A residential building on a farm (farmhouse). For other buildings see below building=farm_auxiliary, building=barn, ... If in yo same as general residential house then you can tag as building=house as well. See also landuse=farmyard for the surrounding ar

Fig 4.9 – *Schema Description examples of the Open Street Map Schema Building Type Layer*

Building class descriptions from OpenStreetMap.

Apartment	A building arranged into individual dwellings, often on separate floors. May also have retail outlets on the ground floor
Church	A building that was built as a church
Garage	A building suitable for the storage of one or possibly more motor vehicle or similar
House	A dwelling unit inhabited by a single household (a family or small group sharing facilities such as a kitchen)
Industrial	A building where some industrial process takes place
Office building	A building where non-specific commercial activities take place
Retail	A building primarily used for selling goods that are sold to the public
Roof	A structure that consists of a roof with open sides, such as a rain shelter, and also gas stations

Fig 4.10 – *Buildings as Visually distinct Objects, as Provided by the OSM Building Type Schema Descriptions*

Whilst the Open Street Map Building Layer Classification schema is also contain Use Class definitions, it also provides a descriptive layer feature schema based on Building Types.

The first aim of the study is to train a CNN to identify Building Types, so we will focus on Mapping AddressBase classes to the appropriate Open Street Map Building Type categories. As a developing open source initiative, much of this data is yet to be populated. We therefore need to adopt a criteria and working methodology for assigning class members to classes of Building Types. In this instance, we adopt a Fuzzy Logic approach, whereby set membership is discerned by applying a measure of the degree of potential set membership characteristics a given set member candidate is seen to display.

4.10 Applying Crisp and Fuzzy Set approaches to Data Filtering:

We will see that the House(Detached and Semi Detached) label Class is both numerous and relatively visually distinct. This will be less so for the problematic Terraces and Metropolitan High street scenes.

Our other Building categories, esp. in an urban setting with a strong mixed use development and urban density focused planning policy in play, are not so easy to categorize for the purposes of a CNN classification task.

We will apply two approaches to this problem:

1 - Apply Filters to the Generic Building Class by adopting the development scale definitions used in UK Town Planning i.e. Major and Minor Developments. This should hopefully provide us with candidate members that possess strong characteristics of the feature class(e.g. Large Block of Modern New Build Flat)

2 - Where these categories are not successful we also apply a Fuzzy Set Membership approach. We select a parameter of choice (e.g. Floor Space, Build Unit Count) that allows for a greater or lesser degree of class membership for the candidate set member. We consider and adjust the Fuzzy Variable Control until we have a satisfactory cut off point and a label that might aid the CNN Building Type Classification Task.

As can be seen from the above, we now have 3 Variations of Flat Class Definition with which to fine tune and determine the greatest recall accuracy with the Inception CNN Model.

The latter 2 categories focus on scale of development, as outlined, for example, in The Town and Country Planning (Development Procedure) (England) Order 2010 Management. Criteria for Developments to be considered in the Major Category include *10+ dwellings / over half a hectare / building(s) exceeds 1000m²*. Minor Dwellings are therefore *1-9 dwellings (unless floorspace exceeds 1000m² / under half a hectare*. Ultimately we are seeking to differentiate between Large Blocks of Flats and smaller Building development Object Types.

Note that we have no properties in the Flat Conversion Category, which accords with our study's focus on New Build Development. These have been excluded as part of the Data Model Build Stage. Please note that our rather clumsy pattern matching may inadvertently also be excluding conversions that are relate not just to single flat, e.g. conversion of offices B1a to C3 residential use(See the Town and Country Planning (General Permitted Development) (England) Order 2015)

Our Set Controls in this category are provided by text from the proposal field and Residential Unit Counts, both provided by the LDD

Key Links

Address Base	https://www.geoplace.co.uk/addresses/national-address-gazetteer/-/asset_publisher/TVLU5AZVOOPK/content/what-is-the-local-land-and-property-gazetteer-llpg-
BS7666 compliancy	https://www.agi.org.uk/agi-groups/standards-committee/bs7666-guidelines
Open Street Map	https://wiki.openstreetmap.org/wiki/Key:building

4.11 Image Buckets for Local Implementation Staging

Having established a criteria and method for creating and populating potential label sets with set members for the Machine Learning Labeling process, we proceed to programmatically implement and apply this workflow to our training labels of interest.

Figures 4.15 and 4.16 illustrate the label selection and image bucket creation(using *shutils* and *os* python libraries) stages in the Jupyter Script Implementation. Figure 4.17 overleaf, is a snapshot of the resultant image label archive.

Fig 4.11 – *Load Label Selection*

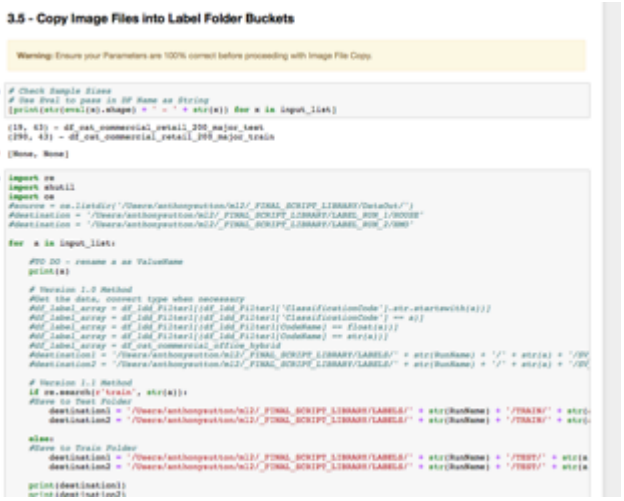


Fig 4.12 – *Create Image Buckets*

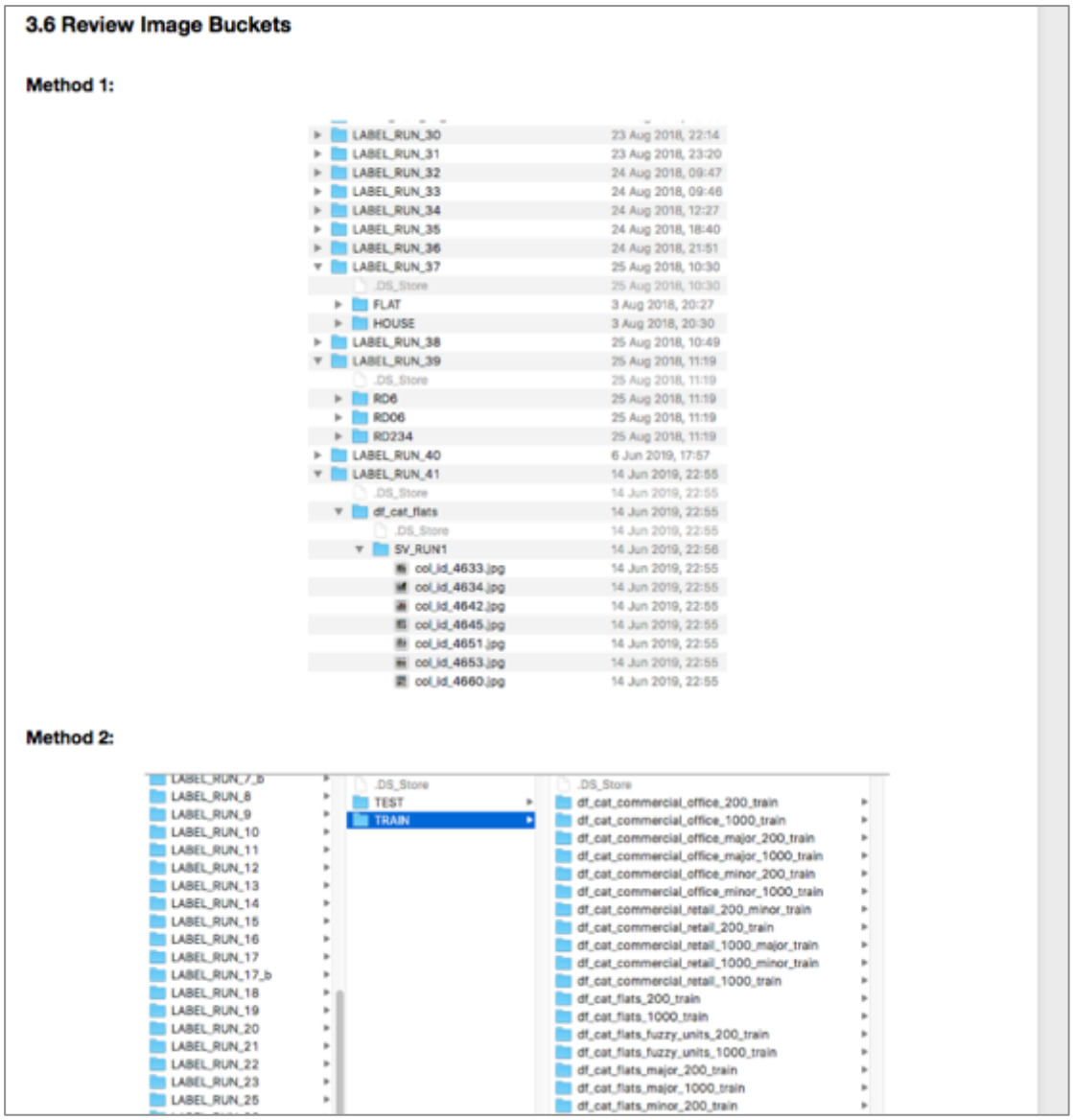


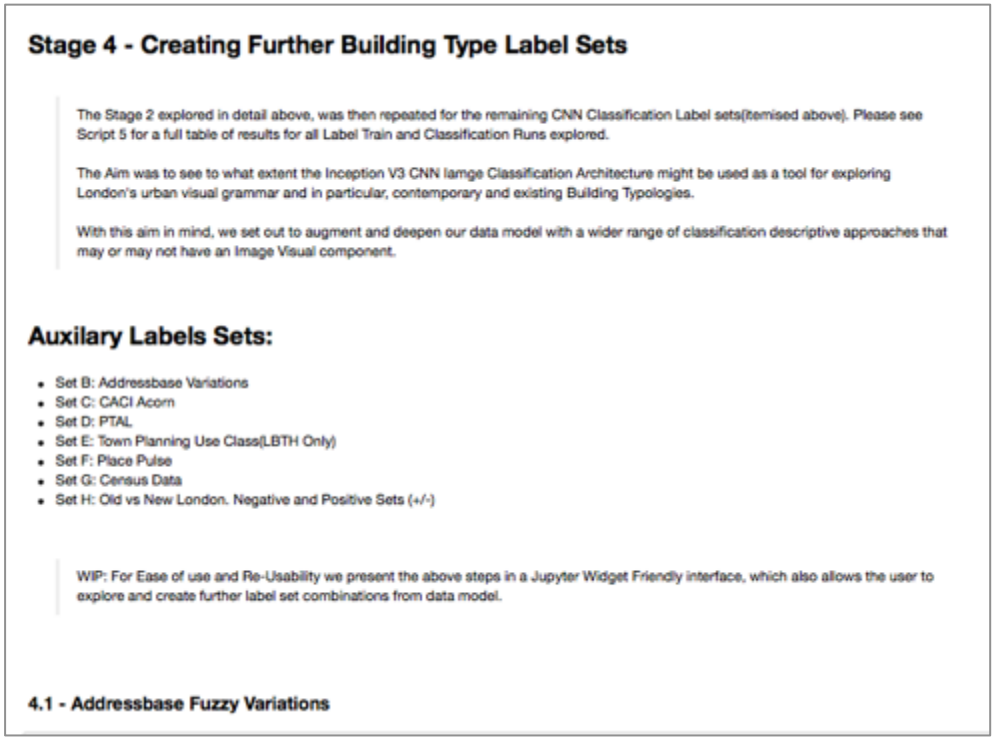
Fig 4.13 – *Snapshot of the Local Machine Image Buckets created during the Label Creation Workflow Stage*

4.12 Auxiliary Descriptors

As mentioned earlier, one of the aims of the project was to create a Jupyter Implementation with helper tools that would allow for generic reuse of the workflow.

The procedure outlined above was then applied to the auxiliary descriptive elements of our visual and descriptive data model of New London, that allows to plug in new feature types, recreate benchmark study training datasets and to experiment with a wide range of possible descriptive labeling approaches, in our motivation to explore large scale image CNN analysis.

Please see metric log table section in the CNN Trainer Section for details on how they performed and compared.



Stage 4 - Creating Further Building Type Label Sets

The Stage 2 explored in detail above, was then repeated for the remaining CNN Classification Label sets(itemised above). Please see Script 5 for a full table of results for all Label Train and Classification Runs explored.

The Aim was to see to what extent the Inception V3 CNN lamge Classification Architecture might be used as a tool for exploring London's urban visual grammar and in particular, contemporary and existing Building Typologies.

With this aim in mind, we set out to augment and deepen our data model with a wider range of classification descriptive approaches that may or may not have an Image Visual component.

Auxilary Labels Sets:

- Set B: Addressbase Variations
- Set C: CACI Acorn
- Set D: PTAL
- Set E: Town Planning Use Class(LBTH Only)
- Set F: Place Pulse
- Set G: Census Data
- Set H: Old vs New London. Negative and Positive Sets (+/-)

WIP: For Ease of use and Re-Usability we present the above steps in a Jupyter Widget Friendly interface, which also allows the user to explore and create further label set combinations from data model.

4.1 - Addressbase Fuzzy Variations

Fig 4.14 – *Examples of Auxiliary Descriptor deployed.*

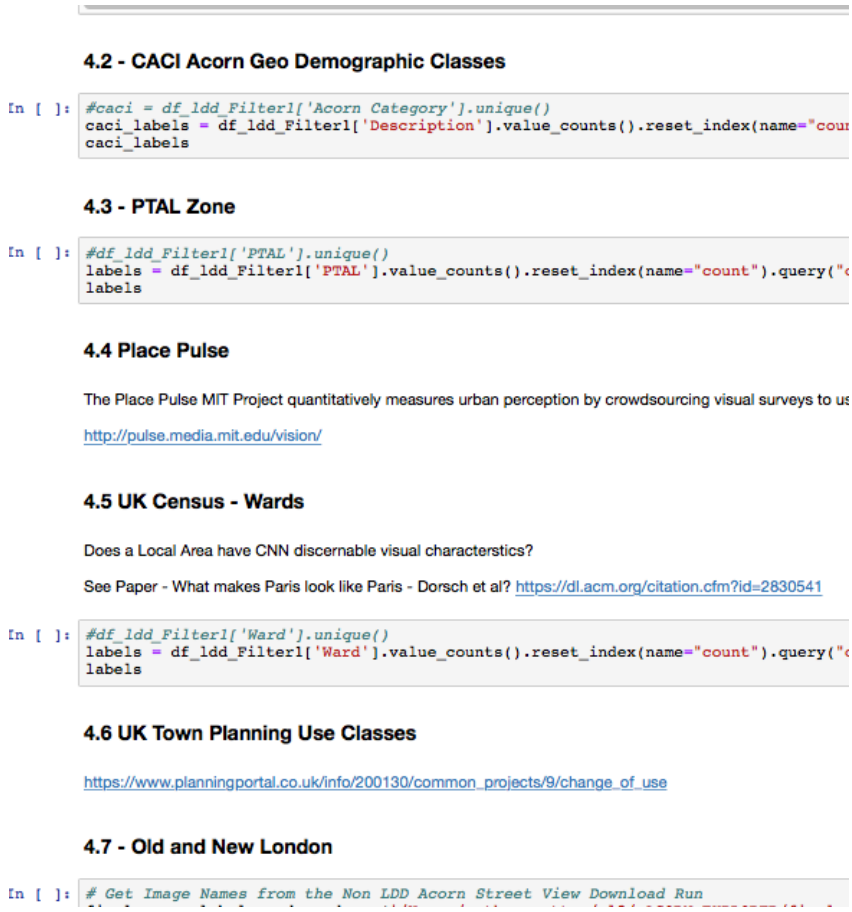


Fig 4.15 – *Examples of Auxiliary Descriptor deployed and in development.*

Stage 5: CNN Training Stage

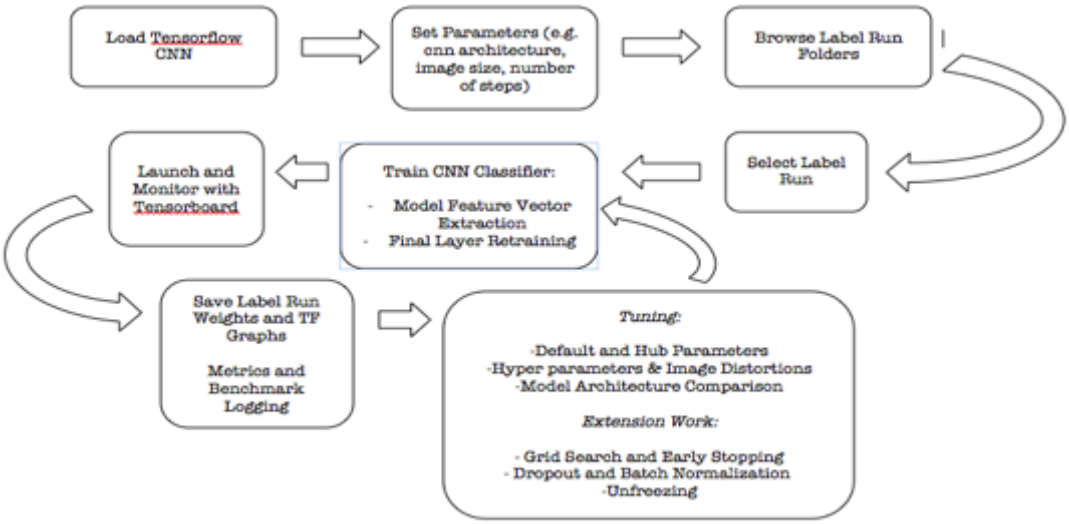


Fig 5.1 *– CNN Training Workflow Stage Overview*

5.1 Stage Overview

- Script to facilitate workflow staging of label set train runs
- Handles Parameter Inputs and Outputs to and from the Low Level TF Training Script
- Includes Misclassified Image Gallery Inspection and other Utilities to assist in large scale batch image classification tasks

5.2 Transfer Learning

Feature Extraction uses the representations learned by a previous network. We add a new classifier, which will be trained from scratch, on top of the pretrained model to repurpose the feature maps (feature vectors) learned previously for our New dataset.

The top layer receives as input a 2048-dimensional vector (assuming Inception V3) for each image. In this script implementation (which uses the low level TensorFlow ver 1.8 api rather than for example keras or eager learning or pre made estimators)) a SoftMax layer(other options might be to...) has been trained on top of this representation. If the SoftMax layer contains N

labels, this corresponds to learning N + 2048* N model parameters for the biases and weights.

Transfer learning can be used to train a model with a secondary & smaller dataset(to improve generalization) and to speed up the training stage.

In our case the models we retrain several model architectures that have been retrained on the ImageNet dataset from the ImageNet Large Scale Visual Recognition Challenge(ILSVRC) - described in further detail later on in this section.

5.3 Bottlenecks

'Bottleneck' is a term used by the Tensorflow api for the layer just before the final output layer that does the classification. (TensorFlow Hub calls this an "image feature vector".) This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize.

Because every image is reused multiple times during training and calculating each bottleneck takes a significant amount of time to generate. By default they're stored in the /tmp/bottleneck directory, and can be reused on subsequent runs(I.e that have will have with same default parameters and architecture) so you don't have to wait for this part again. During the workflow design stage we moved this directory away from the tmp cache, so that we had a benchmarking archive of bottlenecks. Our Jupyter script dynamically manages the file-naming conventions of architecture, label set run names, datetime and parameter setting for each Training run.

All performance results (accuracy, speed and benchmarking) are saved in the metric log tables, found at the end of this chapter.

In subsequent Scripts we will re-use the feature maps generated in the training stages as the source data on which to apply further algorithmic processing for image analysis and categorization.

We use Training Scripts obtained from the TensorFlow online learning repository. These are designed to work with Tensorhub which allows for Pretrained Conv Net architectures to be shared. By in large they are written using the TF-Slim implementation of TensorFlow. TF Slim makes it easy to extend complex models, and to warm start training algorithms by using pieces of pre-existing model checkpoints.

5.4 TF Graphs

Tensoflow Graphs are based on the Data flow programming paradigm, commonly used for parallel computing model. In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. For example, in a TensorFlow graph, the tf.matmul operation would correspond to a single node with two incoming edges (the matrices to be multiplied) and one outgoing edge (the result of the multiplication).

By using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel. By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.

TensorFlow's XLA compiler can use the information in your dataflow graph to generate faster code, for example, by fusing together adjacent operations.

The dataflow graph is a language-independent representation of the code in your model. You can build a dataflow graph in Python, store it in a SavedModel, and restore it in a C++ program for low-latency inference.

Tensorboard visualization of the Tensorflow graph for the MobielNet Model implementation using a 224 px input Image resolution and .50 as the relative size of the model as a fraction of the largest MobileNet.

Inception(MobileNet version mobilenet_v2_100_224/feature_vector

https://machinethink.net/blog/mobilenet-v2/

***Fig 5.2** – TF Dataflow graph of MobileNet V2 with the last layer retrained on LDD Building Types*

A typical TensorFlow graph---especially training graphs with automatically computed gradients---has too many nodes to visualize at once. The graph visualizer makes use of name scopes to group related operations into "super" nodes. You can click on the orange "+" button on any of these super nodes to expand the subgraph inside.

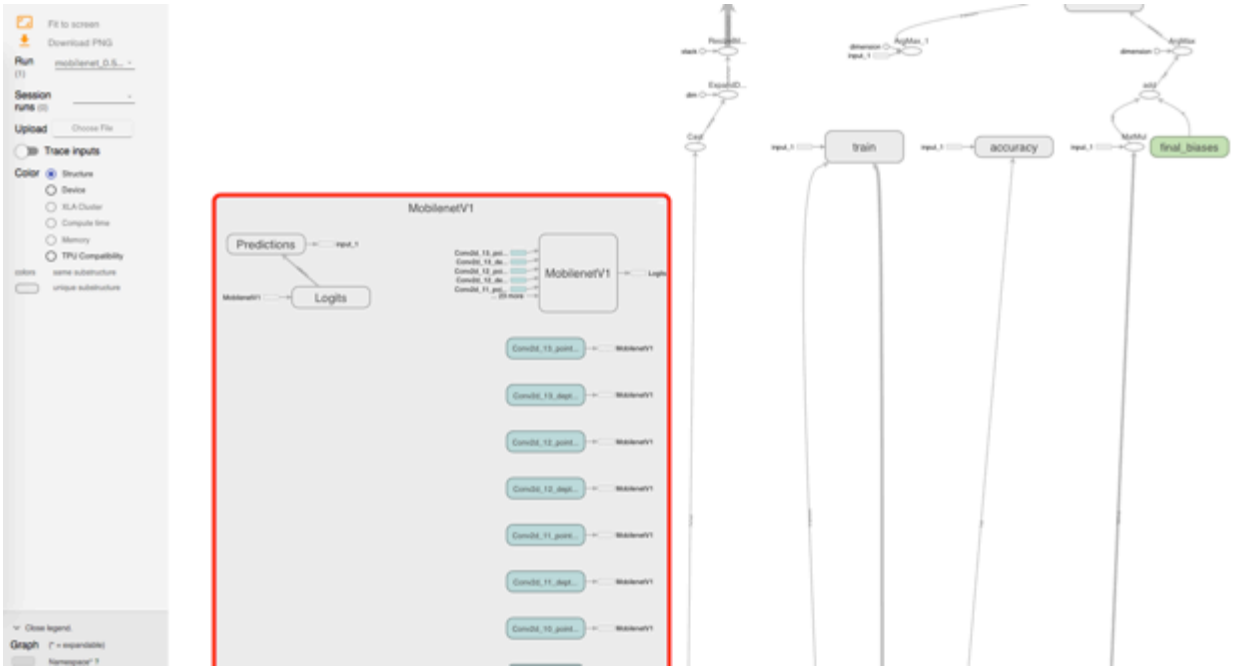


Fig 5.3 *– Clicking Through into the Tensorboard Graph Visualisation of Mobilenet Node reveals its 13 Layer Design. Our inline Confusion Table Modifications to the tf.image lib can be seen on the right.*

5.5 Tensorhub

TensorFlow Hub is a library for the publication, discovery, and consumption of reusable parts of machine learning models. A module is a self-contained piece of a TensorFlow graph, along with its weights and assets, that can be reused across different tasks in a process known as transfer learning.

The collection of pre trained Image Classification models come in two forms, as feature vector and as image classification modules. We use the former to generate vector extract files of the pre trained weights and the latter for carrying out a straightforward image classification task using the full layers and weights of the original CNN Model.

In Pytorch, as we have seen in our Image Preprocessing workflow stage, we use the Caffenet Library of pre trained CNN models.

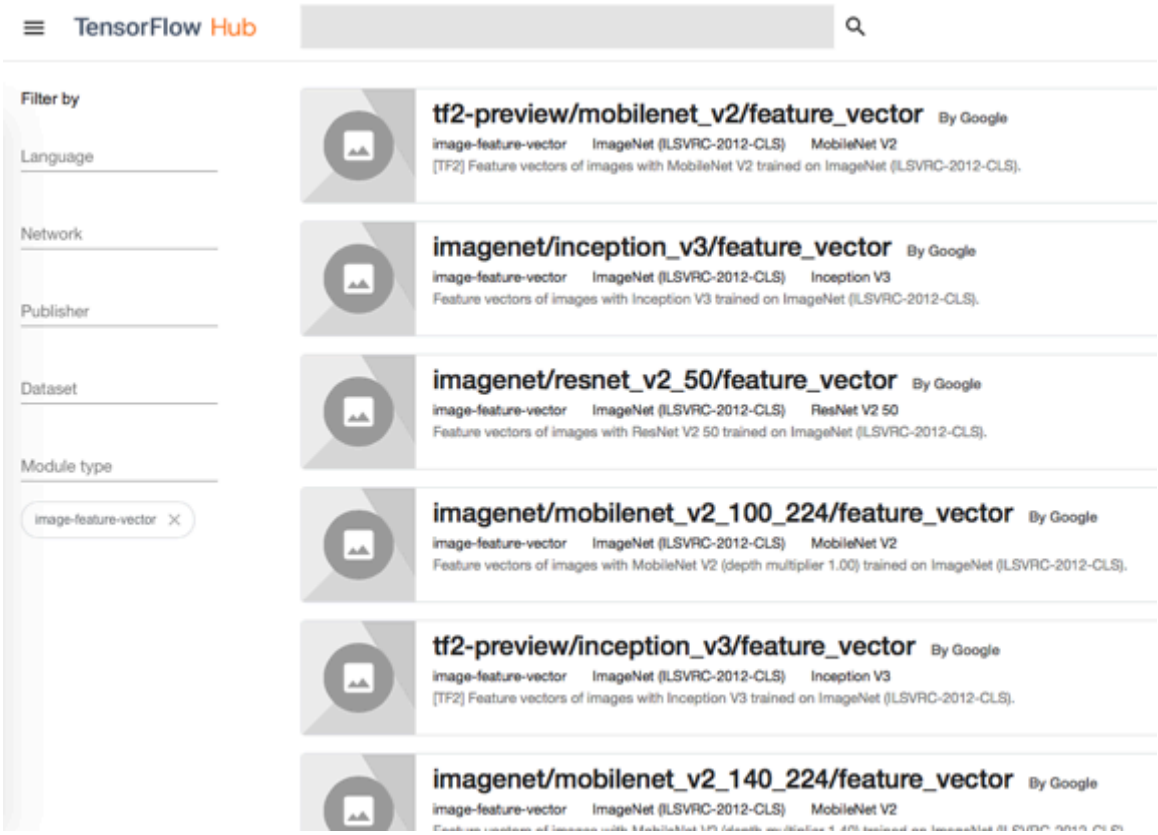


Fig 5.4 *– Tensorflow Hub Modules*

Check the Usage Documentation on TF Hub such as Signature(eg Vector vs Classification and Inputs and Outputs). For instance, num_features and height x width params might need changing, depending on how the original model was built(eg where size of image is important to architecture in question's design aim e.g. speed vs accuracy). The original TF scripts don't accommodate this change, so we must edit the input and output parameters of the script ourselves.

The Usage Docs also recommend specific areas for Fine Tuning: https://tfhub.dev/google/imagenet/nasnet_mobile/feature_vector/3 -.

Fine Tuning such as dropout or batch normalization, are generally used during the latter stages of the model optimization process, once the baseline performance has been achieved and fine tuning is sought after. This level of tuning will be relegated to the Research Extensions, where we rely on on the higher Level Keras api.

e.g.

Fig 5.5 *– TF Hub Module Signature Usage*

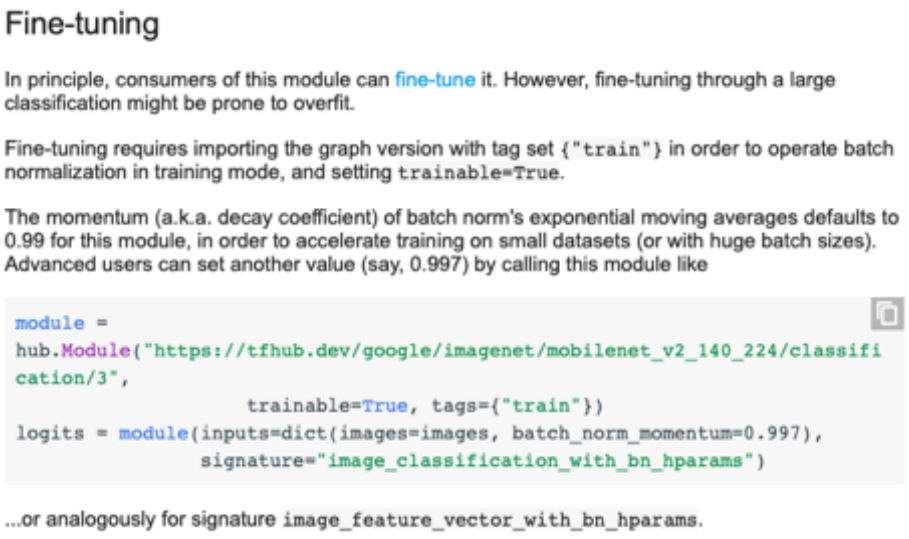


Fig 5.6 *– TF Hub Module Fine Tuning Details*

5.6 Default Architecture - Inception and Mobile Net

Throughout the main implementationworkflow we used model variants on the Inception architecture. Other models were used during benchmarking stages, and can be found documented later in on the chapter.

The Inception v3 CNN Model

"Neurons that fire together, Wire Together" - Hebbes

Successor to GoogLeNet, posted a breakthrough performance on the ImageNet Visual Recognition Challenge(2014)

Rather than building bigger(prone to over fitting and increase in computiational resource requirement):CNN 27 Layers inc Inception Layer

Inception Layer: Allows internal layers to choose filter size relevant to learning the required information. Allows to focus on the lexical Subject of an image, avoids learning noise.

Mobilenets come in various sizes controlled by a multiplier for the depth (number of features) in the convolutional layers. They can also be trained for various sizes of input images to control inference speed. This TF-Hub module uses the TF-Slim implementation of mobilenet_v1 with a depth multiplier of 1.0 and an input size of 224x224 pixels.

5.7 Tensorboard

The script includes TensorBoard summaries that make it easier to understand, debug, and optimize the retraining. For example, you can visualize the graph and statistics, such as how the weights or accuracy varied during training.

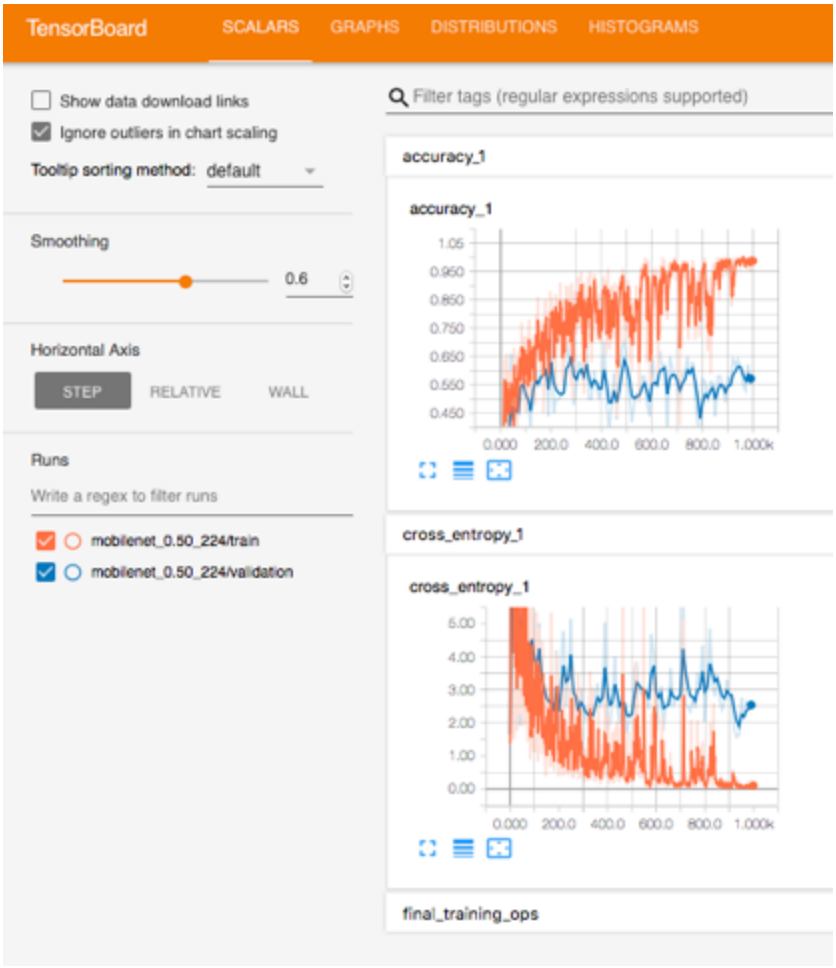


Fig 5.7 *– Tensorboard Menus*

If we compare the validation and training loss and gain of the respective entropy and accuracy curves , we can see the model is probably over fitting. The smoothing tool on the left can be used to remove noise from the visual overview.

5.8 Script Breakdown

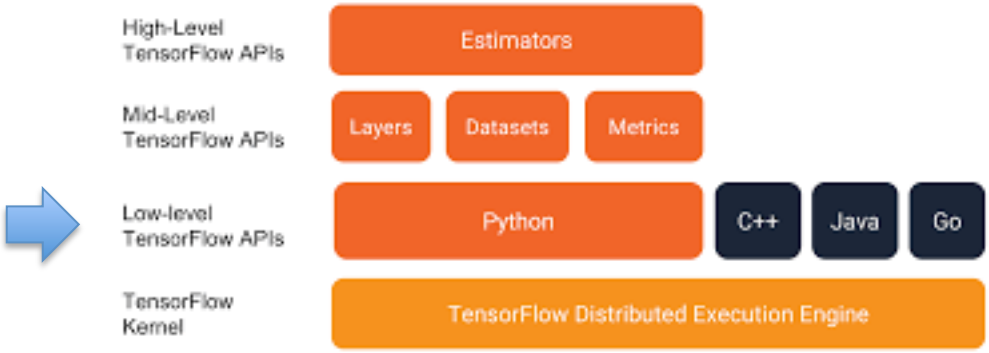


Fig 5.8 *– The Script uses the Python Low Level Tensorflow API*

The script will write out the new model trained on our new Building Type categories to /tmp/output_graph.pb, and a text file containing the labels to /tmp/output_labels.txt. The new model contains both the TF-Hub module inlined into it, and the new classificiation layer

Brief Tour of Script Functions:

Fig 5.9 *– Script Function List. Load Model > File Type Decoding and Encoding>Create Bottlenecks> Run Data Augmentation> Train Final Layer, Log Training Step Events> Run Final Evalutation and Export Model*

```python
def add_final_retrain_ops(class_count, final_tensor_name, bottleneck_tensor,
                          quantize_layer, is_training):
  """Adds a new softmax and fully-connected layer for training and eval.

  We need to retrain the top layer to identify our new classes, so this function
  adds the right operations to the graph, along with some variables to hold the
  weights, and then sets up all the gradients for the backward pass.

  The set up for the softmax and fully-connected layers is based on:
  https://www.tensorflow.org/tutorials/mnist/beginners/index.html

  Args:
    class_count: Integer of how many categories of things we're trying to
        recognize.
    final_tensor_name: Name string for the new final node that produces results.
    bottleneck_tensor: The output of the main CNN graph.
    quantize_layer: Boolean, specifying whether the newly added layer should be
        instrumented for quantization with TF-Lite.
    is_training: Boolean, specifying whether the newly add layer is for training
        or eval.

  Returns:
    The tensors for the training and cross entropy results, and tensors for the
    bottleneck input and ground truth input.
  """
  batch_size, bottleneck_tensor_size = bottleneck_tensor.get_shape().as_list()
  assert batch_size is None, 'We want to work with arbitrary batch size.'
  with tf.name_scope('input'):
    bottleneck_input = tf.placeholder_with_default(
        bottleneck_tensor,
        shape=[batch_size, bottleneck_tensor_size],
        name='BottleneckInputPlaceholder')

    ground_truth_input = tf.placeholder(
        tf.int64, [batch_size], name='GroundTruthInput')

  # Organizing the following ops so they are easier to see in TensorBoard.
  layer_name = 'final_retrain_ops'
```

Fig 5.10 *– The add Final Retrain ops function, which retrains the final Layer of the pre trained transferred model weights*

5.9 Model Training

Here's an example of how to run the label_image example. By convention, all TensorFlow Hub modules accept image inputs with color values in the fixed range [0,1], so you do not need to set the --input_mean or --input_std flags.

```
curl -LO https://github.com/tensorflow/tensorflow/raw/master/tensorflow/examples/label_image/label_image.p
python label_image.py \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--input_layer=Placeholder \
--output_layer=final_result \
--image=$HOME/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

Fig 5.21 *– Script Input Parameters*

To retrain onto new categories we point to a set of sub-folders, each named after one of the new categories and containing only images from that category

Once the bottlenecks are complete we examine step outputs:

Training Accuracy = percent of correct class

Validation Accuracy = performance measure not contained in the training

data. Watch out for overfitting on Training Data

Cross Entropy = loss function/how well the learning process is progressing. The training's objective is to make the loss as small as possible, so you can tell if the learning is working by keeping an eye on whether the loss keeps trending downwards (see Tensorboard), ignoring the short-term noise.

Steps: Default =4000 Max =8000

Epoch: Epoch is considered as number of one pass from entire dataset

Steps: In TensorFlow one step = number of epochs multiplied by examples divided by batch size

Each step chooses ten images at random from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions. Those predictions are then compared against the actual labels to update the final layer's weights through the back-propagation process.

As the process continues we are looking for an improvement, and after all the steps are done, a final test accuracy evaluation is run on a set of images kept separate from the training and validation pictures.

Final Test Accuracy evaluation = Percentage of correctly labels on Test Set.

Varying Step Rate: The rate of improvement in the accuracy slows the longer you train for, and at some point will stop altogether (or even go down due to overfitting), but you can experiment to see what works best for your model.

5.10 Image Distortions (aka Data Augmentation)

A common way of improving the results of image training is by deforming, cropping, or brightening the training inputs in random ways. This has the advantage of expanding the effective size of the training data thanks to all the possible variations of the same images, and tends to help the network learn to cope with all the distortions that will occur in real-life uses of the classifier. The biggest disadvantage of enabling these distortions in our script is that the bottleneck caching is no longer useful, since input images are never reused exactly. This means the training process takes a lot longer (many hours), so it's recommended you try this as a way of polishing your model only after you have one that you're reasonably happy with.

You enable these distortions by passing **--random_crop, --random_scale** and **--random_brightness** to the script. These are all percentage values that control how much of each of the distortions is applied to each image. It's

reasonable to start with values of 5 or 10 for each of them and then experiment to see which of them help with your application. **--flip_left_right** will randomly mirror half of the images horizontally, which makes sense as long as those inversions are likely to happen in your application. For example it wouldn't be a good idea if you were trying to recognize letters, since flipping them destroys their meaning.

5.11 Hyper Parameters

There are several other parameters you can try adjusting to see if they help your results. The --**learning_rate** controls the magnitude of the updates to the final layer during training. Intuitively if this is smaller than the learning will take longer, but it can end up helping the overall precision. That's not always the case though, so you need to experiment carefully to see what works for your case. The **--train_batch_size** controls how many images are examined during each training step to estimate the updates to the final layer.

Learning Rate:

optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation.

The amount that the weights are updated during training is referred to as the step size or the “learning rate.”

Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

This means that a learning rate of 0.1, a traditionally common default value, would mean that weights in the network are updated 0.1 * (estimated weight error) or 10% of the estimated weight error each time the weights are updated.

5.12 Overfitting

Over Fitting

Overfitting occurs when a model fits the data in the training set well, while incurring larger generalization error. To prevent overfitting, the best solution is to use more training data. A model trained on more data will naturally generalize better. When that is no longer possible, the next best solution is to use techniques like regularization(e.g. weight regularisation and drop out). These place constraints on the quantity and type of information your model can store. If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

Prevention:
Cross Validation
More Data
Remove Features
Early Stopping
Regluarixxation
Ensembling(Bagging and Boosting)

- See Classification Stage for More Detail on the above

5.13 User Modifications

- Inline Confusion Matrices and Timer
- Small performance (speed) overhead
- Occasional Anomalous behaviors observed

Fig 5.11 *– Our Inline Confusion Matrice modification accessed via the Images Menu*

5.14 Other Model Architectures

NasNet: Combo of ImageNet and Coco. Dsigner used AutoML method for automated optimal design involving this combination of Image Classification and Object Detection datasets.

QuantNet: Quanitization reduced precision representations of weights. Offer prospect of greater perfomance of models. nference efficiency is particularly important for edge devices, such as mobile and Internet of Things (IoT). Such devices have many restrictions on processing, memory, power-consumption, and storage for models. Interestingly, we found these models to offer greater out of the box perfomance.

integer quantization enables users to take an already-trained floating-point model and fully quantize it to only use 8-bit signed integers (i.e. `int8`). By leveraging this quantization scheme, we can get reasonable quantized model accuracy across many models without resorting to retraining a model with quantization-aware training. With this new tool, models will continue to be 4x smaller, but will see even greater CPU speed-ups.

Related to our Transfer Learning Scenario being problematic see Do Better ImageNet Models Transfer Better? Kornblith et al?

MobNet Neuron Number Variations: Mobilenets come in various sizes controlled by a multiplier for the depth (number of features) in the convolutional layers. They can also be trained for various sizes of input images to control inference speed e.g. Mobilenet_v2_**140**_224 or Mobilenet_v2_**35**_224

ResNet: or Deep Residual Network Solution to the degradation/saturation of accuracy problem (more layers = higher inaccuracy) --> shorcut connections(feeding a layer with the output of previous successive convolutional layers and the respective bypassed input.), allowed 1000s(ie more) layers could be added.

Mobnet ImageNet Classify trick: Mobilenet V2 does not apply the feature depth percentage to the bottleneck layer. Mobilenet V1 did, which made the job of the classification layer harder for small depths. Would it help to cheat and use the scores for the original 1001 ImageNet classes instead of tight bottleneck? You can simply try by replacing *mobilenet_v1.../feature_vectorwith mobilenet_v1.../classification* in the module name

PNasNEt

Other Model Architectures By default the script uses an image feature extraction module with a pretrained instance of the Inception V3 architecture.

This was a good place to start because it provides high accuracy results with moderate running time for the retraining script. But now let's take a look at further options of a TensorFlow Hub module.

On the one hand, that list shows more recent, powerful architectures, such as NASNet (notably nasnet_large and pnasnet_large), which could give you some extra precision.

On the other hand, if you intend to deploy your model on mobile devices or other resource-constrained environments, you may want to trade a little accuracy for much smaller file sizes or faster speeds (also in training). For that, try the different modules implementing the MobileNet V1 or V2 architectures, or also nasnet_mobile.

Training with a different module is easy: Just pass the --tfhub_module flag with the module URL, for example:

This will create a 9 MB model file in /tmp/output_graph.pb with a model that uses the baseline version of MobileNet V2. Opening the module URL in a browser will take you to the module documentation.

If you just want to make it a little faster, you can reduce the size of input images (the second number) from '224' down to '192', '160', or '128' pixels squared, or even '96' (for V2 only). For more aggressive savings, you can choose percentages (the first number) '100', '075', '050', or '035' (that's '025' for V1) to control the "feature depth" or number of neurons per position. The number of weights (and hence the file size and speed) shrinks with the square of that fraction. The MobileNet V1 blogpost and MobileNet V2 page on GitHub report on the respective tradeoffs for Imagenet classification.

Mobilenet V2 does not apply the feature depth percentage to the bottleneck layer. Mobilenet V1 did, which made the job of the classification layer harder for small depths. Would it help to cheat and use the scores for the original 1001 ImageNet classes instead of tight bottleneck? You can simply try by replacing mobilenet_v1.../feature_vector with mobilenet_v1.../classification in the module name.

As before, you can use all of the retrained models with label_image.py. You will need to specify the image size that your model expects, for example:

```
python label_image.py \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--input_layer=Placeholder \
--output_layer=final_result \
--input_height=224 --input_width=224 \
--image=$HOME/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

Full List of Model Architectures available on the hub: https://tfhub.dev/s?module-type=image-feature-vector

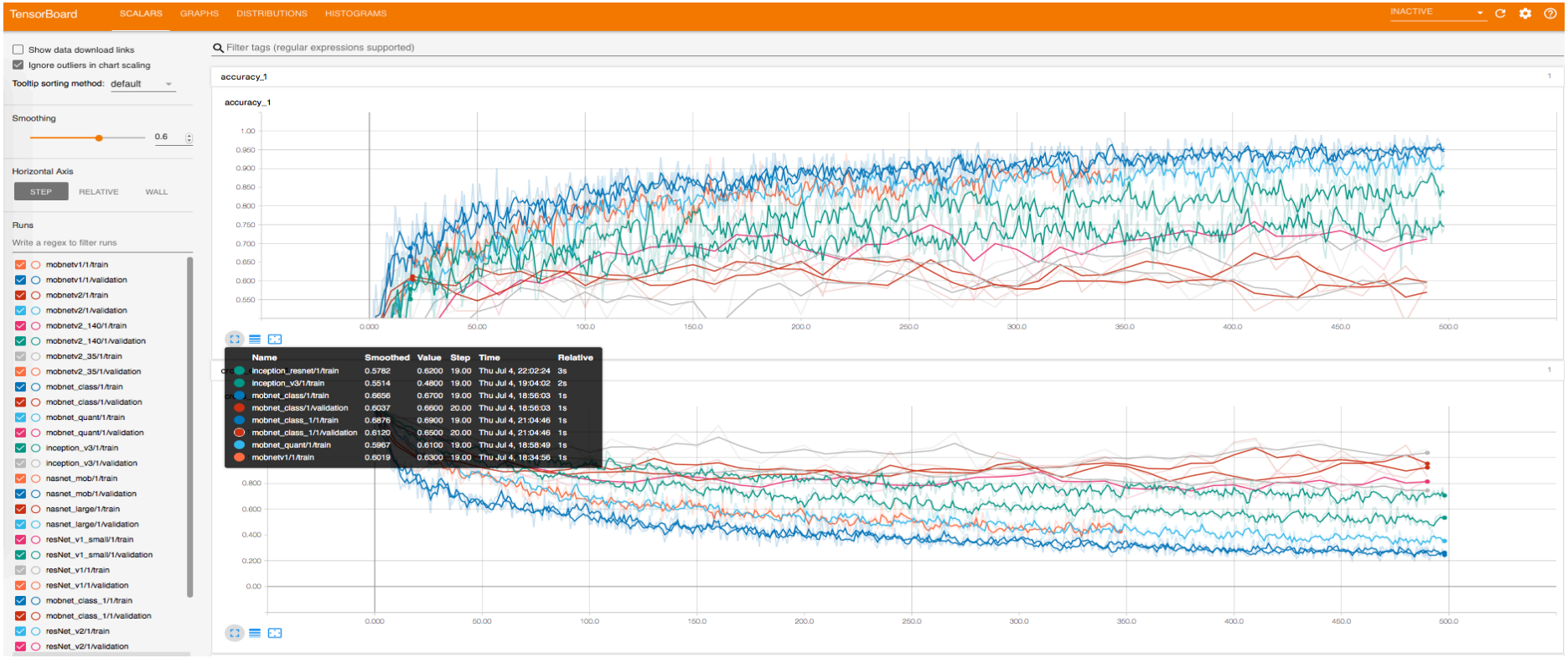


Fig 5.13 *– Architecture Performance Comparison used in our Metric Analysis and Logging Workflow on one of the Best Fit label set Training Runs().*

Signature: The signature of the module specifies what is the purpose for which module is being used for. All the module, comes with the ‘default’ signature and makes use of it, if a signature is not explicitly mentioned. For most of the modules, when ‘default’ signature is used, the internal layers of the model is abstracted from the user. The function used to list out all the signature names of the module is get_signature_names().

Each of the module has some set of expected inputs depending upon the signature of the module being used. Though most of the modules have documented the set of expected inputs in TensorFlow Hub website (particularly for ‘default’ signature), some of them haven’t. In this case, it is easier to obtain the expected input along with it’s size and datatype using the get_input_info_dict().

Expected outputs: In order to build the remaining part of the graph after the TensorFlow Hub’s model is built, it is necessary to know the expected type of output. get_output_info_dict() function is used for this purpose. Note that for ‘default’ signature, there will be usually only one output, but when you use a non-default signature, multiple layers of the graph will be exposed to you.

Quantization of model weights (mobQuant Versions)

Quantization as a size optimization method. This is especially important for on-device applications, where the memory size and number of computations are necessarily limited. Quantization, in a deep learning context, is the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

5.15 Running the Jupyter CNN Training Workflow Steps

Step 1: Start up Virtual Machine and Command Prompts for launching VM instances(Pytorch Requires Python 2 VM Instance), Tensorboard Instances(each train run requires the TBoard server to be shutdown and restarted unless bottlenecks are being re-used) and additional workflow Utilities

Step 2: Launch Custom CNN Train Script in Jupyter

Step 3: Visual Checks on Label Set and Summary OutputRepositories(saved model protobuff, bottlenecks and tensorboard event log)

Step 4: Select Label Set as Input, Training Summary Folder as Output

Step 5: Run and Monitor Training Script, Launch Tensorboard

Step 6: Inspect Training Run Metrics

Step 7: Quick Inspection of Misclassified Images

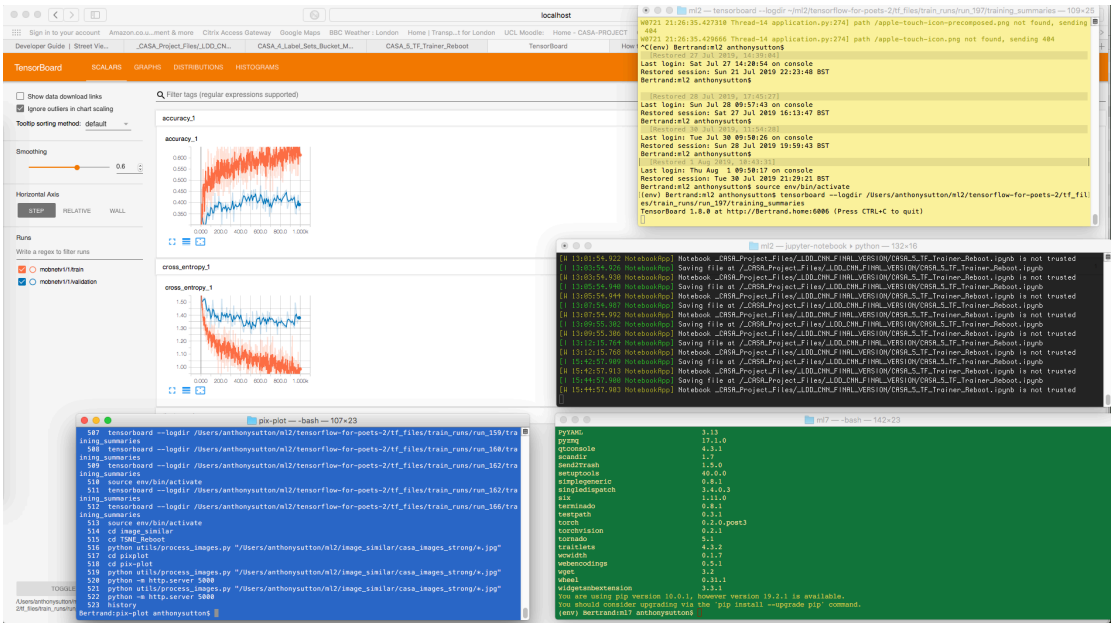


Fig 5.14a *– Typical Console setup during workflow runtime. screens used for jupyter, tensorboard, other python version virtual machiens and for running the 3JS server stage.*

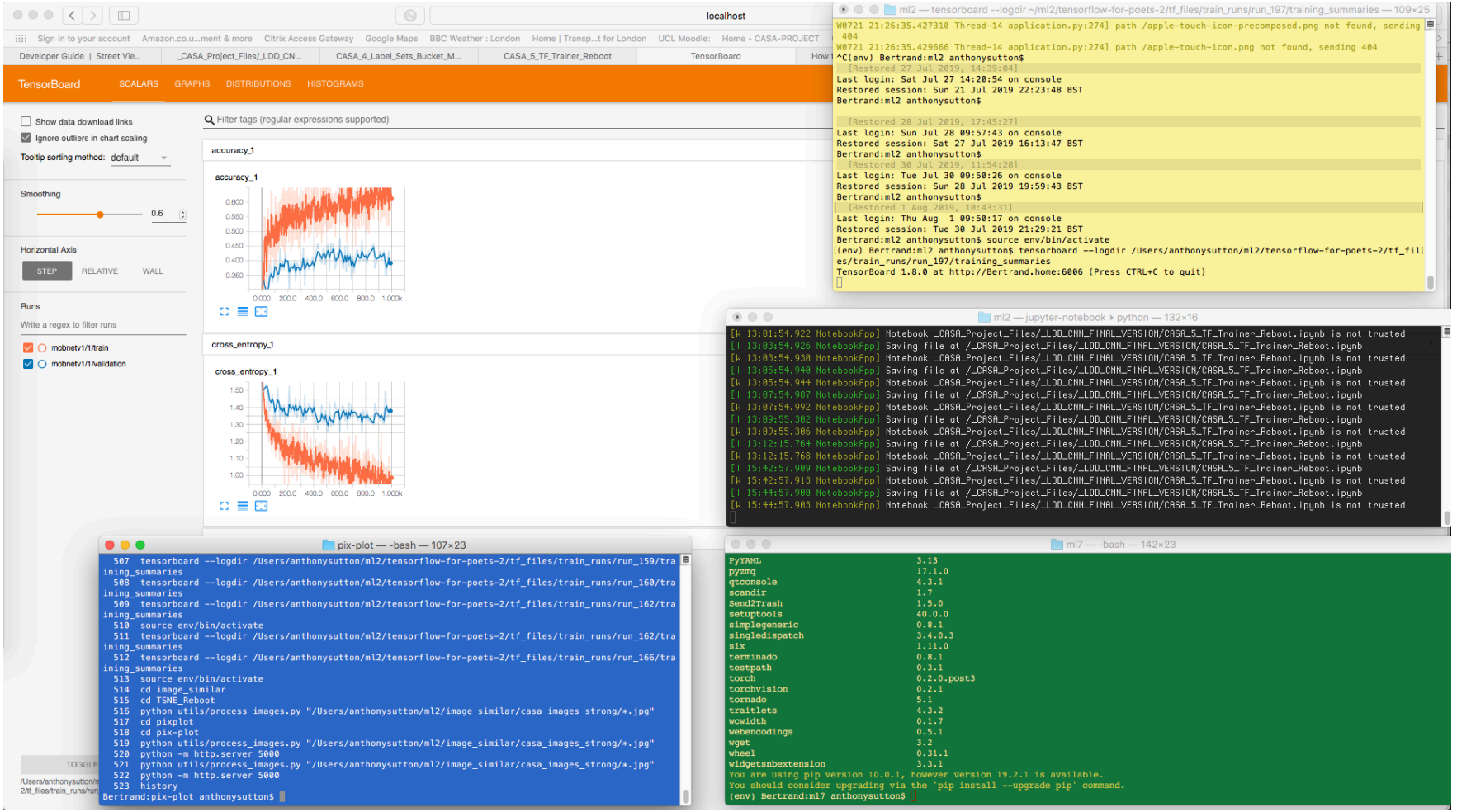
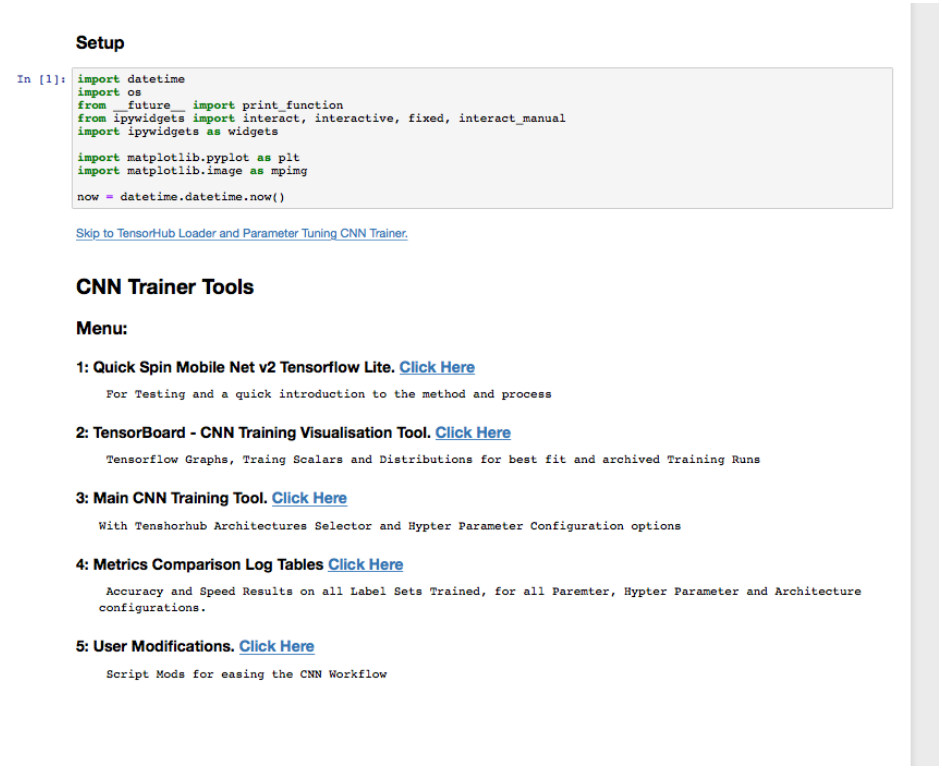


Fig 5.14b – *Fig 5.14a Enlarged*



Setup

In [1]:
```
import datetime
import os
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

now = datetime.datetime.now()
```

Skip to TensorHub Loader and Parameter Tuning CNN Trainer.

CNN Trainer Tools

Menu:

1: Quick Spin Mobile Net v2 Tensorflow Lite. Click Here

For Testing and a quick introduction to the method and process

2: TensorBoard - CNN Training Visualisation Tool. Click Here

Tensorflow Graphs, Traing Scalars and Distributions for best fit and archived Training Runs

3: Main CNN Training Tool. Click Here

With Tenshorhub Architectures Selector and Hypter Parameter Configuration options

4: Metrics Comparison Log Tables Click Here

Accuracy and Speed Results on all Label Sets Trained, for all Paremter, Hypter Parameter and Architecture configurations.

5: User Modifications. Click Here

Script Mods for easing the CNN Workflow

Fig 5.15 *– Jupyter CNN Script Runner Menu*

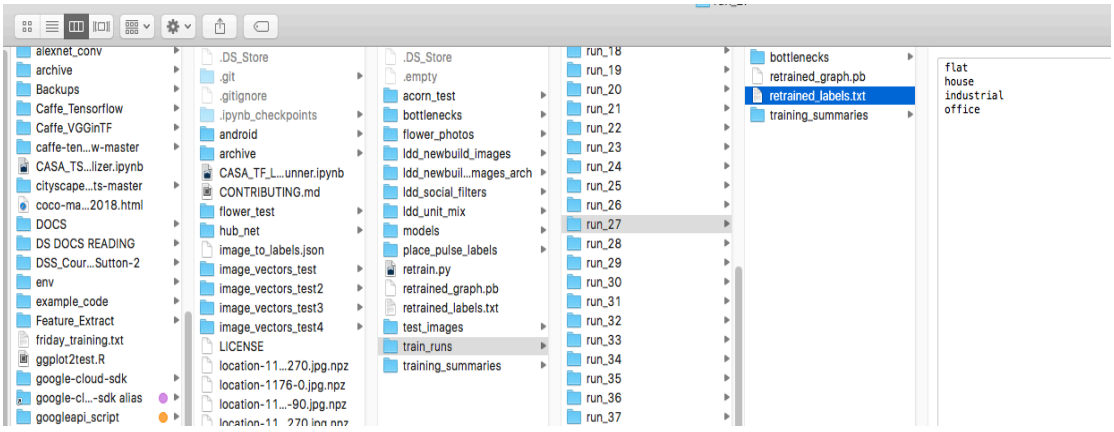


Fig 5.16 *– Checking on the Label and Summary Repositories*

View Label Folders

In [20]:
```
# View Folders from last Label Run
directory = TRAIN_IMAGE_FOLDER_NAME
for filename in os.listdir(directory):
      print(os.path.join(directory, filename))
```

```
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/.DS_Store
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/FLAT
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/HOUSE
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/INDUSTRIAL
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE
```

Select Label Folder:

In [21]:
```
#ENTER TEST IMAGE FOLDER
TEST_FOLDER_NAME = input("ENTER TEST IMAGE FOLDER LOCATION: ")
```

```
ENTER TEST IMAGE FOLDER LOCATION: /Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFI
CE
```

In [22]:
```
directory =  TEST_FOLDER_NAME
for filename in os.listdir(directory):
      print(os.path.join(directory, filename))
```

```
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/.DS_Store
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12289.jpg
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12311.jpg
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12585.jpg
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12866.jpg
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12881.jpg
/Users/anthonysutton/ml2/_FINAL_SCRIPT_LIBRARY/LABELS/LABEL_RUN_2_2_SAFE/OFFICE/col_id_12901.jpg
```

Fig 5.17 *– Loading Inputs and Outputs*

In [22]:
```
%run -i /Users/anthonysutton/ml2/tensorflow-for-poets-2/scripts/retrain_timer.py \
  --bottleneck_dir=/Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/{RUN_FOLDER_NAME}/bottlene
  --how_many_training_steps={STEP_SIZE} \
  --model_dir=/Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/models/ \
  --summaries_dir=/Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/{RUN_FOLDER_NAME}/training_
  --output_graph=/Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/{RUN_FOLDER_NAME}/retrained_
  --output_labels=/Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/{RUN_FOLDER_NAME}/retrained
  --architecture={ARCHITECTURE} \
  --image_dir={TRAIN_IMAGE_FOLDER_NAME}

print('jupyter --> done')
print(now.isoformat())
```

```
INFO:tensorflow:Looking for images in 'FLATS'
INFO:tensorflow:Looking for images in 'HOUSE'
INFO:tensorflow:Looking for images in 'INDUSTRIAL'
INFO:tensorflow:Looking for images in 'OFFICE'
INFO:tensorflow:Looking for images in 'RETAIL'
INFO:tensorflow:Creating bottleneck at /Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/ru
n_83/bottlenecks/FLATS/col_id_10369.jpg_mobilenet_0.50_224.txt
INFO:tensorflow:Creating bottleneck at /Users/anthonysutton/ml2/tensorflow-for-poets-2/tf_files/train_runs/ru
```

Fig 5.18 *– Main Script Run Function*

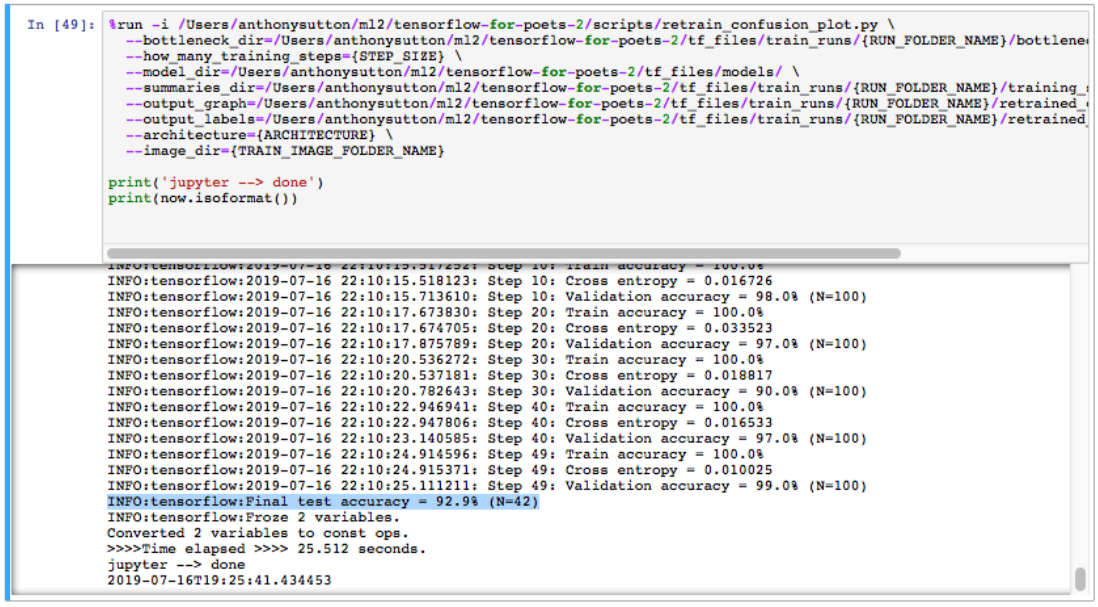


Fig 5.19 *– Inspecting Final Training Evaluation Function Metrics*

The following tensor board visualizations were for a Flat, House, Industrial and Office Building Type label run:

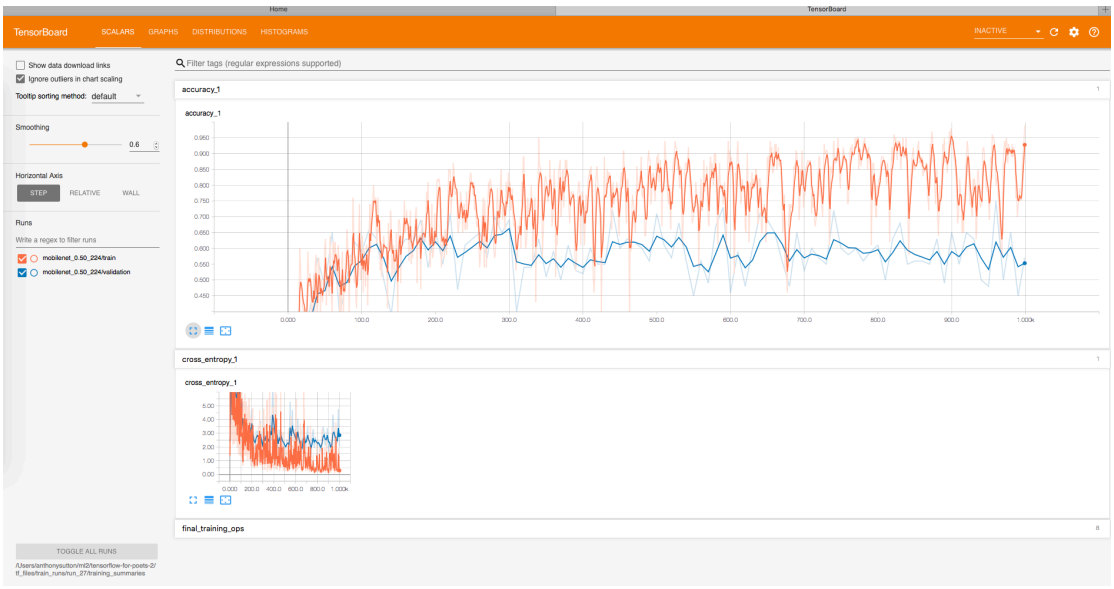


Fig 5.20 *– Accuracy is….*

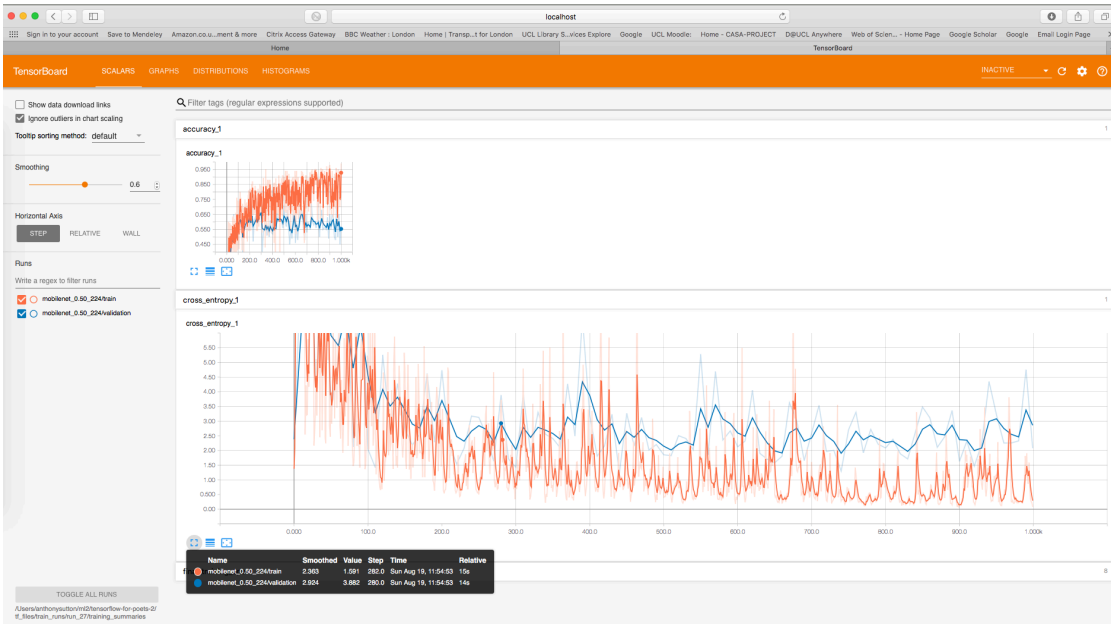


Fig 5.21 *– Entropy Loss is….*

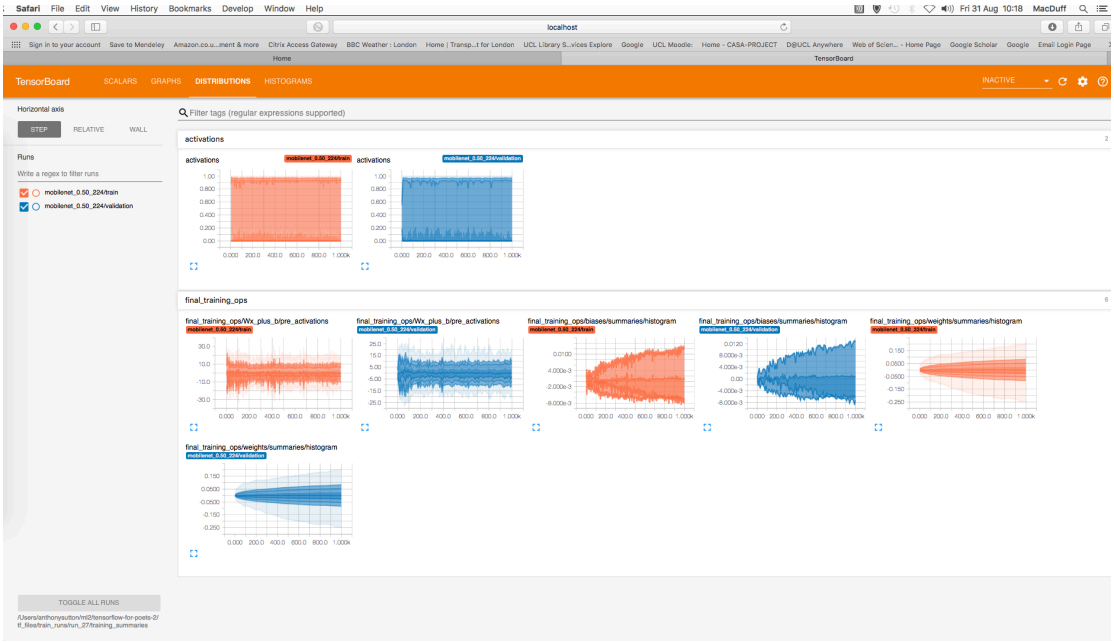


Fig 5.22 *– Distribution of each non-scalar TensorFlow variable across iterations*

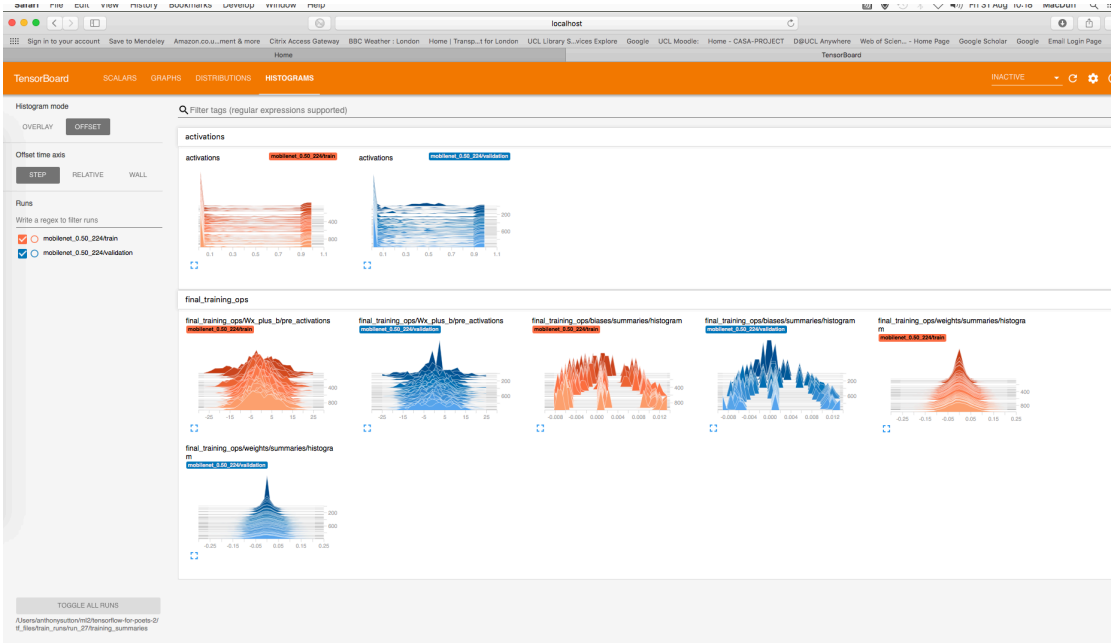


Fig 5.23 *– The TensorBoard Histogram Dashboard displaying distribution of tensor variable in the TensorFlow graph has changed over time*

5.16 Benchmark Metrics and Log Tables

Method:

A trial and error (also known as babysitting) approach was applied to label set runs, hyper parameter and model architecture selection.

More systematic approaches such as grid search, random search, Bayesian optimization, exist in the ML field however were not attempted at this time due to time and resource constraints.

Depending on the outcome of an individual model setting further runs were carried out in similar direction with a view to improving accuracy and/or speed.

Architectures

Figure 2.13 outlines the model architectures deployed and which are described above. The quantized mob net inception variations achieved higher accuracy than the larger Inception Model and Mob Net models on which they based.

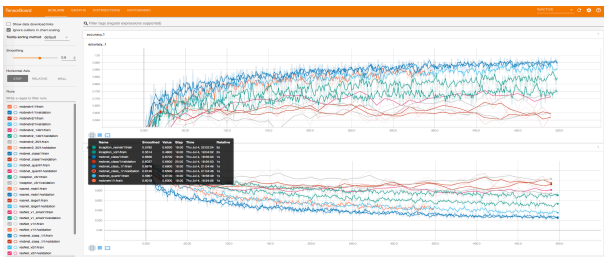


Fig 5.24 *– Architecure comparison (See Fig 5.13 for enlarged version)*

Label Sets

We analyze the relative success and failures of the various label set runs in the next section detailing the classification stage.

Hyper Parameters

Figure 2.13 outlines

For the Hyper Parameter selection TF 2.0 now includes a TF Param component for a comprehensive assessment of optimal model settings although a grid search wrapper could also have been applied using SciKit Learn, if time allowed.

Label Run Details	
Model Architecture	mobilenet_0.50_224, inception v3
Image Size	224
Number of Steps	100 + 1000 + 4000

Label Name	Description	Model	Platform	Location	Step Count	Number of Images	Number of Labels	Test Accuracy	Image Folder Location	Summary Folder Location	Time Taken	Comments
Benchmark Categories:												
FLOWERS	Daisy, Dandelion, Roses, Sunflowers, Tulips	mobnet 0.5	CPU	n/a	4000	3677	5	90.5% (N=359)	tf files	51	Elapsed 391.734 seconds.	
US OSM PROPERTY TYPES	Apartment, Church, House, Industrial, Roof, Office, Retail, Garage	mobnet 0.5	CPU	US Cities (Montreal, New York, Denver)	4000	30000	8	51.6% (N=1782)	lding instance	7	Elapsed 1515.404 seconds	
Trained US OSM PROPERTY TYPES Model applied to LDD	Apartment, Church, House, Industrial, Roof, Office, Retail, Garage	mobnet 0.5	CPU	US Cities (Montreal, New York, Denver)								
LDD HAND LABEL	Weak, Strong, Blank Candidate(London New Build Image, Noisy Street Scene Image, Google Map No Image Found)	mobnet 0.5	CPU	London Wide	4000	374	3	92.9% (N=42)	tf files	14	Elapsed 268.333 seconds.	
Address Base Categories:												
ALL ADD-BASE CLASSES >100 IMAGES	Address Base Tertiary Description Codes (C, C001, CR08, L, PP, R, RD, RD02,03,04,06,U)	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	3029	12	19.1% (N=1380)	5	60	Elapsed 1290.451 seconds.	
ALL ADD-BASE CLASSES and grouped as OSN with no MISC/OTHER Codes	Address Base Tertiary Description Codes (CI, C001, CR08,RD02,03,04,06)	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	4830	7	21.7% (N=875)	31	61	Elapsed 534.096 seconds	Address Base Misc groups identified as problem classifiers
MAJOR MINOR FLATS and ALL HOUSE TYPES	Flat Minor, Flat Major, House Detach, House Semi, House Terrace, Industrial, Office Major, Office Minor, Retail Major, Retail Minor	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	3724	10	27.8% (N=227)	2_2	11	Elapsed 316.105 seconds	Major and Minor divisions not improving Precision
				LBTH, Brent, Croydon	4000							
MAJOR FLATS v HOUSES	Flats, Houses(Grouped)	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	641	2	88.7% (N=53)	37	54	Elapsed 264.752 seconds	High Precision obtained without Offices, Retail and Industrial and Minor Flats
HOUSE vs HOUSE	Semi-Detached, Detached , Terrace(RD02,3,4)	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	834	3	44.1% (N=93)	8	13	Elapsed 275.587 seconds	Low Precision Identified with 3 House Types
FLATS vs TERRACES	RD04, RD06	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	7386	2	80.1% (N=231)	4		Elapsed 375.536 seconds	??????
AB as OSM CLASSES *Best FIT Label	Houses(Grouped), No Minor Flats, No Minor Offices, Retail, Industrial	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	1218	4	64.4% (N=87)	2_3	51	Elapsed 326.727 seconds	OSM Categories and Minor Flat Type ommision achieves above 50 %. Randomness depends on the number of Class Labels in use(100/n = Labels)
				LBTH, Brent, Croydon	4000							
NO RETAIL	House, Flat, Industrial, Office	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	926	4	69.0% (N=87)	2-2-SAFE	22	Elapsed 367.221 seconds	Retail and Offices identified as problem classifier
NO OFFICES	House, Flat, Industrial, Retail	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	994	4	70.5% (N=105)	6_2	10	Elapsed 314.647 seconds	
				LBTH, Brent, Croydon	4000							
HOUSE INDUSTRIAL RETAIL (100 % Auto)	No Offices, No Flats	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	842	3	82.4% (N=85)	9	52	Elapsed 282.231 seconds	
HOUSE, INDUSTRIAL, RETAIL (Hand and TF Labelled)	Outliers Hand Removed	mobnet 0.5	CPU	LBTH, Brent, Croydon	4000	703	3	87.0% (N=115)	16	53	Elapsed 280.964 seconds.	
AB as OSM CLASSES *Best FIT Label	Houses(Grouped), No Minor Flats, No Minor Offices, Retail, Industrial	mobnet 0.5	CPU	London Test and London Train	1000	4234		66.9% (N=441)		81, 46	Elapsed 366.313 seconds.	
AB as OSM CLASSES *Best FIT Label	Houses(Grouped), No Minor Flats, No Minor Offices, Retail, Industrial	mobnet 0.5	CPU	London Test and London Train	4000	4234		66.7% (N=441)		81, 46	Elapsed 227.617 seconds	

Fig 5.25 – *Label Run Summary Log*

CNN Architecture	Default Params		Tuned		Default Params		Tuned		Default Params		Default Params		Default Params		Default Params		Tuned		Label Run ID
	Bottle Necks Cached(2nd Run Metrics)		Random Brightness = 5																
Step Size >>>>>>>>>>>>>>>	500				1000				2000		4000		6000		8000				
Metric >>>>>>>>>>>>>>>>>	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	Accuracy	Time	
Flowers Benchmark (Test Set: n= 359)	93.9	63.2																	
mobnet v1	66.2	44			72.7	103.2			72.7	113	71.4	231.7	67.5	351.2	68.8	454			
mobnet v2	61	52			63.6	105.97			63.6	147.61	64.9	256.5			66.2	526.34			
mobnet v2 035 Neurons	64	45			72.7	105.37													
mobnet v2 140 Neurons	68.8	49			67.5	270.44													
mobnet v1 classification	67.5	50																	
mobnet v2 classification	68.8	44																	
MobNet Quant	74	47	72.7	2533.81 secs	75.3										68.8	464.5			
Inception Full	68.8	112			68.8														
NasNet Mob	62.3	113																	
NasNet Large	66.2	212																	
ResNet v1 Small	70.1	314			75.3	93.2			74	234.7	76.6	397			76.6	850.9			
ResNet v1	70.1	103			71.4	234.9			68.8	238.6	70.1	520.3	71.4	822.53	71.4	979.1			
ResNet v2	71.4	104			74	245			75.3	247.8	72.7	509							
Inception ResNet	64.9	104																	
PNasNet	64.9	177																	

Fig 5.26 – *Model Architecture Comparison*

Summary

Bottleneck Creations (the extraction of the penultimate later of image features) is relatively time-expensive.

Google Dataflow API addresses this workflow issue by processing each image independently and in parallel.