A PROJECT REPORT ON
Automated data labelling system with active learning

A project report submitted in fulfilment for the Diploma Degree in AI & ML

Under
Applied Roots with University of Hyderabad

Project submitted by

Ashish Lotake

Under the Guidance of

Mentor: Neil Fowler
Approved by: Mentor: Neil Fowler

University of Hyderabad

CERTIFICATE OF RECOMMENDATION

We hereby recommend that the thesis entitled “Automated data labelling system with active learning” prepared under my supervision and guidance by Ashish Lotake be accepted in fulfilment of the requirement for awarding the degree of Diploma in AI & ML Under applied roots with University of Hyderabad. The project, in our opinion, is worthy of its acceptance.

_____

Mentor: Neil Fowler

Under Applied roots with

University of Hyderabad

ACKNOWLEDGEMENT

Every project big or small is successful largely due to the effort of a number of wonderful people who have always given their valuable advice or lent a helping hand. I sincerely appreciate the inspiration; support and guidance of all those people who have been instrumental in making this project a success. I, Aishwarya P student of applied roots, is extremely grateful to mentors for the confidence bestowed in me and entrusting my project entitled “Automated data labelling system with active learning” with special reference. At this juncture, I express my sincere thanks to Mentor: Neil Fowler of applied roots for making the resources available at the right time and providing valuable insights leading to the successful completion of our project who even assisted me in completing the project.

Name: Ashish Lotake

Declaration of Authorship

We hereby declare that this thesis titled ”Automated data labelling system with active learning” and the work presented by the undersigned candidate, as part of Diploma Degree in AI & ML.

All information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name: Ashish Lotake

Thesis Title: Automated data labelling system with active learning

Content

1. Introduction

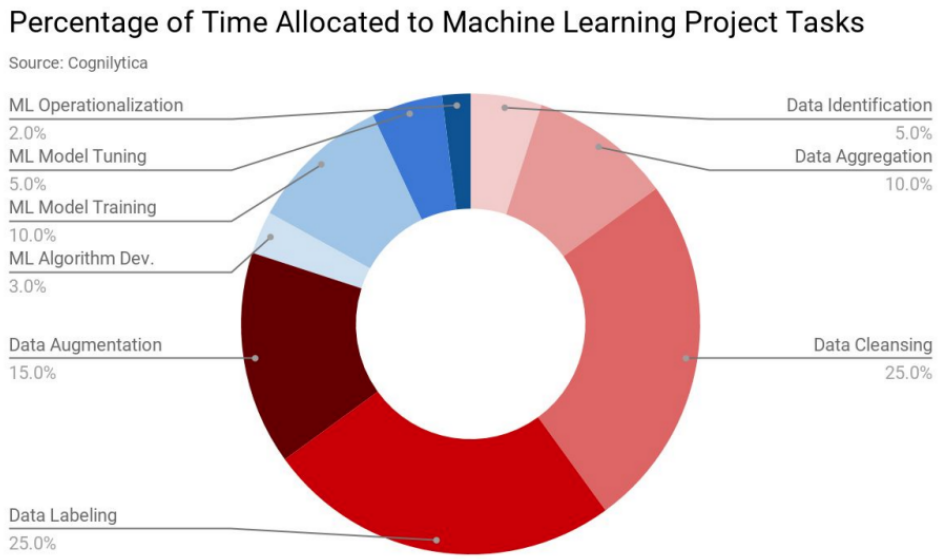
In this project, we need to label the input data from the user, (images in my case) and at the same time allowing the user to label is not satisfied with the result and then retraining the original model, in order to improve the models' generalisation accuracy.

This an important problem to solve, as we can aquire large amount of unlabelled data, but manually labelling is expensive, in order to use supervised algorithm, we need to input both data and it output, and in many neural network we need to give the output, like in CNN, speech recognition and the list goes on. Major takeaway is that labelled data is important.

However we still need an expert/user to validate the result of the model, if the user is satisfied or not with the output.

Business and real world impact

For machine learning models to have good generalisation accuracy we need to feed clean, accurate, complete, and well-labelled data. Around



80% of AI project time is spent on gathering, organising, and labelling data according to analyst firm Cognilytica.

Teams are in a race to collect meaningful data, which must be correctly formatted and labelled for training and deploying models, thus they cannot afford to waste time. For supervised learning algorithms, we must feed relevant data, appropriately labelled with the required output that needs to be learned. One of the most popular use cases of labelling is in NLP (natural language processing). This is because many NLP applications require a large amount of labelled data (for example, Part-of-Speech Tagging and Named Entity Recognition), and labelling this data is quite expensive. The most common workloads for data labelling are object/image recognition, autonomous vehicles, and text and image annotation. The market for third-party Data Labelling solutions is $150M in 2018 growing to over $1B by 2023 according to analyst firm Cognilytica[1]

2. Literature review

Identity Mappings in Deep Residual Networks by Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun[2]

Deeper neural networks require more effort to train. We present a residual learning framework for training networks that are significantly deeper than those previously used. Instead of learning unreferenced functions, we explicitly reformulate the layers as learning residual functions with reference to the layer inputs. We present extensive empirical evidence demonstrating that these residual networks are easier to optimise and can benefit from significantly increased depth. On the ImageNet dataset, we evaluate residual nets with up to 152 layers of depth—-8x deeper than VGG nets but with lower complexity. On the ImageNet test set, an ensemble of these residual nets achieves 3.57% error. This result placed first in the ILSVRC 2015 classification task.

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications[3]

The author introduces MobileNets, which are efficient models for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that builds light weight deep neural networks using depth-wise separable convolutions. The architecture includes two simple global hyperparameters for efficiently balancing latency and accuracy. These hyper-parameters enable the model builder to select the appropriate sized model for their application based on the constraints of the problem, present extensive experiments on resource and accuracy tradeoffs, and demonstrate strong performance when compared to other popular ImageNet classification models. The author then concludes the report by demonstrating the effectiveness of MobileNets

in a variety of applications and use cases such as object detection, finegrain classification, face attributes, and large scale geo-location.

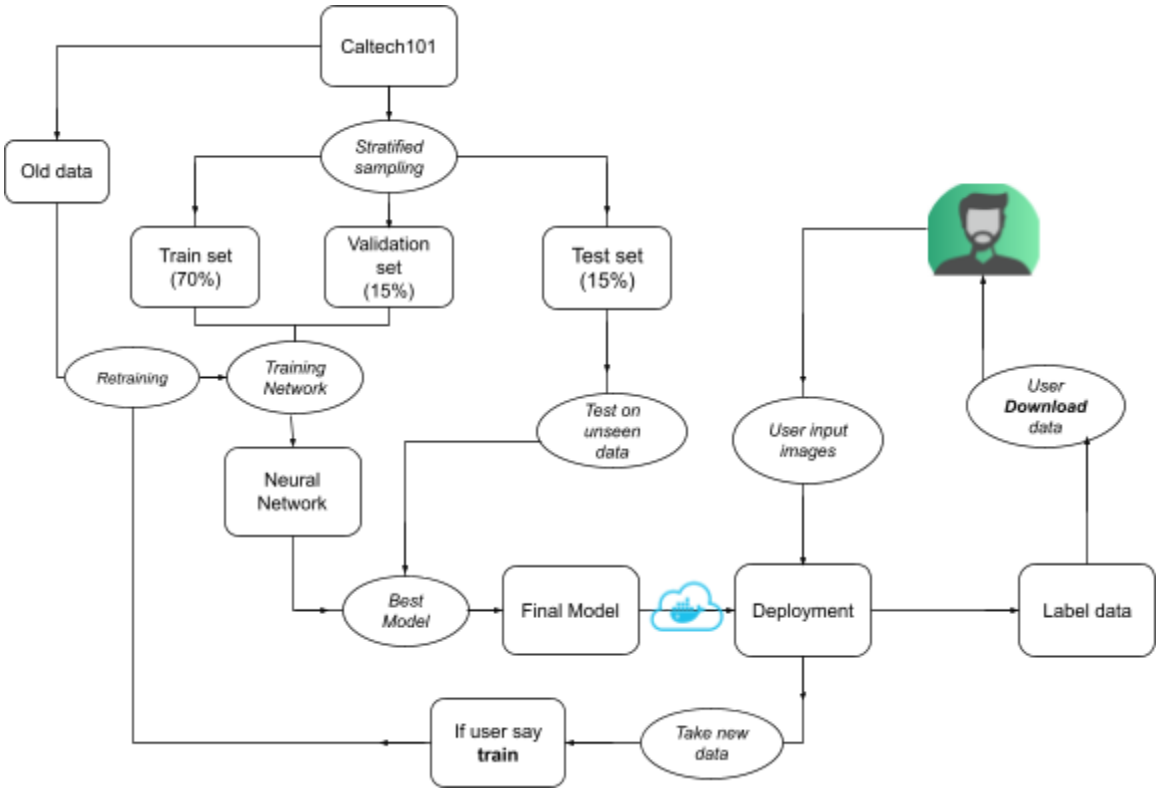
Deep Learning and Transfer Learning Approaches for Image Classification[4]
The author aims to give a brief overview of widely used pre-trained models, and give a high overview about their architecture, about the author, number of parameters, etc. Then the author also explains the need for transfer learning. And author ends his report stating various open source dataset

Transfer learning for image classification using VGG19: Caltech-101 image data set[5]
The author aims to showcase that a single feature extractor whether shallow or deep is not enough to achieve satisfactory results. So, a combined approach using deep learning features and traditional handcrafted features is better for image classification. So the author improved the image classification performance by combining the deep features extracted using popular deep convolutional neural network, VGG19, and various handcrafted feature extraction methods, i.e., SIFT, SURF, ORB, and Shi-Tomasi corner detector algorithm. Further, the extracted features from these methods are classified using various machine learning classification methods, i.e., Gaussian Naïve Bayes, Decision Tree, Random Forest, and eXtreme Gradient Boosting (XGBClassifier) classifier. The experiment is carried out on a benchmark dataset Caltech-101. The experimental results indicate that Random Forest using the combined features give 93.73% accuracy and outperforms other classifiers and methods proposed by other authors

Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates[6]
The author describes a phenomenon known as "super-convergence," in which neural networks can be trained an order of magnitude faster than using traditional training methods. The existence of super-convergence is important for comprehending why deep networks generalise so well. Training with a single learning rate cycle and a high maximum learning rate is a key component of super-convergence. A fundamental insight that enables super-convergence training is that high learning rates regularise

the training, necessitating a reduction in all other forms of regularisation to maintain an optimal regularisation balance.

Methodology for model implementation

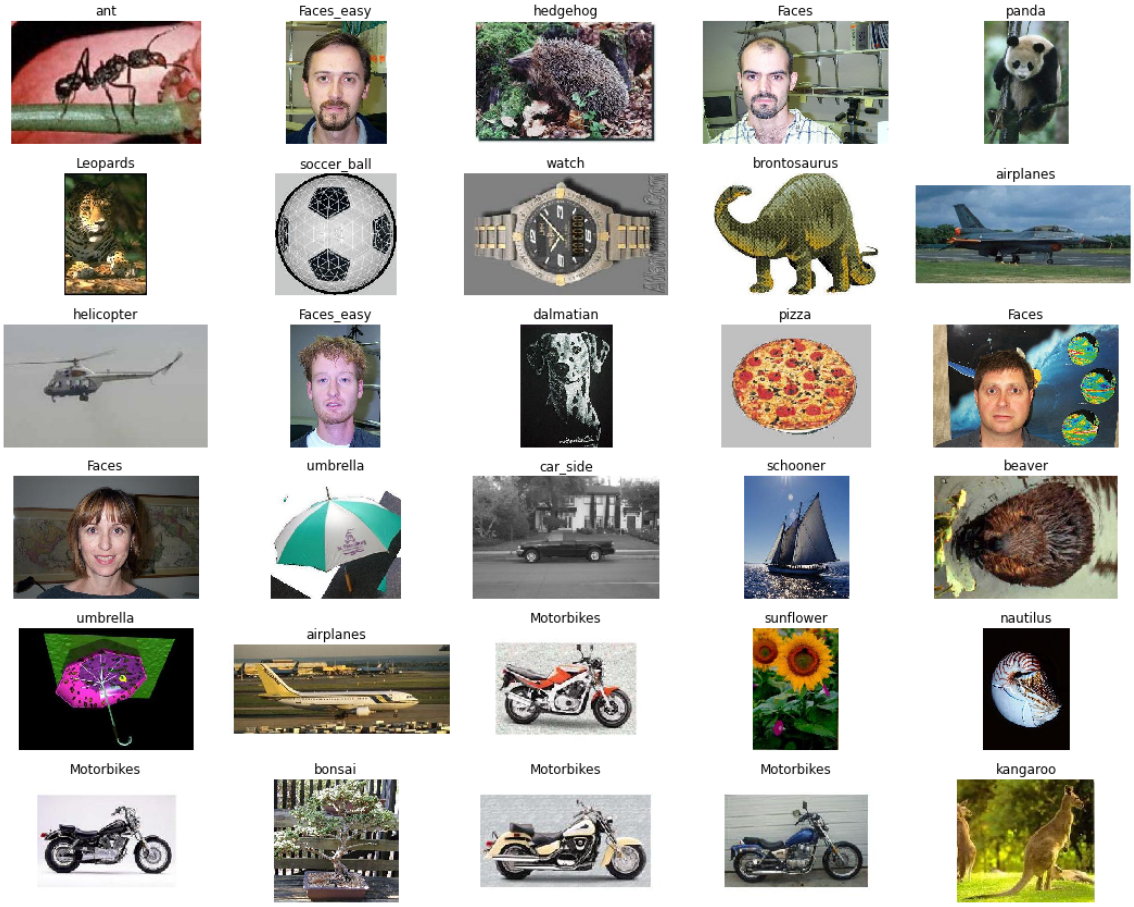


3. Data Description

Caltech-101 dataset:

We will use the Caltech-101 dataset , which consists of 9146 images, split unevenly between 101 distinct object categories and a background category. According to the official website, there are about 40 to 800 images per category, additionally most of the categories have around 50 images in the dataset. Such varying images per category can be a real problem when trying to achieve high accuracy in deep neural network training. The size of each image is roughly 300 x 200 pixels. The format of images are

RGB, meaning we have 3 colour channels. In python image size will be represent as (height , width ,channels)



Datasource Source

This dataset is downloaded from Caltech Data by Caltech Library https://data.caltech.edu.

The data link :- https://data.caltech.edu/records/mzrjq-6wc02

Data Acquisition

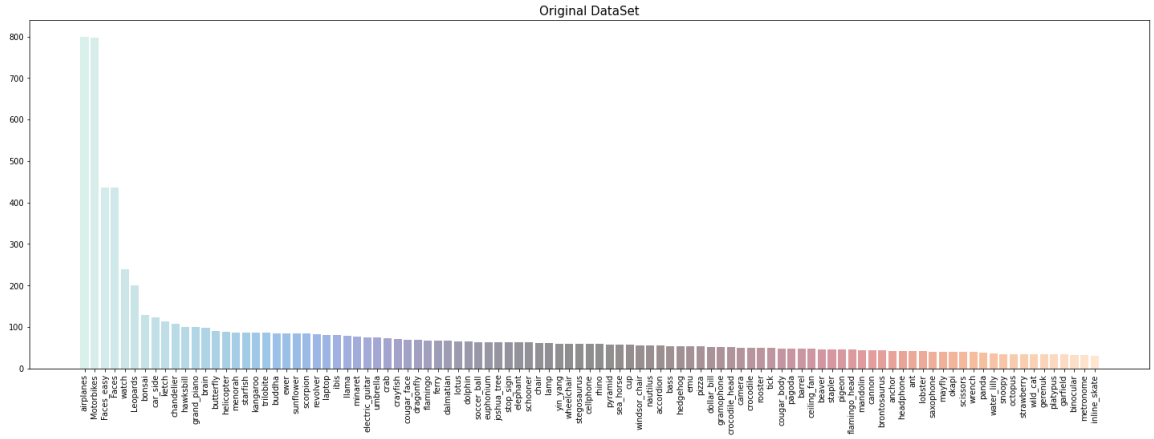
Its a open source data which are be retrieved from Caltech official website
https://data.caltech.edu
Authors :- Li, Andreeto, Ranzato, & Perona. [7]

Tools
For reterviving and processing data

- Python3
- Numpy
- Pandas
- Matplotlib
- Seaborn
- Scikit-learn,
- Jupyter Notebook,
- Keras
- Docker

Class Imbalance

The dataset is relatively small for a deep learning task, it contains only 9146 samples, divided in 101 classes, these are not enough images to get very high accuracy. Furthermore, ignoring the BACKGROUND_Google label and its images, we obtain a total of 8677 images. Apart from the less number of images, another problem is the distribution of the images per class. To get a better idea about the distribution of the images in each category, the following figure might be useful.



We can clearly see that airplanes and Motorbikes categories have the highest number of images, around 800, and the lowest number of images may be as low as 40 for most of the labels. Such an imbalanced dataset, as we will see soon, is one of the major reasons for the bad performance of the network trained from scratch.

4. Approach, Methodology and Results

a. Requirement Analysis

Functional Requirement

- Purpose -> Label the user images and allow the user to retrain the model.
- Input -> We will use RGB images as input for training the underlying model.
- Output-> Label input images from user and place them in respective directory as per label.

hardware requirement

- 32 GB RAM
- 12-16 GB VRAM
- Linux/Unix/Windows
- Internet

Software Requirement

- VSCode or any editor

- Jupyter-lab/Notebook
- Streamlit
- Python3 and Libraries
- Web Browser
- Docker

b. Data Preprocessing

Preprocessing within any machine learning is associated with the transformation of data from one form to another. Usually, preprocessing is conducted to ensure the data utilised is within an appropriate format.

As we will be using CNN, we need to preprocess the data in such a way that

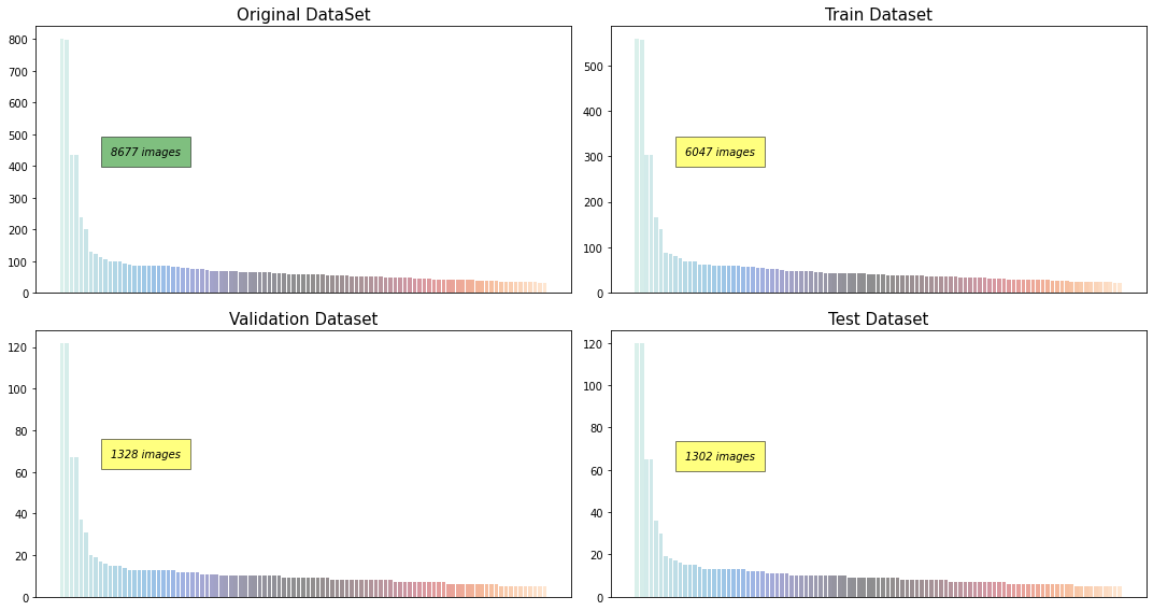
- All the images should have same number of channels
- Normalise the images convert the range from 0-255 to 0-1 for each pixel value,
- Resize all the images to the same size.

When using transfer learning we need to make sure to use the same preprocessing as the underlying convolutional base, for example when using resnet as convolutional base, keras provide resenet preprocessing, which is the same as used in the original network.

- Resizing images
- Normalising pixel values
- All images have 3 colour channels

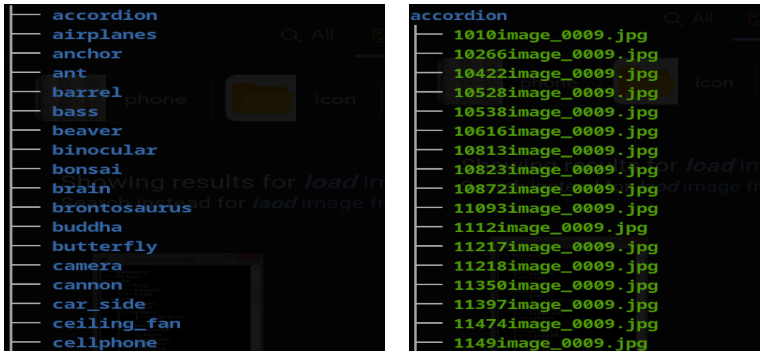
c. Data Splitting

The whole dataset is divided into three parts with 70% for training set which will be used to train the neural network, 15% validation set, which serves to evaluate the performance of the network after each epoch and 15% test data, which is used to test data on unseen data. The split has been implemented using StratifiedShuffleSplit (from sklearn.model selection) to maintain class balancing between the sets. The dataset will be divided into training, validation, and test sets with the proportion mentioned above.



After splitting data, we can see that both TRAIN,TEST and VALIDATION sets are representative of the original dataset that we wanted. Plus this we we can ensure that all the sets have images form each

d. Loading Data



Above images shows the layout of dataset, each class has it own directory.

We will use keras *tf.keras.preprocessing.image_dataset_from_directory* to load images from directory

We will use keras *tf.keras.preprocessing.image_dataset_from_directory* to load images from directory

e. Baseline model

Deep Learning takes a lot of data, which can make decisions about new data. This data is passed through Neural Networks, known as Deep Neural Networks (DNN).In Deep learning, Convolutional-Neural Network (CNN) is a popular type of deep neural networks. CNN eliminates the need for manual feature extraction like traditional features extraction algorithms, such as SIFT, LBP etc. CNN directly extracts the features from a set of raw image data. Related features are not pre-trained; they will learn when the network trains on a group of images. This automated way of feature extraction is the most accurate learning model for computer vision tasks such as object detection, classification, recognition. Traditional Machine Learning approaches done the-feature extraction manually and the classification-algorithm classify the objects separately. But, In Deep Learning approaches the network itself extract the features without user interpretation and also classify the objects.

Convolutional Neural Network formed with the help of different layers to perform the image classification task. The architecture of the CNN contains the different layers. The whole network itself is divided into two parts :- **Convolution base** and the **Classifier**. The features are extracted in the convolution base and the classification is done in the classifier.

Convolution base

1. *Input Layer*: This input layer, accepts raw images, and forwarded to further layers for extracting features.
2. *Convolution Layer*: After input layer, next layer is convolution layer. In this layer a number of filters are applied on images for finding features from images. These features are used for calculating the matches at testing phase.
3. *Pooling*: Extracted features are sent to the pooling layer. This layer captures large images and reduces them, and reduces the parameters to preserve important information. It preserves maximum value from each window.

Classifier

1. *Fully Connected Layer*: The final layer is a fully connected layer, which takes up high- level filtered images and translates them into labels with categories.
2. *Softmax Layer*: This layer is present just before the output layer. This layer gives the probabilities to each class. Those decimal probabilities are in the range of 0 to 1.

We will build an architecture similar to Alexnet**[3]** , which won the ImageNet ILSVRC challenge**[4]** in 2012, it was at this contest that AlexNet showed that deep convolutional neural networks can be used for solving image classification. This network's architecture was very similar to LeNet's, but it was deeper, larger, and featured Convolutional Layers stacked on top of each other.

The research paper that described the internal elements of the CNN architecture also introduced some novel techniques and methods, including effective computing resource utilisation; data augmentation; GPU training; and a variety of methods to avoid overfitting in neural networks.

Here are the types of layers the AlexNet CNN architecture is composed of, along with a brief description:

- *Convolutional layer* :- The mathematical term "convolution" refers to the dot product multiplication of two sets of elements. The convolutional layer's filters/kernels and array of image data are affected by the convolution operation in deep learning. A convolutional layer is therefore just a layer that contains the convolution operation that takes place between the filters and the images that are processed by a convolutional neural network.

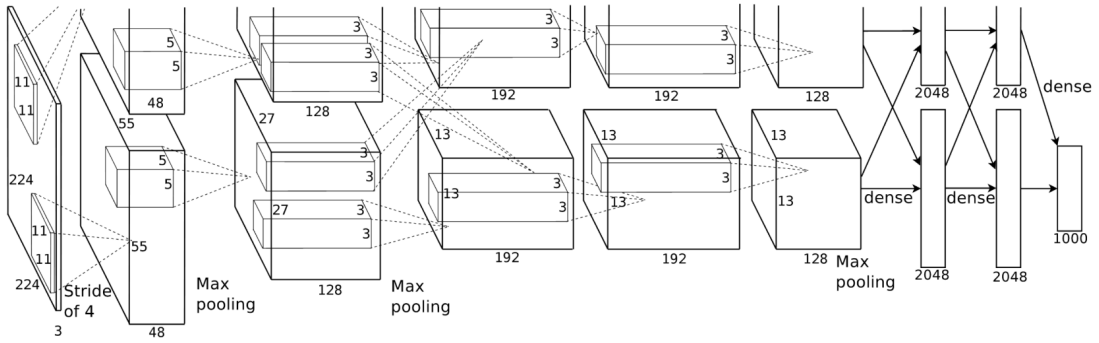
- *Batch Normalisation* :- By adding an additional layer that operates on the inputs from the previous layer, batch normalisation is a technique that reduces the impact of unstable gradients within a neural network. The operations first standardise and normalise the input values before scaling and shifting operations transform the input values.
- *Max Pooling* :- With max pooling, the output is the highest pixel value of pixels that are contained within a unit's receptive field within a sub-sampling layer. The max-pooling operation below produces an average of the pixels inside the kernel's receptive field by sliding across the input data with a 2x2 window.

- *Flatten layer* :- Takes an input shape and flattens the input image data into a one-dimensional array.
- *Dense Layer* :- A dense layer has an embedded number of arbitrary units/neurons within. Each neuron is a perceptron.

Other techniques utilised in the AlexNet CNN:

- *Activation Function* :- a mathematical process that produces a normalised output from the outcome or signals of neurons. An activation function is a part of a neural network whose function is to add non-linearity to the network. The neural network can have more representational power and solve complex functions with the addition of an activation function.
- *ReLU* :- A particular activation process that modifies a neuron's value output. The transformation imposed by ReLU on values from a neuron is represented by the formula y=max(0,x). Any negative values from the neuron are clamped down to 0 by the ReLU activation function, while positive values are left unaltered. Within a neural network, the outcome of this mathematical transformation is used as the input to the next layer and as the output of the current layer.
- *Softmax Activation Function* :- a specific kind of activation function used to determine the probability distribution of a group of numbers contained within an input vector. The result of a softmax activation function is a vector whose set of values represents the likelihood that a class or event will occur. The vector's values add up to 1 in total.
- *Dropout* :- The dropout technique reduces the quantity of interconnecting neurons in a neural network at random. Each neuron has a chance to be dropped from the collated contributions from connected neurons at each training step.

Lets see the overall Architecture



The first convolutional layer applies 96 kernels with an 11 x 11 x 3 pixel size and a 4 pixel stride to the 224 x 224 x 3 input image. The output of the first convolutional layer serves as the input for the second convolutional layer, which filters it using 256 kernels of size 5 x 5 x 48. Without any pooling or normalisation layers in between, the third, fourth, and fifth convolutional layers are connected to one another. The (normalised, pooled) outputs of the second convolutional layer are connected to 384 kernels of size 3 x 3 x 256 in the third convolutional layer. The fifth convolutional layer has 256 kernels of size 3 x 3 x 192, and the fourth convolutional layer has 384 kernels of that size. Each of the fully connected layers contains 4096 neurons..

The eight main layers in AlexNet are divided into two groups: the first five layers are convolutional layers, some of which are followed by max-pooling layers, and the final three layers are fully connected. A 1000-dimensional vector containing the scores for each class is produced by the final layer. To avoid the problems of "killing" the gradients and co-adaptation in the hidden layers, the network uses a non-saturating ReLU activation function and Dropout.
Prior to beginning the training, we must modify the AlexNet structure to account for our problem's use of only 101 categories as opposed to the default value of 1000. Therefore, we add a new layer of the same type with a 101-dimensional output to the network to replace the final fully connected layer.

Training from scratch

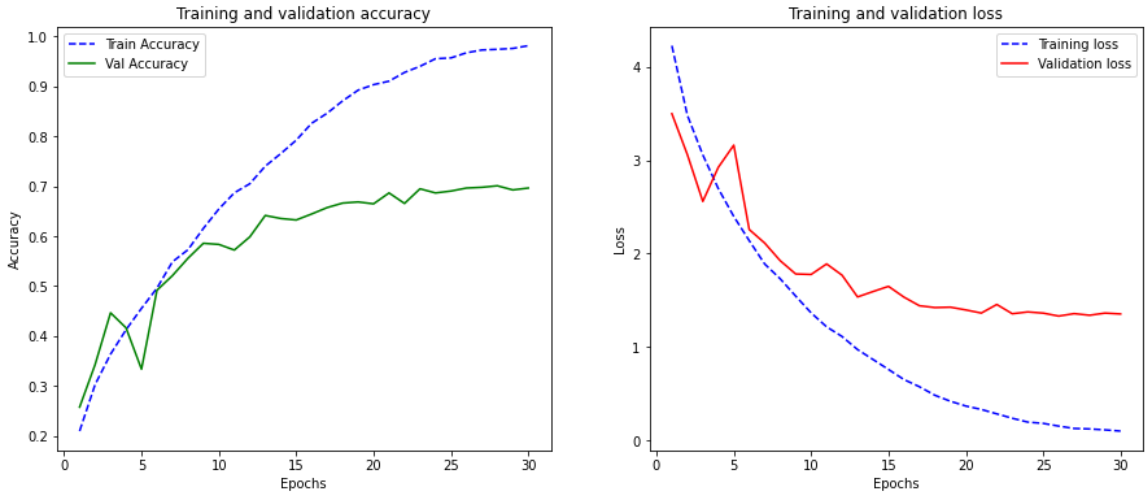
Code: Implement model from scratch

```
input = keras.Input(shape=(227,227,3))
x = layers.Rescaling(1./255)(input) #
```

```
# 1st layer
x = layers.Conv2D(filters=90, kernel_size=(11,11), strides=(4,4),
activation='relu')(x)
x = layers.MaxPool2D(pool_size=(3,3), strides=(2,2))(x)
x = layers.BatchNormalization()(x)
# 2nd layer
x = layers.Conv2D(filters= 256, kernel_size=(5,5),strides=(1,1),
activation='relu', padding="valid")(x)
x = layers.MaxPool2D(pool_size=(3,3), strides=(2,2))(x)
x = layers.BatchNormalization()(x)
# 3rd layer
x = layers.Conv2D(filters= 384, kernel_size=(3,3),strides=(1,1),
activation='relu', padding='valid')(x)
x = layers.BatchNormalization()(x)
# 4th layer
x = layers.Conv2D(filters= 384, kernel_size=(3,3), strides=(1,1),
activation='relu', padding='valid')(x)
x = layers.BatchNormalization()(x)
# 5th layer
x = layers.Conv2D(filters= 256, kernel_size=(3,3),  strides=(1,1),
activation='relu', padding='valid')(x)
x = layers.MaxPool2D(pool_size=(3,3), strides=(2,2))(x)
x = layers.BatchNormalization()(x)
# flattening
x = layers.Flatten()(x)
# 1st dense layer
x = layers.Dense(4096, activation='relu')(x)
x = layers.Dropout(0.5)(x)
x = layers.BatchNormalization()(x)
# 2nd dense layer
x = layers.Dense(4096, activation='relu')(x)
x = layers.Dropout(0.5)(x)
x = layers.BatchNormalization()(x)
# output softmax layer
output = layers.Dense(101, activation='softmax')(x)
alexnet_scratch = keras.Model(inputs = input, outputs=output)
alexnet_scratch.compile(loss='sparse_categorical_crossentropy',
optimizer='sgd', metrics=['accuracy'])
```

We should keep an eye on a number of useful variables while a neural network is being trained. The loss is the first quantity that has to be monitored during training because it is assessed on individual batches during the forward pass. The second quantity is the validation/training accuracy because it provides important information about the degree of overfitting in the model. The different hyperparameter settings were examined using loss and accuracy plots in order to gain understanding of how to modify them for more effective learning.

At the beginning, we train the network with the default values provided
In keras code. In particular we use the following parameters 30 epochs with the Cross Entropy loss function. The gradients are updated at each step according to the Stochastic Gradient Descent (SGD) with default learning rate and momentum, with a batch size of 32.



Train and validation accuracy when model ran with default hyper parameters defined in keras api

Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
0.9818	0.0992	0.6965	1.3532	0.691	1.434

f. Hyperparameter tuning

Deep learning relies heavily on hyperparameter optimization.

The reason for this is that neural networks are extremely difficult to configure and require a large number of parameters to be set. Furthermore, individual models can be very slow to train.

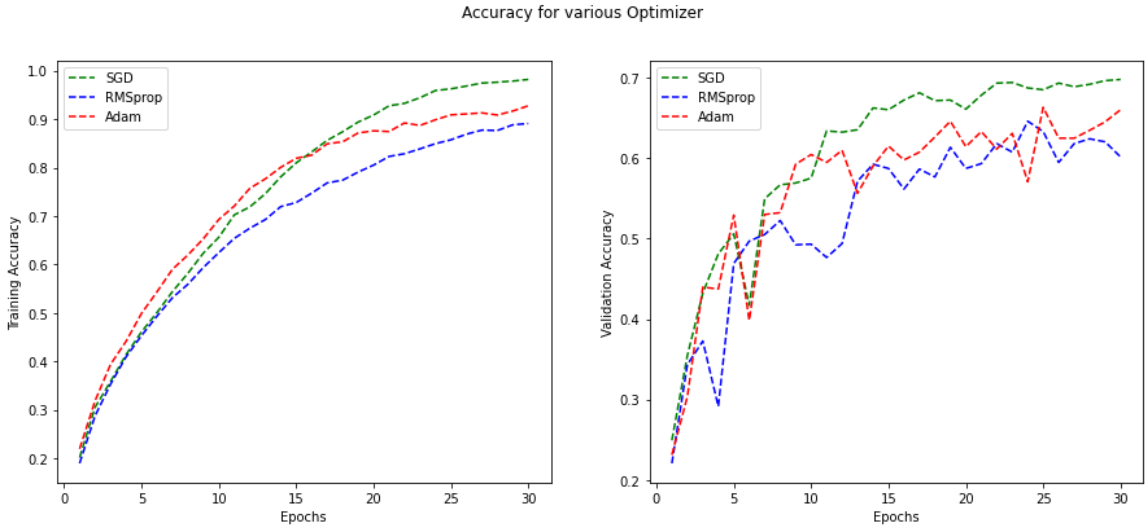
We will try to optimise few of the important hyperparameters for CNN

1. Optimization Algorithm
2. Learning Rate
3. Number of Epochs
4. Momentum

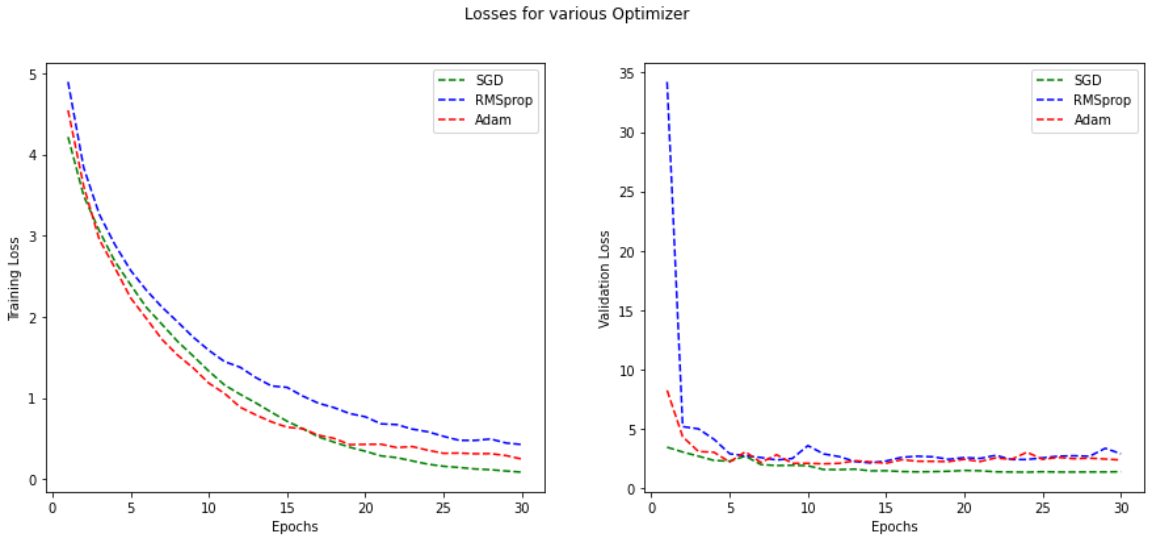
Optimization Algorithm

Optimizers are techniques that alter the neural network's weights and learning rate in order to minimise losses. Three optimization algorithms will be tested. The optimizers you employ determine how you should modify the weights or learning rates of your neural network to minimise losses. The goal of optimization algorithms or strategies is to minimise losses and deliver the most precise results.
It is common to pre-select an optimization algorithm to train your network and tune its parameters.
We will try three optimizer algorithm for our architecture :

- SGD
- Adma
- RMSProp



Train and validation accuracy when model ran with various optimization algorithm with default hyperparameters



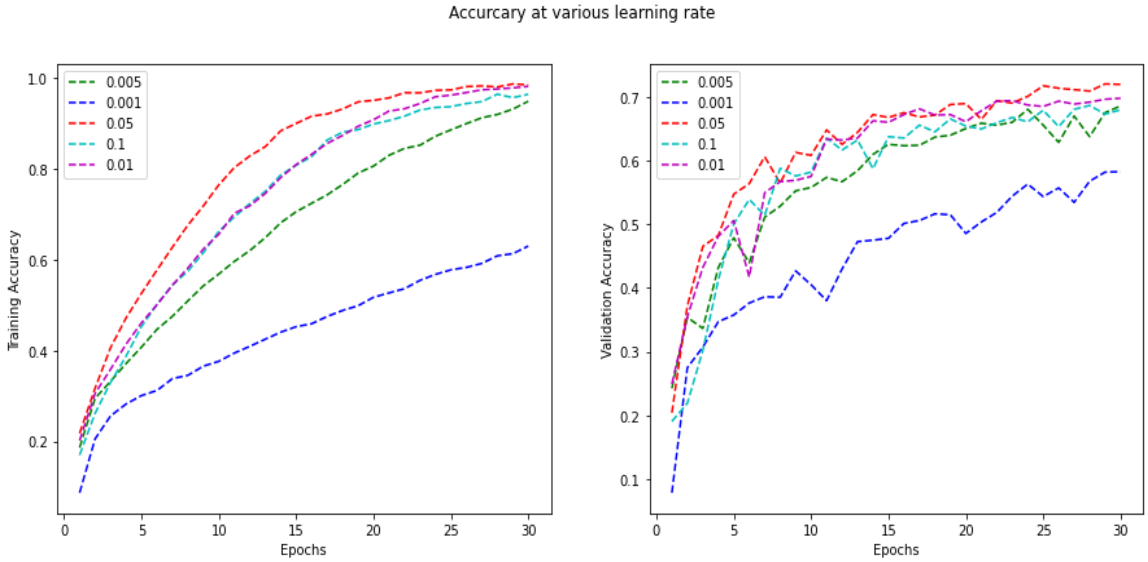
Train and validation loss when model ran with various optimization algorithm with default hyperparameters

	Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
SGD	**0.9818**	**0.08767**	**0.6980**	**1.3981**	**0.694**	**1.439**
RMS	0.8909	0.4286	0.617	2.9087	0.578	2.990
ADAM	0.9276	0.2492	0.6604	2.385	0.635	2.990

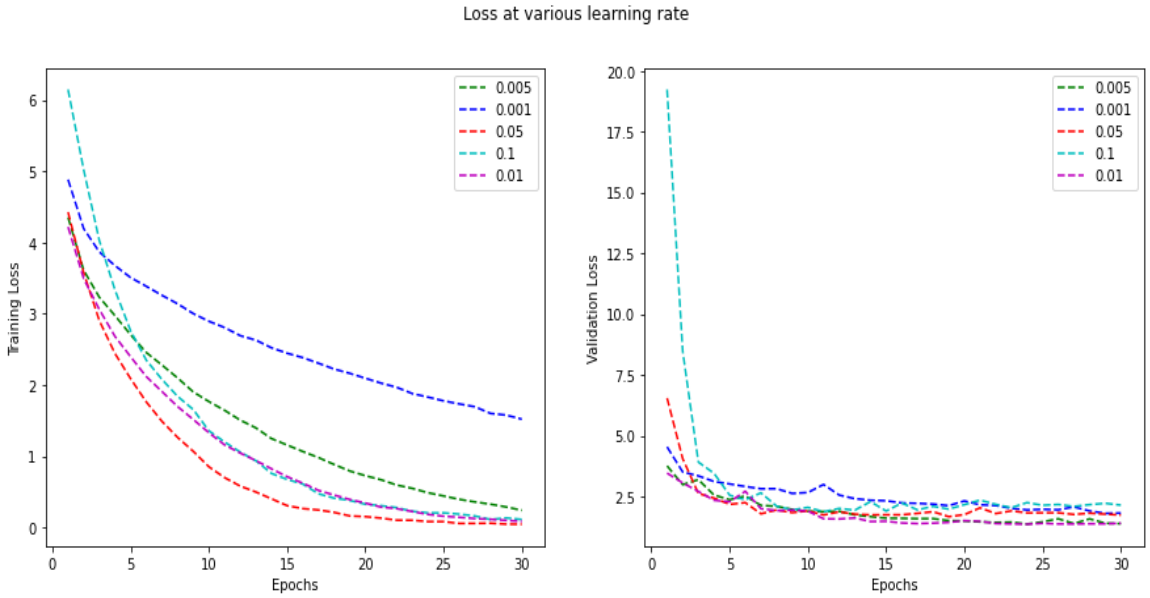
As we can see SGD is performing better when compared to other optimizers, this is because we are using architecture similar to alexnet, which also uses SGD.

Selecting Learning rate

The learning rate is a hyperparameter that determines how much to alter the model each time the model weights are updated in response to the estimated error. It can be difficult to choose the learning rate because a value that is too small could lead to a lengthy training process that may get stuck, whereas a value that is too large could lead to learning a suboptimal set of weights too quickly or to an unstable training process. When configuring your neural network, the learning rate might be the most crucial hyperparameter.



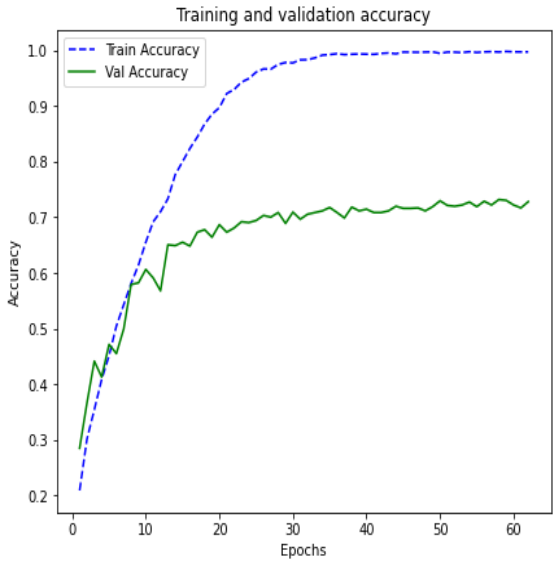
Train and validation accuracy when model ran with various learning rate with optimizer =SGD

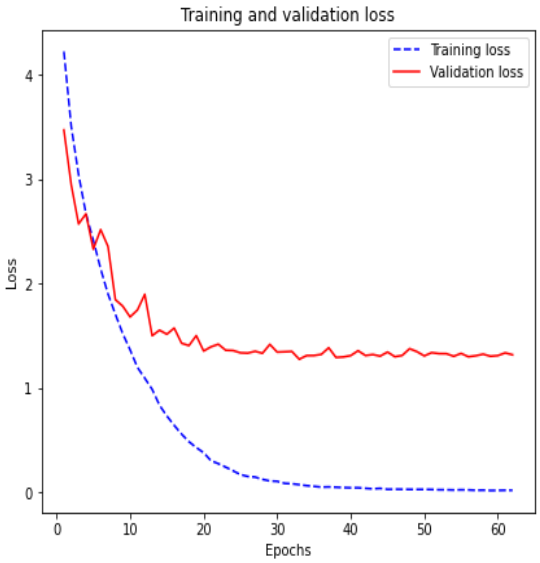


	Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
Learning rate						
0.001	0.6301	1.5218	0.5828	1.8176	0.571	1.821
0.005	0.9491	0.2435	0.6860	1.3802	0.665	1..452
0.01	0.9818	0.08767	0.6980	**1.3981**	0.694	**1.439**
0.05	**0.985**	**0.0481**	**0.7199**	1.737	**0.707**	1.780
0.1	0.9644	0.115	0.68	2.1428	0.675	2.119

Here learning rate 0.05, looks good on training data, but the loss on validation and test set are significantly greater when compared to learning rate 0.01.
The accuracy across all 3 sets, the difference between them for learning rate 0.05 and 0.01 is around 1-2%, but loss for learning rate 0.05 is third highest for validation and test.
Seems like the model is overfitting to training data with learning rate 0.05.
So we will be using a learning rate = 0.01 going forward.

Selecting optimal Epochs





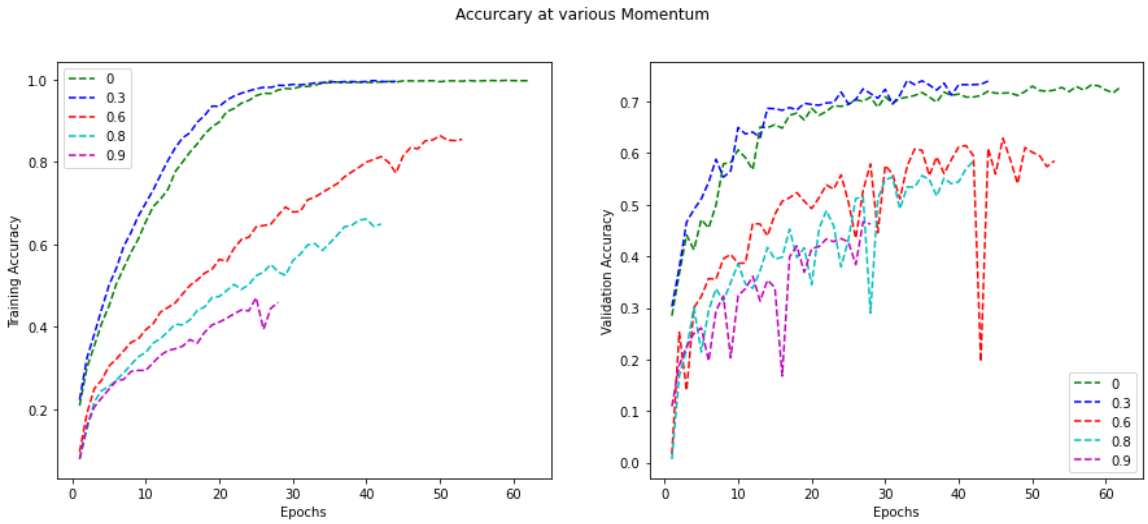
The model with **learning rate =0.01** and optimizer as **SGD** for **100 epochs with early stopping** as callback, results after **62 epochs.**

Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
0.9975	0.0168	0.7282	1.316	0.707	1.440

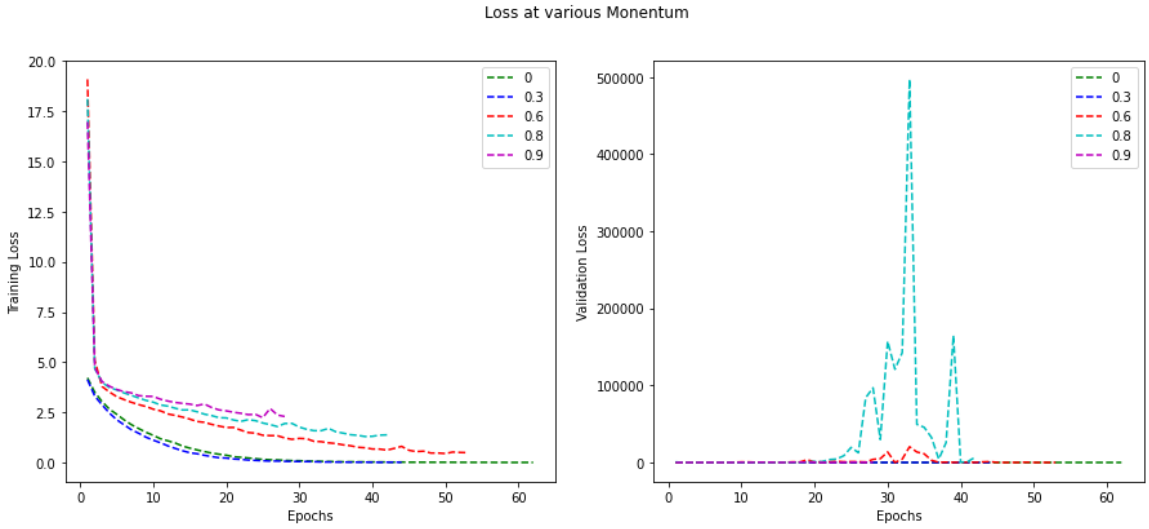
As we can see the result seems pretty good, the accuracy and loss for validation and test set are pretty close.

Selecting Optimal Momentum

The learning rate controls how much to update the weight at the end of each batch, and the momentum controls how much to let the previous update influence the current weight update. Momentum can smooth the progression of the learning algorithm that, in turn, can accelerate the training process.



Train and validation accuracy when model ran with various momentum with learning rate=0.01 and optimizer =SGD



Train and validation loss when model ran with various momentum with learning rate=0.01 and optimizer =SGD

Result after running for 62 epochs with early stopping:-

epochs	*Momentum*	*Train Acc*	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
62	*0*	**0.9975**	**0.0168**	0.7282	**1.316**	0.707	**1.440**
44	*0.3*	0.9957	0.0225	**0.7395**	1.352	**0.712**	1.421
53	*0.6*	0.8553	0.5040	0.5851	20.22	0.577	4239.4
42	*0.8*	0.6497	1.3890	0.5843	6907.3	0.550	220229
28	*0.9*	0.4594	2.3027	0.4623	250.79	0.446	30.720

Momentum = 0.3, seem to be better choice, as it have achieved a good result in far less epochs as compared when momentum = 0, and models with momentum =0.3 seems to have good generalisation compared to other learning rate

Finally, after finding the best configuration based on our dataset are as follow:

1. Optimizer = SGD
2. Learning rate = 0.01
3. Momentum = 0.3
4. Epochs = 62

The final accuracy on the test set obtained with this setting is 71.2%.

	Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
At start	0.9818	0.0992	0.6965	1.3532	0.691	1.434
After Hyperparameter Tuning	**0.9957**	**0.0225**	**0.7395**	1.352	**0.712**	**1.421**

We can see that we were able to increase the models' validation and test accuracy by 2%, just by tweaking a few hyperparameters.

The various plots we saw above show that training a network from scratch with such few samples does not result in a good model that learns well from the training set in addition to being capable of well generalising on unseen data.

To summarise, training a network from scratch is a difficult and time-consuming task in the case of small datasets.

There are many to we can go from here:

1. Oversampling and Undersampling
2. Add new data
3. Transfer Learning

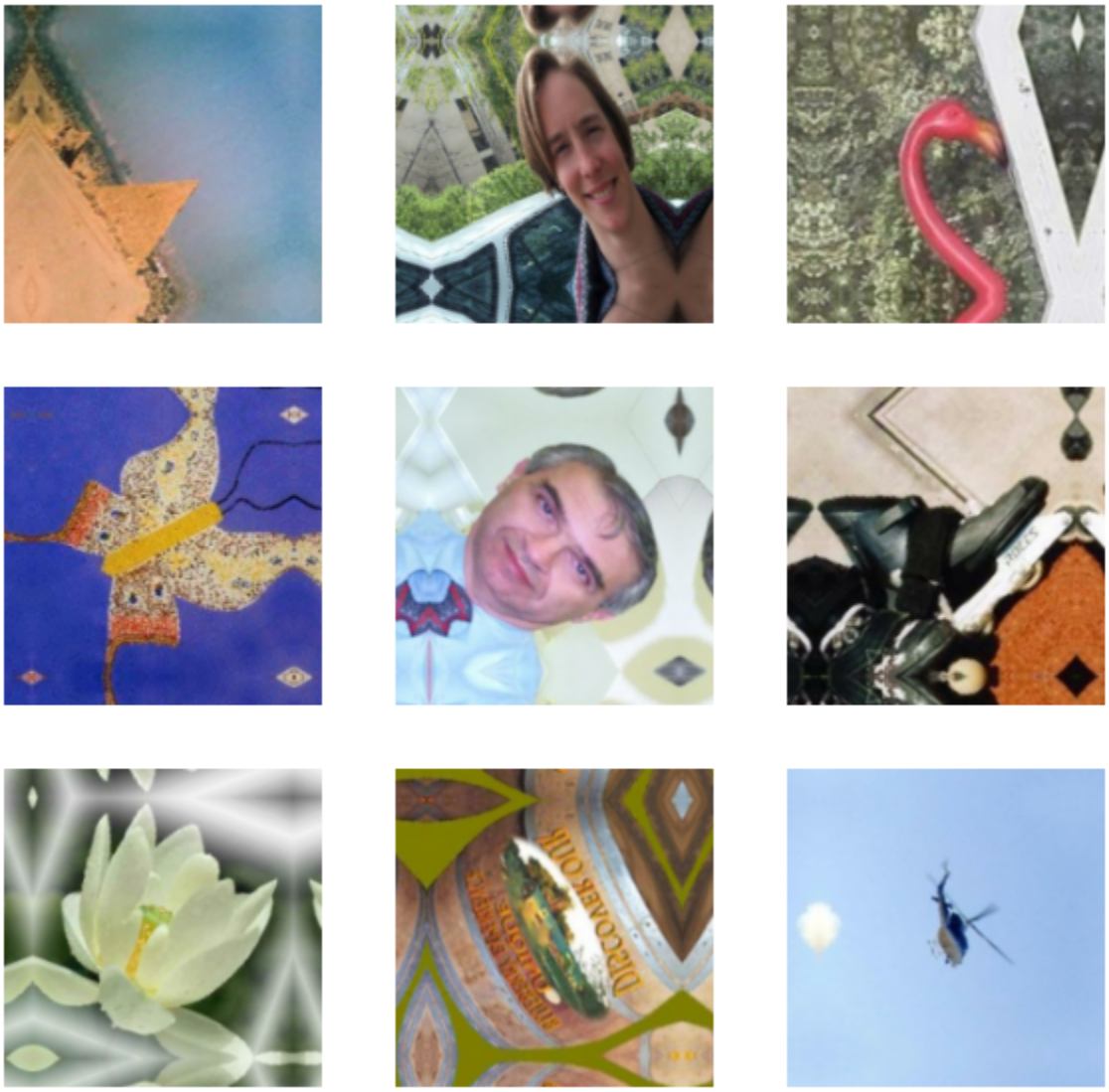
g. Data Augmentation

Image data augmentation is a method for artificially increasing the size of a training dataset by producing altered versions of the dataset's images. This is done by applying domain-specific techniques to examples from the training data that create new and different training examples. The ability of fit models to generalise what they have learned to new images can be improved by training deep learning neural network models on more data. Additionally, augmentation techniques can produce variations of the images. Image data augmentation involves producing altered versions of training dataset images that fall under the same class as the original image. Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more. For example, a horizontal flip of a picture of a cat may make sense, because the photo could have been taken from the left or right. A vertical flip of the photo of a cat does not make sense and would probably not be appropriate given that the model is very unlikely to see a photo of an upside-down cat. As a result, selecting the precise data augmentation methods to be used for a training dataset requires careful consideration of both the training dataset and the problem domain. Image data augmentation is typically only applied to the training dataset, and not to the validation or test dataset. This is different from data preparation such as image resizing and pixel scaling; they must be performed consistently across all datasets that interact with the model.

The following Augmentation will be used:

- Horizontal Flip Augmentation
- Random Rotation Augmentation
- Random Zoom Augmentation

Augmented images using tf.keras layer

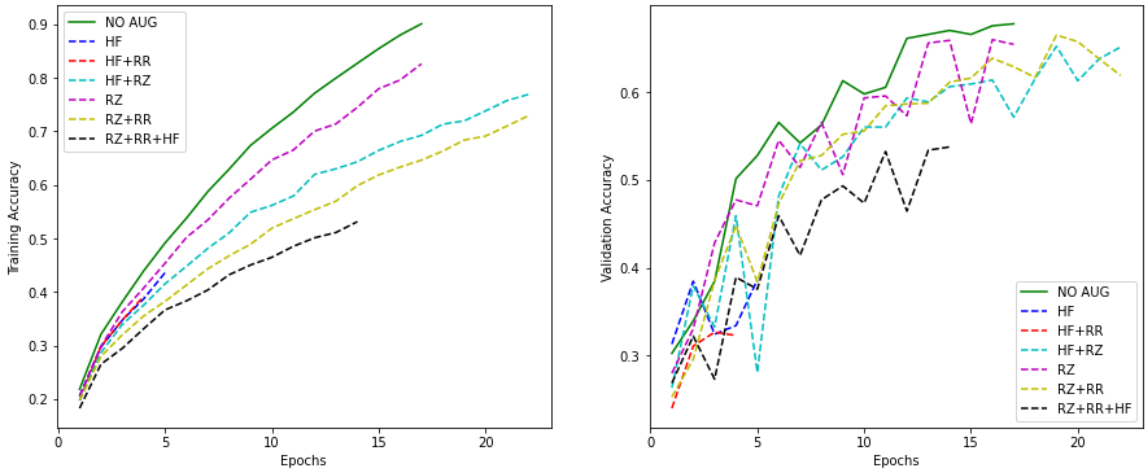


Result after running at following configuration

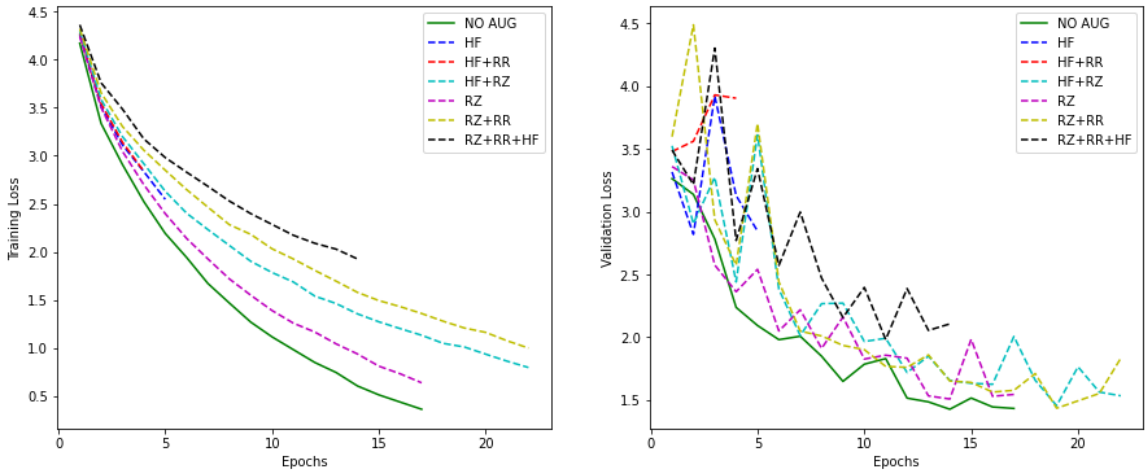
- Learning Rate = 0.01

- Momentum = 0.3
- Epochs = 62
- Optimizer = SGD
- With early stopping as callback

Training Accuracy using various Augmentation Layers



Training Loss using various Augmentation Layer



Data Augmentation	*Train Acc*	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*

No Augmentation	**0.9004**	**0.3648**	**0.678**	**1.433**	**0.677**	**1.422**
Horizontal Flip	0.4361	2.5468	0.386	2.846	0.396	2.817
Horizontal Flip + Random Rotation	0.3919	2.8384	0.323	3.904	0.307	3.9
Horizontal Flip + Random Zoom	0.7686	0.7986	0.651	1.534	0.669	1.492
Random Zoom	0.8257	0.6397	0.654	1.545	0.646	1.586
Random Zoom + Random Rotation	0.7288	1.0009	0.619	1.833	0.618	1.827
Random Zoom + Random Rotation + Horizontal Flip	0.5312	1.9271	0.538	2.106	0.534	2.087

Looking at the above charts and table, it looks like the data augmentation doesn't perform well with our models' architecture.

h. Handling Oversampling and Undersampling

Imbalanced datasets are those where there is a severe skew in the class distribution.

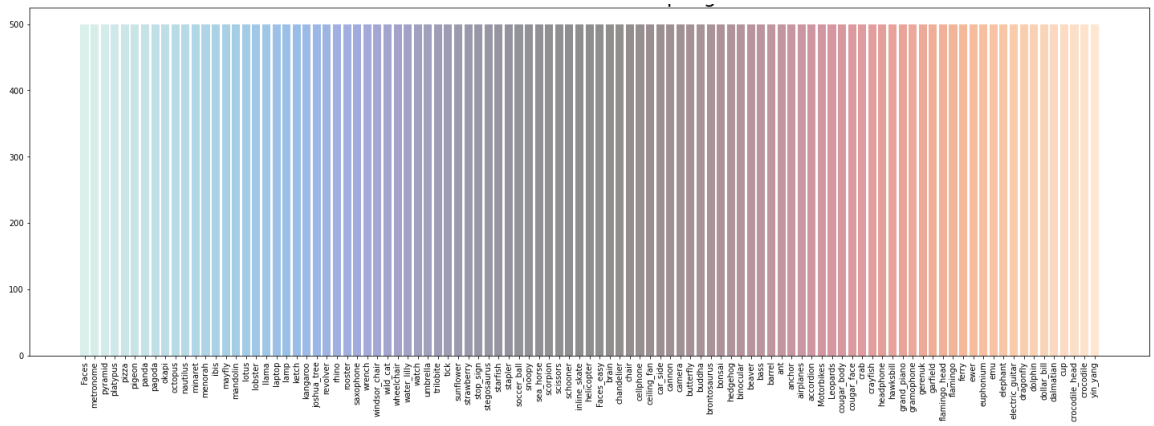
This bias can influence any machine learning algorithms, resulting in completely ignoring the minority.
I will follow the Data Level Approach to treat the Imbalanced Data by using Re-Sampling technique.

One method for dealing with class imbalance is to randomly resample the training dataset. To randomly resample an imbalanced dataset, the two main approaches are to delete examples from the majority class, known as undersampling, and to duplicate examples from the minority class, known as oversampling.

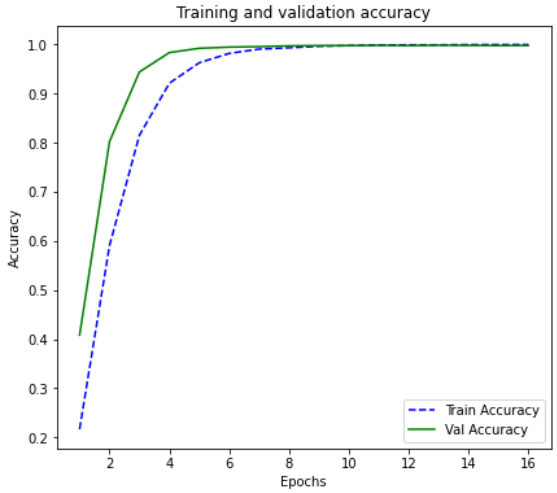
Before going forward:

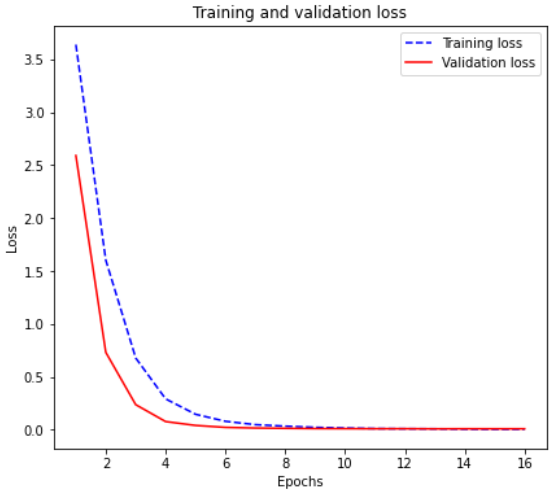
- Overfitting can occur when random oversampling duplicates examples from the minority class in the training dataset.
- Random undersampling removes examples from the majority class, which can result in the loss of information vital to a model.

Results after applying undersampling and oversampling .

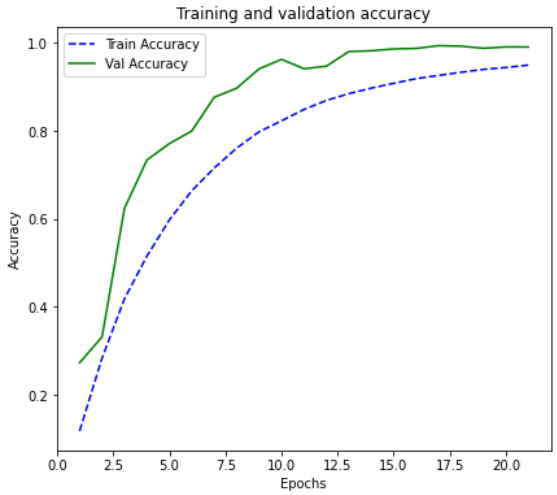


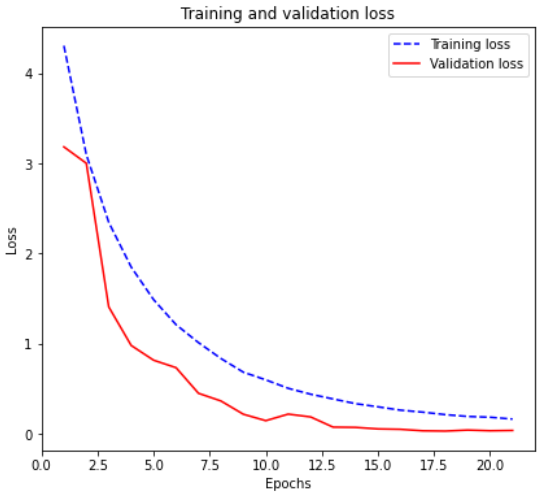
Training & Validation Accuracy & Loss on OverSampled Data





Training & Validation Accuracy & Loss on OverSampled Data with Data Augmentation





	Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
No Augmentation	0.993	0.0068	0.997	0.0105	0.998	0.001
Augmentation	0.9449	0.168	0.990	0.0387	0.993	0.030

Looks like we hit a jackpot! all the accuracy and losses are what we desired.
But these results are too good to be true, there may be many reason:-

- As more than 95 classes were less than 200 images, and only 2 classes have more than 500 images, oversampling these minority classes may have resulted in overfitting to these classes, so as a result of this, high performance.
- Maybe data leak, but not sure, we have taken care of that using stratified sampling.

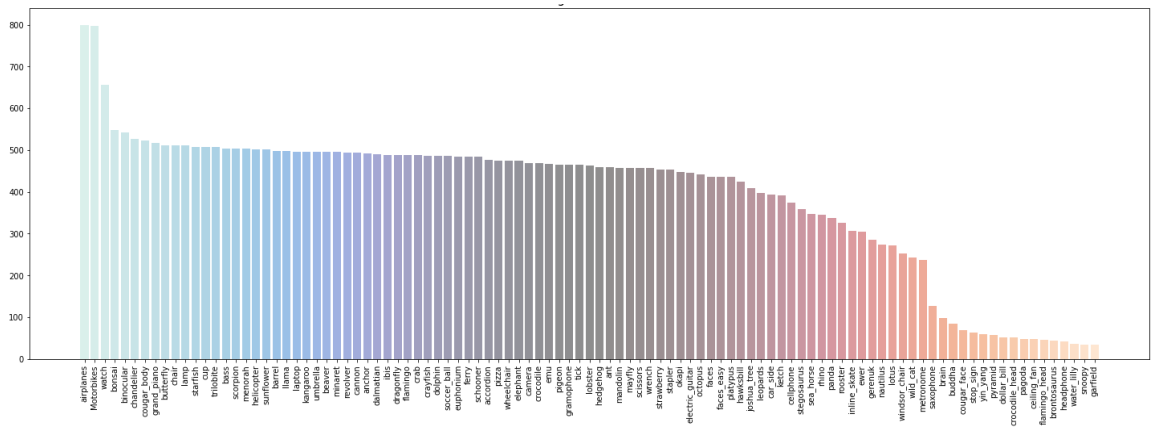
i. Add New Data

For adding new images, I decided to use ImageNet dataset to extend my dataset, ImageNet is widely used for benchmarking image classification models. It contains 14 million images in more than 20 000 categories. The ImageNet project does not own any of the images but they provide a URL list for every image.

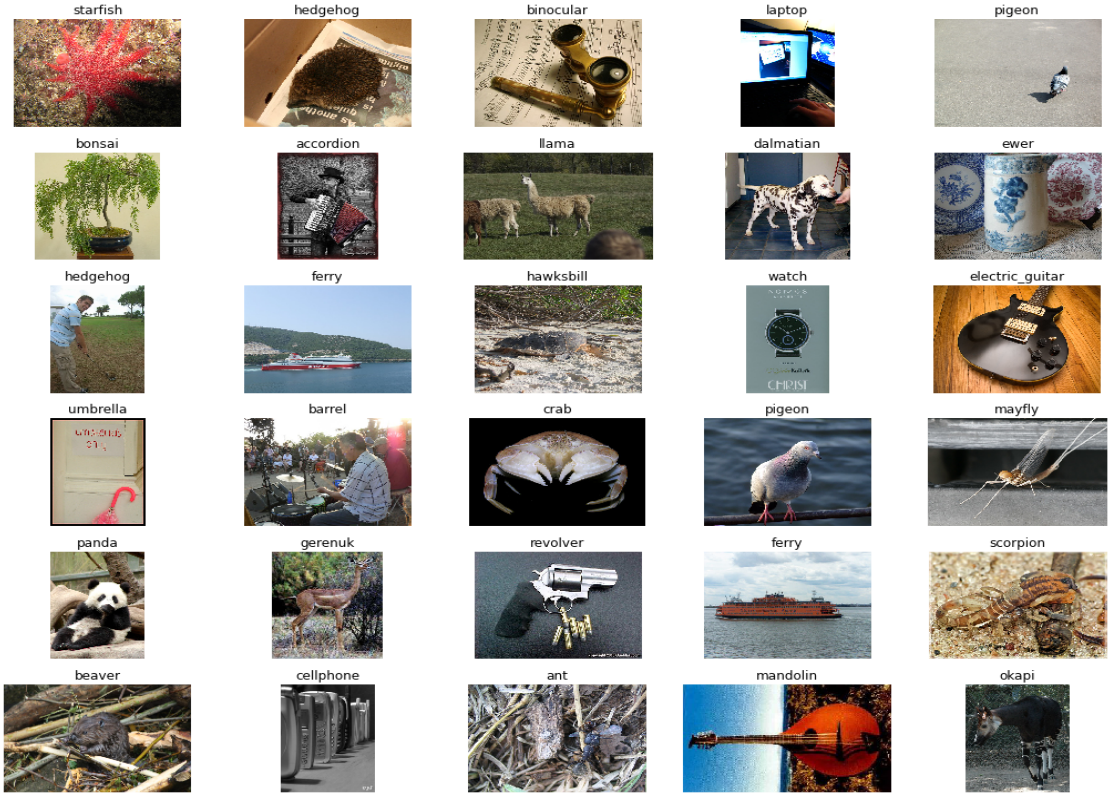
I used script from this source
https://github.com/mf1024/ImageNet-Datasets-Downloader

I upload the my extended dataset to kaggele :-
https://www.kaggle.com/datasets/ashishlotake/extendedcaltech101

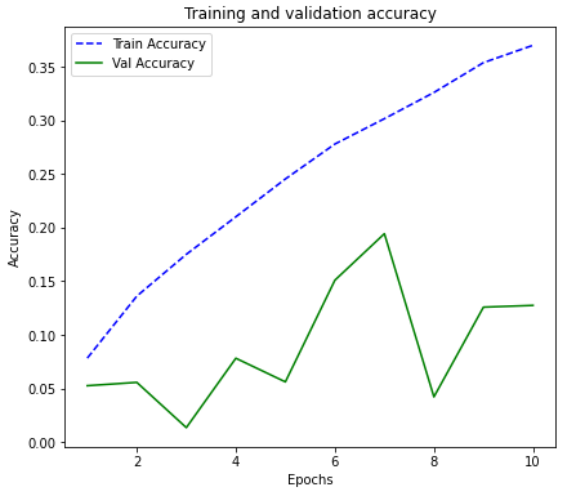
Results after applying increasing images from ImageNet Dataset .

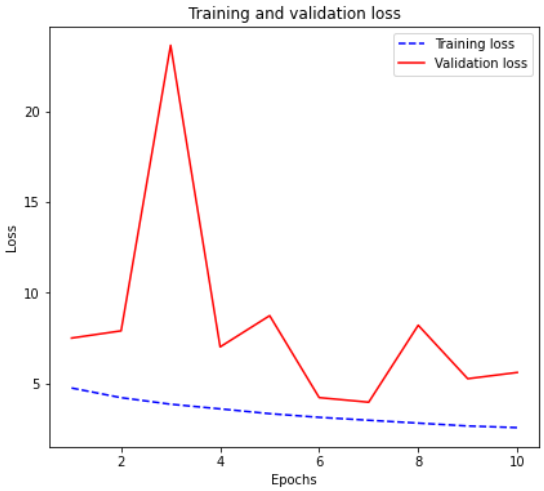


Sample images from extended data

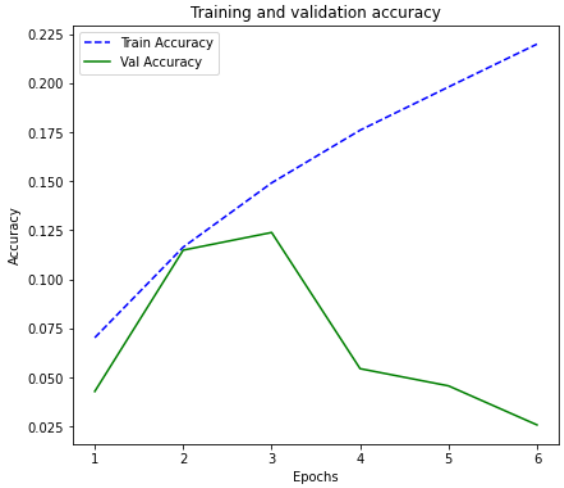


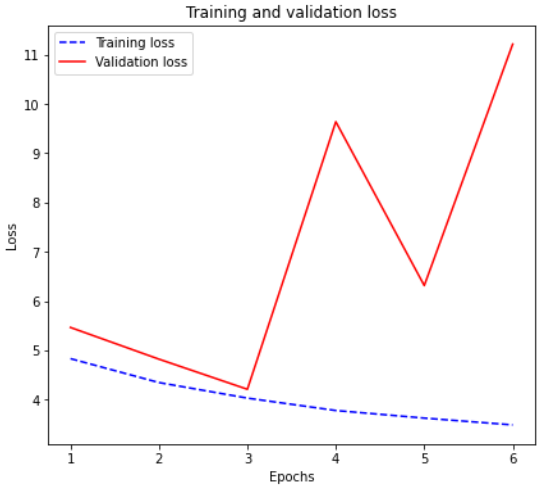
Result after using tuned Architecture



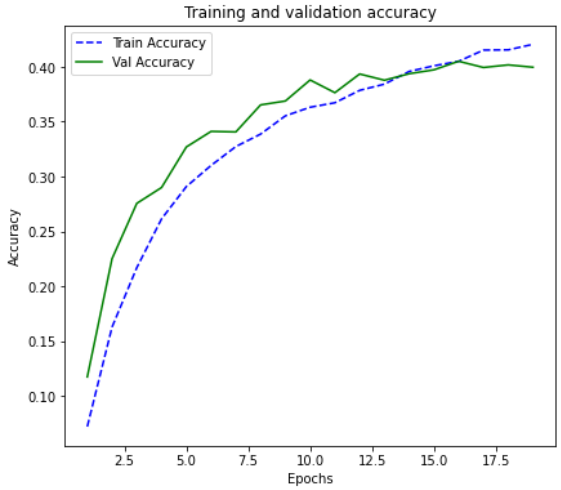


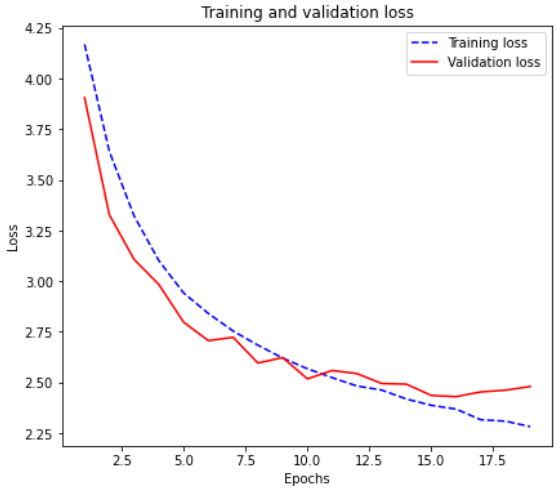
Result after using tuned Architecture + Data Augmentation





Result after using Simple Architecture + Data Augmentation





As we can see the models' performance doesn't seem to improve, maybe the images from the ImageNet and caltech101 have different distribution. This wasn't a fruitful endeavour.

Adding data augmentation, trying to oversample images or extendinding data the model performance doesn't seem to improve, even after hyperparameter tuning the best we could achieve is:-

Train Acc	*Train Loss*	*Val Acc*	*Val Loss*	*Test Acc*	*Test Loss*
0.9957	0.0225	0.7395	1.352	0.712	1.421

j. Transfer learning

Neural networks typically need huge amounts of data in order to be operationally effective, in this case for classification. In order to address the issue of low accuracy in training from scratch, it's possible to use Transfer Learning which uses weights from pre-trained model (already learned on a large dataset, in our case Imagenet) and it is used as starting point for training on our small dataset (i.e. Caltech-101).

Its common and highly effective approach when doing deep learning on small image datasets is to use a pretrained model. A pretrained model is a model that was previously trained on a large dataset, typically on a large-scale image-classification task like imagenet.

If this original dataset is large enough and general enough, the spatial hierarchy of features learned by the pretrained model can effectively act as a generic model of the visual world, and hence, its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those
of the original task.

There are two ways to use a pretrained model: feature extraction and fine-tuning. We will be using feature extraction.

Convnets used for image classification comprise two parts: series of pooling and convolution layers, and at the end with a densely
connected classifier, the first part is called the convolutional base of the model.

Feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output, because the representations learned by the convolutional base are likely to be more generic and, therefore, more reusable. Whereas the representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained.

There are a lot of pre-trained models available, the reasons for using pre-trained models are, training from scratch requires more computational power to train huge models on large datasets, in addition to too much time is required to train the network

from scratch up to a number of weeks. Training the new network with pre-trained weights can speed up the learning process.

Before feeding the net, the images are preprocessed in the same way the images were preprocessed for the pretrained model.

In this report we we will be testing out following models, due to various reasons :

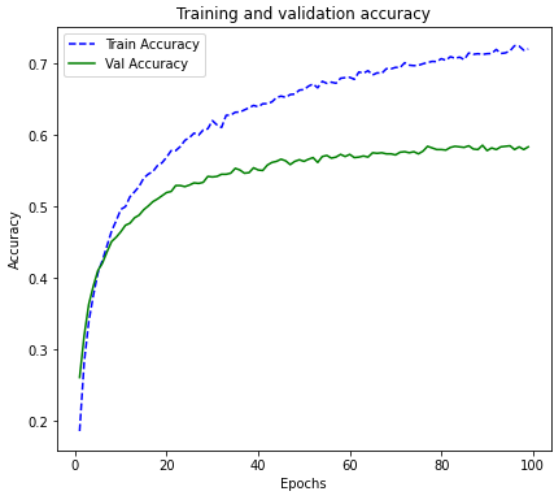
1. Size
2. No of parameters
3. Speed on CPU and GPU
4. Accuracy
5. Deepth

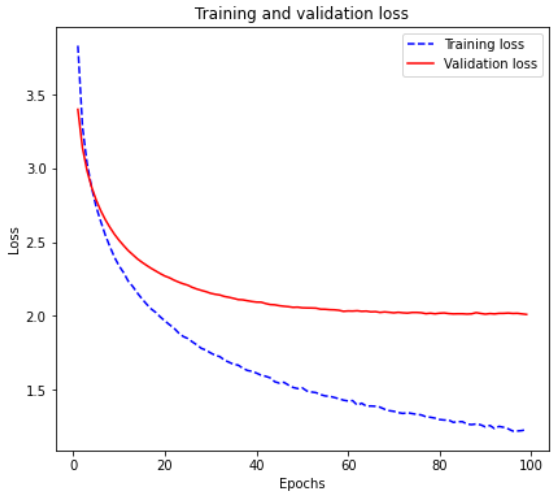
Model	*Size (MB)*	*Parameters*
MobileNetV3	~44	4.5M
ResNet50V2	~98	25.6M
InceptionV3	~92	23.9M

As we have a very small data set, training the full network is not possible, and we will be using weights from a model which was trained on imagenet dataset, as out of 101 classes in clatect101 around 80-85 classes are also in imagenet dataset.

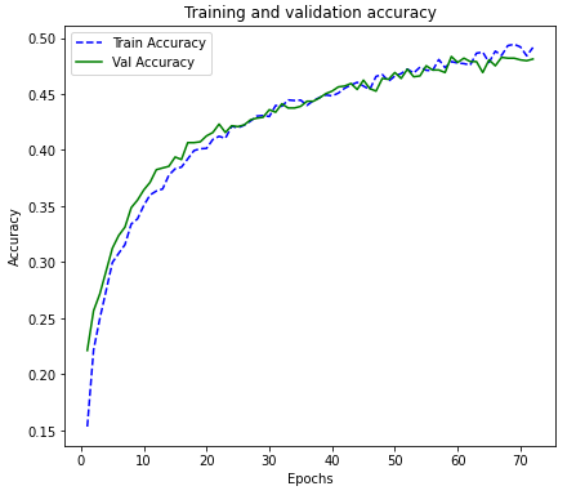
MobileNet

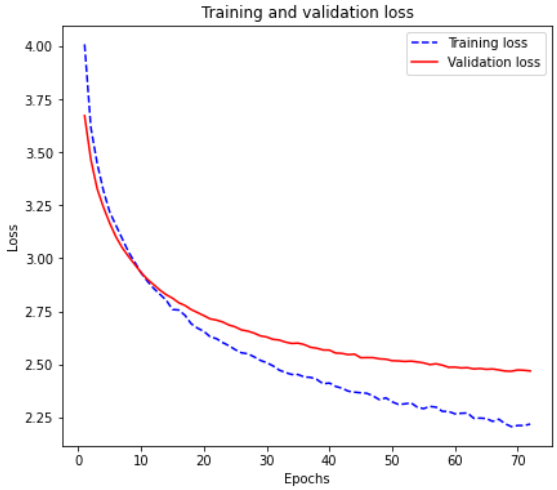
No-Augmentation for 200 epochs with early stopping





Augmentation for 200 epochs with early stopping



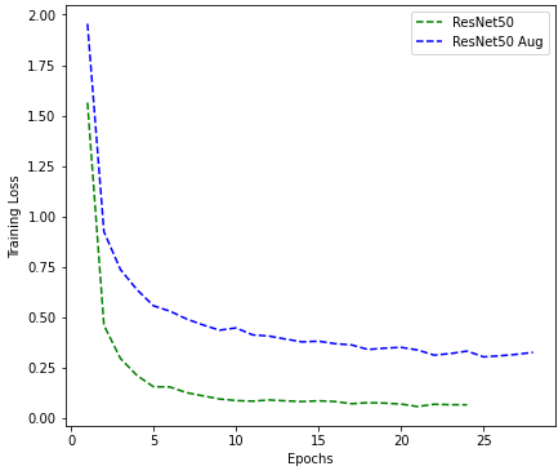


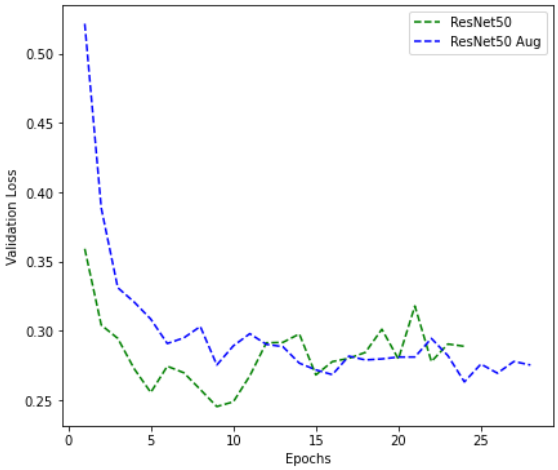
For my scenario, MobileNet seems to be performing worse than the baseline model.

Both Resnet50 and Inception have achieved Train and Val accuracy of greater than 90% less focus on loss.

Resnet50

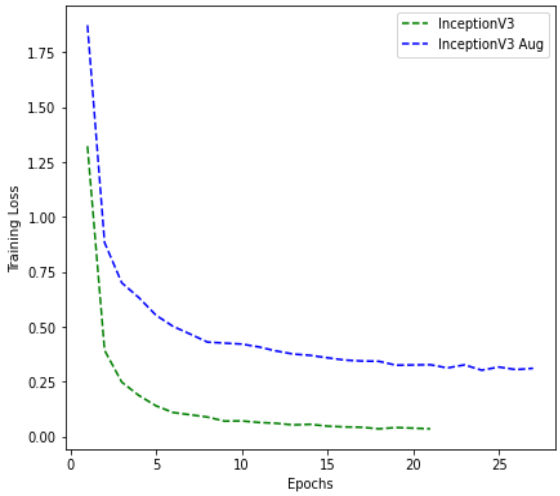
Result for 200 epochs with early stopping

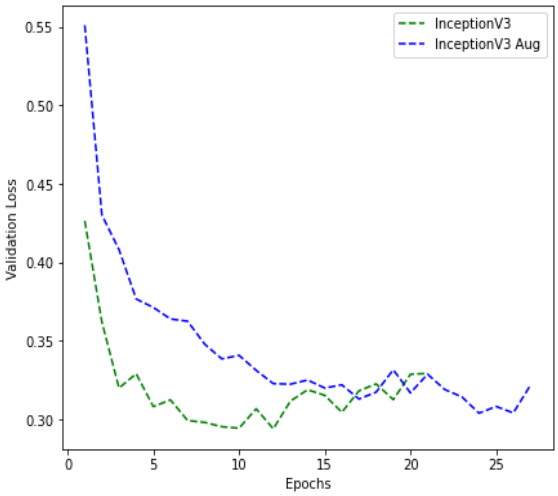




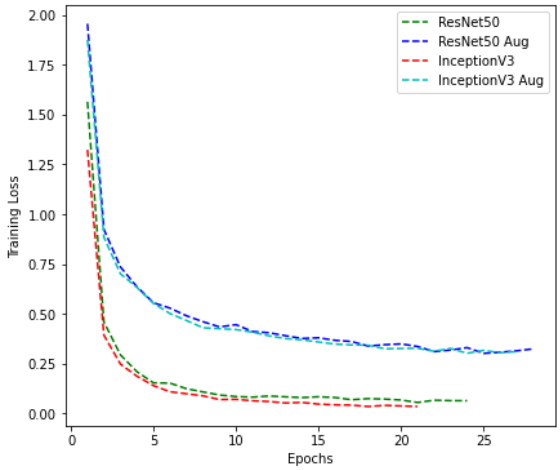
InceptionV3

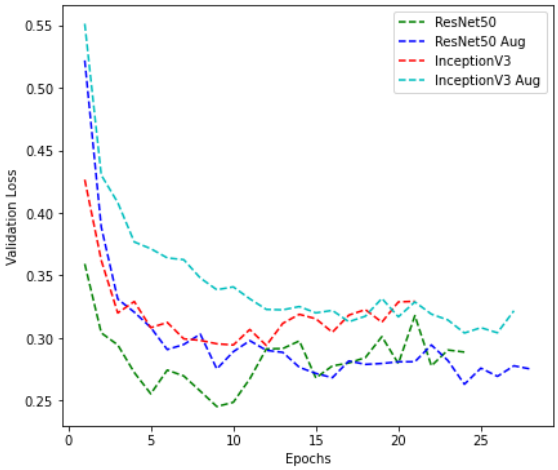
Result for 200 epochs with early stopping





Comparison





	Val acc	*Val Loss*	*Test Acc*	*Test Loss*
ResNet	**0.9307**	0.288	**0.94**	**0.239**
ResNet Aug	0.9232	**0.2753**	0.922	0.2657
Inception	0.9307	0.3294	0.9240	0.3352
Inception Aug	0.9133	0.3129	0.918	0.3043

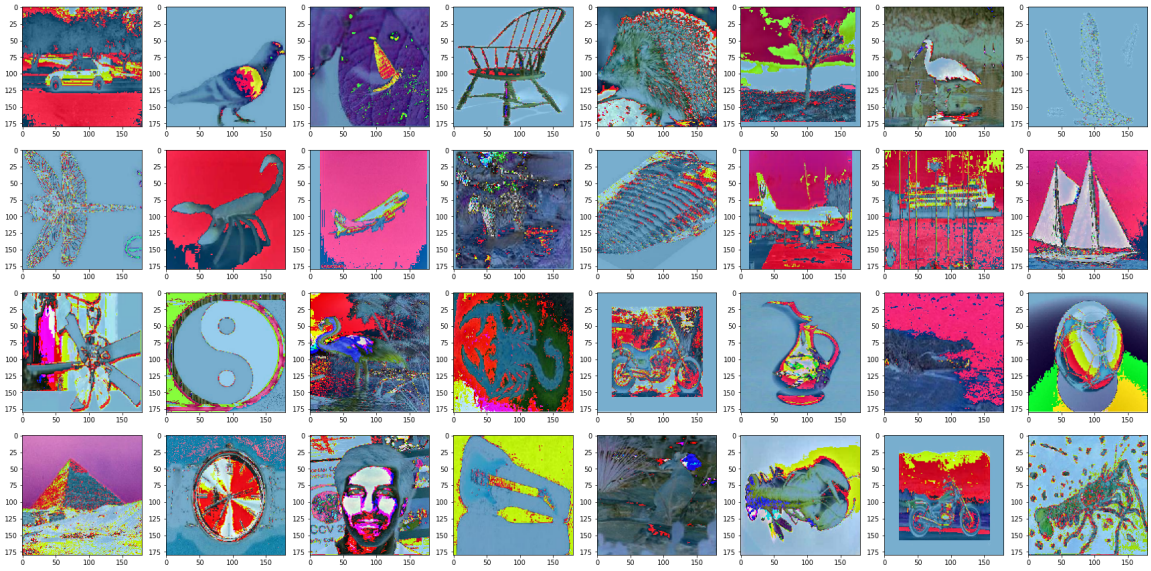
It looks like Resnet with and without data augmentation seem to be performing well, with the default configuration, and the size of this is around ***~100 MB***

Finalising the model

After doing various experiments from hyperparameter tuning, adding new data to trying out various pre-trained models on ImagenNet with and with data augmentation, we decided to go with ResNet50V2 from keras as the results were great, around 20% improvement from our baseline model on both test and validation set.

As stated earlier, that we need to preprocess images base on the pre-trained model we choose, let's look at preprocessed images from resnet50 convolutional base:-

Result after passing images using ResNet50 preprocessing from imagenet



For retraining the model we will load the model and continue the model training where it was left, without compiling, so all the weights learned by the classifier will be copied from the previous model and the model will start learning from that weight.

We will copy a few data points from the original dataset and then copy new images to that and we will start retraining by combining new and old.

k. Models Interpretability

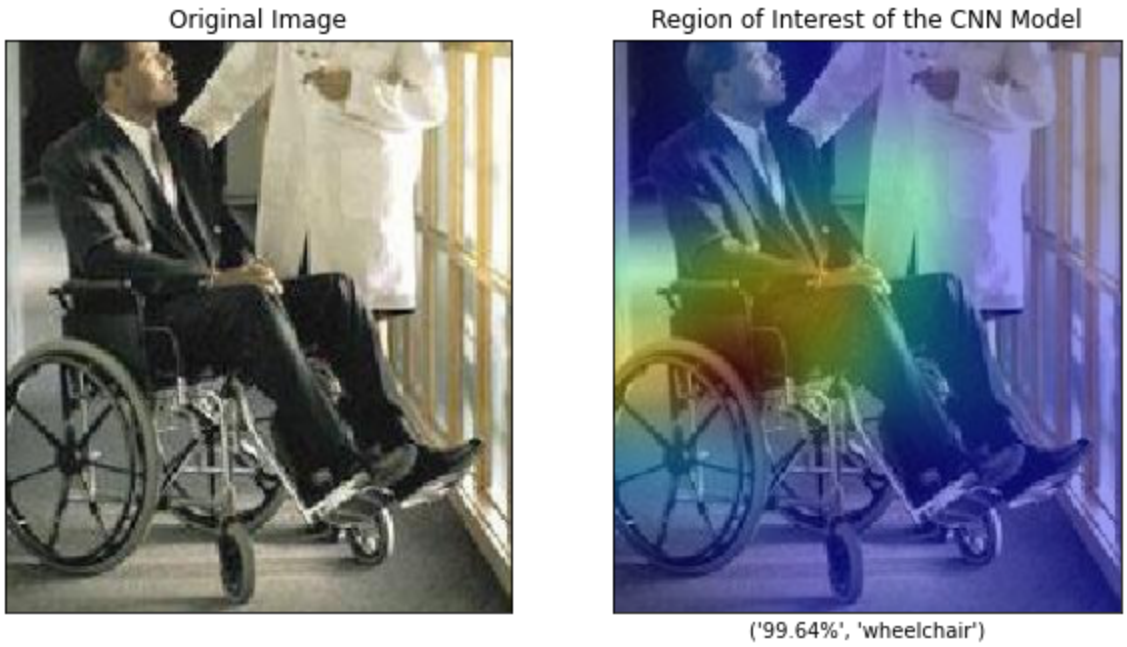
When developing a computer vision application, one of the most fundamental issues is interpretability: why did your classifier think a particular image contained a guitar when all you see is a spanner?

Deep learning models are frequently referred to as "black boxes" because they learn representations that are difficult to extract and present in a human-readable format. Although this is true for some deep learning models, it is clearly not true for convnets. The representations learned by convnets are highly interpretable.

We will be focusing on *Visualising heatmaps of class activation in an image*.[8] is useful for understanding which parts of a given image led a convnet to its final classification decision. This is helpful for “debugging” the decision process of a convnet, particularly in the case of a classification mistake.

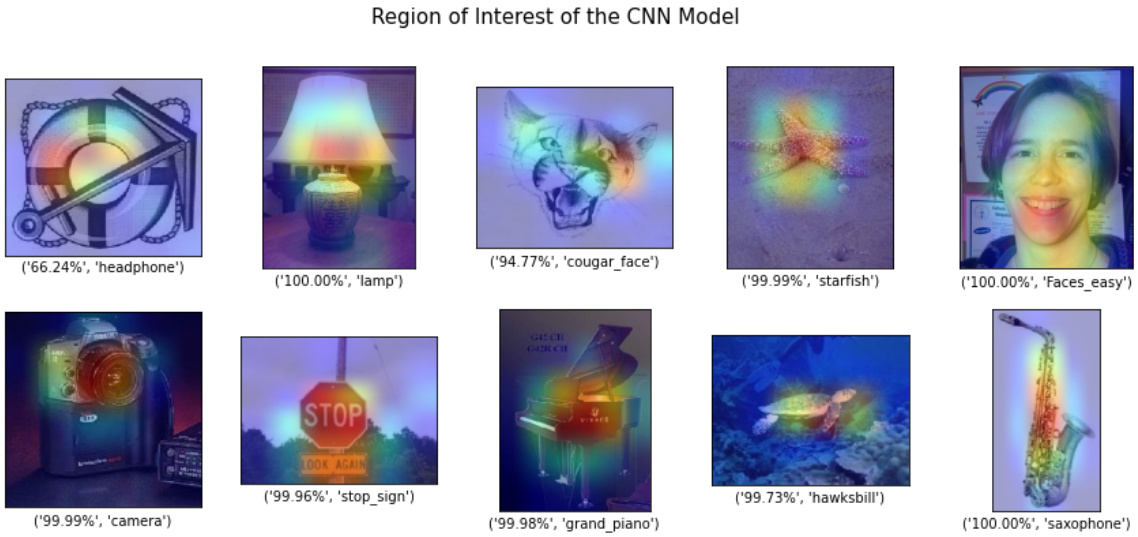
Class activation map (CAM) visualisation is a broad family of methods that involves creating heatmaps of class activation over input images.
A class activation heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class under consideration.

CAM: Visual Explanations from Deep Networks via Gradient-based Localization.[9] Grad-CAM consists of taking the output feature map of a convolution layer, given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel. Intuitively, one way to understand this trick is to imagine that you’re weighting a spatial map of “how intensely the input image activates different channels” by “how important each channel is with regard to the class,” resulting in a spatial map of “how intensely the input image activates the class.”



By looking at the above images, we can say that after the model detected a person and wheel in close proximity, the model decided it was a wheelchair with 99.64% confidence.

Result for various images



5. Conclusion

We have successfully created an active labelling system, using transfer learning which can achieve an accuracy of above 90% on unseen data.

Allowing the user to input raw data, label if the model performs poorly and then give the user correctly classified images , by placing them into the correct directory.

6. References

Reference

[1]-https://go.cloudfactory.com/hubfs/02-Contents/3-Reports/Cognilytica-Research-Report-2019.pdf?_hsmi=75150081

[2]-https://arxiv.org/abs/1512.03385v1

[3]-https://arxiv.org/pdf/1704.04861.pdf

[4]-https://www.researchgate.net/profile/Hemantha-Kumar-Kalluri/publication/333666150_Deep_Learning_and_Transfer_Learning_Approaches_for_Image_Classification

[5]-https://link.springer.com/article/10.1007/s12652-021-03488-z

[6]-https://arxiv.org/abs/1708.07120v3

[7]-https://data.caltech.edu/records/mzrjq-6wc02

[8]-https://www.manning.com/books/deep-learning-with-python

[9]-https://arxiv.org/abs/1610.02391

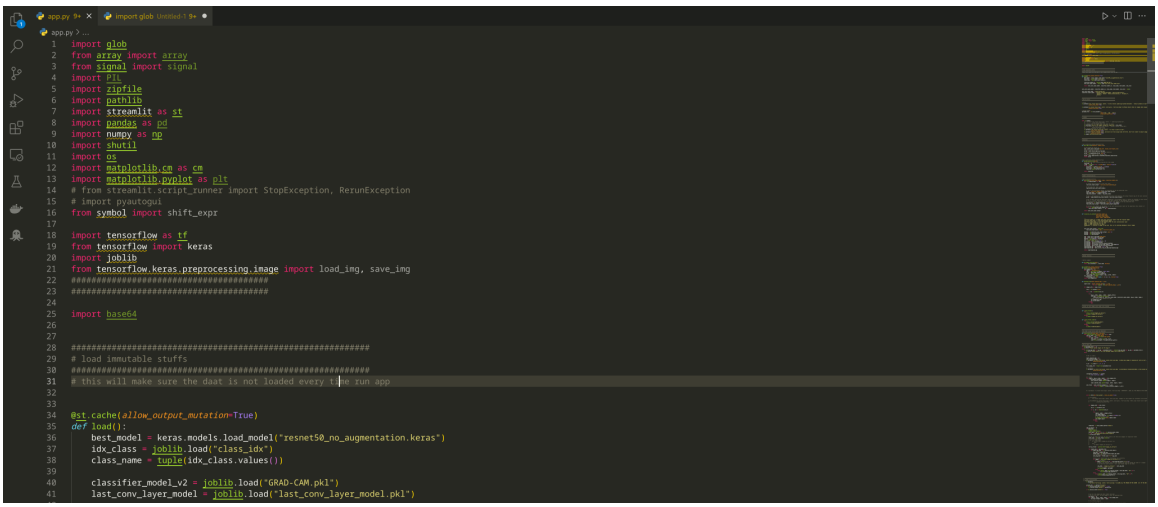
7. Deployment

Deployment is the project's final phase, in which we deploy the entire machine learning pipeline into a production system, in a real-time scenario. In this final stage, we must deploy this machine learning pipeline in order to make the research available to end users.

The model will be used in a real-world scenario, taking continuous raw input from the user and then giving them labelled data as output.

App Building:

In this step a web application developed using Streamlit. It provides a frontend to the user for uploading multiple images.
Streamlit is an open source python framework for building web apps for Machine Learning and Data Science. It simply converts simple Python scripts to functional, interactive, and shareable web apps within minutes. Because it's designed toward showcasing data science products, it natively supports the data models and graphs with major data science libraries, such as pandas and matplotlib. Thus, it requires little learning curve if you already know Python well.



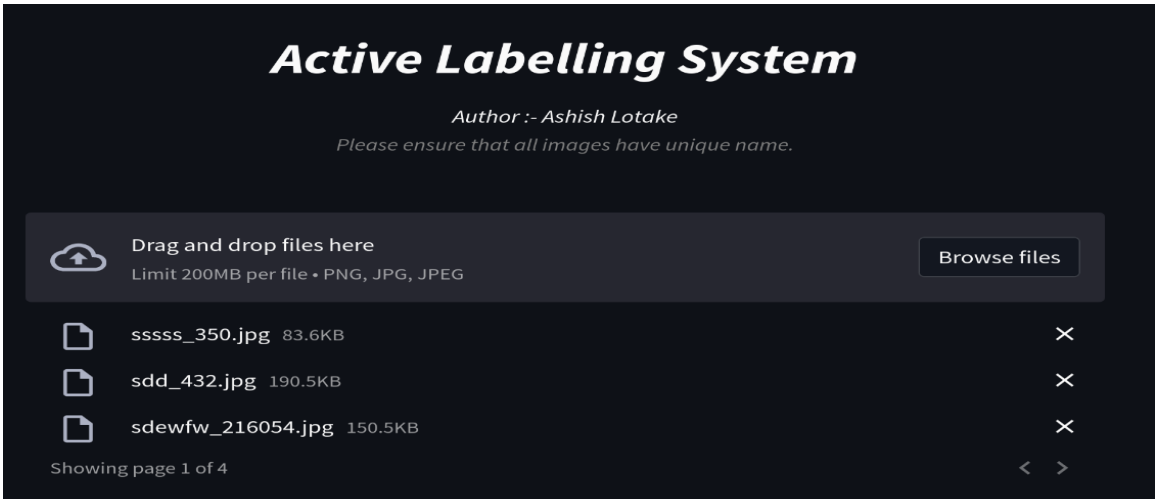
We can our web app using streamlit in local with the following command:

```
Streamlit run app.py
```

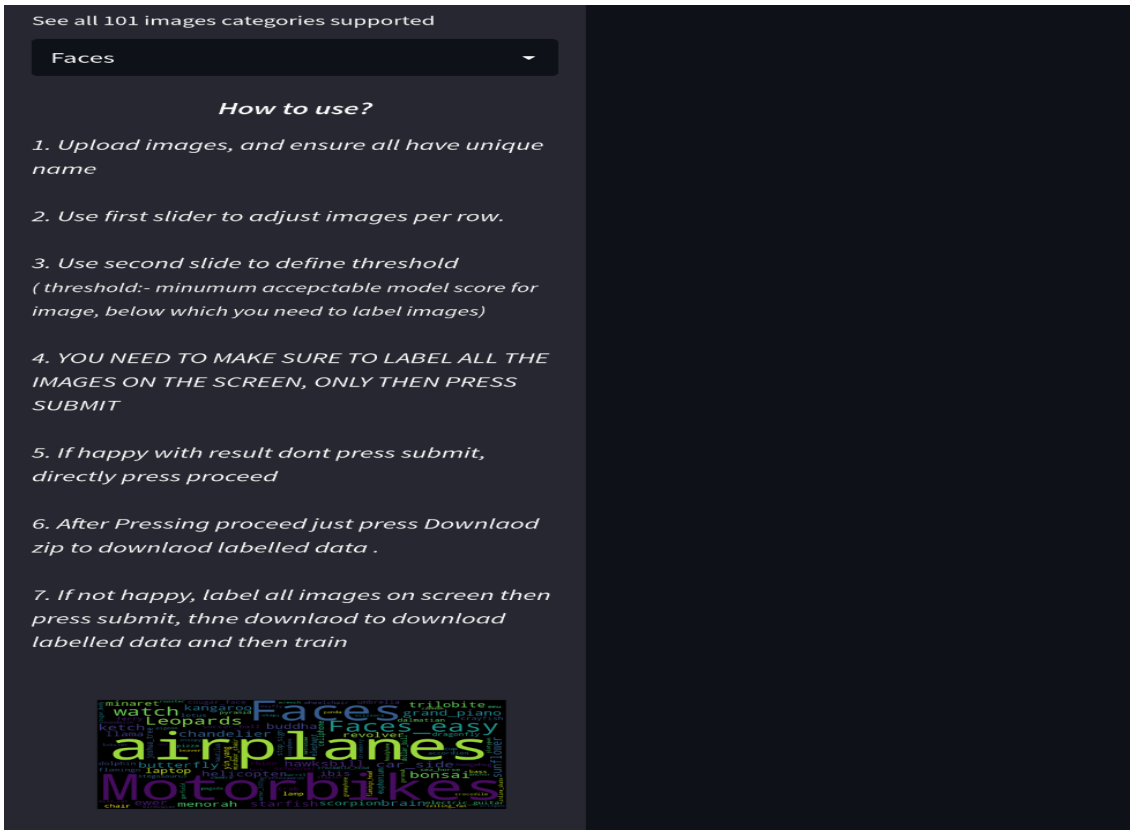
Functioning of Streamlit app:-

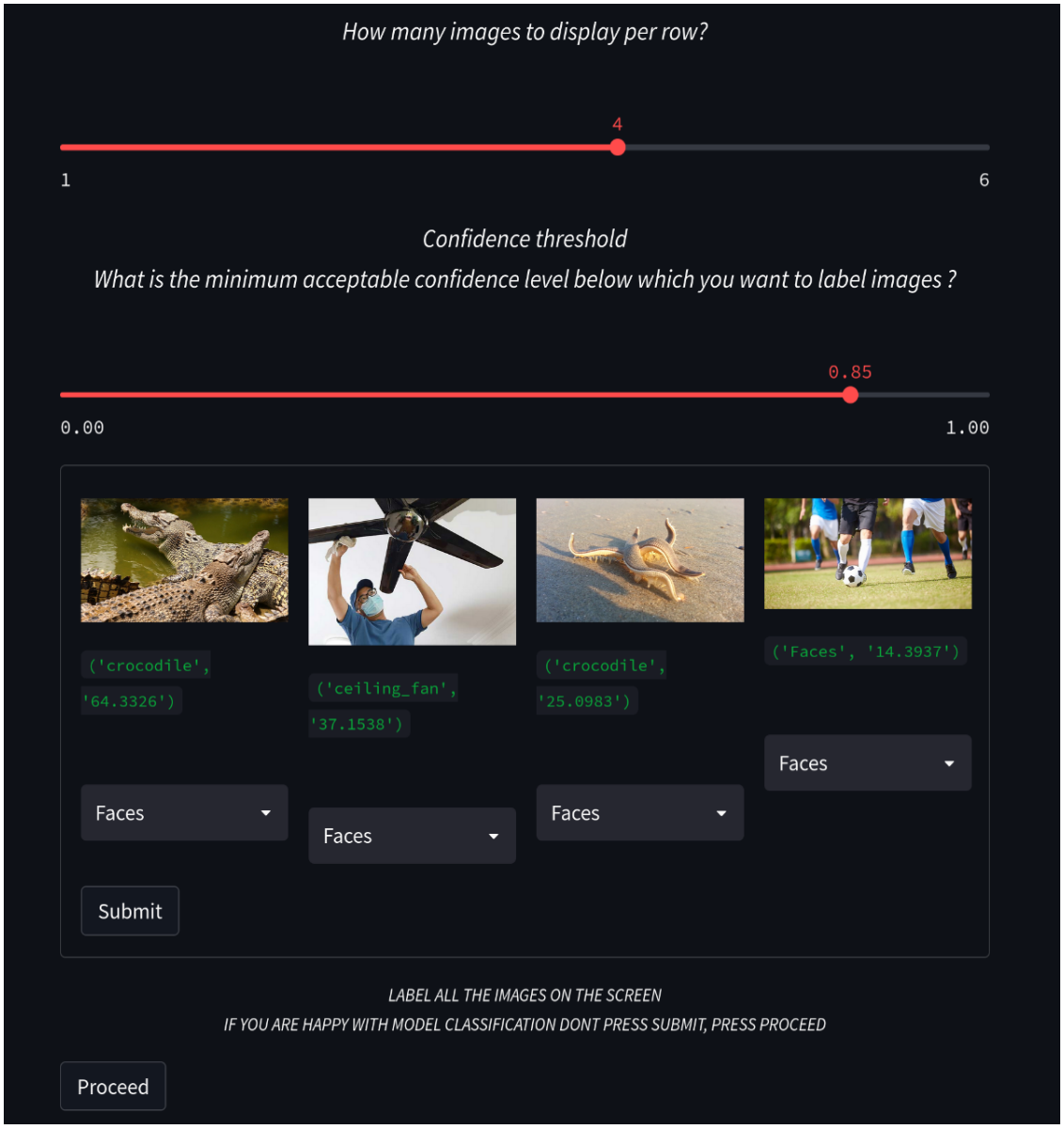
Title and Image Uploader

Flowing code renders title and streamlit have a built in function to allow used to upload multiple images, streamlit also allow to use html inside markdown.



Slidebar



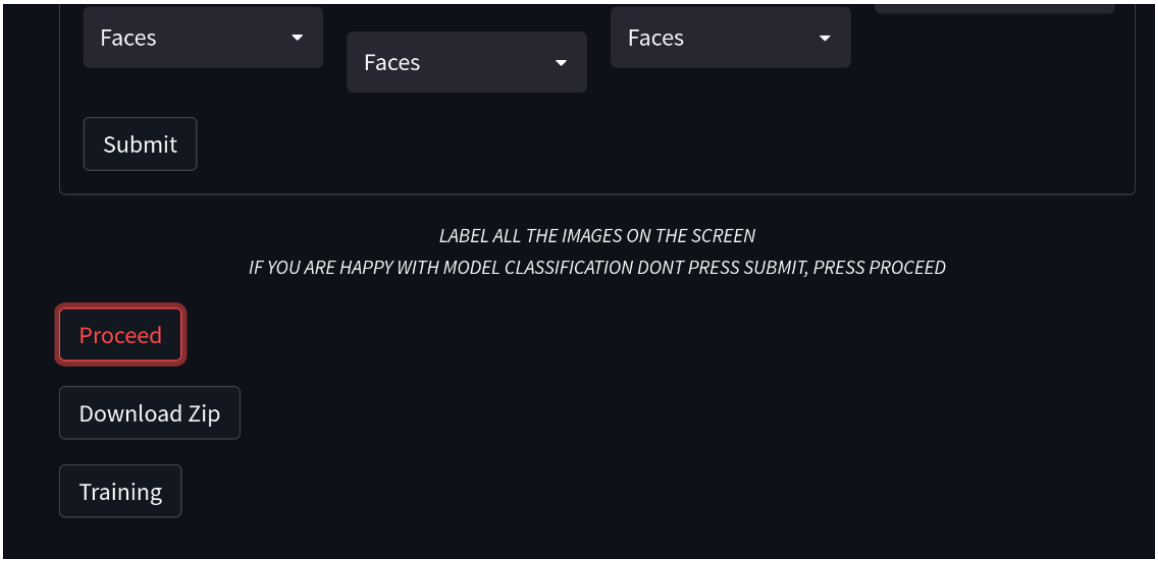


Giving users two sliders, one for adjusting how to images per row, then second one it to set the minimum threshold below which the user will label images.

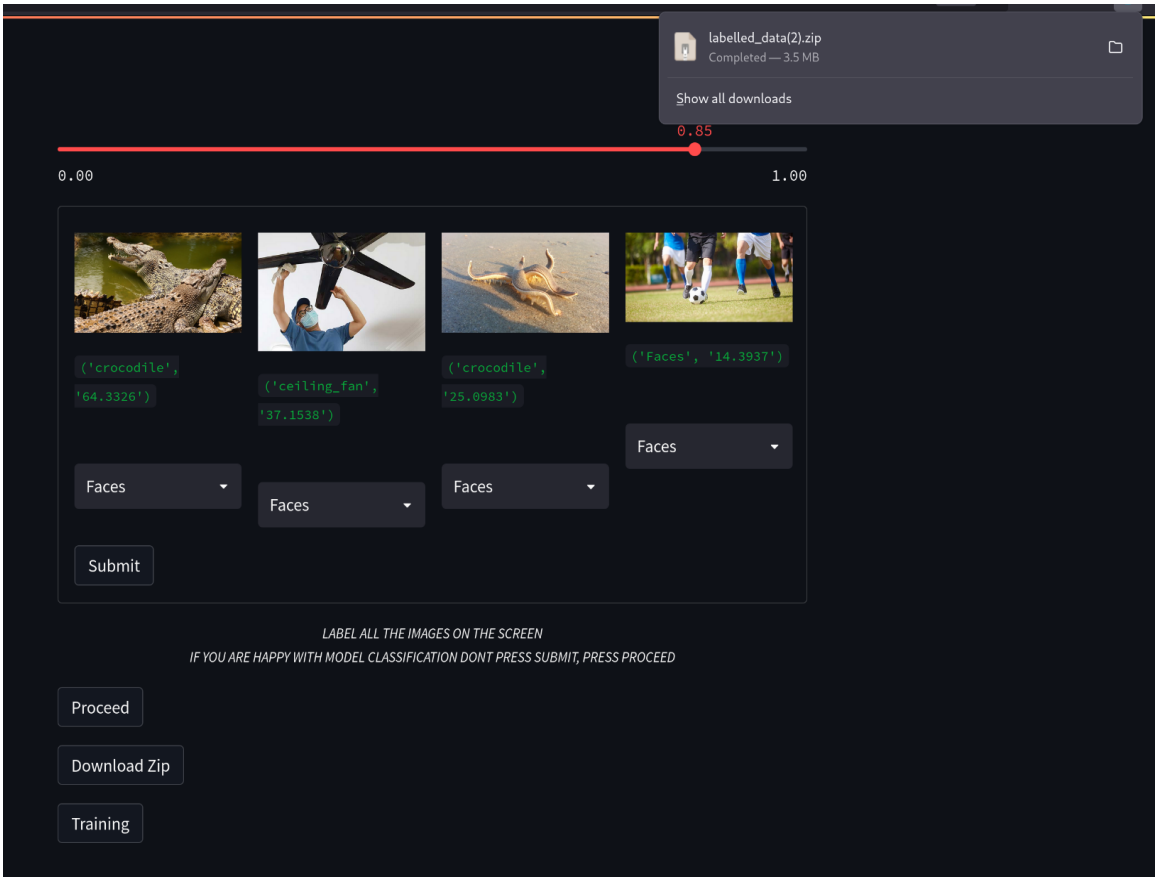
Creating a streamlit form inside which user upload images will be shown and below each image models prediction and model confidence along with a select box, from which the user can select the correct class for an image when model misclassified. And when the user has labelled all the images then he/she is supposed to press submit or don't label any image on screen just press proceed.

If the user pressed submit then the following button will be shown, at this moment there

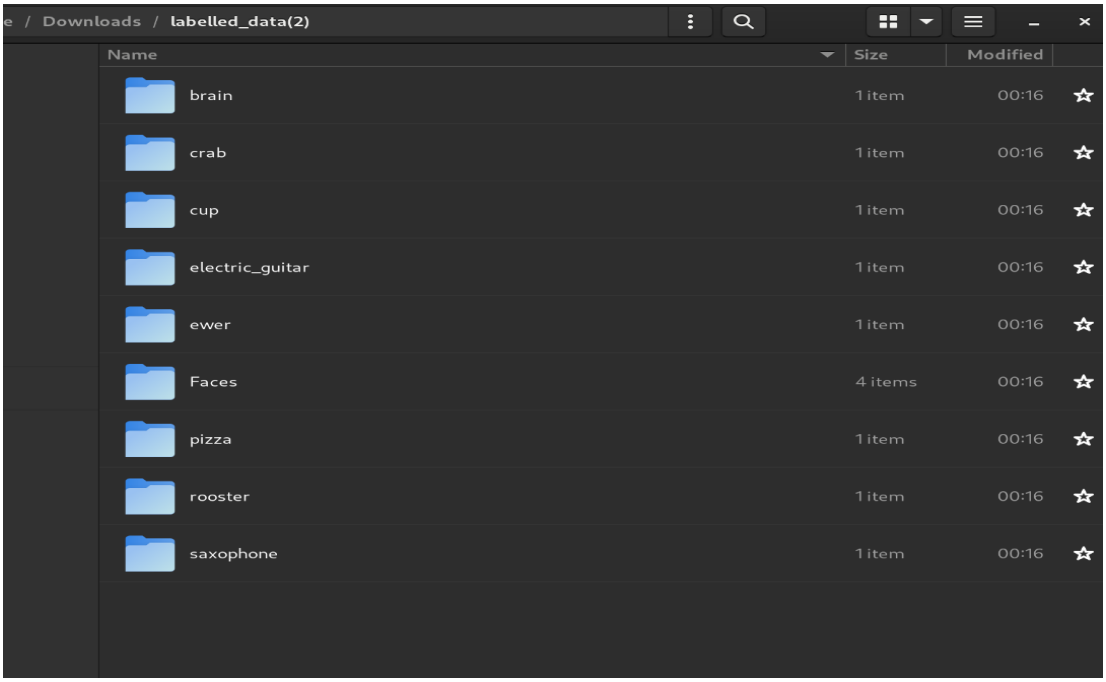
is nothing to train.



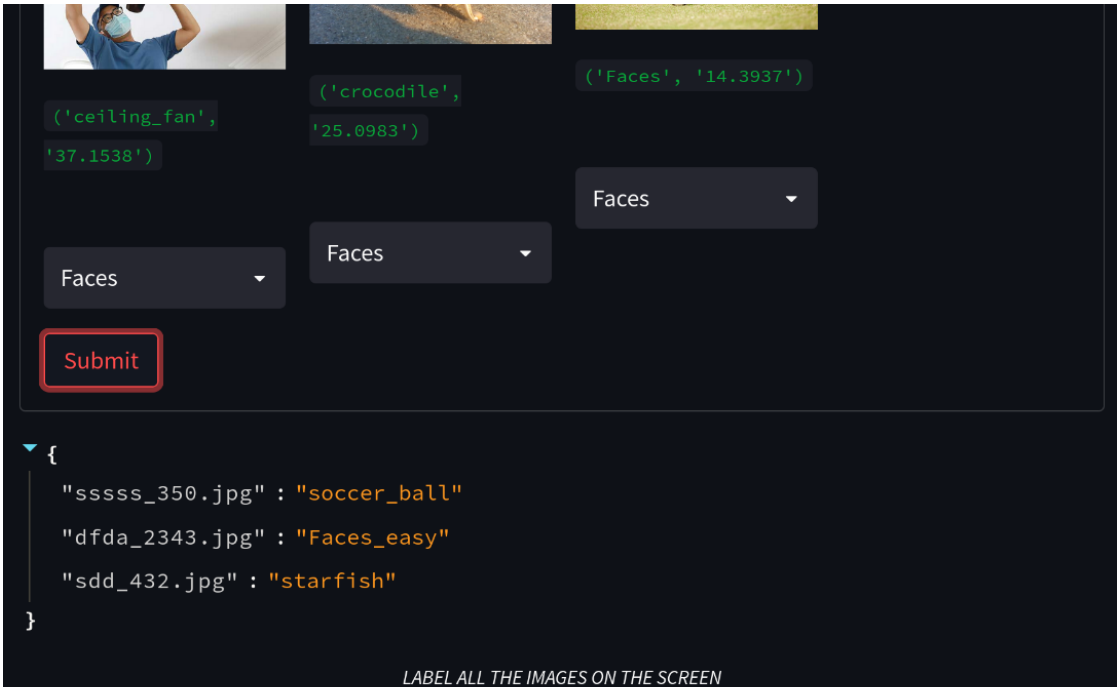
When user press download button a zip file will be downloaded



This zip file will have folder for each category, and images placed inside them correctly



Now if use label the images on screen then:-



Now after that if user presses proceed then training then models' retraining will begin:-

Now that your app is ready, it is time for deployment, before that we will wrap everything into a docker container, so we can take this docker image and deploy it anywhere.

Dockerizing

Dockerizing is the process of packing, deploying, and running applications using Docker containers.

Docker is an open source tool that ships your application as a single package with all necessary functionalities. Docker allows you to pack your application with everything it needs to run (such as libraries) and ship it as a single package - a container. Containers are made up of images that specify their exact contents.

Reason for creating a docker instead of direct deployment :-

- Easy to use:- Docker simplified the way we deploy applications. You do not distribute the software as source code - you send the binary image of a part of your disk
- Fast:- Docker containers are just sandboxed environments that run on the kernel. You can create and run the containers in seconds
- Able to create a reproducible environment:- Wrapping everything into containers means that the application you build runs on other devices without friction.

Step for Dockerizing your stremlit app :-

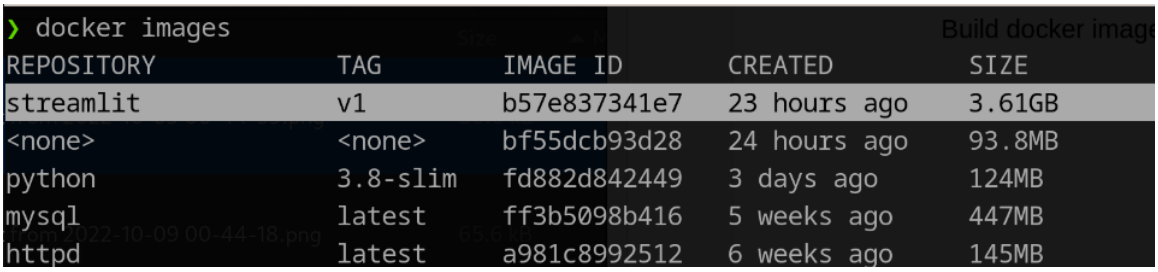
Create a **Dockerfile** in folder root directory:-

```
FROM python:3.8-slim
EXPOSE 8501
WORKDIR /app
COPY . .
RUN pip3 install -r requirements.txt
ENTRYPOINT ["streamlit", "run", "app.py", "--server.port=8501",
"--server.address=0.0.0.0"]
```

- Pull python3.8 slim images from docker hub
- Expose the port 8501 so to access streamlit from browser
- Create a new directory
- Copy everything from root directory to app folder
- Install the project dependencies
- Configure a container that will run as an executable

Build docker image:-

```
docker build -t streamlit:v1 .
Docker images
```



Run the Docker container

```
docker run -p 8501:8501 streamlit:v1
```

Your streamlit app has been dockerized and now it can be deployed anywhere,and you can watch while it launches.
Now we can access the web app by going to localhost:8501