Google landmark recognition

ABSTRACT:

Google landmark recognition is a research problem in the fields of image recognition and computer vision. It contains of large dataset of images in which each image needs to labeled with a landmark if any landmark is present in those images. In this paper we want to design a model which solves this problem. We discuss about different state-of-art image recognition models and there performances. We will be discussing different image classification techniques to benchmark an efficient landmark recognition model.

INTRODUCTION:

Image classification has gained a lot of attention in last decade regarding the importance of its applications on different domains like image searches in e-commerce applications, automated image organization in image applications, facial detection applications, object detection for AI and robots, and many more.

In day to day life people often take many pictures with landscapes and different landmarks. when looking back into these pictures after a long time they might not realize where the picture is take. what the landmark in the picture is. A machine learning model can help in these situations. By implementing a image classification model we can predict the to which landmark the current image in the matched and then retrieving the details of the matched image helps to get the details of the query image.

Even the outlook solution to these problem looks simple to im- plement such a model is a challenging task. There are large number of landmarks in the world. Recognising a image into one of those large set of classifications is a real challenge. In a regular image classification problem even to classify a image into two labels thou- sands of images are used to train each class. Collecting such a large number of images for each landmark image is also not possible. So the images available for most of the landmarks are very small. This makes this task even more difficult to solve. These challenges makes the landmark recognition task as a research field in areas of computer vision, image recognition and image classification.

RELATED WORKS:

[1] Karen Simonyan et al. worked on how the depth of the convolution network effects the accuracy of the large scale image recognition tasks. They also made study on how the deep convolution networks performs on implementing the convolution filters of very small sizes (3x3) on basis of ImageNet datasets. This model pro- duced state-of-art results with the ImageNet dataset and and also used in many other image classification problems as a pre-trained model on large set of images.

[2] Hideaki Yanagisawa et.al. examined the effectiveness of object proposals for manga (Japanese comics) object detection. This paper focuses on acquiring metadata from manga images for automatic recognition of manga content. This paper examines the effectiveness of manga object detection by comparing Fast R-CNN, Faster R-CNN, and SSD. R-CNN and Fast R-CNN uses Selective Search technique to generate region proposals. Faster R-CNN generates region proposals using CNN layers called Region Proposal Network (RPN). Single Shot MultiBox Detector (SSD) performs object classification and box adjustment for small sections. These methods proved to be effective to natural images but they were not tried for classification of manga images. The manga objects to be detected are panel layout, speech balloon, character face, and text. Manga109 dataset is used for training and evaluation of detectors. Evaluation results showed that R-CNN and Fast R-CNN are effective for objects with clear boundaries such as panel layout and speech balloons, and Faster R-CNN is effective for objects whose boundaries are unclear such as character face and text. SSD showed difficulty in detecting manga objects in whole image which can be improved by dividing image into small regions.

[3] H. Noh et al. proposed a local descriptor extraction feature for a large scale image retrieval system.Most of the image

recognition models deals with extracting a global feature vectors This approach extracts local features. It is a CNN based annotation detection of images specially trained on landmark image dataset to extract the local features for the given images.

[4] He, Kaiming et al. proposed a residual Networks for deep learning networks which actually overcomes the problem of deep neural networks. It's been a convention that usually increasing the network depth improves the performance of the model as the net- work understands the data better and provides better generalization to the model. But it also increases a lot of computation and degradation i.e the accuracy gets saturated and starts decreasing. This is not due to over fitting but due to depth of the layers. To overcome this problem they made residual networks which propagates the original input from previous layers along with the changes value of the input. This is used to as a replacement for the identity function for the model they build over a plain shallow model and a much deeper model of the same shallow model with more layers and expecting same at least as good results from the deeper model when compared to the shallow model.

[5] Nitesh Ramakrishnan et.al. Proposed an object recognition algorithm to classify and detect objects such as sharp edged knife, utensil, and water bottle present in visual images using a new CNN model with modified VGG architecture to improve the training and testing accuracy of the existing models such as AlexNet, ZFNet and VGG13. To gain better performance in accuracy, a lightweight 11 layer CNN model based on VGG architecture was proposed. It consists of an input layer, multiple convolutional layers, max- pooling layers, fully-connected layers followed by an output layer with 3 nodes to classify different objects like knife, utensil and water bottles. The paper sows that due to limited weight parameters the proposed model executes faster in Intel CPU. The proposed model overall achieved 98% training accuracy and 92.2% testing accuracy. The detection accuracy of this model is good as well as the training parameters are less and the computation hardware required is minimum. The performance of the proposed model was found to be better when compared with the other models considering sharp objects detection.

[6] B. Cao et al. proposed an interesting observation in the field of image retrieval. Instead of using global descriptor only they combine the advantages of local descriptor along with the local descriptor for the image retrieval. Where global features increases the recall of the retrieval task and local features matches the best of those retrieved images and rank them which increases the precision of the retrieval model.

[7] M. Gogoi et al. in this paper gives an introduction to auto encoders and stacked auto-encoders, with SVM and softmax regression as classification layer. The paper focuses more on stacked auto encoders. The dataset used is popular MNIST of handwritten letters. Main focus is on implementation and comparison of unsupervised learning model such as auto encoder in conjunction with a classification layer on top. The effect of varying number of neurons in the hidden layers of autoencoder is studied. The performance

is measured with regard to classification accuracy. The weights in the hidden layers are adjusted using supervised learning algorithm in the backpropagation step. The models in comparison are 2SAE (having 2 encoding layers) with Softmax layer and 3SAE (having 3 encoding layers) with Softmax layer and Auto-encoder with SVM as classification layer. The best performing model was Auto-encoder with SVM as classification layer on top, having accuracy as 99.8 % and error rate as 0.2 %. The paper clearly states the performance of particular model is best only for the current dataset and might not be the same for other datasets.

[8] Y. Li et al . proposed the model based on the combination of convolutional auto-encoder and convolutional neural network termed as a semi supervised model to assist image classification. The main highlight of this model is that it gets good performance even when the number labelled data points reduced by more than half. The usage of convolutional auto-encoder is that it can capture the 2d structure of the image. CAE models include convolutional, pool- ing, fully-connected, deconvolution, unpooling and loss layers. The competence of the model is that it can work limited number of labelled data too. The dataset used is a collection of dogs and images.

[9] Girshick el al. in this paper made a model for object detection. In object detection it draws a bounding box around the object of interest. This is challenging to implement with a typical CNN because the length of the object can be variable and the number of occurrences are also not limited. To address this problem various algorithms have been proposed. First we take a look at Region based Convolutional Networks (RCNN). These RCNNs take input an image and set of proposed regions of interest. These regions are obtained with the help of a selective search algorithm. Now each of these regions is connected to a CNN for extracting the features. Next we connect these features to an SVM to confirm the presence of an object. Next we then try to create boundaries around the object. This method although highly accurate has huge performance impact due to the region proposal algorithm. So quicker methods like SPP nets, fast-RCNN, faster-RCNN have been proposed.

Google landmark recognition

[10] Jie Hu et al. proposed Squeeze and Excitation networks that provide an improvement over existing Convolutional neural networks (CNNs). In CNNs the focus is on constructing features by combining the spatial and channel wise information of the image at each layer. However CNNs do not focus on inter channel relationships. This is the problem that the proposed method aims to deal with. To capture these inter channel relationships the authors make use of a Squeeze and Excitation (SE-Block). The SE Block has two primary functions. First is the Squeeze: which produces a channel descriptor by converting each feature map into a unique value, which can be achieved either by pooling. The second is the Excitation operation: which learns non linear interaction between channels by applying activations to the output of Squeeze operation. Finally the output is generated by scalar multiplying the value generated by Excitation operation and the feature map generated by convolution operation. This output is subsequently fed to the next layers. This methodology can be easily incorporated to any existing CNN. The only change includes replacing the existing architectural blocks with the new SE-Blocks. The effect on performance by the addition of these blocks is very minimal considering the improved model.

Test 1:

Dataset:

Source of the dataset is: https://www.kaggle.com/google/google-landmarks-dataset

The dataset available is in the form of a .csv file which includes SNO, ID of the image, URL to download the image class label of the image (landmark_id). Total number of images in original dataset consists of 1 Million images.

Sample 1:

From this dataset around 1,30,0000 Images are taken, while maintaining the proportion and properties of data to be same as the original dataset. The CSV contains image URLs, and for each image URL there is landmark ID and ID field. ID is unique identifier of an image. The landmark IDs can also be called as class labels. The sample contains class labels from 1000 to 2999.

The images are downloaded from the URLs available in the csv file. Further, three folders are created i.e. for train_images_model, test_from_train_images and validation_images_model. In data preprocessing the directory structure is created in such a way that each of these folders contain subfolders with class name as folder name. Each subfolder consists the images corresponding to a particular that class.

After that data cleaning is done. This step involves removing all the corrupted image files. These are identified when an error was thrown while running the code.

For each of the 2000 classes, the train and validation split are 80-20 i.e. 80 % will be train and 20 % will be in the validation. For every class 1% of images is taken as test along with other images which are not in these 2000 classes are also added to make the problem reflecting the original dataset test images.

After data cleaning the training set contains 108181 images and validation set contains around 19826 images. For test, the images are picked from the training set. There 1885 images in the test set. The model learns on the train dataset and after each epoch the model is tested on validation dataset to check for model performance, overfitting etc.

Sample 2:

In second sample of dataset we took classes from 5000 to 6999 classes. These classes consists of 1,70,000 training images. We found that one of the class alone has 50000 images and two more classes consists of 30,000 images. From the past experience of the project the model was able to train from lesser images also and this huge images of a single class might make the model more towards that single class. To overcome that problem we reduced the size of these classes to 3000 images. So that they are big when compared to remaining classes and the model is efficient enough to learn about a class with 3000 images. These conclusion was made based on observing the results from the previous experiments of the models.

After reducing the images of some classes the dataset is reduced to 1,20,000 training images which belongs to 2000 classes. We again did the data cleaning part which we done on the previous dataset sample to overcome the error while proceding with futher training and testing of the model. As part of this we have to remove all the images which are corrupted and unable to process by the PIL library. These images are removed using PIL library itself. After this data cleaning step we left with around 90,000 images. These images are divided into training dataset and validation dataset with a split of 80% and 20% respectively.

After the datacleaning as we are using datagenerators from keras library we need to classify the images into their respective class labels and create folders for all their classes. So we created 2000 folders named with 5000 to 6000 folder names. These folders are created in two root folders which are Trian_images_model and validation_image_model for training and validation. The respective Train images and validation images will be moved to their respective folders in these root folders.

Final sample:

As to make the sample much reliable as the final model we combined all the three sample we trailed. Which resembles the 50% of original Google Landmark Recognition Dataset. The original dataset consists of nearly 1 Million images which are classified into 14,000 classes. Our final sample consists of nearly Half Million images which are classified into 6,000 classes. These gives the complete challenges of the real Google Landmark recognition challenge. The number of classes are large with some classes are having even less than 10 images to train.

As we already cleaned the data for different samples we are able to merge these datasets with all the data pre-processing steps are already done. Created a new folders for Train_images_model, Validation_images_model and merged all the three previous sample and make a single sample out of all the samples. These sample consists of 523603 of overall train images

before data Pre-processing. The train and validation split is same as we did for the previous samples that is 80% train images and 20% validation images.

Approach 1

Model Training

VGG16, VGG19, Inception, etc are some of the CNN architectures could be found in the Keras library. To predict the landmarks we used the VGG16 pre-trained on Google ImageNet dataset. VGG16 is a convolutional neural network model which achieves 92.7% test accuracy in ImageNet dataset. ImageNet is a dataset consisting around 22,000 categories approx of over 15 million labeled high-resolution images. Generic features from our images were successfully captured using ImageNet Weights which made VGG16 (with ImageNet weights) as our model of choice in the process of transfer learning. Transfer Learning is a process of storing the knowledge that has been gained while solving one problem and applying it to some other different problem but a related problem. Our approach is to use feature extractors from the bottleneck layers of VGG16 and train the top 4 layers to classify the landmarks using the features that are more relevant to the project.

The Four layers that are initialized for training are 1 Flattening, 2 ReLU activation and 1 Softmax. Flattening is basically a conversion to vector from a two-dimensional matrix of features, that can be provided into a fully connected classifier. The ReLU (rectified linear activation function) is a linear function that checks if the input is positive or not. If it is positive it will output the input directly otherwise if it is less than 0 it will output zero. It is used to overcome non-linearity in a data. The final output layer in a neural network is the softmax that is responsible for performing multi-class classification. The final classification is in the range between 0 and 1, and the one with highest value is taken as the output. Overall their sum is 1.

The entire VGG16 model was compiled with the 4 initialized layers were added on top of VGG16 bottom layers. To make the network learn in a better way for the images outside the images consisting in the ImageNet dataset, we chose to use only bottom 16 layer weights and fine tune the top layers on our dataset. This saves a lot of computation instead of training on complete layers.

Number of layers to train

We used VGG16 architecture till the last convolutional layer i.e till 16^{th} layer, and added 4 layers (fully-connected layers) on top as sequential layers. So overall the model contains 20 layers. The best accuracies were achieved with only with layers 1 Flattening and three dense layers that are 2 ReLU activation, 1 softmax and weights from bottom 16 layers which are frozen.

Google landmark recognition

Optimizers

Optimizers are basically algorithms that are used to reduce losses by changing some of the attributes of the network such as learning rate and weights. Adagrad, momemtum SGD and Adam are some of the optimizers we tried. For these optimizers, Decay, Momentum and Learning rate are the hyperparameters that we tuned. Adam turned to be working the best for our problem after few trials.

Image Augmentation

A large amount of training data is required by deep neural networks to achieve good accuracy. Image augmentation is used when we have a very little training data and with that constraint we want to build a powerful image classifier. Image augmentation helps to increase the performance. Rotation, shifts, shear and flips, etc. are some of the different ways of processing or combinations of these processing that Image augmentation applies on the images. We observed the effect of each parameter on images and observed the effect of each on the results of our model.

Batch Size

Throughout the network, the number of samples that will be propagated is defined by the batch size. Keeping in mind the memory constraints of our systems batch size was to be decided as the size of the dataset is very large. We were able to go up to batch size of 431 with fine accuracy and run times.

Number of Epochs

An epoch is basically one cycle through the full training dataset and training a neural network usually takes more than a few epochs. If the training data is given as an input to a network for more than one epoch in different patterns, we hope for a better result when given a new unseen input test data. When after few epochs the validation accuracy shows no change we stop, otherwise we will be just overfitting the data. With a specific setting of given hyper-parameters once good accuracy levels were achieved, we increased the epochs from 16 to 25 ,32 and 100 which would give better results on the model. Finally we set it to 100. Each epoch has 340 steps which takes 890ms to 1.2 secs per step. So overall, each epoch took roughly 290 secs to 310 secs to execute.

Google landmark recognition

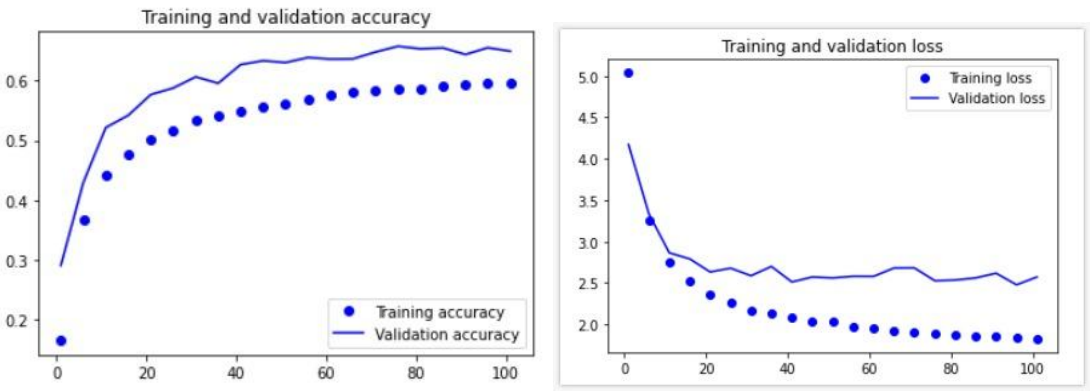
Final Model

Our model trained with VGG16 architecture could give us an **accuracy of 43% on testing dataset**. The final model had the parameters as given below:

Sequential Layers (4)	1 Flattening, 2 Relu Activation and 1 Softmax
Optimizers	Adam
Augmented Images	(rescale = 1. /255, rotation_range = 30, width_shift_range = 0.2, height_shift_range = 0.2, zoom_range = 0.5)
Batch Size	431
Number of Epochs	100
Train Accuracy	67%
Validation Accuracy	60%
Test Accuracy	**43%**

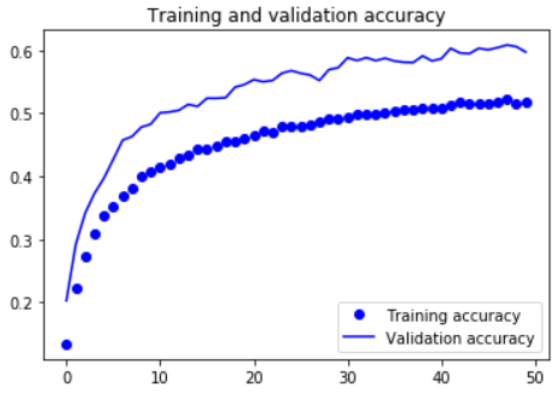
Apart from accuracies shown above we consider another metric namely GAP (Global Average Precision) which is better and more suitable for such problems. The issue with plain test accuracy is, it simply calculates based on number of images correctly classified over the total images. But what it doesn't consider is "with what confidence does the model classify a specific image". This useful detail that comes in handy when an image is classified but with very less confidence. So using GAP metric we set a threshold, and any image classified with confidence less than threshold, we set its confidence to 0. Using this metric our model generated a GAP score of 0.44

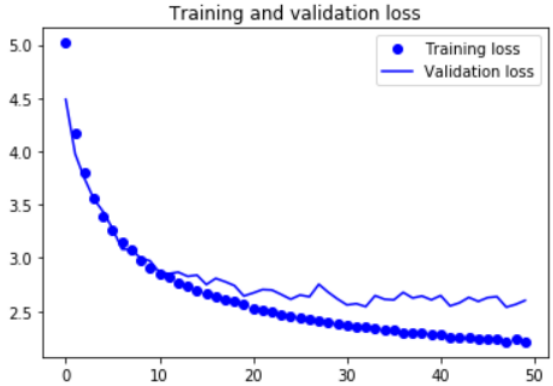
- The graph below shows the accuracy to number of epochs and loss to number of epochs for VGG 16 with dataset with 2000 classes (1000 to 2999)



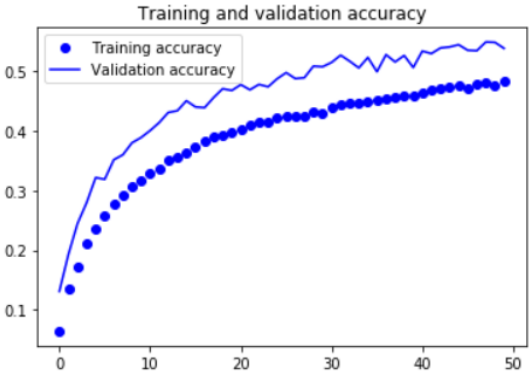
Google landmark recognition

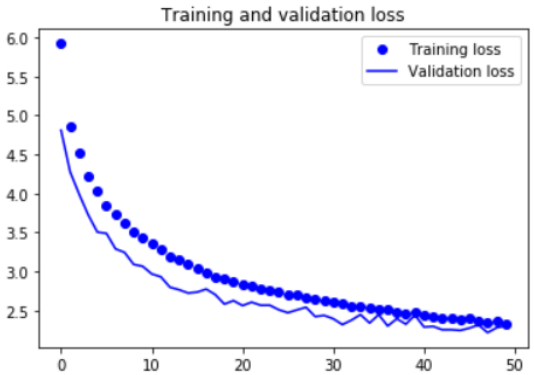
- The graph below shows the accuracy to number of epochs and loss to number of epochs for VGG 16 with dataset with 2000 classes (3000 to 4999)





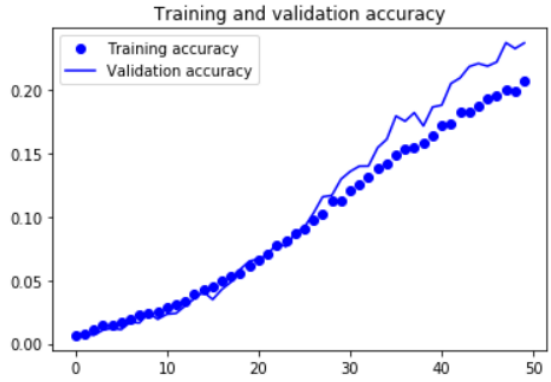
The graph below shows the accuracy to number of epochs and loss to number of epochs for VGG 16 with dataset with 2000 classes (5000 to 6999)

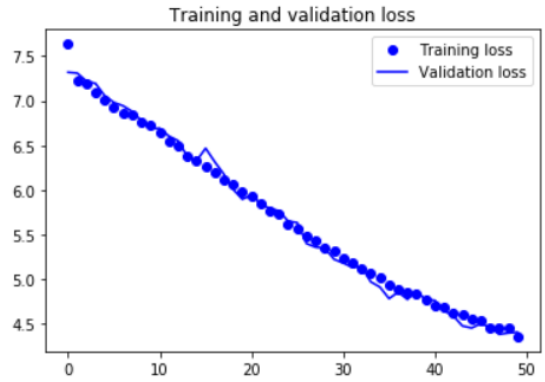




Google landmark recognition

The graph below shows the accuracy to number of epochs and loss to number of epochs for VGG 16 with dataset with 6000 classes.





Approach 2:

Problem with approach 1:

The problem with a convolution neural network is they can be easily fooled when a image which is an outlier given these to a CNN, it predict these images too with some confidence and to some class. This decreases the GAP score as the wrong images are classified and sometimes with higher confidences. To overcome this problem we are proposing to include the local features along with global features to finalize the confidence of the prediction.

DELF:

To get the local features of an image we are using DELF (deep local features) from google which is trained on landmarks images. This is available as a hub in tensorflow library. This module takes two image and makes the key features inside each image and make a geometric verification of these key features between both the images and produces an inlier score. This inlier score is a metric for the match of local features between both the images.

Methodology:

So, with delf and the data we have with us like train images with labeled classes and prediction we made on the test images with their respective confidences we came up with an approach to improve the False Positives of the classifications. For each image the model will predict it into a particular class with a particular confidence score came from softmax layer. If this image is misclassified with more confidence these values damage the overall GAP score. So, we need to give lesser confidence to the misclassified images but how to know the image is misclassified? To get an idea about whether it is misclassified or not we took the images from the train set with the label of test image predicted. Let's say we take K images from the train image of the predicted class of the test image. These k images are individually passed along with the test images to the DELF module. This DELF modules gives an inlier score for each pair of images (pair of train_image of predicted class, test_image). The inlier score of all the images are can be analyzed by averaging all the values, taking the maximum of all the values, taking minimum of all the values. These averages, max or min depends upon the K we choose. In our case due to lack of time and computation we choose K=2,5. So as K is very less the sample, we are testing for this inlier score is very small. So, we used max of all the values we got from the DELF. If these k can be increases to much bigger values average of those inlier scores would give a better result. We can also make this K= number of train images available to that particular class as well but it requires a lot more time and computation.

Now K has chosen but we have to choose the images in test set to which we have to apply our approach so that it gives a better score. We observed that most of the images with confidence score > 0.9 are mostly correctly classified. So, there is no point to cross check these images with local features again as it can only decrease the confidence which is not, we intended. We can check for these images as well if we can increase the K value. But with smaller K It is not a good idea. So, we started with images which are classified with lesser confidence (<0.4) and started finding the inlier scores of these images with its predicted class labels if inlier score of these images have at least 3 score then we are not changing the confidence but out of K trails if we didn't find at least 3 inliers it reflects the image is misclassified. So, we further decreased the confidence to 0. This confidence assignment can be completely flexible. It can be 0.1, 0.15 anything whichever gives a better score for the model. This deals with images with low confidence and misclassified and changes them to very low confidence. The next step is to select images with an average confidence score, there is no rule to define these metrics we divide the confidences score low as <0.4, medium as 0.4<x<0.6 and high as 0.6<x<0.8. we defined different minimum inlier score required to different confidence score. For medium confidence score we use the max of inlier score out of all the images tested should be at least 3(can be changed whichever gives the best) and we decreased the confidence if it is less than that. The confidence with higher values if doesn't have minimum local score matching is decreased too.

These local feature score work better with the better size of the image resolutions. In our case the input image size is 96*96 as out dataset images are. But increasing it would give better results but requires more time to update the complete dataset. The sample of images from the train can also be random, fixed or the best handpicked image to get better results. This approach gives much better results when the test images have more and more outlier or images which are not belonging to the classes which we have.

Results:

The previous GAP score after the first approach is 44% and after using local features it increased by 6% and final GAP is 50%.

Test 1.5:

Approach 3:

As the dataset has is very large and the number of classes to be classified are very large in number the 16 layered VGG16 model is not enough to learn the complete the structures of all the 6000 classes of the images as well as the images of some classes are very less the model should be powerful enough to define the classification with lesser to train. This also involves the depth of the network we want to use. As referred from many studies the depth of the convolution increases the efficiency of the model.

On this point if we increase the depth of the model but this results in the model to be complete learning the identity function instead of the hidden structures in the data and this might also lead to increase in the complexity of the network along with the number of parameters to learn the model. To overcome these problems, we used a residual based CNN model. These residual models consist of 175 layers and comparatively less number of parameters to train than a normal CNN based models. This is possible due to the residual operation it does in between the blocks of the CNN layers. So, there are different Residual Models available to choose with 50 blocks, 101 blocks and 151 blocks named as ResNet50, ResNet101, ResNet151. We used ResNet50 as our model of choice to use.

Google landmark recognition

Model:

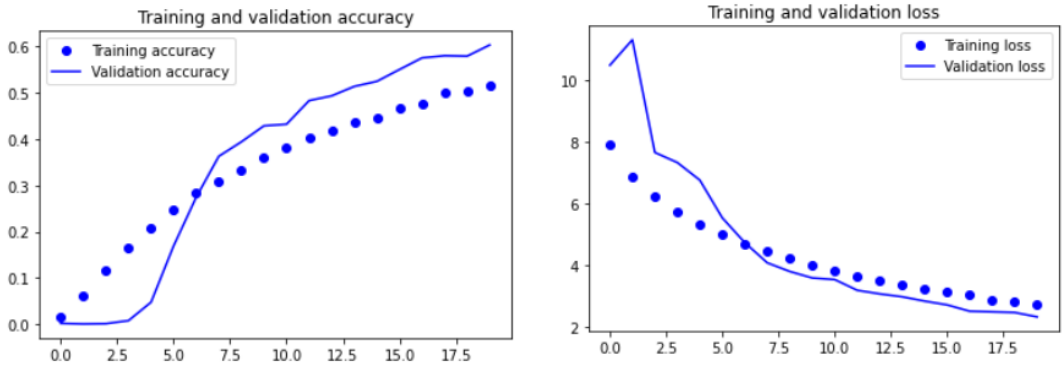
The ResNet 50 architecture consists of the following:

A convolution with a kernel size of (7 * 7) and 64 different kernels all with a stride of size 2 giving us 1 layer. Next, we have a max pooling with also a stride size of 2. In the next convolution there is a 1 * 1, 64 kernels following this 3 * 3, 64 kernel and at last a 1 * 1, 256 kernel, these three layers are repeated in total 3 time so giving us 9 layers in this step. Next, we see kernel of 1 * 1,128 after that a kernel of 3 * 3, 128 and at last a kernel of 1 * 1,512 this step was repeated 4 time so giving us 12 layers in this step. After that there is a kernel of 1 * 1,256 and two more kernels with 3 * 3, 256 and 1 * 1, 1024 and this is repeated 6 times giving us a total of 18 layers. And then again, a 1 * 1, 512 kernel with two more of 3 * 3, 512 and 1 * 1, 2048 and this was repeated 3 times giving us a total of 9 layers. After that we do average pool and end it with a fully connected layer containing 6000 nodes and at the end a softmax function so this gives us 1 layer.

We don't actually count the activation functions and the max/ average pooling layers. So in total it gives us a 1 + 9 + 12 + 18 + 9 + 1 = 50 network layers.

ResNet first introduced the concept of skip connection. In skip connection we perform stacking convolution layers but we now also add the original input to the output of the convolution block. We are using Keras library to load the pre-trained ResNet-50. We are loading the pre-trained ImageNet weights. We set include_top=False to not include the final pooling and fully connected layer in the original model. We added Global Average Pooling Layer, a dropout layer and a dense layer with softmax activation function to the ResNet-50 model.After this, we combined ResNet50 model with Delf. To get the local features of an image we are using DELF (deep local features) from Google which is trained on landmarks images. This is available as a hub in tensorflow library. Detailed information about it has already been explained in the approach 2. We ran our model for 20 epochs and got a good accuracy.

Given below are two graphs for the above model. The graph on the left represents the accuracy and the graph on the right represents loss.



Google landmark recognition

Approach 4:

The ensemble technique is used to combine approaches from different models to improve the overall performance. The models that we have combined are Resnet-50, VGG-16 and VGG-19.

The VGG-16 and Resnet-50 are already explained in above approaches. The number of classes taken is 6000.

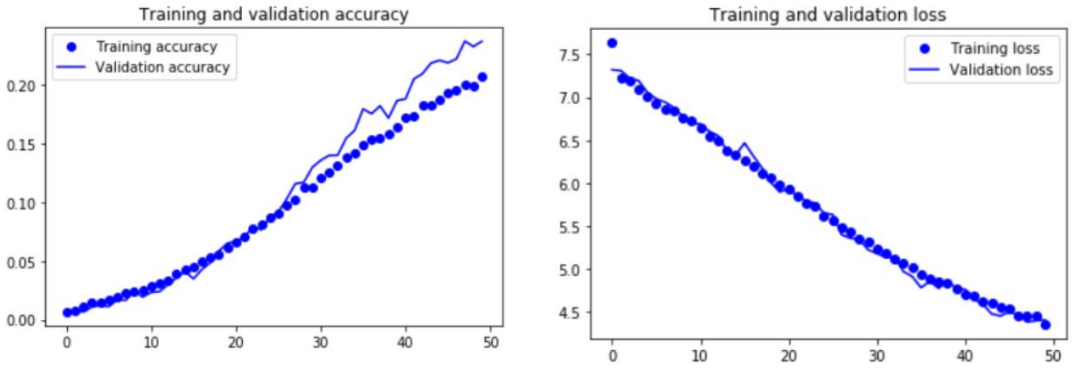
The VGG-19 model is using weights from the pretrained ImageNet model. The top-layer is made false here. First one a sequential model is created and on the top of that 3 layers are added. The first one is flattening layer, second is a dense layer with 1024 neurons and activation functions as relu, and finally another dense layer is added with SoftMax activation.

The technique used for ensemble is averaging. For each model given an image the output contains probabilities of 6000 classes. So, for a given image there are 3 lists of probabilities given by 3 models. After this step after, from the same index in each list, probabilities are taken and their average (column wise average) is calculated. Finally, the list that contains 6000 average probabilities and from that list the highest probability value is taken. This selected probability values class is defined as the class label for that image with the average as the confidence score

Given below are two graphs for VGG-19. The graph is made using the dataset having 6000 classes. The first graph is for training and validation accuracy and the second graph is for Training and validation loss.

In Graph below, it can be seen that as the number of epochs increase the training and validation accuracies also increase.

In Graph below, it can be seen that as the number of epochs increase the training and validation loss also decrease.



Approach 5:

The above ensemble model is of VGG 16, VGG 19 and ResNet 50. In order to further improve the GAP score we try to include the local features along with the global features to finalize the confidence of prediction. This is achieved by the DELF method that was discussed as a part of

Google landmark recognition

Approach 2. So the final model now consists of the model discussed in Approach 4 along with the combination of DELF discussed in Approach 2. All the parameters of the individual models remain consistent through this approach.

This approach gave us a GAP score of 0.4108

RESULTS: The actual result achieved is a GAP score of 0.376. The results achieved by our models are given below in the table.

DATASET	VGG 16	DELF+VGG16	RESNET	VGG16+19+RESNET (Ensemble)	Ensemble + DELF
run1 (2k)	0.4412	NA	NA	NA	NA
run2(2k)	0.5703	NA	NA	NA	NA
run3(2k)	0.4904	NA	NA	NA	NA
run4(avg)	0.50063	NA	NA	NA	NA
run5(6k)	0.3816	0.40748	0.38175	0.41566	0.4108

CONCLUSIONS and FUTURE DIRECTIONS:

Conclusion:

We researched on the topic Google Landmark Recognition and performed classification on around half a million images to classify it to around 6000 classes.
We started with VGG 16 architecture model to classify 2000 classes of 3 different samples which gave us GAP score of 0.50063 on average.

As our model scored much higher than the top scorers shown on Kaggle, we concluded that as our dataset size is small it gave us a very high accuracy compared to other results. Thus, to get closer to the comparison with the original dataset we moved onto increasing our dataset size to 6000 classes which actually decreased our accuracy and got us in a position to compare better with other results. Thus we moved on to classify 6000 classes by starting with VGG 16 architecture again which gave us GAP score of 0.3816.

As wrong images were classified and sometimes with higher confidences which decreased the GAP score, we proposed our own method to include the local features along with global features along with the VGG architecture to finalize the confidence of the prediction which increased the accuracy and gave us a GAP score of 0.40748. After that we then used a residual based CNN model ResNet 50 which gave us a GAP score of 0.38175.

Then we performed ensemble techniques to increase the overall performance by combining the models Resnet-50, VGG-16 and VGG-19 which gave us a GAP score of 0.41566. The test dataset had a lot of images that were noisy, had different orientations and ranges in which they were shot or many images were not even landmarks. So thus, an important insight about these kinds of projects is the ranking strategies of the confidence scores. Techniques like using DELF are very helpful to improve the confidence scores of both True Positives and False Positives in these scenarios.

For future directions and improvements, the following suggestions can be implemented.

1. Assuming there are enough resources to handle the huge dataset, it would be more appropriate to consider the whole dataset while using the above models as this would give the most accurate results possible.
2. More complex models can be selected like Squeeze and Excitation nets, which not only

consider information related to pixels in images but also include the channel wise information(R, G, B).
3. Along with DELF and local features, we can also embed existing models with clustering techniques and object detection techniques, to further improve the confidence scores and bring up the overall Global Average Precision scores.
4. Instead of just working with 96x96 images, it's advisable to use higher resolution images, and also consider various orientations, filters, etc... Of the same image.

REFERENCES:

1. Karen simonyan and Andrew Zisserman. "Very deep convolutional networks for large scale image recognition." ICLR, 2015.

2. Ramakrishnan, Nitesh, Kamalakannan, Anandhanarayanan, Chelliah, Balika Rajamanickam and Govindaraj. (2019). "Computer Vision Framework for Visual sharp Object Detection using Deep Learning Model, International Journal of Engineering and Advanced Technology (IJEAT)"

3. H. Noh, A. Araujo, J. Sim, T. Weyand, and B. Han. "Largescale image retrieval with attentive deep local features." ICCV, 2017.
4. He, Kaiming et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016).
5. H. Yanagisawa, T. Yamashita and H. Watanabe, "A study on object detection method from manga images using CNN," 2018 International Workshop on Advanced Image Technology (IWAIT), Chiang Mai, 2018.
6. B. Cao, A. Araujo, and J. Sim. Unifying Deep Local and Global Features for Image Search. arXiv:2001.05027, 2020.
7. M. Gogoi and S. A. Begum, "Image Classification Using Deep Autoencoder," 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), Coimbatore, 2017.
8. Y. Li and H. Yeh, "A semi-supervised learning model based on convolutional autoencoder and convolutional neural network for image classification," 2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), Taipei, Taiwan, 2019.
9. R. Girshick, J. Donahue, T. Darrell and J. Malik, "Region-Based Convolutional Networks for Accurate Object Detection and Segmentation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 38, no. 1, pp. 142-158, 1 Jan. 2016.
10. J. Hu, L. Shen and G. Sun, "Squeeze-and-Excitation Networks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, 2018, pp. 7132-7141.

APPENDICES:

A) Complete Information about the data set.

Source of the dataset is: https://www.kaggle.com/google/google-landmarks-dataset

The dataset available is in the form of a .csv file which includes SNO, ID of the image, URL to download the image class label of the image (landmark_id). Total number of images in original dataset consists of 1 Million images.

From this dataset around 1,30,0000 Images are taken, while maintaining the proportion and properties of data to be same as the original dataset. The CSV contains image URLs, and for each

image URL there is landmark ID and ID field. ID is unique identifier of an image. The landmark IDs can also be called as class labels.

We have taken 3 samples of the original dataset. Sample 1 contains class labels from 1000 to 2999. Sample 2 contains class labels from 3000 to 4999. Sample 3 contains class labels from 5000 to 6999.

For each sample the images are downloaded from the URLs available in the .csv file. Further, three folders are created i.e. for train_images_model, test_from_train_images and validation_images_model. In data preprocessing the directory structure is created in such a way that each of these folders contains subfolders with class name as folder name. Each subfolder consists of the images corresponding to a particular that class. Cleaning involves removing all the corrupted image files. These are identified when an error was thrown while running the code. These are sometimes not possible to open.

For each of the 2000 classes, the train and validation split are 80-20 i.e. 80 % will be train and 20 % will be in the validation. For every class 1% of images are taken as test along with other images which are not in these 2000 classes are also added to make the problem reflecting the original dataset test images.

After cleaning the final samples have the data as follows:
Sample 1 has train data: 108181 images, Validation data: 19826 images
Sample 2 has train data: 99724 images, Validation data: 21400 images
Sample 3 has train data: ~70000 images, Validation data: 19380 images

As to make the sample much reliable as the final model we combined the entire three samples we trained. This resembles the 50% of original Google Landmark Recognition Dataset. This sample consists of 523603 of overall train images before data Pre-processing. The train and validation split is same as we did for the previous samples that is 80% train images and 20% validation images. The original dataset consists of nearly 1 Million images which are classified into 14,000 classes. Our final sample consists of nearly Half Million images which are classified into 6,000 classes. These gives the complete challenges of the real Google Landmark recognition challenge. The number of classes are large with some classes are having even less than 10 images to train.

B) Source/Pseudo Code.

In [2]:
```
cd "E:\ME-COMPUTERS\THIRD SEM\ADM\project"
```

```
E:\ME-COMPUTERS\THIRD SEM\ADM\project
```

In [3]:
```
import numpy as np
import pandas as pd
import sys, requests, shutil, os

data=pd.read_csv("train.csv")
data_test=pd.read_csv("test.csv")
```

In [4]:
```
landmark_list = [str(x) for x in list(range(1000,7000))]
data_sample = data[data['landmark_id'].isin(landmark_list)]
counts=data_sample.landmark_id.value_counts()
sum(data_sample.landmark_id.value_counts())
```

Out[4]:
```
523603
```

In [5]:
```
import re
TARGET_SIZE = 96 #imports images of resolution 96x96

'''change URLs to resize images to target size'''
def overwrite_urls(df):
    def reso_overwrite(url_tail, reso=TARGET_SIZE):
        pattern = 's[0-9]+'
        search_result = re.match(pattern, url_tail)
        if search_result is None:
            return url_tail
        else:
            return 's{}'.format(reso)

    def join_url(parsed_url, s_reso):
        parsed_url[-2] = s_reso
        return '/'.join(parsed_url)

    df = df[df.url.apply(lambda x: len(x.split('/'))>1)]
    parsed_url = df.url.apply(lambda x: x.split('/'))
    train_url_tail = parsed_url.apply(lambda x: x[-2])
    resos = train_url_tail.apply(lambda x: reso_overwrite(x, reso=TARGE
T_SIZE))

    overwritten_df = pd.concat([parsed_url, resos], axis=1)
    overwritten_df.columns = ['url', 's_reso']
    df['url'] = overwritten_df.apply(lambda x: join_url(x['url'], x['s_
reso']), axis=1)
    return df

data_sample_resize = overwrite_urls(data_sample)
print ('1. URLs overwritten')

'''Split to test and train'''
data_test = pd.DataFrame(columns = ['id','url','landmark_id'])
data_training_all = pd.DataFrame(columns = ['id','url','landmark_id'])
percent_test = 0.05

import random
random.seed(42)
for landmark_id in set(data_sample_resize['landmark_id']):
    n=1
    t = data_sample_resize[(data_sample_resize.landmark_id == landmark_
id)] #get all images for a landmark id
    i = 0
    r =[]
    #print(len(t.id))
    while i < len(t.id) and i<6000:
        it = i
        r.append(t.id.iloc[it])  #create a list of all these images
        i += 1

    test = random.sample(r,int(percent_test*len(r))) #randomly pick a s
ample of 1% images from list 'r'
    training = list(set(r) - set(test))  #get the remaining images
    data_t = data_sample_resize[data_sample_resize.id.isin(test)] #hold
```

```
    out dataset
        data_tr = data_sample_resize[data_sample_resize.id.isin(training)]
    #training dataset
        data_test = data_test.append(data_t)
        data_training_all = data_training_all.append(data_tr)
        n+=1

    print ('2. train and test set created')


    '''Split into train and validation set'''
    data_valid = pd.DataFrame(columns = ['id','url','landmark_id'])
    data_train = pd.DataFrame(columns = ['id','url','landmark_id'])
    percent_validation = 0.2 #takes 20% from each class as holdout data
    import random
    random.seed(42)
    for landmark_id in set(data_training_all['landmark_id']):
        n=1
        t = data_training_all[(data_training_all.landmark_id == landmark_i
    d)]
        i = 0
        r =[]
        while i < len(t.id):
            it = i
            r.append(t.id.iloc[it])
            i += 1

        valid = random.sample(r,int(percent_validation*len(r)))
        train = list(set(r) - set(valid))
        data_v = data_training_all[data_training_all.id.isin(valid)]
        data_t = data_training_all[data_training_all.id.isin(train)]
        data_valid = data_valid.append(data_v)
        data_train = data_train.append(data_t)
        n+=1

    print ('3. train and validation set created')
```

```
1. URLs overwritten
2. train and test set created
3. train and validation set created
```

In [ ]:
```
train_sample=data_train.to_csv()
test_sample=data_test.to_csv()
valdi_sample=data_test.to_csv()
```

Create directories 'train_images_model', 'validation_images_model', 'test_images_from_train' before running the code ahead.

In [ ]:
```
def fetch_image(path,folder):
    try:
        url=path
        response=requests.get(url, stream=True)
        with open(folder + '/image.jpg', 'wb') as out_file:
            shutil.copyfileobj(response.raw, out_file)
        del response
    except:
        print("error")
'''TRAIN SET - fetch images for the resized URLs and save in the alread
y created directory train_images_model'''
i=0
for link in data_train['url']:
    if i%10000==0:
        print(i)
    #looping over links to get images
    if os.path.exists('train_images_model/'+str(data_train['id'].iloc
[i])+'.jpg'):
        i+=1
        print(i)
        continue
    fetch_image(link,'train_images_model')
    try:
        os.rename('train_images_model/image.jpg','train_images_model/'+
str(data_train['id'].iloc[i])+ '.jpg')
    except:
        print("not found")
    i+=1
#     if(i==50):   #uncomment to test in your machine
#         break
print('4. train images fetched')
i=0
for link in data_valid['url']:              #looping over links to get
images
    if os.path.exists('validation_images_model/'+str(data_valid['id'].i
loc[i])+'.jpg'):
        i+=1
        continue
    fetch_image(link,'validation_images_model')
    try:
        os.rename('validation_images_model/image.jpg','validation_image
s_model/'+ str(data_valid['id'].iloc[i])+ '.jpg')
    except:
        print("not found")
    i+=1
#     if(i==50):   #uncomment to test in your machine
#         break
print('5. Validation images fetched')

i=0
for link in data_test['url']:              #looping over links to get i
mages
    if os.path.exists('test_images_from_train/'+str(data_test['id'].ilo
c[i])+'.jpg'):
        i+=1
```

```
        continue
    fetch_image(link,'test_images_from_train')
    try:
        os.rename('test_images_from_train/image.jpg','test_images_from_
train/'+ str(data_test['id'].iloc[i])+ '.jpg')
    except:
        print("not found")
    i+=1
#     if(i==50):   #uncomment to test in your machine
#         break
print('6. Test images fetched')
```

Data Preprocessing

Creating folders for each landmark ID (Class label)

In [ ]:
```
##create folders for landmark IDs in Training folder
import pandas as pd
import os
import shutil
from shutil import copyfile
import urllib

train_data = data_train

temp = pd.DataFrame(data_train.landmark_id.value_counts())
temp.reset_index(inplace=True)
temp.columns = ['landmark_id','count']

def createfolders(dataset,folder):
    i = 0
    while i < len(dataset):
        landmark = str(dataset.landmark_id.iloc[i])
        path = folder + '/'+ landmark
        if not os.path.exists(path):
            os.makedirs(path)
        i+=1
createfolders(temp,'train_images_model')
available = [int((x[0].split('/'))[-1]) for x in os.walk(r'train_images
_model/') if len((x[0].split('/'))[-1]) > 0]
new = [str(x) for x in range(1000,6999) if x not in available]
for i in new:
    path = 'train_images_model/' + i
    if not os.path.exists(path):
        os.makedirs(path)
print ('Train folders created')

rootdirpics = r'train_images_model/'
rootdirfolders = r'train_images_model/'

def transformdata(data,path1, path2):

    n = 1
    for landmark_id in set(data['landmark_id']):
        t = data[(data.landmark_id == landmark_id)]
        i = 1
        r =[]
        while i <= len(t.id):
            it = i - 1
            r.append(t.id.iloc[it])
            i += 1
        for files in os.listdir(rootdirpics):    # loop through startfo
lders
            inpath = path1 + files
            folder = str(landmark_id)
            outpath = path2 + folder
            if ((files.split('.')[0] in r) & (os.path.getsize(inpath) >
1000)):
#                 print('move')
                shutil.move(inpath, outpath)
            elif ((files.split('.')[0] in r) & (os.path.getsize(inpath)
```

```
<= 1000)):
                os.remove(inpath)
        n+=1

transformdata(train_data,rootdirpics, rootdirfolders)
print ('Train images moved')
```

In [ ]:
```
##create folders for landmark IDs in Validation folder

temp = pd.DataFrame(data_valid.landmark_id.value_counts())
temp.reset_index(inplace=True)
temp.columns = ['landmark_id','count']
createfolders(temp,'validation_images_model')
print ('Validation folders created')

#make folders for landmark ID which had no images in validation sets - required for codes running next
available = [int((x[0].split('/'))[-1]) for x in os.walk(r'validation_images_model/') if len((x[0].split('/'))[-1]) > 0]
new = [str(x) for x in range(1000,6999) if x not in available]
for i in new:
    path = 'validation_images_model/' + i
    if not os.path.exists(path):
        os.makedirs(path)

rootdirpics = r'validation_images_model/'
rootdirfolders = r'validation_images_model/'
transformdata(data_valid,rootdirpics, rootdirfolders)
print ('Validation images moved')
```

In [ ]:
```
#remove corrupted images from the dataset
def cleaning(dir):
    i = 1000
    count = 0
    while i <= 6999:
        f = str(i)
        print (f)
        for root, dirs, files in os.walk(dir +'/'+ f):  # loop through startfolders
            for pic in files:
                p=dir+'/'+f+'/'+pic
                try:
                    im=Image.open(p)
                except IOError:
                    count+=1
                    print(p)
                    os.remove(p)
            i += 1
    print(count)
```

In [ ]:
```
cleaning("/train_images_model")
cleaning("/validation_images_model")
```

In [ ]:
```
# Make sure we are pointing to the directory that has all the files nec
essary
cd 'drive/My Drive'
```

```
/content/drive/My Drive
```

In [ ]:
```
ls
```

```
bottleneck_fc_model.h5              test_images_from_train/
bottleneck_features_train.npy       test_sample.csv
bottleneck_features_validation.npy  train_images_model/
class_indices.npy                   train_sample.csv
moved_images/                       validation_images_model/
moved_images_1/                     valid_sample.csv
```

In [ ]:
```
import numpy as np
import pandas as pd
import sys, requests, shutil, os

data_train=pd.read_csv("train_sample.csv")
data_valid=pd.read_csv("valid_sample.csv")
data_test=pd.read_csv("test_sample.csv")
```

In [ ]:
```
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator, im
g_to_array, load_img
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
#from tensorflow.keras.utils.np_utils import to_categorical
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model
import csv
import os
#import cv2
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import math
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import tensorflow as tf
import tensorflow.keras
```

In [ ]:

```
'''
        There are few images that are not possible to open because they are corrupted or
    they do exhibit some properties which are not applicable to standard algorithms,
    these may result in algorithm going to infinite loop trying to process it.All those images
    are removed using the function below.
'''
from PIL import Image
import os

def filter_images(dir):
    i = 1000
    count = 0
    while i <= 1000:
        f = str(i)
        for root, dirs, files in os.walk(dir +'/'+ f):
            for pic in files:
                p=dir+'/'+f+'/'+pic
                try:
                    im=Image.open(p)    # if the image opening throws error
                except IOError:
                    count+=1
                    os.remove(p)        # catch it and delete it
            i += 1

filter_images('train_images_model')
filter_images('validation_images_model')
filter_images('test_images_from_train')
```

In [ ]:
```
# This cell counts the total number of train images and validation imag
es in the entire dataset
# It stores the counts of train data 108181 training samples belonging
to 1000 classes and
# 19826 test samples in the respective variables nb_train_samples and n
b_validation_samples



train_data_dir = 'train_images_model'                    # points to dir
ectory having train images
validation_data_dir = 'validation_images_model'          # points to dir
ectory having validation images


def count(dir):
    i = 1000
    count = []
    while i <= 2999:
        f = str(i)
        print (f)
        for root, dirs, files in os.walk(dir +'/'+ f):  # os. walk() is
used to iterate through all folders of a directory
            for pic in files:
                count.append(f)
            i += 1
    print (len(count))
    return ([len(count),count])

nb_train_samples = count(train_data_dir)
nb_validation_samples = count(validation_data_dir)
```

In [ ]:
```
#Importing few other files needed
import pandas as pd
import os
import shutil
from shutil import copyfile
import urllib
```

In [ ]:
```
# here we build the VGG 16 model, which is pretrained using the imagene
t dataset,
# This is achieved by setting weights = 'imagenet' as a parameter while
building the model

img_width, img_height = 96, 96 #dimensions used in the model
top_model_weights_path = 'bottleneck_fc_model.h5' # this is used to sav
e weights in later stages, avoid recomputations
epochs = 5
batch_size = 431 #GCD
def save_bottleneck_features():
    datagen = ImageDataGenerator(rescale=1. / 255,
                                 rotation_range=30,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range = 0.5,
                                 brightness_range = [0.5,1.5])

    # VGG 16 FramerWork
    model = applications.VGG16(include_top=False, weights='imagenet', i
nput_shape=(96,96,3))
    print ('start1')
    generator = datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)

    print ('start2')
    bottleneck_features_train = model.predict_generator(generator, nb_t
rain_samples[0] // batch_size)
    print ('bottleneck_features_trained')


    with open('bottleneck_features_train.npy', 'wb') as features_train_
file:
        np.save(features_train_file, bottleneck_features_train)
    print ('Train done')


    datagen = ImageDataGenerator(rescale=1. / 255)
    generator = datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)
    print ('validation predict start')
    bottleneck_features_validation = model.predict_generator(generator,
nb_validation_samples[0] // batch_size)

    with open('bottleneck_features_validation.npy', 'wb') as features_v
alidation_file:
        np.save(features_validation_file, bottleneck_features_validatio
```

```
n)
    print ('validation done')
save_bottleneck_features()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras
-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 0s 0us/step
start1
Found 108181 images belonging to 2000 classes.
start2
WARNING:tensorflow:From <ipython-input-16-f4d1cbcba41b>:29: Model.pre
dict_generator (from tensorflow.python.keras.engine.training) is depr
ecated and will be removed in a future version.
Instructions for updating:
Please use Model.predict, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "

bottleneck_features_trained
Train done
Found 19826 images belonging to 2000 classes.
validation predict start
validation done
```

In [ ]:
```
# counts of train and validation labels
train_labels = np.array(nb_train_samples[1])
train_labels = [str(int(train_label) - 1000) for train_label in train_l
abels]
print(len(train_labels))
train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
print(len(train_data))
validation_data = np.load(open('bottleneck_features_validation.npy', 'r
b'))
print(len(validation_data))
validation_labels = np.array(nb_validation_samples[1])
print(len(validation_labels))
```

```
108181
108181
19826
19826
```

In [ ]:
```
epochs = 5
batch_size = 431
import numpy as np

def train_top_model():
    train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
    train_labels = np.array(nb_train_samples[1])
    train_labels = [str(int(train_label) - 1000) for train_label in tra
in_labels]
    # we do label-1000, so that class lable starts from 0, as we have l
abels starting from 1000

    validation_data = np.load(open('bottleneck_features_validation.npy
', 'rb'))
    validation_labels = np.array(nb_validation_samples[1])
    validation_labels = [str(int(validation_label) - 1000) for validati
on_label in validation_labels]

    model = Sequential()
    model.add(Flatten(input_shape=train_data.shape[1:]))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(256, activation='relu'))
    n_class = 2000 # max. classes given to the model
    model.add(Dense(n_class, activation='softmax'))
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy', metrics=['accuracy
'])
    train_labels = to_categorical(train_labels, n_class)
    validation_labels = to_categorical(validation_labels, n_class)

    print ('model fit starting')

    model.fit(train_data, train_labels,epochs=epochs,batch_size=batch_s
ize,validation_data=(validation_data, validation_labels))
    model.save_weights(top_model_weights_path)

train_top_model()
```

```
model fit starting
Epoch 1/5
251/251 [==============================] - 3s 13ms/step - loss: 4.733
7 - accuracy: 0.2031 - val_loss: 4.0237 - val_accuracy: 0.2669
Epoch 2/5
251/251 [==============================] - 3s 10ms/step - loss: 3.780
3 - accuracy: 0.3092 - val_loss: 3.5328 - val_accuracy: 0.3804
Epoch 3/5
251/251 [==============================] - 3s 10ms/step - loss: 3.337
8 - accuracy: 0.3613 - val_loss: 3.2535 - val_accuracy: 0.4197
Epoch 4/5
251/251 [==============================] - 3s 10ms/step - loss: 3.042
9 - accuracy: 0.4007 - val_loss: 3.1162 - val_accuracy: 0.4549
Epoch 5/5
251/251 [==============================] - 2s 10ms/step - loss: 2.810
3 - accuracy: 0.4318 - val_loss: 3.0494 - val_accuracy: 0.4714
```

In [ ]:
```
# This is where we fine tune the pretrained model according to our data set

img_width, img_height = 96, 96
top_model_weights_path = 'bottleneck_fc_model.h5'
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 100
def trainCNN():

    # build the VGG16 network

    base_model = applications.VGG16(weights='imagenet',include_top= False,input_shape=(96,96,3))

    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    n_class = 2000
    top_model.add(Dense(n_class, activation='softmax'))
    top_model.load_weights(top_model_weights_path) #Load the weights initialized in previous steps


    model = Model(base_model.input, top_model(base_model.output))

    # set the first 16 layers to non-trainable (weights will not be updated)
    # 1 conv layer and three dense layers will be trained
    for layer in model.layers[:16]:
        layer.trainable = False

    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(lr=0.001, beta_1=0.9,beta_2=0.999,epsilon=1e-8, decay=0.0),
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    np.save('class_indices.npy', train_generator.class_indices)
```

```
    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    print ('Model fit begins...')
    model.fit_generator(
        train_generator,
        steps_per_epoch=340,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=100,
        callbacks=[ModelCheckpoint(filepath=top_model_weights_path, sav
e_best_only=True, save_weights_only=True)]
        )

    model.save_weights(top_model_weights_path)
    # final weights are saved in bottleneck_fc_model.h5 file

trainCNN()
```

In [ ]:
```
base_model = applications.VGG16(weights='imagenet',include_top= False,i
nput_shape=(96,96,3))
print(base_model.input)
```

```
Tensor("input_2:0", shape=(None, 96, 96, 3), dtype=float32)
```

In [ ]:
```
from tensorflow.python.platform import app
import argparse
import os
import sys
import time
from time import *
import io
import tensorflow as tf
```

In [ ]:
```
top_model_weights_path = 'bottleneck_fc_model.h5'        # final weights
of the model
train_data_dir = 'train_images_model'
testfile = 'test_images_from_train'
subfile = 'result.csv'                                   # predictions o
f the model
```

In [12]:
```
from keras import backend as K
import cv2

# here we send the test images to the model
# the model predicts the classes with some confidence score
# these tuples are stored in a file result.csv that are used in later p
arts of the code

def predict(image_path):
    print ('starting...')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)

    train_datagen = ImageDataGenerator(rescale=1. / 255)

    generator = train_datagen.flow_from_directory(train_data_dir, batch
_size=batch_size)
    label_map = (generator.class_indices)

    n_class = 2000

    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks'])

        file_counter = 0
        for root, dirs, files in os.walk(image_path):
            for pic in files:
                t1 = clock()

                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark
                base_model = applications.VGG16(weights='imagenet', inc
lude_top=False, input_shape=(96, 96, 3))

                top_model = Sequential()
                top_model.add(Flatten(input_shape=base_model.output_sha
pe[1:]))
                top_model.add(Dense(256, activation='relu'))
                top_model.add(Dense(256, activation='relu'))
                top_model.add(Dense(n_class, activation='softmax'))

                model = Model(base_model.input, top_model(base_model.ou
```

```
tput))
                model.load_weights(top_model_weights_path)

                prediction = model.predict(image)

                class_predicted = prediction.argmax(axis=1)
                inID = class_predicted[0]
                inv_map = {v: k for k, v in label_map.items()}
                label = inv_map[inID]

                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                out = str(label) + ' '+ scor
                #print(score)

                newFileWriter.writerow([os.path.splitext(pic)[0], out])
                K.clear_session()

predict(testfile)
```

In [ ]:
```
result = pd.read_csv('result.csv')      # predicted values
```

In [ ]:
```
test = pd.read_csv('test_sample.csv')   # ground truth
```

In [11]:
```
# final accuracy of the model as tested on test dataset
import pandas as pd
def test_accuracy(pred,truth):
    result=pd.read_csv(pred)
    #print(result.head(20))
    test=pd.read_csv(truth)
    count=0
    for i in result["id"]:
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #b=list(b[0].split(" "))
        b[0]=int(b[0])
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              count+=1
    print(count/len(result))
    return count/len(result)
test_accuracy("result.csv","test_sample.csv")
```

```
0.43471074380165287
```

Out[11]:
```
0.43471074380165287
```

In [4]:
```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("test_sample.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=1100
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
```

In [5]:
```
GAP_metric('result.csv')
```

```
1100 491.92584301563505 1815 789
0.4472053118323955
```

Out[5]:
```
0.4472053118323955
```

In [ ]:
```
#ls
```

In [ ]:
```
cd 'RUN2'
```

```
/content/drive/My Drive/RUN2
```

In [ ]:
```
#!unzip admset213.zip
```

In [ ]:
```
cd admrun2
```

```
/content/drive/My Drive/RUN2/admrun2
```

In [ ]:
```
import numpy as np
import pandas as pd
import sys, requests, shutil, os

data_train=pd.read_csv("train_sample.csv")
data_valid=pd.read_csv("validation_sample.csv")
data_test=pd.read_csv("test_sample.csv")
```

In [ ]:
```
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator, im
g_to_array, load_img
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
#from tensorflow.keras.utils.np_utils import to_categorical
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model
import csv
import os
#import cv2
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import math
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import tensorflow as tf
import tensorflow.keras
```

In [ ]:

```
'''
        There are few images that are not possible to open because they are corrupted or
    they do exhibit some properties which are not applicable to standard algorithms,
    these may result in algorithm going to infinite loop trying to process it.All those images
    are removed using the function below.
'''
from PIL import Image
import os

def filter_images(dir):
    i = 1000
    count = 0
    while i <= 1000:
        f = str(i)
        for root, dirs, files in os.walk(dir +'/'+ f):
            for pic in files:
                p=dir+'/'+f+'/'+pic
                try:
                    im=Image.open(p)    # if the image opening throws error
                except IOError:
                    count+=1
                    os.remove(p)        # catch it and delete it
            i += 1

filter_images('train_images_model')
filter_images('validation_images_model')
filter_images('test_images_from_train')
```

In [ ]:
```
# This cell counts the total number of train images and validation images in the entire dataset
# It stores the counts of train data 108181 training samples belonging to 1000 classes and
# 19826 test samples in the respective variables nb_train_samples and nb_validation_samples



train_data_dir = 'train_images_model'                    # points to directory having train images
validation_data_dir = 'validation_images_model'          # points to directory having validation images


def count(dir):
    i = 3000
    count = []
    while i <= 4999:
        f = str(i)
        #print (f)
        for root, dirs, files in os.walk(dir +'/'+ f):  # os. walk() is used to iterate through all folders of a directory
            for pic in files:
                count.append(f)
            i += 1
    print (len(count))
    return ([len(count),count])

nb_train_samples = count(train_data_dir)
nb_validation_samples = count(validation_data_dir)
```

```
99724
21400
```

In [ ]:
```
#Importing few other files needed
import pandas as pd
import os
import shutil
from shutil import copyfile
import urllib
```

In [ ]:
```
# here we build the VGG 16 model, which is pretrained using the imagene
t dataset,
# This is achieved by setting weights = 'imagenet' as a parameter while
building the model

img_width, img_height = 96, 96 #dimensions used in the model
top_model_weights_path = 'bottleneck_fc_model.h5' # this is used to sav
e weights in later stages, avoid recomputations
epochs = 5
batch_size = 428 #GCD
def save_bottleneck_features():
    datagen = ImageDataGenerator(rescale=1. / 255,
                                 rotation_range=30,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range = 0.5,
                                 brightness_range = [0.5,1.5])

    # VGG 16 FramerWork
    model = applications.VGG16(include_top=False, weights='imagenet', i
nput_shape=(96,96,3))
    print ('start1')
    generator = datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)

    print ('start2')
    bottleneck_features_train = model.predict_generator(generator, nb_t
rain_samples[0] // batch_size)
    print ('bottleneck_features_trained')


    with open('bottleneck_features_train.npy', 'wb') as features_train_
file:
        np.save(features_train_file, bottleneck_features_train)
    print ('Train done')


    datagen = ImageDataGenerator(rescale=1. / 255)
    generator = datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)
    print ('validation predict start')
    bottleneck_features_validation = model.predict_generator(generator,
nb_validation_samples[0] // batch_size)

    with open('bottleneck_features_validation.npy', 'wb') as features_v
alidation_file:
        np.save(features_validation_file, bottleneck_features_validatio
```

```
n)
    print ('validation done')
save_bottleneck_features()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras
-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 2s 0us/step
start1
Found 99724 images belonging to 2000 classes.
start2
WARNING:tensorflow:From <ipython-input-20-cd1ff807f0ca>:27: Model.pre
dict_generator (from tensorflow.python.keras.engine.training) is depr
ecated and will be removed in a future version.
Instructions for updating:
Please use Model.predict, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "

bottleneck_features_trained
Train done
Found 21400 images belonging to 2000 classes.
validation predict start
validation done
```

In [ ]:
```
# counts of train and validation labels
train_labels = np.array(nb_train_samples[1])
train_labels = [str(int(train_label) - 3000) for train_label in train_l
abels]
print(len(train_labels))
train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
print(len(train_data))
validation_data = np.load(open('bottleneck_features_validation.npy', 'r
b'))
print(len(validation_data))
validation_labels = np.array(nb_validation_samples[1])
print(len(validation_labels))
```

```
99724
99724
21400
21400
```

In [ ]:
```
epochs = 5
batch_size = 428
import numpy as np

def train_top_model():
    train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
    train_labels = np.array(nb_train_samples[1])
    train_labels = [str(int(train_label) - 3000) for train_label in tra
in_labels]
    # we do label-1000, so that class lable starts from 0, as we have l
abels starting from 1000

    validation_data = np.load(open('bottleneck_features_validation.npy
', 'rb'))
    validation_labels = np.array(nb_validation_samples[1])
    validation_labels = [str(int(validation_label) - 3000) for validati
on_label in validation_labels]

    model = Sequential()
    model.add(Flatten(input_shape=train_data.shape[1:]))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(256, activation='relu'))
    n_class = 2000 # max. classes given to the model
    model.add(Dense(n_class, activation='softmax'))
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy', metrics=['accuracy
'])
    train_labels = to_categorical(train_labels, n_class)
    validation_labels = to_categorical(validation_labels, n_class)

    print ('model fit starting')

    model.fit(train_data, train_labels,epochs=epochs,batch_size=batch_s
ize,validation_data=(validation_data, validation_labels))
    model.save_weights(top_model_weights_path)

train_top_model()
```

```
model fit starting
Epoch 1/5
233/233 [==============================] - 4s 17ms/step - loss: 5.155
3 - accuracy: 0.1441 - val_loss: 4.3747 - val_accuracy: 0.2388
Epoch 2/5
233/233 [==============================] - 4s 15ms/step - loss: 4.200
1 - accuracy: 0.2429 - val_loss: 3.8240 - val_accuracy: 0.3244
Epoch 3/5
233/233 [==============================] - 4s 15ms/step - loss: 3.731
1 - accuracy: 0.2988 - val_loss: 3.6022 - val_accuracy: 0.3578
Epoch 4/5
233/233 [==============================] - 4s 15ms/step - loss: 3.417
5 - accuracy: 0.3392 - val_loss: 3.2435 - val_accuracy: 0.4210
Epoch 5/5
233/233 [==============================] - 4s 15ms/step - loss: 3.173
0 - accuracy: 0.3724 - val_loss: 3.3218 - val_accuracy: 0.4196
```

In [ ]:
```
base_model = applications.VGG16(weights='imagenet',include_top= False,i
nput_shape=(96,96,3))
print(base_model.input)
```

```
Tensor("input_2:0", shape=(None, 96, 96, 3), dtype=float32)
```

In [ ]:
```
# This is where we fine tune the pretrained model according to our data
set

img_width, img_height = 96, 96
top_model_weights_path = 'bottleneck_fc_model.h5'
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 50
def trainCNN():

    # build the VGG16 network

    base_model = applications.VGG16(weights='imagenet',include_top= Fal
se,input_shape=(96,96,3))

    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    n_class = 2000
    top_model.add(Dense(n_class, activation='softmax'))
    top_model.load_weights(top_model_weights_path) #Load the weights in
itialized in previous steps


    model = Model(base_model.input, top_model(base_model.output))

    # set the first 16 layers to non-trainable (weights will not be upd
ated)
    # 1 conv layer and three dense layers will be trained
    for layer in model.layers[:16]:
        layer.trainable = False

    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(lr=0.001, beta_1=0.9,beta_2
=0.999,epsilon=1e-8, decay=0.0),
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    np.save('class_indices.npy', train_generator.class_indices)
```

```
    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    print ('Model fit begins...')
    model.fit_generator(
        train_generator,
        steps_per_epoch=340,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=100,
        callbacks=[ModelCheckpoint(filepath=top_model_weights_path, sav
e_best_only=True, save_weights_only=True)]
        )

    model.save_weights(top_model_weights_path)
    # final weights are saved in bottleneck_fc_model.h5 file

trainCNN()
```

```
Compilation done.
Found 99724 images belonging to 2000 classes.
Found 21400 images belonging to 2000 classes.
Model fit begins...
WARNING:tensorflow:From <ipython-input-24-6b4d59130c1f>:65: Model.fit
_generator (from tensorflow.python.keras.engine.training) is deprecat
ed and will be removed in a future version.
Instructions for updating:
Please use Model.fit, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/50
340/340 [==============================] - 450s 1s/step - loss: 5.019
1 - accuracy: 0.1338 - val_loss: 4.4888 - val_accuracy: 0.2026
Epoch 2/50
340/340 [==============================] - 411s 1s/step - loss: 4.167
8 - accuracy: 0.2232 - val_loss: 3.9779 - val_accuracy: 0.2918
Epoch 3/50
340/340 [==============================] - 409s 1s/step - loss: 3.801
4 - accuracy: 0.2725 - val_loss: 3.7460 - val_accuracy: 0.3413
Epoch 4/50
340/340 [==============================] - 403s 1s/step - loss: 3.563
2 - accuracy: 0.3087 - val_loss: 3.5531 - val_accuracy: 0.3729
Epoch 5/50
340/340 [==============================] - 394s 1s/step - loss: 3.394
8 - accuracy: 0.3369 - val_loss: 3.4336 - val_accuracy: 0.3968
Epoch 6/50
340/340 [==============================] - 394s 1s/step - loss: 3.267
3 - accuracy: 0.3519 - val_loss: 3.2745 - val_accuracy: 0.4268
Epoch 7/50
340/340 [==============================] - 390s 1s/step - loss: 3.150
4 - accuracy: 0.3697 - val_loss: 3.0824 - val_accuracy: 0.4572
Epoch 8/50
340/340 [==============================] - 394s 1s/step - loss: 3.079
3 - accuracy: 0.3812 - val_loss: 3.0777 - val_accuracy: 0.4631
Epoch 9/50
340/340 [==============================] - 395s 1s/step - loss: 2.982
9 - accuracy: 0.3991 - val_loss: 2.9978 - val_accuracy: 0.4780
Epoch 10/50
340/340 [==============================] - 396s 1s/step - loss: 2.912
4 - accuracy: 0.4081 - val_loss: 2.9729 - val_accuracy: 0.4827
Epoch 11/50
340/340 [==============================] - 393s 1s/step - loss: 2.857
8 - accuracy: 0.4146 - val_loss: 2.8633 - val_accuracy: 0.5006
Epoch 12/50
340/340 [==============================] - 392s 1s/step - loss: 2.828
3 - accuracy: 0.4186 - val_loss: 2.8531 - val_accuracy: 0.5020
Epoch 13/50
340/340 [==============================] - 396s 1s/step - loss: 2.769
1 - accuracy: 0.4294 - val_loss: 2.8668 - val_accuracy: 0.5047
Epoch 14/50
340/340 [==============================] - 399s 1s/step - loss: 2.734
8 - accuracy: 0.4348 - val_loss: 2.8277 - val_accuracy: 0.5141
Epoch 15/50
340/340 [==============================] - 393s 1s/step - loss: 2.692
7 - accuracy: 0.4423 - val_loss: 2.8393 - val_accuracy: 0.5109
Epoch 16/50
340/340 [==============================] - 391s 1s/step - loss: 2.670
8 - accuracy: 0.4444 - val_loss: 2.7507 - val_accuracy: 0.5239
Epoch 17/50
340/340 [==============================] - 394s 1s/step - loss: 2.631
7 - accuracy: 0.4492 - val_loss: 2.8086 - val_accuracy: 0.5238
Epoch 18/50
340/340 [==============================] - 397s 1s/step - loss: 2.612
7 - accuracy: 0.4548 - val_loss: 2.7775 - val_accuracy: 0.5248
Epoch 19/50
340/340 [==============================] - 393s 1s/step - loss: 2.588
```

```
5 - accuracy: 0.4554 - val_loss: 2.7405 - val_accuracy: 0.5415
Epoch 20/50
340/340 [==============================] - 388s 1s/step - loss: 2.567
6 - accuracy: 0.4613 - val_loss: 2.6432 - val_accuracy: 0.5456
Epoch 21/50
340/340 [==============================] - 382s 1s/step - loss: 2.524
7 - accuracy: 0.4652 - val_loss: 2.6720 - val_accuracy: 0.5534
Epoch 22/50
340/340 [==============================] - 392s 1s/step - loss: 2.505
0 - accuracy: 0.4715 - val_loss: 2.7038 - val_accuracy: 0.5500
Epoch 23/50
340/340 [==============================] - 393s 1s/step - loss: 2.502
7 - accuracy: 0.4702 - val_loss: 2.6991 - val_accuracy: 0.5521
Epoch 24/50
340/340 [==============================] - 403s 1s/step - loss: 2.465
0 - accuracy: 0.4800 - val_loss: 2.6570 - val_accuracy: 0.5636
Epoch 25/50
340/340 [==============================] - 391s 1s/step - loss: 2.454
8 - accuracy: 0.4802 - val_loss: 2.6127 - val_accuracy: 0.5676
Epoch 26/50
340/340 [==============================] - 370s 1s/step - loss: 2.442
8 - accuracy: 0.4799 - val_loss: 2.6527 - val_accuracy: 0.5635
Epoch 27/50
340/340 [==============================] - 440s 1s/step - loss: 2.427
9 - accuracy: 0.4827 - val_loss: 2.6358 - val_accuracy: 0.5607
Epoch 28/50
340/340 [==============================] - 398s 1s/step - loss: 2.412
9 - accuracy: 0.4870 - val_loss: 2.7533 - val_accuracy: 0.5520
Epoch 29/50
340/340 [==============================] - 383s 1s/step - loss: 2.391
9 - accuracy: 0.4908 - val_loss: 2.6785 - val_accuracy: 0.5692
Epoch 30/50
340/340 [==============================] - 366s 1s/step - loss: 2.380
0 - accuracy: 0.4916 - val_loss: 2.6147 - val_accuracy: 0.5724
Epoch 31/50
340/340 [==============================] - 397s 1s/step - loss: 2.363
6 - accuracy: 0.4948 - val_loss: 2.5595 - val_accuracy: 0.5881
Epoch 32/50
340/340 [==============================] - 387s 1s/step - loss: 2.350
6 - accuracy: 0.4975 - val_loss: 2.5713 - val_accuracy: 0.5836
Epoch 33/50
340/340 [==============================] - 382s 1s/step - loss: 2.355
1 - accuracy: 0.4981 - val_loss: 2.5431 - val_accuracy: 0.5883
Epoch 34/50
340/340 [==============================] - 363s 1s/step - loss: 2.342
2 - accuracy: 0.4985 - val_loss: 2.6465 - val_accuracy: 0.5835
Epoch 35/50
340/340 [==============================] - 357s 1s/step - loss: 2.324
4 - accuracy: 0.5001 - val_loss: 2.6128 - val_accuracy: 0.5874
Epoch 36/50
340/340 [==============================] - 350s 1s/step - loss: 2.319
7 - accuracy: 0.5029 - val_loss: 2.6066 - val_accuracy: 0.5829
Epoch 37/50
```

In [ ]:
```
from tensorflow.python.platform import app
import argparse
import os
import sys
import time
from time import *
import io
import tensorflow as tf
```

In [ ]:
```
ls
```

```
admset213.zip                        test.csv
bottleneck_fc_model.h5               test_images_from_train/
bottleneck_features_train.npy        test_sample.csv
bottleneck_features_validation.npy   train.csv
class_indices.npy                    train_images_model/
result.csv                           train_sample.csv
RUN/                                 validation_images_model/
run2.ipynb                           validation_sample.csv
```

In [ ]:
```
top_model_weights_path = 'bottleneck_fc_model.h5'       # final weights
of the model
train_data_dir = 'train_images_model'
testfile = 'test_images_from_train'
subfile = 'result.csv'                                  # predictions o
f the model
```

In [ ]:
```
from keras import backend as K
import cv2
top_model_weights_path = 'bottleneck_fc_model.h5'       # final weights
of the model
train_data_dir = 'train_images_model'
testfile = 'test_images_from_train'
subfile = 'result2.csv'                                  # predictions
of the model
def predict(image_path):
    print ('starting')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)
    #train_datagen = ImageDataGenerator(rescale=1. / 255)
    #generator = train_datagen.flow_from_directory(train_data_dir, batc
h_size=batch_size)
    #label_map = (generator.class_indices)
    label_map={}
    for i in range(2000):
      label_map[str(3000+i)]=i
    n_class = 2000
    base_model = applications.VGG16(weights='imagenet', include_top=Fal
se, input_shape=(96, 96, 3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(n_class, activation='softmax'))
    model = Model(base_model.input,top_model(base_model.output))
    model.load_weights(top_model_weights_path)
    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks','confidence'])
        file_counter = 0
        for root, dirs, files in os.walk(image_path):  # loop through s
tartfolders
            i=0
            for pic in files:
                i+=1
                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                # important! otherwise the predictions will be '0'
                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark
```

```
                prediction = model.predict(image)


                class_predicted = prediction.argmax(axis=1)
                #class_predicted = np.argmax(prediction,axis=1)
                #print (pic, class_predicted)


                inID = class_predicted[0]
                #print inID

                inv_map = {v: k for k, v in label_map.items()}
                #print class_dictionary
                label = inv_map[inID]
                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                #out = str(label) + ' '+ scor
                print (i,score,label)
                newFileWriter.writerow([os.path.splitext(pic)[0], int(l
abel),scor])

predict(testfile)
```

In [ ]:
```
#result = pd.read_csv('result.csv')      # predicted values
```

In [ ]:
```
#test = pd.read_csv('test_sample.csv')  # ground truth
```

In [ ]:
```
# final accuracy of the model as tested on test dataset
import pandas as pd
def test_accuracy(pred,truth):
    result=pd.read_csv(pred)
    #print(result.head(20))
    test=pd.read_csv(truth)
    count=0
    for i in result["id"]:
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #b=list(b[0].split(" "))
        b[0]=int(b[0])
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              count+=1
    print(count/len(result))
    return count/len(result)
test_accuracy("result2.csv","test_sample.csv")
```

```
0.6106811145510835
```

Out[ ]:
```
0.6106811145510835
```

In [ ]:
```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("test_sample.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
GAP_metric("result2.csv")
```

```
5168 2947.5271513731795 5168 3156
0.5703419410551818
```

Out[ ]:
```
0.5703419410551818
```

In [ ]:
```
#GAP_metric('result.csv')
```

In [ ]:
```

```

In [ ]:
```
# Make sure we are pointing to the directory that has all the files nec
essary
import numpy as np
import pandas as pd
import sys, requests, shutil, os
```

In [ ]:
```
cd "/content/sample_data/dataset2"
```

```
/content/sample_data/dataset2
```

In [ ]:
```
pwd
```

Out[ ]:
```
'/content/drive/My Drive/ADM_2/dataset2'
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
```

In [ ]:
```
import numpy as np
import pandas as pd
import sys, requests, shutil, os

data_train=pd.read_csv("train_sample.csv")
data_valid=pd.read_csv("valid_sample.csv")
data_test=pd.read_csv("test_sample.csv")
```

In [ ]:
```
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator, im
g_to_array, load_img
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense, GlobalAver
agePooling2D
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
#from tensorflow.keras.utils.np_utils import to_categorical
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model
import csv
import os
#import cv2
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import math
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import tensorflow as tf
import tensorflow.keras
```

In [ ]:
```
# This cell counts the total number of train images and validation images in the entire dataset
# It stores the counts of train data 108181 training samples belonging to 1000 classes and
# 19826 test samples in the respective variables nb_train_samples and nb_validation_samples



train_data_dir = 'train_images_model'                    # points to directory having train images
validation_data_dir = 'validation_images_model'          # points to directory having validation images


def count(dir):
    i = 5000
    count = []
    while i <= 6999:
        f = str(i)
        #print (f)
        for root, dirs, files in os.walk(dir +'/'+ f):  # os. walk() is used to iterate through all folders of a directory
            for pic in files:
                count.append(f)
            i += 1
    print (len(count))
    return ([len(count),count])

nb_train_samples = count(train_data_dir)
nb_validation_samples = count(validation_data_dir)
```

```
63750
19380
```

In [ ]:
```
#Importing few other files needed
import pandas as pd
import os
import shutil
from shutil import copyfile
import urllib
```

In [ ]:
```
# here we build the VGG 16 model, which is pretrained using the imagene
t dataset,
# This is achieved by setting weights = 'imagenet' as a parameter while
building the model

img_width, img_height = 96, 96 #dimensions used in the model
top_model_weights_path = 'bottleneck_fc_model.h5' # this is used to sav
e weights in later stages, avoid recomputations
epochs = 5
batch_size = 510 #GCD
def save_bottleneck_features():
    datagen = ImageDataGenerator(rescale=1. / 255,
                                 rotation_range=30,
                                 width_shift_range=0.2,
                                 height_shift_range=0.2,
                                 zoom_range = 0.5,
                                 brightness_range = [0.5,1.5])

    # VGG 16 FramerWork
    model = applications.VGG16(include_top=False, weights='imagenet', i
nput_shape=(96,96,3))
    print ('start1')
    generator = datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)

    print ('start2')
    bottleneck_features_train = model.predict_generator(generator, nb_t
rain_samples[0] // batch_size)
    print ('bottleneck_features_trained')


    with open('bottleneck_features_train.npy', 'wb') as features_train_
file:
        np.save(features_train_file, bottleneck_features_train)
    print ('Train done')


    datagen = ImageDataGenerator(rescale=1. / 255)
    generator = datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)
    print ('validation predict start')
    bottleneck_features_validation = model.predict_generator(generator,
nb_validation_samples[0] // batch_size)

    with open('bottleneck_features_validation.npy', 'wb') as features_v
alidation_file:
        np.save(features_validation_file, bottleneck_features_validatio
```

```
n)
    print ('validation done')
save_bottleneck_features()
```

```
start1
Found 63750 images belonging to 2000 classes.
start2
WARNING:tensorflow:From <ipython-input-6-43ba42158bdc>:27: Model.pred
ict_generator (from tensorflow.python.keras.engine.training) is depre
cated and will be removed in a future version.
Instructions for updating:
Please use Model.predict, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "

bottleneck_features_trained
Train done
Found 19380 images belonging to 2000 classes.
validation predict start
validation done
```

In [ ]:

In [ ]:
```
# counts of train and validation labels
train_labels = np.array(nb_train_samples[1])
train_labels = [str(int(train_label) - 5000) for train_label in train_l
abels]
print(len(train_labels))
train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
print(len(train_data))
validation_data = np.load(open('bottleneck_features_validation.npy', 'r
b'))
print(len(validation_data))
validation_labels = np.array(nb_validation_samples[1])
print(len(validation_labels))
```

```
63920
63920
19380
19380
```

In [ ]:
```
epochs = 5
batch_size = 510
import numpy as np

def train_top_model():
    train_data = np.load(open('bottleneck_features_train.npy', 'rb'))
    train_labels = np.array(nb_train_samples[1])
    train_labels = [str(int(train_label) - 5000) for train_label in tra
in_labels]
    # we do label-1000, so that class lable starts from 0, as we have l
abels starting from 1000

    validation_data = np.load(open('bottleneck_features_validation.npy
', 'rb'))
    validation_labels = np.array(nb_validation_samples[1])
    validation_labels = [str(int(validation_label) - 5000) for validati
on_label in validation_labels]

    model = Sequential()
    model.add(Flatten(input_shape=train_data.shape[1:]))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(256, activation='relu'))
    n_class = 2000 # max. classes given to the model
    model.add(Dense(n_class, activation='softmax'))
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy', metrics=['accuracy
'])
    train_labels = to_categorical(train_labels, n_class)
    validation_labels = to_categorical(validation_labels, n_class)

    print ('model fit starting')

    model.fit(train_data, train_labels,epochs=epochs,batch_size=batch_s
ize,validation_data=(validation_data, validation_labels))
    model.save_weights(top_model_weights_path)

train_top_model()
```

```
model fit starting
Epoch 1/5
125/125 [==============================] - 2s 15ms/step - loss: 5.737
7 - accuracy: 0.0910 - val_loss: 4.7274 - val_accuracy: 0.1746
Epoch 2/5
125/125 [==============================] - 2s 14ms/step - loss: 4.737
9 - accuracy: 0.1807 - val_loss: 4.0420 - val_accuracy: 0.2614
Epoch 3/5
125/125 [==============================] - 2s 13ms/step - loss: 4.161
1 - accuracy: 0.2398 - val_loss: 3.5902 - val_accuracy: 0.3257
Epoch 4/5
125/125 [==============================] - 2s 13ms/step - loss: 3.736
2 - accuracy: 0.2842 - val_loss: 3.3032 - val_accuracy: 0.3641
Epoch 5/5
125/125 [==============================] - 2s 13ms/step - loss: 3.404
6 - accuracy: 0.3229 - val_loss: 3.2014 - val_accuracy: 0.3829
```

In [ ]:
```
base_model = applications.VGG16(weights='imagenet',include_top= False,i
nput_shape=(96,96,3))
print(base_model.input)
```

```
Tensor("input_12:0", shape=(None, 96, 96, 3), dtype=float32)
```

In [ ]:
```
pwd
```

Out[ ]:
```
'/content/drive/My Drive/ADM_2/dataset2'
```

In [ ]:
```
# This is where we fine tune the pretrained model according to our data set

img_width, img_height = 96, 96
top_model_weights_path = 'bottleneck_fc_model.h5'
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 50
def trainCNN():

    # build the VGG16 network

    base_model = applications.VGG16(weights='imagenet',include_top= False,input_shape=(96,96,3))

    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    n_class = 2000
    top_model.add(Dense(n_class, activation='softmax'))
    top_model.load_weights(top_model_weights_path) #Load the weights initialized in previous steps


    model = Model(base_model.input, top_model(base_model.output))

    # set the first 16 layers to non-trainable (weights will not be updated)
    # 1 conv layer and three dense layers will be trained
    for layer in model.layers[:16]:
        layer.trainable = False

    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(lr=0.001, beta_1=0.9,beta_2=0.999,epsilon=1e-8, decay=0.0),
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    np.save('class_indices.npy', train_generator.class_indices)
```

```
    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    print ('Model fit begins...')
    history=model.fit_generator(
        train_generator,
        steps_per_epoch=150,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=75,
        callbacks=[ModelCheckpoint(filepath=top_model_weights_path, sav
e_best_only=True, save_weights_only=True)]
        )

    model.save_weights(top_model_weights_path)
    # final weights are saved in bottleneck_fc_model.h5 file

trainCNN()
```

```
Compilation done.
Found 63750 images belonging to 2000 classes.
Found 19380 images belonging to 2000 classes.
Model fit begins...

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/50
150/150 [==============================] - 148s 985ms/step - loss: 5.
9212 - accuracy: 0.0644 - val_loss: 4.8089 - val_accuracy: 0.1310
Epoch 2/50
150/150 [==============================] - 143s 956ms/step - loss: 4.
8514 - accuracy: 0.1346 - val_loss: 4.2770 - val_accuracy: 0.1929
Epoch 3/50
150/150 [==============================] - 142s 948ms/step - loss: 4.
5148 - accuracy: 0.1729 - val_loss: 3.9892 - val_accuracy: 0.2443
Epoch 4/50
150/150 [==============================] - 139s 927ms/step - loss: 4.
2142 - accuracy: 0.2106 - val_loss: 3.7237 - val_accuracy: 0.2791
Epoch 5/50
150/150 [==============================] - 140s 931ms/step - loss: 4.
0307 - accuracy: 0.2354 - val_loss: 3.5047 - val_accuracy: 0.3213
Epoch 6/50
150/150 [==============================] - 146s 975ms/step - loss: 3.
8505 - accuracy: 0.2577 - val_loss: 3.4891 - val_accuracy: 0.3185
Epoch 7/50
150/150 [==============================] - 142s 949ms/step - loss: 3.
7308 - accuracy: 0.2775 - val_loss: 3.2901 - val_accuracy: 0.3512
Epoch 8/50
150/150 [==============================] - 139s 928ms/step - loss: 3.
6172 - accuracy: 0.2911 - val_loss: 3.2440 - val_accuracy: 0.3595
Epoch 9/50
150/150 [==============================] - 140s 931ms/step - loss: 3.
5052 - accuracy: 0.3068 - val_loss: 3.0912 - val_accuracy: 0.3795
Epoch 10/50
150/150 [==============================] - 141s 942ms/step - loss: 3.
4322 - accuracy: 0.3153 - val_loss: 3.0661 - val_accuracy: 0.3883
Epoch 11/50
150/150 [==============================] - 140s 935ms/step - loss: 3.
3501 - accuracy: 0.3284 - val_loss: 2.9684 - val_accuracy: 0.4000
Epoch 12/50
150/150 [==============================] - 141s 943ms/step - loss: 3.
2860 - accuracy: 0.3363 - val_loss: 2.9319 - val_accuracy: 0.4133
Epoch 13/50
150/150 [==============================] - 141s 939ms/step - loss: 3.
1907 - accuracy: 0.3507 - val_loss: 2.7982 - val_accuracy: 0.4303
Epoch 14/50
150/150 [==============================] - 142s 946ms/step - loss: 3.
1548 - accuracy: 0.3554 - val_loss: 2.7664 - val_accuracy: 0.4332
Epoch 15/50
150/150 [==============================] - 140s 934ms/step - loss: 3.
0962 - accuracy: 0.3622 - val_loss: 2.7238 - val_accuracy: 0.4499
Epoch 16/50
150/150 [==============================] - 141s 937ms/step - loss: 3.
0455 - accuracy: 0.3719 - val_loss: 2.7355 - val_accuracy: 0.4396
Epoch 17/50
150/150 [==============================] - 139s 927ms/step - loss: 2.
9857 - accuracy: 0.3818 - val_loss: 2.7746 - val_accuracy: 0.4383
Epoch 18/50
150/150 [==============================] - 140s 933ms/step - loss: 2.
9306 - accuracy: 0.3895 - val_loss: 2.7031 - val_accuracy: 0.4553
Epoch 19/50
150/150 [==============================] - 140s 936ms/step - loss: 2.
```

```
9060 - accuracy: 0.3934 - val_loss: 2.5824 - val_accuracy: 0.4701
Epoch 20/50
150/150 [==============================] - 139s 929ms/step - loss: 2.
8671 - accuracy: 0.3964 - val_loss: 2.6264 - val_accuracy: 0.4671
Epoch 21/50
150/150 [==============================] - 140s 936ms/step - loss: 2.
8368 - accuracy: 0.4030 - val_loss: 2.5629 - val_accuracy: 0.4769
Epoch 22/50
150/150 [==============================] - 139s 927ms/step - loss: 2.
8175 - accuracy: 0.4086 - val_loss: 2.6068 - val_accuracy: 0.4683
Epoch 23/50
150/150 [==============================] - 141s 937ms/step - loss: 2.
7713 - accuracy: 0.4139 - val_loss: 2.5673 - val_accuracy: 0.4773
Epoch 24/50
150/150 [==============================] - 139s 926ms/step - loss: 2.
7538 - accuracy: 0.4145 - val_loss: 2.5677 - val_accuracy: 0.4733
Epoch 25/50
150/150 [==============================] - 140s 933ms/step - loss: 2.
7314 - accuracy: 0.4214 - val_loss: 2.5093 - val_accuracy: 0.4867
Epoch 26/50
150/150 [==============================] - 139s 925ms/step - loss: 2.
6997 - accuracy: 0.4235 - val_loss: 2.4719 - val_accuracy: 0.4970
Epoch 27/50
150/150 [==============================] - 139s 925ms/step - loss: 2.
6954 - accuracy: 0.4230 - val_loss: 2.5081 - val_accuracy: 0.4870
Epoch 28/50
150/150 [==============================] - 138s 921ms/step - loss: 2.
6642 - accuracy: 0.4248 - val_loss: 2.5417 - val_accuracy: 0.4885
Epoch 29/50
150/150 [==============================] - 139s 928ms/step - loss: 2.
6431 - accuracy: 0.4320 - val_loss: 2.4231 - val_accuracy: 0.5079
Epoch 30/50
150/150 [==============================] - 141s 938ms/step - loss: 2.
6354 - accuracy: 0.4299 - val_loss: 2.4371 - val_accuracy: 0.5068
Epoch 31/50
150/150 [==============================] - 142s 948ms/step - loss: 2.
6028 - accuracy: 0.4389 - val_loss: 2.3947 - val_accuracy: 0.5141
Epoch 32/50
150/150 [==============================] - 142s 946ms/step - loss: 2.
5847 - accuracy: 0.4438 - val_loss: 2.3196 - val_accuracy: 0.5262
Epoch 33/50
150/150 [==============================] - 141s 937ms/step - loss: 2.
5555 - accuracy: 0.4469 - val_loss: 2.3764 - val_accuracy: 0.5161
Epoch 34/50
150/150 [==============================] - 141s 941ms/step - loss: 2.
5435 - accuracy: 0.4461 - val_loss: 2.4482 - val_accuracy: 0.5049
Epoch 35/50
150/150 [==============================] - 140s 936ms/step - loss: 2.
```

In [ ]:

In [ ]:
```
from tensorflow.python.platform import app
import argparse
import os
import sys
import time
from time import *
import io
import tensorflow as tf
import cv2
```

In [ ]:
```
top_model_weights_path = 'bottleneck_fc_model.h5'        # final weights
of the model
train_data_dir = 'train_images_model'
testfile = 'test_images_from_train'
subfile = 'result2.csv'                                   # predictions
of the model
```

In [ ]:
```
from keras import backend as K
import cv2

# here we send the test images to the model
# the model predicts the classes with some confidence score
# these tuples are stored in a file result.csv that are used in later p
arts of the code

def predict(image_path):
    print ('starting...')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)

    train_datagen = ImageDataGenerator(rescale=1. / 255)

    generator = train_datagen.flow_from_directory(train_data_dir, batch
_size=batch_size)
    label_map = (generator.class_indices)

    n_class = 2000

    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks'])

        file_counter = 0
        base_model = applications.VGG16(weights='imagenet', include_top
=False, input_shape=(96, 96, 3))

        top_model = Sequential()
        top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
        top_model.add(Dense(256, activation='relu'))
        top_model.add(Dense(256, activation='relu'))
        top_model.add(Dense(n_class, activation='softmax'))

        model = Model(base_model.input, top_model(base_model.output))
        model.load_weights(top_model_weights_path)
        for root, dirs, files in os.walk(image_path):
            for pic in files:

                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark
```

```
                prediction = model.predict(image)

                class_predicted = prediction.argmax(axis=1)
                inID = class_predicted[0]
                inv_map = {v: k for k, v in label_map.items()}
                label = inv_map[inID]

                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                out = str(label) + ' '+ scor
                #print(score)

                newFileWriter.writerow([os.path.splitext(pic)[0], out])


predict(testfile)
```

In [ ]:
```
result = pd.read_csv('result2.csv')      # predicted values
```

In [ ]:
```
from keras import backend as K
import cv2
top_model_weights_path = 'bottleneck_fc_model.h5'       # final weights
of the model
train_data_dir = 'train_images_model'
testfile = 'test_images_from_train'
subfile = 'result2.csv'                                  # predictions
of the model
def predict(image_path):
    print ('starting')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)
    #train_datagen = ImageDataGenerator(rescale=1. / 255)
    #generator = train_datagen.flow_from_directory(train_data_dir, batc
h_size=batch_size)
    #label_map = (generator.class_indices)
    label_map={}
    for i in range(2000):
      label_map[str(5000+i)]=i
    n_class = 2000
    base_model = applications.VGG16(weights='imagenet', include_top=Fal
se, input_shape=(96, 96, 3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(n_class, activation='softmax'))
    model = Model(base_model.input,top_model(base_model.output))
    model.load_weights(top_model_weights_path)
    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks','confidence'])
        file_counter = 0
        for root, dirs, files in os.walk(image_path):  # loop through s
tartfolders
            i=0
            for pic in files:
                i+=1
                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                # important! otherwise the predictions will be '0'
                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark
```

```
                prediction = model.predict(image)


                class_predicted = prediction.argmax(axis=1)
                #class_predicted = np.argmax(prediction,axis=1)
                #print (pic, class_predicted)


                inID = class_predicted[0]
                #print inID

                inv_map = {v: k for k, v in label_map.items()}
                #print class_dictionary
                label = inv_map[inID]
                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                #out = str(label) + ' '+ scor
                #print (i,score,label)
                newFileWriter.writerow([os.path.splitext(pic)[0], int(l
abel),scor])

predict(testfile)
```

In [ ]:
```
test = pd.read_csv('test_sample.csv')  # ground truth
```

In [ ]:
```
# final accuracy of the model as tested on test dataset
import pandas as pd
def test_accuracy(pred,truth):
    result=pd.read_csv(pred)
    #print(result.head(20))
    test=pd.read_csv(truth)
    count=0
    for i in result["id"]:
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #b=list(b[0].split(" "))
        b[0]=int(b[0])
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              count+=1
    print(count/len(result))
    return count/len(result)
test_accuracy("result2.csv","test_sample.csv")
```

```
0.5410830022998119
```

Out[ ]:
```
0.5410830022998119
```

In [ ]:
```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("test_sample.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
GAP_metric("result.csv")
```

In [ ]:
```
GAP_metric('result2.csv')
```

```
4783 2345.7565041894886 4783 2588
0.4904362333659813
```

Out[ ]:
```
0.4904362333659813
```

In [ ]:
```
!unzip "/content/drive/My Drive/ADM/dataset3.zip" -d "/content/sample_data/dataset"
```

```
Archive:  /content/drive/My Drive/ADM/dataset3.zip
replace /content/sample_data/dataset/dataset3/test_images_from_train/00fb450622785388.jpg? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

In [ ]:
```
# Make sure we are pointing to the directory that has all the files necessary
import numpy as np
import pandas as pd
import sys, requests, shutil, os
import numpy as np
from shutil import copyfile
import urllib
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
#from tensorflow.keras.utils.np_utils import to_categorical
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model
import csv
import os
import cv2
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import math
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import tensorflow as tf
import tensorflow.keras
from keras.optimizers import SGD, Adam
```

In [ ]:
```
cd "/content/sample_data/dataset/dataset3"
```

```
/content/sample_data/dataset/dataset3
```

In [ ]:
```
# This is where we fine tune the pretrained model according to our data set
img_width, img_height = 96, 96
save_model_weights = "VGG16_weights.h5"
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 100
def train_VGG16():



    base_model = applications.VGG16(weights='imagenet',include_top= False,input_shape=(96,96,3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(256, activation='relu'))
    top_model.add(Dense(256, activation='relu'))
    n_class = 6000
    top_model.add(Dense(n_class, activation='softmax'))

    model = Model(base_model.input, top_model(base_model.output))
    # set the first 16 layers to non-trainable (weights will not be updated)
    # 1 conv layer and three dense layers will be trained
    for layer in model.layers[:16]:
        layer.trainable = False
    #model.load_weights("/content/sample_data/dataset/dataset3/VGG16_weights.h5")
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(lr=0.001, beta_1=0.9,beta_2=0.999,epsilon=1e-8, decay=0.0),
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    #np.save('class_indices.npy', train_generator.class_indices)

    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
```

```
        class_mode='categorical')

    print ('Model fit begins...')
    history1=model.fit_generator(
        train_generator,
        steps_per_epoch=150,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=75,
        callbacks=[ModelCheckpoint(filepath="vgg16_weights_tf_dim_order
ing_tf_kernels-notop.h5",
                                   save_best_only=True, save_weights_on
ly=True)]
        )

    model.save_weights(save_model_weights)
    # final weights are saved in bottleneck_fc_model.h5 file
    return history1
history1=train_VGG16()
```

```
Compilation done.
Found 246794 images belonging to 6000 classes.
Found 54916 images belonging to 6000 classes.
Model fit begins...
```

```
/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/100
150/150 [==============================] - 108s 722ms/step - loss: 7.
4300 - accuracy: 0.0090 - val_loss: 7.1718 - val_accuracy: 0.0139
Epoch 2/100
150/150 [==============================] - 109s 729ms/step - loss: 6.
9714 - accuracy: 0.0192 - val_loss: 6.9208 - val_accuracy: 0.0205
Epoch 3/100
150/150 [==============================] - 108s 722ms/step - loss: 6.
6725 - accuracy: 0.0275 - val_loss: 6.6716 - val_accuracy: 0.0251
Epoch 4/100
150/150 [==============================] - 109s 726ms/step - loss: 6.
5231 - accuracy: 0.0335 - val_loss: 6.4537 - val_accuracy: 0.0350
Epoch 5/100
150/150 [==============================] - 109s 726ms/step - loss: 6.
4007 - accuracy: 0.0410 - val_loss: 6.3077 - val_accuracy: 0.0434
Epoch 6/100
150/150 [==============================] - 109s 727ms/step - loss: 6.
2400 - accuracy: 0.0529 - val_loss: 6.3571 - val_accuracy: 0.0443
Epoch 7/100
150/150 [==============================] - 110s 731ms/step - loss: 6.
1298 - accuracy: 0.0607 - val_loss: 6.1201 - val_accuracy: 0.0615
Epoch 8/100
150/150 [==============================] - 110s 731ms/step - loss: 5.
9918 - accuracy: 0.0681 - val_loss: 6.0070 - val_accuracy: 0.0675
Epoch 9/100
150/150 [==============================] - 109s 727ms/step - loss: 5.
8889 - accuracy: 0.0771 - val_loss: 5.9660 - val_accuracy: 0.0698
Epoch 10/100
150/150 [==============================] - 110s 731ms/step - loss: 5.
7831 - accuracy: 0.0857 - val_loss: 5.6975 - val_accuracy: 0.0823
Epoch 11/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
7154 - accuracy: 0.0882 - val_loss: 5.6943 - val_accuracy: 0.0902
Epoch 12/100
150/150 [==============================] - 109s 726ms/step - loss: 5.
6435 - accuracy: 0.0931 - val_loss: 5.5814 - val_accuracy: 0.0971
Epoch 13/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
5917 - accuracy: 0.0958 - val_loss: 5.5662 - val_accuracy: 0.1003
Epoch 14/100
150/150 [==============================] - 109s 729ms/step - loss: 5.
5090 - accuracy: 0.0989 - val_loss: 5.4818 - val_accuracy: 0.1073
Epoch 15/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
4479 - accuracy: 0.1081 - val_loss: 5.4121 - val_accuracy: 0.1115
Epoch 16/100
150/150 [==============================] - 109s 724ms/step - loss: 5.
4146 - accuracy: 0.1098 - val_loss: 5.3636 - val_accuracy: 0.1151
Epoch 17/100
150/150 [==============================] - 109s 725ms/step - loss: 5.
3553 - accuracy: 0.1113 - val_loss: 5.3788 - val_accuracy: 0.1195
Epoch 18/100
150/150 [==============================] - 110s 730ms/step - loss: 5.
3490 - accuracy: 0.1151 - val_loss: 5.4474 - val_accuracy: 0.1133
Epoch 19/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
```

```
2971 - accuracy: 0.1209 - val_loss: 5.3283 - val_accuracy: 0.1272
Epoch 20/100
150/150 [==============================] - 109s 726ms/step - loss: 5.
2446 - accuracy: 0.1248 - val_loss: 5.2383 - val_accuracy: 0.1331
Epoch 21/100
150/150 [==============================] - 110s 730ms/step - loss: 5.
2162 - accuracy: 0.1228 - val_loss: 5.1954 - val_accuracy: 0.1388
Epoch 22/100
150/150 [==============================] - 110s 730ms/step - loss: 5.
2105 - accuracy: 0.1253 - val_loss: 5.1831 - val_accuracy: 0.1293
Epoch 23/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
1667 - accuracy: 0.1315 - val_loss: 5.1527 - val_accuracy: 0.1325
Epoch 24/100
150/150 [==============================] - 109s 726ms/step - loss: 5.
1345 - accuracy: 0.1336 - val_loss: 5.2340 - val_accuracy: 0.1307
Epoch 25/100
150/150 [==============================] - 109s 725ms/step - loss: 5.
1241 - accuracy: 0.1340 - val_loss: 5.2166 - val_accuracy: 0.1418
Epoch 26/100
150/150 [==============================] - 108s 722ms/step - loss: 5.
1025 - accuracy: 0.1361 - val_loss: 5.0651 - val_accuracy: 0.1508
Epoch 27/100
150/150 [==============================] - 109s 728ms/step - loss: 5.
0498 - accuracy: 0.1392 - val_loss: 5.1113 - val_accuracy: 0.1483
Epoch 28/100
150/150 [==============================] - 109s 726ms/step - loss: 5.
0443 - accuracy: 0.1408 - val_loss: 5.1296 - val_accuracy: 0.1529
Epoch 29/100
150/150 [==============================] - 108s 723ms/step - loss: 5.
0499 - accuracy: 0.1398 - val_loss: 5.0342 - val_accuracy: 0.1507
Epoch 30/100
150/150 [==============================] - 108s 721ms/step - loss: 4.
9938 - accuracy: 0.1462 - val_loss: 5.0420 - val_accuracy: 0.1569
Epoch 31/100
150/150 [==============================] - 109s 726ms/step - loss: 4.
9983 - accuracy: 0.1471 - val_loss: 5.0137 - val_accuracy: 0.1583
Epoch 32/100
150/150 [==============================] - 109s 724ms/step - loss: 4.
9832 - accuracy: 0.1482 - val_loss: 5.0043 - val_accuracy: 0.1606
Epoch 33/100
150/150 [==============================] - 110s 730ms/step - loss: 4.
9514 - accuracy: 0.1527 - val_loss: 4.9065 - val_accuracy: 0.1719
Epoch 34/100
150/150 [==============================] - 109s 725ms/step - loss: 4.
9365 - accuracy: 0.1472 - val_loss: 4.9690 - val_accuracy: 0.1697
Epoch 35/100
150/150 [==============================] - 110s 732ms/step - loss: 4.
9152 - accuracy: 0.1490 - val_loss: 4.8987 - val_accuracy: 0.1738
Epoch 36/100
150/150 [==============================] - 110s 731ms/step - loss: 4.
8905 - accuracy: 0.1535 - val_loss: 4.8897 - val_accuracy: 0.1688
Epoch 37/100
150/150 [==============================] - 109s 727ms/step - loss: 4.
8836 - accuracy: 0.1577 - val_loss: 5.0219 - val_accuracy: 0.1615
Epoch 38/100
```

```
150/150 [==============================] - 109s 728ms/step - loss: 4.
8446 - accuracy: 0.1567 - val_loss: 4.9855 - val_accuracy: 0.1672
Epoch 39/100
150/150 [==============================] - 109s 724ms/step - loss: 4.
8623 - accuracy: 0.1564 - val_loss: 4.9100 - val_accuracy: 0.1745
Epoch 40/100
150/150 [==============================] - 108s 722ms/step - loss: 4.
8319 - accuracy: 0.1631 - val_loss: 4.9599 - val_accuracy: 0.1672
Epoch 41/100
150/150 [==============================] - 108s 723ms/step - loss: 4.
8018 - accuracy: 0.1641 - val_loss: 4.8263 - val_accuracy: 0.1813
Epoch 42/100
150/150 [==============================] - 109s 724ms/step - loss: 4.
8038 - accuracy: 0.1625 - val_loss: 4.8219 - val_accuracy: 0.1865
Epoch 43/100
150/150 [==============================] - 109s 727ms/step - loss: 4.
8120 - accuracy: 0.1608 - val_loss: 4.8227 - val_accuracy: 0.1813
Epoch 44/100
150/150 [==============================] - 109s 724ms/step - loss: 4.
8053 - accuracy: 0.1621 - val_loss: 4.7299 - val_accuracy: 0.1912
Epoch 45/100
150/150 [==============================] - 108s 723ms/step - loss: 4.
7757 - accuracy: 0.1666 - val_loss: 4.9076 - val_accuracy: 0.1785
Epoch 46/100
150/150 [==============================] - 108s 723ms/step - loss: 4.
7433 - accuracy: 0.1683 - val_loss: 4.8026 - val_accuracy: 0.1888
Epoch 47/100
150/150 [==============================] - 109s 724ms/step - loss: 4.
7614 - accuracy: 0.1668 - val_loss: 4.7838 - val_accuracy: 0.1893
Epoch 48/100
150/150 [==============================] - 109s 729ms/step - loss: 4.
7155 - accuracy: 0.1739 - val_loss: 4.8190 - val_accuracy: 0.1835
Epoch 49/100
150/150 [==============================] - 109s 728ms/step - loss: 4.
7411 - accuracy: 0.1712 - val_loss: 4.7230 - val_accuracy: 0.1908
Epoch 50/100
150/150 [==============================] - 109s 729ms/step - loss: 4.
7027 - accuracy: 0.1734 - val_loss: 4.7255 - val_accuracy: 0.1964
Epoch 51/100
150/150 [==============================] - 110s 732ms/step - loss: 4.
7041 - accuracy: 0.1739 - val_loss: 4.7440 - val_accuracy: 0.1863
Epoch 52/100
150/150 [==============================] - 111s 738ms/step - loss: 4.
6709 - accuracy: 0.1782 - val_loss: 4.9017 - val_accuracy: 0.1861
Epoch 53/100
150/150 [==============================] - 110s 734ms/step - loss: 4.
6747 - accuracy: 0.1765 - val_loss: 4.6901 - val_accuracy: 0.2022
Epoch 54/100
150/150 [==============================] - 110s 735ms/step - loss: 4.
6791 - accuracy: 0.1747 - val_loss: 4.6284 - val_accuracy: 0.2114
Epoch 55/100
150/150 [==============================] - 111s 739ms/step - loss: 4.
6236 - accuracy: 0.1842 - val_loss: 4.8191 - val_accuracy: 0.1897
Epoch 56/100
150/150 [==============================] - 110s 735ms/step - loss: 4.
6483 - accuracy: 0.1785 - val_loss: 4.8465 - val_accuracy: 0.1816
```

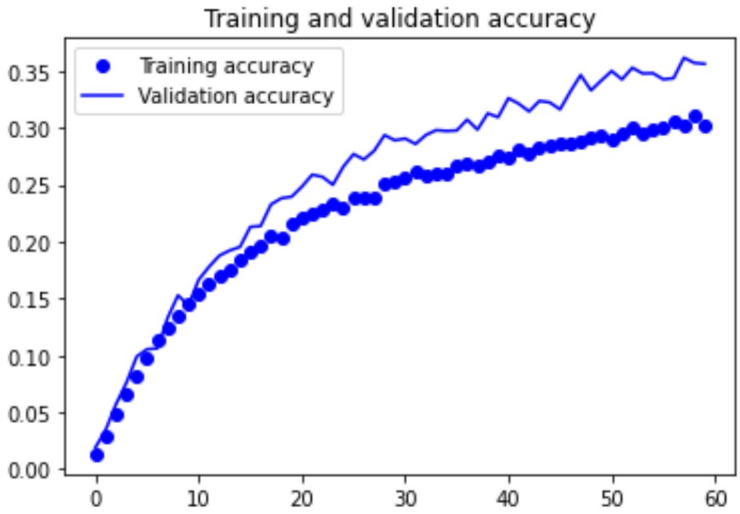
```
Epoch 57/100
150/150 [==============================] - 111s 737ms/step - loss: 4.
6349 - accuracy: 0.1788 - val_loss: 4.6319 - val_accuracy: 0.1984
Epoch 58/100
150/150 [==============================] - 111s 740ms/step - loss: 4.
6469 - accuracy: 0.1795 - val_loss: 4.8030 - val_accuracy: 0.1935
Epoch 59/100
150/150 [==============================] - 111s 738ms/step - loss: 4.
6271 - accuracy: 0.1807 - val_loss: 4.6667 - val_accuracy: 0.2045
Epoch 60/100
150/150 [==============================] - 111s 737ms/step - loss: 4.
5969 - accuracy: 0.1824 - val_loss: 4.7158 - val_accuracy: 0.1982
Epoch 61/100
150/150 [==============================] - 111s 739ms/step - loss: 4.
6065 - accuracy: 0.1827 - val_loss: 4.7126 - val_accuracy: 0.2009
Epoch 62/100
150/150 [==============================] - 111s 739ms/step - loss: 4.
5993 - accuracy: 0.1829 - val_loss: 4.8235 - val_accuracy: 0.1881
Epoch 63/100
150/150 [==============================] - 111s 739ms/step - loss: 4.
5754 - accuracy: 0.1884 - val_loss: 4.6701 - val_accuracy: 0.2016
Epoch 64/100
150/150 [==============================] - 111s 739ms/step - loss: 4.
6165 - accuracy: 0.1832 - val_loss: 4.6406 - val_accuracy: 0.2095
Epoch 65/100
150/150 [==============================] - 110s 734ms/step - loss: 4.
5615 - accuracy: 0.1873 - val_loss: 4.6796 - val_accuracy: 0.2053
Epoch 66/100
150/150 [==============================] - 110s 737ms/step - loss: 4.
5861 - accuracy: 0.1844 - val_loss: 4.7908 - val_accuracy: 0.2001
Epoch 67/100
150/150 [==============================] - 110s 734ms/step - loss: 4.
5303 - accuracy: 0.1902 - val_loss: 4.6051 - val_accuracy: 0.2157
Epoch 68/100
150/150 [==============================] - 110s 734ms/step - loss: 4.
5441 - accuracy: 0.1920 - val_loss: 4.6863 - val_accuracy: 0.2141
Epoch 69/100
150/150 [==============================] - 110s 734ms/step - loss: 4.
5555 - accuracy: 0.1838 - val_loss: 4.6632 - val_accuracy: 0.2169
Epoch 70/100
150/150 [==============================] - 109s 728ms/step - loss: 4.
5475 - accuracy: 0.1881 - val_loss: 4.6371 - val_accuracy: 0.2157
Epoch 71/100
150/150 [==============================] - 109s 730ms/step - loss: 4.
5341 - accuracy: 0.1900 - val_loss: 4.6168 - val_accuracy: 0.2135
Epoch 72/100
150/150 [==============================] - 110s 735ms/step - loss: 4.
5506 - accuracy: 0.1891 - val_loss: 4.6198 - val_accuracy: 0.2141
Epoch 73/100
150/150 [==============================] - 109s 729ms/step - loss: 4.
4899 - accuracy: 0.1952 - val_loss: 4.6055 - val_accuracy: 0.2180
Epoch 74/100
150/150 [==============================] - 109s 730ms/step - loss: 4.
5188 - accuracy: 0.1936 - val_loss: 4.7596 - val_accuracy: 0.2003
Epoch 75/100
150/150 [==============================] - 110s 730ms/step - loss: 4.
```

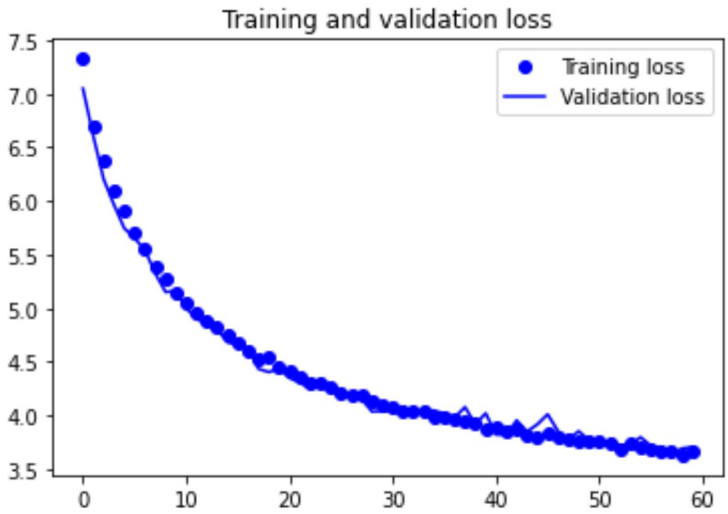
```
5118 - accuracy: 0.1942 - val_loss: 4.7052 - val_accuracy: 0.2155
Epoch 76/100
150/150 [==============================] - 108s 720ms/step - loss: 4.
5407 - accuracy: 0.1928 - val_loss: 4.4593 - val_accuracy: 0.2325
Epoch 77/100
150/150 [==============================] - 109s 725ms/step - loss: 4.
5090 - accuracy: 0.1949 - val_loss: 4.7124 - val_accuracy: 0.2168
Epoch 78/100
150/150 [==============================] - 108s 718ms/step - loss: 4.
4910 - accuracy: 0.1924 - val_loss: 4.6655 - val_accuracy: 0.2123
Epoch 79/100
150/150 [==============================] - 108s 721ms/step - loss: 4.
4860 - accuracy: 0.1936 - val_loss: 4.5779 - val_accuracy: 0.2245
Epoch 80/100
150/150 [==============================] - 110s 731ms/step - loss: 4.
4877 - accuracy: 0.1953 - val_loss: 4.5497 - val_accuracy: 0.2271
Epoch 81/100
150/150 [==============================] - 109s 726ms/step - loss: 4.
4845 - accuracy: 0.1953 - val_loss: 4.7317 - val_accuracy: 0.2072
Epoch 82/100
150/150 [==============================] - 108s 722ms/step - loss: 4.
4688 - accuracy: 0.1976 - val_loss: 4.5545 - val_accuracy: 0.2273
Epoch 83/100
150/150 [==============================] - 111s 743ms/step - loss: 4.
4392 - accuracy: 0.1977 - val_loss: 4.5983 - val_accuracy: 0.2251
Epoch 84/100
150/150 [==============================] - 110s 731ms/step - loss: 4.
4504 - accuracy: 0.1988 - val_loss: 4.6340 - val_accuracy: 0.2145
```

In [ ]:
```
history1
```

Out[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x7f04acdab198>
```

In [ ]:
```
acc = history1.history['accuracy']
val_acc = history1.history['val_accuracy']
loss = history1.history['loss']
val_loss = history1.history['val_loss']
epochs = range(60)
plt.figure()
plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.show()
epochs = range(60)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





In [ ]:

In [3]:
```
cd "/content/sample_data/dataset3"
```

```
/content/sample_data/dataset3
```

In [4]:
```
import os
```

In [11]:
```
from absl import logging

import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageOps
from scipy.spatial import cKDTree
from skimage.feature import plot_matches
from skimage.measure import ransac
from skimage.transform import AffineTransform
from six import BytesIO

import tensorflow as tf

import tensorflow_hub as hub
from six.moves.urllib.request import urlopen
```

In [6]:
```
delf = hub.load('https://tfhub.dev/google/delf/1').signatures['default
']
```

In [7]:
```
def run_delf(image):
  try:
    np_image = np.array(image)
    float_image = tf.image.convert_image_dtype(np_image, tf.float32)

    return delf(
      image=float_image,
      score_threshold=tf.constant(100.0),
      image_scales=tf.constant([0.25, 0.3536, 0.5, 0.7071, 1.0, 1.4142,
2.0]),
      max_feature_num=tf.constant(1000))
  except:
    return 0
```

In [8]:
```
def delf_work(path1, path2):
  image1 = Image.open(path1)
  image1 = ImageOps.fit(image1, (96, 96), Image.ANTIALIAS)
  image2 = Image.open(path2)
  image2 = ImageOps.fit(image2, (96, 96), Image.ANTIALIAS)
  result1 = run_delf(image1)
  result2 = run_delf(image2)

  if (type(result1) is int or type(result2) is int) and (result1==0 or
result2 == 0):
    return 0

  distance_threshold = 0.8

  # Read features.
  num_features_1 = result1['locations'].shape[0]
  #print("Loaded image 1's %d features" % num_features_1)

  num_features_2 = result2['locations'].shape[0]
  #print("Loaded image 2's %d features" % num_features_2)

  # Find nearest-neighbor matches using a KD tree.
  d1_tree = cKDTree(result1['descriptors'])
  _, indices = d1_tree.query(
      result2['descriptors'],
      distance_upper_bound=distance_threshold)

  # Select feature locations for putative matches.
  locations_2_to_use = np.array([
      result2['locations'][i,]
      for i in range(num_features_2)
      if indices[i] != num_features_1
  ])
  locations_1_to_use = np.array([
      result1['locations'][indices[i],]
      for i in range(num_features_2)
      if indices[i] != num_features_1
  ])

  # Perform geometric verification using RANSAC.
  try:
    _, inliers = ransac(
        (locations_1_to_use, locations_2_to_use),
        AffineTransform,
        min_samples=3,
        residual_threshold=20,
        max_trials=1000)
  except:
    inliers = [0]

  if inliers is None:
    print('Found 0 inliers')
    return 0
  print('Found %d inliers' % sum(inliers))
  return sum(inliers)
```

In [9]:
```
import pandas as pd
```

In [10]:
```
def local_feature(path):
  x = 0
  v=[]
  result = pd.read_csv(path)
  for i in result['id']:
    a = result.loc[result['id'] == i, "landmarks"]
    c = result.loc[result['id'] == i, "confidence"]
    a = a.values
    c = c.values
    #print(a,c)
    if c<0.4:
      p2 = '/content/sample_data/dataset3/train_images_model/'+str(a
[0])
      r = os.listdir(p2)

      if(len(r)>2):
        s=0
        for j in range(2):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s


      if avg < 3:
        result.loc[result.id == i, "confidence"] = 0
      elif avg >= 10:
        result.loc[result.id == i, "confidence"] = 0.6
    elif c>0.4 and c<0.6:
      p2 = '/content/sample_data/dataset3/train_images_model/'+str(a
[0])
      r = os.listdir(p2)

      if(len(r)>2):
        s=0
        for j in range(2):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
    elif c==1:
      p2 = '/content/sample_data/dataset3/train_images_model/'+str(a
[0])
      r = os.listdir(p2)
```

```
      if(len(r)>5):
        s=0
        for j in range(5):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s

      if avg < 3:
        result.loc[result.id == i, "confidence"] = 0
    #v.append([x,avg,c])
    print(x)
    x+=1



  result.to_csv('delf_test_best.csv')
  #return v
  #print(GAP_metric_2(result))
```

In [46]:

```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("/content/sample_data/dataset3/test_sample3.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M/,s/2,total,true)
    print(s/M)
    return s/M
```

In [34]:
```
GAP_metric("/content/sample_data/dataset3/delf_test_best.csv")
```

```
5305 2161.695945001019 13067 4100
0.4074827417532552
```

Out[34]:
```
0.4074827417532552
```

In [ ]:

```
# Make sure we are pointing to the directory that has all the files nec
essary
import numpy as np
import pandas as pd
import sys, requests, shutil, os
import numpy as np
from shutil import copyfile
import urllib
from tensorflow.keras.preprocessing.image import ImageDataGenerator, im
g_to_array, load_img
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense, GlobalAver
agePooling2D
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
#from tensorflow.keras.utils.np_utils import to_categorical
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model
import csv
import os
import cv2
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import math
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
import argparse
import random
import tensorflow as tf
import tensorflow.keras
from keras.optimizers import SGD, Adam
```

In [51]:
```
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("/content/drive/My Drive/ADM/test_sample3.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
```

In [ ]:
```
img_height,img_width = 96,96
num_classes = 6000
#If imagenet weights are being loaded,
#input must have a static square shape (one of (128, 128), (160, 160),
(192, 192), or (224, 224))
base_model = applications.resnet50.ResNet50(weights= 'imagenet', includ
e_top=False, input_shape= (img_height,img_width,3))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras
-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_noto
p.h5
94773248/94765736 [==============================] - 2s 0us/step
```

In [ ]:
```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.7)(x)
predictions = Dense(num_classes, activation= 'softmax')(x)
model = Model(inputs = base_model.input, outputs = predictions)
```

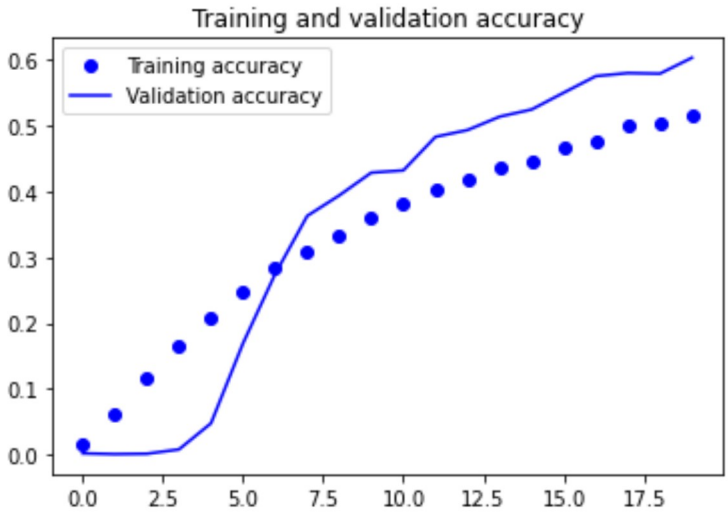
In [ ]:
```
adam = Adam(lr=0.0001)
model.compile(optimizer= adam, loss='categorical_crossentropy', metrics
=['accuracy'])
```

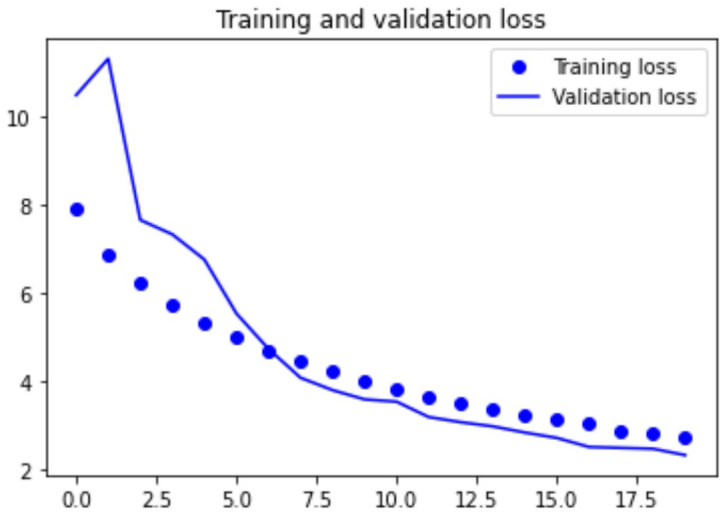
In [ ]:
```
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 20
train_datagen = ImageDataGenerator(rescale=1. / 255, rotation_range=90,
width_shift_range=0.2, height_shift_range=0.2, zoom_range = 0.5)

valid_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory( train_data_dir, ta
rget_size=(img_height, img_width), batch_size=batch_size,
                                                    class_mode='categor
ical')

#np.save('class_indices.npy', train_generator.class_indices)

validation_generator = valid_datagen.flow_from_directory(validation_dat
a_dir, target_size=(img_height, img_width), batch_size=batch_size,
                                                         class_mode='ca
tegorical')

print ('Model fit begins...')
history1=model.fit_generator( train_generator, steps_per_epoch=150, epo
chs=epochs,validation_data=validation_generator,
                             validation_steps=75,
                             callbacks=[ModelCheckpoint(filepath="resne
t50_weights_tf_dim_ordering_tf_kernels_notop.h5",
                                                        save_best_only=
True, save_weights_only=True)]
    )
model.save_weights("resNet_weights.h5")
```

```
Found 246794 images belonging to 6000 classes.
Found 54916 images belonging to 6000 classes.
Model fit begins...
WARNING:tensorflow:From <ipython-input-7-01e48294d2ea>:21: Model.fit_
generator (from tensorflow.python.keras.engine.training) is deprecate
d and will be removed in a future version.
Instructions for updating:
Please use Model.fit, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/20
150/150 [==============================] - 161s 1s/step - loss: 7.914
6 - accuracy: 0.0154 - val_loss: 10.5048 - val_accuracy: 0.0021
Epoch 2/20
150/150 [==============================] - 154s 1s/step - loss: 6.861
3 - accuracy: 0.0602 - val_loss: 11.3307 - val_accuracy: 0.0010
Epoch 3/20
150/150 [==============================] - 151s 1s/step - loss: 6.251
2 - accuracy: 0.1160 - val_loss: 7.6732 - val_accuracy: 0.0015
Epoch 4/20
150/150 [==============================] - 148s 988ms/step - loss: 5.
7626 - accuracy: 0.1641 - val_loss: 7.3462 - val_accuracy: 0.0081
Epoch 5/20
150/150 [==============================] - 146s 973ms/step - loss: 5.
3588 - accuracy: 0.2090 - val_loss: 6.7752 - val_accuracy: 0.0478
Epoch 6/20
150/150 [==============================] - 143s 955ms/step - loss: 4.
9996 - accuracy: 0.2490 - val_loss: 5.5506 - val_accuracy: 0.1695
```

In [ ]:
```
acc = history1.history['accuracy']
val_acc = history1.history['val_accuracy']
loss = history1.history['loss']
val_loss = history1.history['val_loss']
epochs = range(20)
plt.figure()
plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.show()
epochs = range(20)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





In [ ]:
```
from keras import backend as K
import cv2
#top_model_weights_path = 'bottleneck_fc_model.h5'       # final weight
s of the model
#train_data_dir = 'train_images_model'
testfile = '/content/sample_data/test_set'
subfile = 'finalresnet.csv'                                 # prediction
s of the model
def predict(image_path):
    print ('starting')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)

    img_height,img_width = 96,96
    num_classes = 6000
    base_model = applications.resnet50.ResNet50(weights= 'imagenet', in
clude_top=False, input_shape= (img_height,img_width,3))
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dropout(0.7)(x)
    predictions = Dense(num_classes, activation= 'softmax')(x)
    model = Model(inputs = base_model.input, outputs = predictions)
    model.load_weights("/content/sample_data/resNet_weights.h5")

    label_map={}
    for i in range(6000):
      label_map[str(1000+i)]=i
    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks','confidence'])
        file_counter = 0
        for root, dirs, files in os.walk(image_path):  # loop through s
tartfolders
            i=0
            for pic in files:
                i+=1
                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                # important! otherwise the predictions will be '0'
                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark


                prediction = model.predict(image)
```

```
                class_predicted = prediction.argmax(axis=1)
                #class_predicted = np.argmax(prediction,axis=1)
                #print (pic, class_predicted)


                inID = class_predicted[0]
                #print inID

                inv_map = {v: k for k, v in label_map.items()}
                #print class_dictionary
                label = inv_map[inID]
                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                #out = str(label) + ' '+ scor
                print (i,score,label)
                newFileWriter.writerow([os.path.splitext(pic)[0], int(l
abel),scor])

predict(testfile)
```

In [ ]:
```
test_accuracy("/content/sample_data/dataset3/finalresnet.csv","/content
/sample_data/dataset3/test_sample3.csv")
```

```
0.36458253615979186
```

Out[ ]:
```
0.36458253615979186
```

In [52]:
```
GAP_metric("finalresnet.csv")
```

```
5305 2025.234982992889 13067 3176
0.38175965749159074
```

Out[52]:
```
0.38175965749159074
```

Approach 4

In [ ]:
```
# This is where we fine tune the pretrained model according to our data
set
img_width, img_height = 96, 96
save_model_weights = "VGG19_weights.h5"
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 50
def train_VGG19():

    # build the Resnet50 network

    base_model = applications.VGG19(weights='imagenet',include_top= Fal
se,input_shape=(96,96,3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    #top_model.add(Dense(1024, activation='relu'))
    top_model.add(Dense(1024, activation='relu'))
    n_class = 6000
    top_model.add(Dense(n_class, activation='softmax'))

    model = Model(base_model.input, top_model(base_model.output))
    model.load_weights("/content/sample_data/dataset/dataset3/VGG19_wei
ghts.h5")
    # set the first 16 layers to non-trainable (weights will not be upd
ated)
    # 1 conv layer and three dense layers will be trained
    for layer in model.layers[:19]:
        layer.trainable = False
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(lr=0.001, beta_1=0.9,beta_2
=0.999,epsilon=1e-8, decay=0.0),
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    #np.save('class_indices.npy', train_generator.class_indices)

    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
```

```
        class_mode='categorical')

    print ('Model fit begins...')
    history1=model.fit_generator(
        train_generator,
        steps_per_epoch=150,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=75,
        callbacks=[ModelCheckpoint(filepath="/content/sample_data/datas
et/dataset3/VGG19_weights.h5",
                                   save_best_only=True, save_weights_on
ly=True)]
        )

    model.save_weights(save_model_weights)
    # final weights are saved in bottleneck_fc_model.h5 file
    return history1
#history1=train_VGG19()
```

In [ ]:
```
# This is where we fine tune the pretrained model according to our data set
img_width, img_height = 96, 96
save_model_weights = "VGG19_weights.h5"
train_data_dir = 'train_images_model'
validation_data_dir = 'validation_images_model'
batch_size = 200
epochs = 50
def train_VGG19():

    base_model = applications.VGG19(weights='imagenet',include_top= False,input_shape=(96,96,3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    top_model.add(Dense(1024, activation='relu'))
    n_class = 6000
    top_model.add(Dense(n_class, activation='softmax'))

    model = Model(base_model.input, top_model(base_model.output))
    # set the first 16 layers to non-trainable (weights will not be updated)
    # 1 conv layer and three dense layers will be trained
    #for layer in model.layers[:19]:
        #layer.trainable = False

    model.compile(loss='categorical_crossentropy',
                  optimizer="adam",
                  metrics=['accuracy'])
    print ('Compilation done.')

    train_datagen = ImageDataGenerator(rescale=1. / 255,
                                       rotation_range=90,
                                        width_shift_range=0.2,
                                        height_shift_range=0.2,
                                        zoom_range = 0.5)

    valid_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    #np.save('class_indices.npy', train_generator.class_indices)

    validation_generator = valid_datagen.flow_from_directory(
        validation_data_dir,
        target_size=(img_height, img_width),
        batch_size=batch_size,
        class_mode='categorical')

    print ('Model fit begins...')
    history=model.fit_generator(
        train_generator,
```

```
        steps_per_epoch=150,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=75,
        callbacks=[ModelCheckpoint(filepath="vgg19_weights_tf_dim_order
ing_tf_kernels_notop.h5",
                                   save_best_only=True, save_weights_on
ly=True)]
        )

    model.save_weights(save_model_weights)
    # final weights are saved in bottleneck_fc_model.h5 file

train_VGG19()
```

```
Compilation done.
Found 246794 images belonging to 6000 classes.
Found 54916 images belonging to 6000 classes.
Model fit begins...
```

```
/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/50
  2/150 [..............................] - ETA: 23s - loss: 8.6620 -
accuracy: 0.0025   WARNING:tensorflow:Callbacks method `on_train_batc
h_end` is slow compared to the batch time (batch time: 0.1096s vs `on
_train_batch_end` time: 0.1996s). Check your callbacks.
150/150 [==============================] - 135s 901ms/step - loss: 7.
6355 - accuracy: 0.0070 - val_loss: 7.3191 - val_accuracy: 0.0073
Epoch 2/50
150/150 [==============================] - 136s 906ms/step - loss: 7.
2322 - accuracy: 0.0079 - val_loss: 7.3097 - val_accuracy: 0.0081
Epoch 3/50
150/150 [==============================] - 137s 911ms/step - loss: 7.
1940 - accuracy: 0.0106 - val_loss: 7.2201 - val_accuracy: 0.0066
Epoch 4/50
150/150 [==============================] - 136s 906ms/step - loss: 7.
0848 - accuracy: 0.0142 - val_loss: 7.1933 - val_accuracy: 0.0107
Epoch 5/50
150/150 [==============================] - 136s 904ms/step - loss: 7.
0080 - accuracy: 0.0148 - val_loss: 7.0558 - val_accuracy: 0.0122
Epoch 6/50
150/150 [==============================] - 135s 902ms/step - loss: 6.
9318 - accuracy: 0.0169 - val_loss: 6.9810 - val_accuracy: 0.0111
Epoch 7/50
150/150 [==============================] - 137s 913ms/step - loss: 6.
8661 - accuracy: 0.0193 - val_loss: 6.9431 - val_accuracy: 0.0170
Epoch 8/50
150/150 [==============================] - 136s 907ms/step - loss: 6.
8365 - accuracy: 0.0223 - val_loss: 6.8802 - val_accuracy: 0.0164
Epoch 9/50
150/150 [==============================] - 136s 910ms/step - loss: 6.
7540 - accuracy: 0.0239 - val_loss: 6.7766 - val_accuracy: 0.0242
Epoch 10/50
150/150 [==============================] - 139s 929ms/step - loss: 6.
7297 - accuracy: 0.0253 - val_loss: 6.6865 - val_accuracy: 0.0193
Epoch 11/50
150/150 [==============================] - 138s 919ms/step - loss: 6.
6405 - accuracy: 0.0285 - val_loss: 6.6925 - val_accuracy: 0.0233
Epoch 12/50
150/150 [==============================] - 137s 911ms/step - loss: 6.
5424 - accuracy: 0.0311 - val_loss: 6.6038 - val_accuracy: 0.0240
Epoch 13/50
150/150 [==============================] - 136s 909ms/step - loss: 6.
4881 - accuracy: 0.0338 - val_loss: 6.5527 - val_accuracy: 0.0299
Epoch 14/50
150/150 [==============================] - 136s 909ms/step - loss: 6.
3866 - accuracy: 0.0390 - val_loss: 6.3771 - val_accuracy: 0.0356
Epoch 15/50
150/150 [==============================] - 135s 901ms/step - loss: 6.
3309 - accuracy: 0.0425 - val_loss: 6.3268 - val_accuracy: 0.0409
Epoch 16/50
150/150 [==============================] - 134s 896ms/step - loss: 6.
2574 - accuracy: 0.0452 - val_loss: 6.4683 - val_accuracy: 0.0349
Epoch 17/50
150/150 [==============================] - 135s 898ms/step - loss: 6.
1980 - accuracy: 0.0493 - val_loss: 6.3155 - val_accuracy: 0.0429
Epoch 18/50
```

```
150/150 [==============================] - 135s 902ms/step - loss: 6.
1131 - accuracy: 0.0529 - val_loss: 6.1765 - val_accuracy: 0.0487
Epoch 19/50
150/150 [==============================] - 135s 899ms/step - loss: 6.
0668 - accuracy: 0.0557 - val_loss: 6.0082 - val_accuracy: 0.0587
Epoch 20/50
150/150 [==============================] - 133s 888ms/step - loss: 5.
9786 - accuracy: 0.0613 - val_loss: 5.8976 - val_accuracy: 0.0652
Epoch 21/50
150/150 [==============================] - 133s 887ms/step - loss: 5.
9341 - accuracy: 0.0662 - val_loss: 5.9427 - val_accuracy: 0.0669
Epoch 22/50
150/150 [==============================] - 133s 885ms/step - loss: 5.
8453 - accuracy: 0.0713 - val_loss: 5.8477 - val_accuracy: 0.0699
Epoch 23/50
150/150 [==============================] - 134s 892ms/step - loss: 5.
7747 - accuracy: 0.0775 - val_loss: 5.7931 - val_accuracy: 0.0797
Epoch 24/50
150/150 [==============================] - 133s 885ms/step - loss: 5.
7276 - accuracy: 0.0810 - val_loss: 5.7685 - val_accuracy: 0.0766
Epoch 25/50
150/150 [==============================] - 133s 890ms/step - loss: 5.
6220 - accuracy: 0.0877 - val_loss: 5.6520 - val_accuracy: 0.0874
Epoch 26/50
150/150 [==============================] - 133s 885ms/step - loss: 5.
5747 - accuracy: 0.0909 - val_loss: 5.6348 - val_accuracy: 0.0919
Epoch 27/50
150/150 [==============================] - 133s 888ms/step - loss: 5.
4889 - accuracy: 0.0982 - val_loss: 5.4013 - val_accuracy: 0.1031
Epoch 28/50
150/150 [==============================] - 134s 896ms/step - loss: 5.
4394 - accuracy: 0.1019 - val_loss: 5.3602 - val_accuracy: 0.1160
Epoch 29/50
150/150 [==============================] - 136s 904ms/step - loss: 5.
3552 - accuracy: 0.1125 - val_loss: 5.3497 - val_accuracy: 0.1171
Epoch 30/50
150/150 [==============================] - 136s 906ms/step - loss: 5.
3146 - accuracy: 0.1129 - val_loss: 5.2257 - val_accuracy: 0.1299
Epoch 31/50
150/150 [==============================] - 135s 903ms/step - loss: 5.
2299 - accuracy: 0.1214 - val_loss: 5.1839 - val_accuracy: 0.1360
Epoch 32/50
150/150 [==============================] - 135s 898ms/step - loss: 5.
1933 - accuracy: 0.1254 - val_loss: 5.1353 - val_accuracy: 0.1402
Epoch 33/50
150/150 [==============================] - 136s 906ms/step - loss: 5.
1207 - accuracy: 0.1312 - val_loss: 5.1437 - val_accuracy: 0.1403
Epoch 34/50
150/150 [==============================] - 136s 905ms/step - loss: 5.
0656 - accuracy: 0.1386 - val_loss: 4.9735 - val_accuracy: 0.1548
Epoch 35/50
150/150 [==============================] - 137s 916ms/step - loss: 5.
```

In [ ]:
```
history1=train_VGG19()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras
-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80142336/80134624 [==============================] - 3s 0us/step
Compilation done.
Found 246794 images belonging to 6000 classes.
Found 54916 images belonging to 6000 classes.
Model fit begins...
WARNING:tensorflow:From <ipython-input-10-36232a1718ee>:61: Model.fit
_generator (from tensorflow.python.keras.engine.training) is deprecat
ed and will be removed in a future version.
Instructions for updating:
Please use Model.fit, which supports generators.

/usr/local/lib/python3.6/dist-packages/PIL/Image.py:932: UserWarning:
Palette images with Transparency expressed in bytes should be convert
ed to RGBA images
  "Palette images with Transparency expressed in bytes should be "
```

```
Epoch 1/50
150/150 [==============================] - 168s 1s/step - loss: 4.324
7 - accuracy: 0.2232 - val_loss: 4.3614 - val_accuracy: 0.2536
Epoch 2/50
150/150 [==============================] - 165s 1s/step - loss: 4.355
1 - accuracy: 0.2267 - val_loss: 4.3898 - val_accuracy: 0.2499
Epoch 3/50
150/150 [==============================] - 161s 1s/step - loss: 4.322
7 - accuracy: 0.2280 - val_loss: 4.3827 - val_accuracy: 0.2486
Epoch 4/50
150/150 [==============================] - 159s 1s/step - loss: 4.310
5 - accuracy: 0.2300 - val_loss: 4.3039 - val_accuracy: 0.2622
Epoch 5/50
150/150 [==============================] - 155s 1s/step - loss: 4.315
0 - accuracy: 0.2278 - val_loss: 4.3515 - val_accuracy: 0.2594
Epoch 6/50
150/150 [==============================] - 158s 1s/step - loss: 4.294
5 - accuracy: 0.2338 - val_loss: 4.4560 - val_accuracy: 0.2574
Epoch 7/50
150/150 [==============================] - 153s 1s/step - loss: 4.272
7 - accuracy: 0.2311 - val_loss: 4.3346 - val_accuracy: 0.2649
Epoch 8/50
150/150 [==============================] - 153s 1s/step - loss: 4.252
9 - accuracy: 0.2370 - val_loss: 4.3187 - val_accuracy: 0.2658
Epoch 9/50
150/150 [==============================] - 152s 1s/step - loss: 4.204
3 - accuracy: 0.2414 - val_loss: 4.3300 - val_accuracy: 0.2657
Epoch 10/50
150/150 [==============================] - 151s 1s/step - loss: 4.223
7 - accuracy: 0.2358 - val_loss: 4.2773 - val_accuracy: 0.2771
Epoch 11/50
150/150 [==============================] - 148s 989ms/step - loss: 4.
2017 - accuracy: 0.2366 - val_loss: 4.3536 - val_accuracy: 0.2705
Epoch 12/50
150/150 [==============================] - 149s 992ms/step - loss: 4.
1664 - accuracy: 0.2436 - val_loss: 4.2500 - val_accuracy: 0.2763
Epoch 13/50
150/150 [==============================] - 149s 994ms/step - loss: 4.
1161 - accuracy: 0.2474 - val_loss: 4.2715 - val_accuracy: 0.2791
Epoch 14/50
150/150 [==============================] - 148s 988ms/step - loss: 4.
1468 - accuracy: 0.2443 - val_loss: 4.2244 - val_accuracy: 0.2718
Epoch 15/50
150/150 [==============================] - 150s 997ms/step - loss: 4.
1408 - accuracy: 0.2417 - val_loss: 4.3501 - val_accuracy: 0.2750
Epoch 16/50
150/150 [==============================] - 148s 985ms/step - loss: 4.
1249 - accuracy: 0.2449 - val_loss: 4.2765 - val_accuracy: 0.2767
Epoch 17/50
150/150 [==============================] - 149s 995ms/step - loss: 4.
1284 - accuracy: 0.2476 - val_loss: 4.1974 - val_accuracy: 0.2805
Epoch 18/50
150/150 [==============================] - 150s 1s/step - loss: 4.098
4 - accuracy: 0.2517 - val_loss: 4.2331 - val_accuracy: 0.2855
Epoch 19/50
150/150 [==============================] - 150s 997ms/step - loss: 4.
```

```
0700 - accuracy: 0.2485 - val_loss: 4.2437 - val_accuracy: 0.2821
Epoch 20/50
150/150 [==============================] - 148s 989ms/step - loss: 4.
0976 - accuracy: 0.2441 - val_loss: 4.2028 - val_accuracy: 0.2842
Epoch 21/50
150/150 [==============================] - 148s 988ms/step - loss: 4.
0724 - accuracy: 0.2503 - val_loss: 4.2207 - val_accuracy: 0.2893
Epoch 22/50
150/150 [==============================] - 148s 986ms/step - loss: 4.
0676 - accuracy: 0.2516 - val_loss: 4.2042 - val_accuracy: 0.2865
Epoch 23/50
150/150 [==============================] - 147s 977ms/step - loss: 4.
0636 - accuracy: 0.2511 - val_loss: 4.1884 - val_accuracy: 0.2867
Epoch 24/50
150/150 [==============================] - 146s 972ms/step - loss: 4.
0412 - accuracy: 0.2523 - val_loss: 4.1754 - val_accuracy: 0.2923
Epoch 25/50
150/150 [==============================] - 147s 978ms/step - loss: 4.
0134 - accuracy: 0.2541 - val_loss: 4.2471 - val_accuracy: 0.2871
Epoch 26/50
150/150 [==============================] - 148s 984ms/step - loss: 4.
0222 - accuracy: 0.2563 - val_loss: 4.1647 - val_accuracy: 0.2949
Epoch 27/50
150/150 [==============================] - 149s 991ms/step - loss: 4.
0123 - accuracy: 0.2552 - val_loss: 4.2309 - val_accuracy: 0.2845
Epoch 28/50
150/150 [==============================] - 145s 966ms/step - loss: 4.
0298 - accuracy: 0.2538 - val_loss: 4.1833 - val_accuracy: 0.2952
Epoch 29/50
150/150 [==============================] - 146s 973ms/step - loss: 4.
0110 - accuracy: 0.2547 - val_loss: 4.2303 - val_accuracy: 0.2861
Epoch 30/50
150/150 [==============================] - 147s 979ms/step - loss: 4.
0129 - accuracy: 0.2519 - val_loss: 4.2476 - val_accuracy: 0.2929
Epoch 31/50
150/150 [==============================] - 148s 986ms/step - loss: 3.
9622 - accuracy: 0.2603 - val_loss: 4.1767 - val_accuracy: 0.2906
Epoch 32/50
150/150 [==============================] - 147s 980ms/step - loss: 3.
9781 - accuracy: 0.2600 - val_loss: 4.2052 - val_accuracy: 0.2877
Epoch 33/50
150/150 [==============================] - 146s 976ms/step - loss: 3.
9807 - accuracy: 0.2601 - val_loss: 4.1705 - val_accuracy: 0.2981
Epoch 34/50
150/150 [==============================] - 145s 970ms/step - loss: 3.
9524 - accuracy: 0.2638 - val_loss: 4.2247 - val_accuracy: 0.3037
Epoch 35/50
150/150 [==============================] - 148s 984ms/step - loss: 3.
9467 - accuracy: 0.2659 - val_loss: 4.1448 - val_accuracy: 0.3019
Epoch 36/50
150/150 [==============================] - 147s 980ms/step - loss: 3.
9619 - accuracy: 0.2629 - val_loss: 4.1406 - val_accuracy: 0.3039
Epoch 37/50
150/150 [==============================] - 149s 991ms/step - loss: 3.
9583 - accuracy: 0.2633 - val_loss: 4.1300 - val_accuracy: 0.2945
Epoch 38/50
```

In [ ]:
```
from keras import backend as K
import cv2
#top_model_weights_path = 'bottleneck_fc_model.h5'       # final weight
s of the model
train_data_dir = 'train_images_model'
testfile = '/content/sample_data/test_set'
subfile = 'resultvgg199.csv'                                  # predictio
ns of the model
def predict(image_path):
    print ('starting')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)

    img_height,img_width = 96,96
    num_classes = 6000
    base_model = applications.VGG19(weights='imagenet',include_top= Fal
se,input_shape=(96,96,3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model.output_shape[1:]))
    #top_model.add(Dense(1024, activation='relu'))
    top_model.add(Dense(1024, activation='relu'))
    n_class = 6000
    top_model.add(Dense(n_class, activation='softmax'))

    model = Model(base_model.input, top_model(base_model.output))
    model.load_weights("/content/VGG19_weights.h5")

    label_map={}
    for i in range(6000):
      label_map[str(1000+i)]=i
    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks','confidence'])
        file_counter = 0
        for root, dirs, files in os.walk(image_path):  # loop through s
tartfolders
            i=0
            for pic in files:
                i+=1
                #loop folder and convert image
                path = image_path + '/' + pic


                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)

                # important! otherwise the predictions will be '0'
                image = image / 255

                image = np.expand_dims(image, axis=0)

                #classify landmark
```

```
                prediction = model.predict(image)


                class_predicted = prediction.argmax(axis=1)
                #class_predicted = np.argmax(prediction,axis=1)
                #print (pic, class_predicted)


                inID = class_predicted[0]
                #print inID

                inv_map = {v: k for k, v in label_map.items()}
                #print class_dictionary
                label = inv_map[inID]
                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                #out = str(label) + ' '+ scor
                print (i,score,label)
                newFileWriter.writerow([os.path.splitext(pic)[0], int(l
abel),scor])

predict(testfile)
```

In [ ]:
```
testfile = '/content/sample_data/test_set'
subfile = 'finalensemble.csv'                                    # predicti
ons of the model
def predict(image_path):
    print ('starting')
    path, dirs, files = next(os.walk(image_path))
    file_len = len(files)
    print('Number of Testimages:', file_len)


    img_height,img_width = 96,96
    num_classes = 6000

    base_model = applications.resnet50.ResNet50(weights= 'imagenet', in
clude_top=False, input_shape= (img_height,img_width,3))
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dropout(0.7)(x)
    predictions = Dense(num_classes, activation= 'softmax')(x)
    model = Model(inputs = base_model.input, outputs = predictions)
    model.load_weights("/content/resNet_weights.h5")


    base_model2 = applications.VGG19(weights='imagenet',include_top= Fa
lse,input_shape=(96,96,3))
    top_model = Sequential()
    top_model.add(Flatten(input_shape=base_model2.output_shape[1:]))
    top_model.add(Dense(1024, activation='relu'))
    top_model.add(Dense(num_classes, activation='softmax'))
    model2 = Model(base_model2.input, top_model(base_model2.output))
    model2.load_weights("/content/VGG19_weights.h5")


    base_model3 = applications.VGG16(weights='imagenet',include_top= Fa
lse,input_shape=(96,96,3))
    top_model2 = Sequential()
    top_model2.add(Flatten(input_shape=base_model3.output_shape[1:]))
    top_model2.add(Dense(256, activation='relu'))
    top_model2.add(Dense(256, activation='relu'))
    top_model2.add(Dense(num_classes, activation='softmax'))
    model3 = Model(base_model3.input, top_model2(base_model3.output))
    model3.load_weights("/content/drive/My Drive/ADM/VGG16_weights.h5")

    label_map={}
    for i in range(6000):
      label_map[str(1000+i)]=i
    with open(subfile, 'w') as csvfile:
        newFileWriter = csv.writer(csvfile)
        newFileWriter.writerow(['id', 'landmarks','confidence'])
        file_counter = 0
        for root, dirs, files in os.walk(image_path):  # loop through s
tartfolders
            i=0
            for pic in files:
                i+=1
```

```
                path = image_path + '/' + pic
                orig = cv2.imread(path)
                image = load_img(path, target_size=(96, 96))
                image = img_to_array(image)
                image = image / 255
                image = np.expand_dims(image, axis=0)
                #classify landmark

                prediction1 = model.predict(image)
                prediction2 = model2.predict(image)
                prediction3 = model3.predict(image)

                prediction = np.average(a=[prediction1,prediction2,pred
iction3],axis=0)
                class_predicted = prediction.argmax(axis=1)
                inID = class_predicted[0]
                inv_map = {v: k for k, v in label_map.items()}
                label = inv_map[inID]
                score = max(prediction[0])
                scor = "{:.2f}".format(score)
                print (i,score,label)
                newFileWriter.writerow([os.path.splitext(pic)[0], int(l
abel),scor])

predict(testfile)
```

In [ ]:
```
import pandas as pd
def test_accuracy(pred,truth):
    result=pd.read_csv(pred)
    #print(result.head(20))
    test=pd.read_csv(truth)
    count=0
    for i in result["id"]:
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #b=list(b[0].split(" "))
        b[0]=int(b[0])
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              count+=1
    print(count/len(result))
    return count/len(result)
test_accuracy("/content/resultvgg19.csv","/content/drive/My Drive/ADM/t
est_sample3.csv")
```

```
0.31906614785992216
```

Out[ ]:
```
0.31906614785992216
```

In [ ]:
```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("/content/drive/My Drive/ADM/test_sample3.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
GAP_metric("/content/VGG19_result_6000.csv")
```

```
5305 1693.6816389038179 13067 2595
0.3192613833937451
```

Out[ ]:
```
0.3192613833937451
```

Ensemble VGG16,VGG19,ResNet50

In [ ]:
```
import pandas as pd
def test_accuracy(pred,truth):
    result=pd.read_csv(pred)
    #print(result.head(20))
    test=pd.read_csv(truth)
    count=0
    for i in result["id"]:
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #b=list(b[0].split(" "))
        b[0]=int(b[0])
        if i in test.id.values:
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              count+=1
    print(count/len(result))
    return count/len(result)
test_accuracy("/content/resultEnsemble.csv","/content/drive/My Drive/ADM/test_sample3.csv")
```

```
0.37659753577714855
```

Out[ ]:
```
0.37659753577714855
```

In [53]:
```
GAP_metric("finalensemble.csv")
```

```
5305 2205.1114568364956 13067 4921
0.41566662711338276
```

Out[53]:
```
0.41566662711338276
```

In [ ]:
```

```

In [2]:
```
from absl import logging

import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageOps
from scipy.spatial import cKDTree
from skimage.feature import plot_matches
from skimage.measure import ransac
from skimage.transform import AffineTransform
from six import BytesIO

import tensorflow as tf

import tensorflow_hub as hub
from six.moves.urllib.request import urlopen
```

In [3]:
```
delf = hub.load('https://tfhub.dev/google/delf/1').signatures['default
']
```

In [4]:
```
def delf_work(path1, path2):
  image1 = Image.open(path1)
  image1 = ImageOps.fit(image1, (96, 96), Image.ANTIALIAS)
  image2 = Image.open(path2)
  image2 = ImageOps.fit(image2, (96, 96), Image.ANTIALIAS)
  result1 = run_delf(image1)
  result2 = run_delf(image2)

  if (type(result1) is int or type(result2) is int) and (result1==0 or
result2 == 0):
    return 0

  distance_threshold = 0.8

  # Read features.
  num_features_1 = result1['locations'].shape[0]
  #print("Loaded image 1's %d features" % num_features_1)

  num_features_2 = result2['locations'].shape[0]
  #print("Loaded image 2's %d features" % num_features_2)

  # Find nearest-neighbor matches using a KD tree.
  d1_tree = cKDTree(result1['descriptors'])
  _, indices = d1_tree.query(
      result2['descriptors'],
      distance_upper_bound=distance_threshold)

  # Select feature locations for putative matches.
  locations_2_to_use = np.array([
      result2['locations'][i,]
      for i in range(num_features_2)
      if indices[i] != num_features_1
  ])
  locations_1_to_use = np.array([
      result1['locations'][indices[i],]
      for i in range(num_features_2)
      if indices[i] != num_features_1
  ])

  # Perform geometric verification using RANSAC.
  try:
    _, inliers = ransac(
        (locations_1_to_use, locations_2_to_use),
        AffineTransform,
        min_samples=3,
        residual_threshold=20,
        max_trials=1000)
  except:
    inliers = [0]

  if inliers is None:
    print('Found 0 inliers')
    return 0
  print('Found %d inliers' % sum(inliers))
  return sum(inliers)
```

In [5]:
```
def local_feature(path):
  x = 0
  v=[]
  result = pd.read_csv(path)
  for i in result['id']:
    a = result.loc[result['id'] == i, "landmarks"]
    c = result.loc[result['id'] == i, "confidence"]
    a = a.values
    c = c.values
    #print(a,c)
    if c<0.4:
      p2 = '/content/sample_data/dataset/dataset3/train_images_model/'+
str(a[0])
      r = os.listdir(p2)

      if(len(r)>2):
        s=0
        for j in range(2):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s


      if avg < 3:
        result.loc[result.id == i, "confidence"] = 0
      elif avg >= 10:
        result.loc[result.id == i, "confidence"] = 0.6
    elif c>0.4 and c<0.6:
      p2 = '/content/sample_data/dataset/dataset3/train_images_model/'+
str(a[0])
      r = os.listdir(p2)

      if(len(r)>2):
        s=0
        for j in range(2):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
    elif c==1:
      p2 = '/content/sample_data/dataset/dataset3/train_images_model/'+
str(a[0])
      r = os.listdir(p2)
```

```
      if(len(r)>5):
        s=0
        for j in range(5):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s
      else:
        s=0
        for j in range(len(r)):
          s = max(s,delf_work('/content/sample_data/test_set/'+str
(i)+'.jpg', p2+'/'+str(r[j]) ))
        avg = s

      if avg < 3:
        result.loc[result.id == i, "confidence"] = 0
    #v.append([x,avg,c])
    print(x)
    x+=1



  result.to_csv('delf_test_best.csv')
  #return v
  #print(GAP_metric_2(result))
```

In [6]:
```
# This call calulates the Global Average Precision Metric
#
import pandas as pd
def GAP_metric(path):
    result=pd.read_csv(path)
    test=pd.read_csv("/content/drive/My Drive/ADM/test_sample3.csv")
    result=result.sort_values(by='confidence', ascending=False)
    M=0
    true=0
    total=0
    s=0
    relevance=0
    for i in result['id']:
        total+=1
        a=result.loc[result['id'] == i, "landmarks"]
        b=a.values
        #if i in test['id']:
        if i in test.id.values:
          M+=1
          c=test.loc[test['id'] == i, "landmark_id"]
          d=c.values
          if b[0]==d[0]:
              true+=1
              relevance=1
        s+=(true/total)*relevance
        relevance=0
    print(M,s,total,true)
    print(s/M)
    return s/M
GAP_metric("/content/delf_test_best.csv")
```

```
5305 2179.42304373032 13067 4921
0.4108243249256022
```

Out[6]:
```
0.4108243249256022
```

In [ ]:
```
```

Google landmark recognition

C) READ ME.

Code Explanation: We have submitted 7 ipynb files. Each file is explained below:

dataset.ipynb:
The data is available as .csv files in which the images are given as the URL links. So we have to download the images from these links and convert them into required smaller sizes to compatible for computation and later we have to pre-processing these images into folders of train and validation folders named with their respective class labels. All this is done in a separate .ipnby file named as dataset.ipynb.

Approach_1_on_1000_to_3000_VGG16.ipynb:
In this we implemented VGG16 on a dataset of images belonging to class labels of 1000 to 2999. This is the first approach we explained in the test 1.

Approach_1_on_3000_to_5000_VGG16.ipynb:
In this we implemented VGG16 on a dataset of images belonging to class labels of 3000 to 4999. This is done as a part of modification of test 1. As we have to make different samples to run the experiment and average the score we got.

Approach_1_on_5000_to_7000_VGG16.ipynb:
In this we implemented VGG16 on a dataset of images belonging to class labels of 5000 to 6999. This is done as a part of modification of test 1. As we have to make different samples to run the experiment and average the score we got.

Approach_1_on_6000_VGG16.ipynb:
In this we implemented VGG16 on a dataset of images belonging to class labels of 1000 to 6999. This is done as a part of modification of test 1. As we have to make different samples to run the experiment and average the score we got. This is the bigger and final dataset used for all the remaining experiments.

Approach_2_on_6000_VGG16_DELF.ipynb:
In this we implemented DELF and a novel algorithm to re-compute the confidence scores of the predictions. We have used DELF hub which is available in the tensorflow_hub module. All the libraries required are available in the file itself.

Approach_3_on_6000_RESNET_and_Approach_4_Ensemble_of_VGG16_VGG19_RESNET.ipynb:
In this we implemented RESNET50 for images classification on the final dataset this is our Approach 3 we discussed in the test 1.5. In approach 4 we have created vgg19 as well as one more CNN backbone model to our ensemble model along with vgg16 we trained in the approach1. All these models are ensemble and an average layers are added over them and the test images are predicted over them.

Approach_5_on_6000_ENSEMBLE_DELF.pynb:
In this we implemented approach 4 model along with local features using the ideas of approach 2 i.e., DELF with Ensemble.

Google landmark recognition

Environment and Execution Steps:

1) First we need to create the train and validation images from the given url links.
2) Open the dataset.ipnby in the local machine in the jupyter notebook and change the working directory to the directory in which the train.csv and test.csv are available.
3) Execute the dataset.ipnby file to resize the images to 96x96 sizes, download these images to local machine.
4) After downloading the images create train_images_model, validation_images_model, test_images_from_train folders in the current working directory and continue executing the data Pre-processing module in the dataset.ipnby. All these images are sent to the respective class labels in the train and validation folder. The last cells of this file have a function named cleaning. This function removes the images which are corrupted so that this resolves the error in the later part of the execution.
5) This also created train_sample.csv, validation_sample.csv, test_sample.csv which later used to calculate the performance of the models.
6) This entire directory where the entire folders we created need to be zipped and uploaded into the Google drive to access these in the colab notebooks.
7) Upload all remaining .ipnby files into to the colab notebooks and start running notebooks one by one. The colab should be linked with drive of the respective user to access these zip files. While unzipping the dataset from the drive make sure to change the directory to your drive folder where you upload the dataset zip folder.