Capstone Project

Machine Learning Engineer Nanodegree
Author - Sagarnil Das

Definition

Project Overview

Vision loss is a problem that affects people all around the world. As computers are getting better at understanding images due to advances in computer vision, the concept of a virtual assistant for the blind that could read text[1], identify/spot objects[2], or even describe a whole scene in natural language[3] is becoming increasingly realistic. In this project I created an application which can detect objects via webcam in real-time and state their name.

The application uses a classifier pre-trained on the MS-COCO[4] dataset. It comprises 90 classes of images. Besides that, I have also successfully trained my own classifier to detect custom objects.

Problem Statement

The goal is to create a real time object detection application using Tensorflow's object detection API. The steps are as follows.

- Download the Tensorflow object detection API[5].
- Currently, the code is written so that it only detects objects in images. So the next step would be to modify the code so that it uses the webcam to detect objects in real time using openCV. I will use a pre trained SSD MobileNet[6] model for this purpose on the MS-COCO dataset.
- The next target is to make a completely new classifier and train it on a custom object. After that it should detect that custom object in real time.
- Implement the pyttsx module in python to make an artificially generated voice speak out loud the name of the object.

Metrics

Accuracy is a very common metric for both binary and multiclass classifiers. Here it is a multiclass classification problem as we are dealing with many classes of images. The accuracy takes into account both true positives and true negatives with equal weights. Another evaluation metrics is the validation and training loss. The lower the loss is the better. Loss is not in percentage as opposed to accuracy and it is a summation of the errors made for each example in training or validation sets. In the case of neural networks the loss is usually negative log-likelihood in case of classification problem. For accuracy, the formula is:

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

Processing delay is also a metric that can affect the user experience largely. It can be broken down into two components, as the image processing is done in two steps:

$$processing\ delay \approx classification\ delay + extraction\ delay$$

The classification delay is the time taken by the classifier to detect objects. It is important by itself as the application provides feedback to the user immediately after it detects objects.

The extraction delay is the time taken by the application to start speaking after the object is detected and classified by the classifier.

Analysis

Data Exploration

The COCO (Common Objects in Context) dataset has hundreds of thousands of richly annotated images; the annotations are not described here because only the images are used. More accurately, only a subset of the images are used, the 2014 training images, as the COCO-Text 1.0 dataset has annotations only for that particular subset. Of the 82,783 images, 63,686 contain text; altogether there are 173,589 text instances, which is more than enough to train the classifier. The images are colored and have around 600 * 400 pixels each. This is the dataset, the model is pre trained on. Afterwards, when I build my own custom classifier, I will scrap images of Mickey Mouse from the web and train an object detection model on those images.

Fig.1: Sample images from MS-COCO dataset

Fig.2: Sample images of Mickey Mouse

Exploratory Visualization:

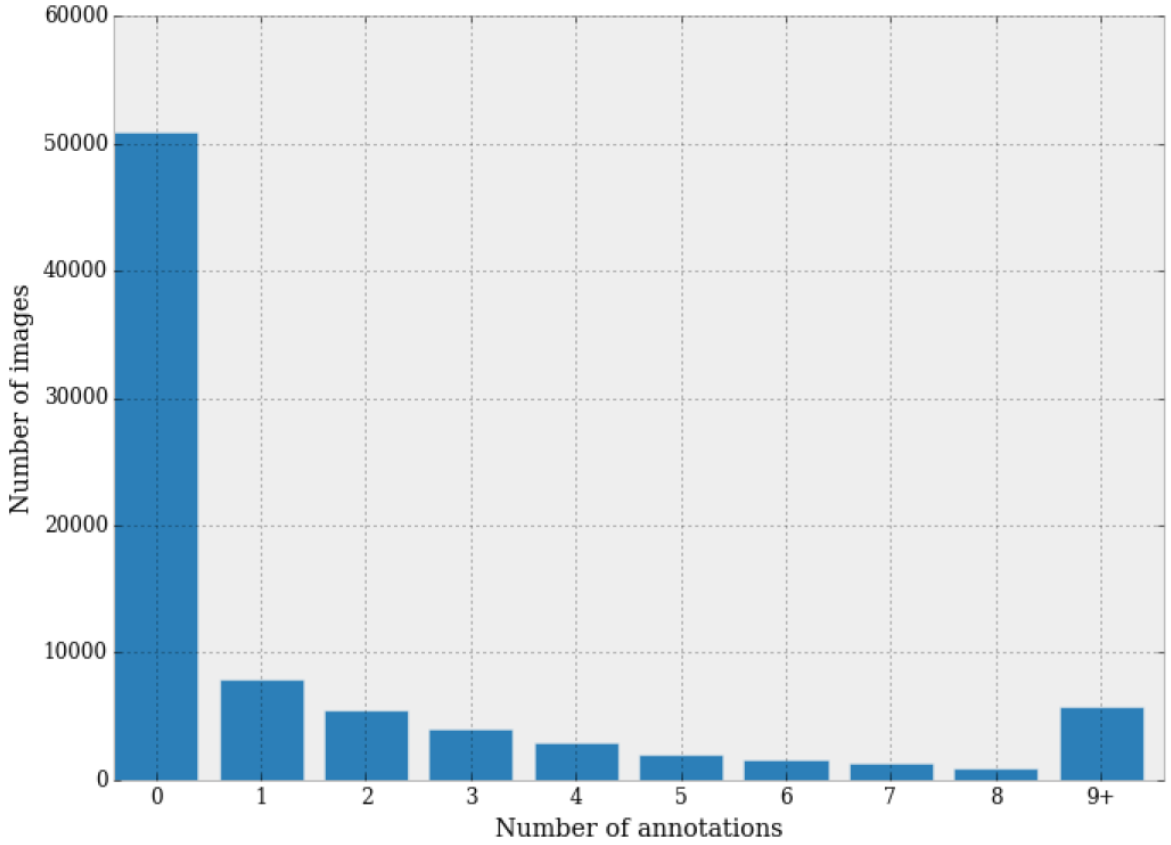


Fig.2

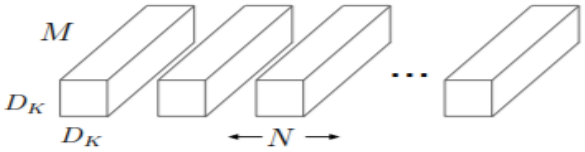
Fig 2 shows us the distribution of the COCO-data. Of course for the Mickey Mouse images, number of classes=1. So there's no question of distribution even though this was the more challenging task. Here, the distribution is right-tailed. (The 9+ column is large only because it contains the sum of all of the other columns which were cut off from the plot.) If the distribution was left-tailed, that would suggest that the annotations are mostly assigned to a few of the images, which could result in overfitting.

Algorithms and Techniques

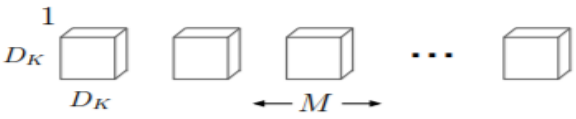
The algorithm used in this project is a Convolutional Neural Network, which is the state of the art classifier for most image processing tasks, including object detection and recognition. Convolutional neural networks have become irreplaceable in the computer vision field ever since AlexNet popularized the deep convolutional neural network by winning the ImageNet challenge. The advent of neural networks since that time has been tremendous. Huge research has been done to achieve higher accuracy with Convolutional Neural Networks by making deeper and more complicated networks. However, with increase in accuracy of these more complex networks, the major trade off is the application performance with respect to size and speed. This is where, one of the newer models named 'MobileNets' has become hugely successful with respect to speed with slight tradeoff in accuracy which is a very important factor in modern industries like Robotics and Artificial Intelligence.

The Mobilenet model is based on depth wise separable convolutions which are a form of factorized convolutions which factorize a standard convolution into a depth wise convolution and a 1×1 convolution called a point wise convolution. For MobileNets the depth wise convolution applies a single filter to each

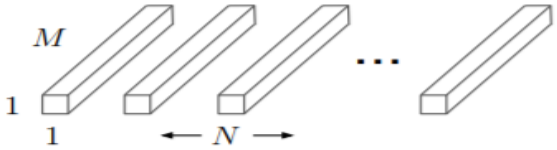
input channel. The point wise convolution then applies a 1×1 convolution to combine the outputs the depth wise convolution. A standard convolution both filters and combines inputs into a new set of outputs in one step. The depth wise separable convolution splits this into two layers, a separate layer for filtering and a separate layer for combining. This factorization has the effect of drastically reducing computation and model size.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1 × 1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

The standard convolution operation has the effect of filtering features based on the convolutional kernels and combining features in order to produce a new representation. The filtering and combination steps can be split into two steps via the use of factorized convolutions called depth wise separable convolutions for substantial reduction in computational cost. In the Mobilenet architecture, All layers are followed by a batchnorm and ReLU nonlinearity with the exception of the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification. Figure 3 contrasts a layer with regular convolutions, batchnorm and ReLU nonlinearity to the factorized layer with depth wise convolution, 1 × 1 pointwise convolution as well as batchnorm and ReLU after each convolutional layer. Down sampling is handled with strided convolution in the depth wise convolutions as well as in the first layer. A final average pooling reduces the spatial resolution to 1 before the fully connected layer. Counting depth wise and point wise convolutions as separate layers, MobileNet has 28 layers.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw / s1	3 × 3 × 32 dw	112 × 112 × 32
Conv / s1	1 × 1 × 32 × 64	112 × 112 × 32
Conv dw / s2	3 × 3 × 64 dw	112 × 112 × 64
Conv / s1	1 × 1 × 64 × 128	56 × 56 × 64
Conv dw / s1	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw / s2	3 × 3 × 128 dw	56 × 56 × 128
Conv / s1	1 × 1 × 128 × 256	28 × 28 × 128
Conv dw / s1	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw / s2	3 × 3 × 256 dw	28 × 28 × 256
Conv / s1	1 × 1 × 256 × 512	14 × 14 × 256
5× Conv dw / s1	3 × 3 × 512 dw	14 × 14 × 512
5× Conv / s1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw / s2	3 × 3 × 512 dw	14 × 14 × 512
Conv / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw / s2	3 × 3 × 1024 dw	7 × 7 × 1024
Conv / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Avg Pool / s1	Pool 7 × 7	7 × 7 × 1024
FC / s1	1024 × 1000	1 × 1 × 1024
Softmax / s1	Classifier	1 × 1 × 1000

Fig.3 – MobileNet Architecture

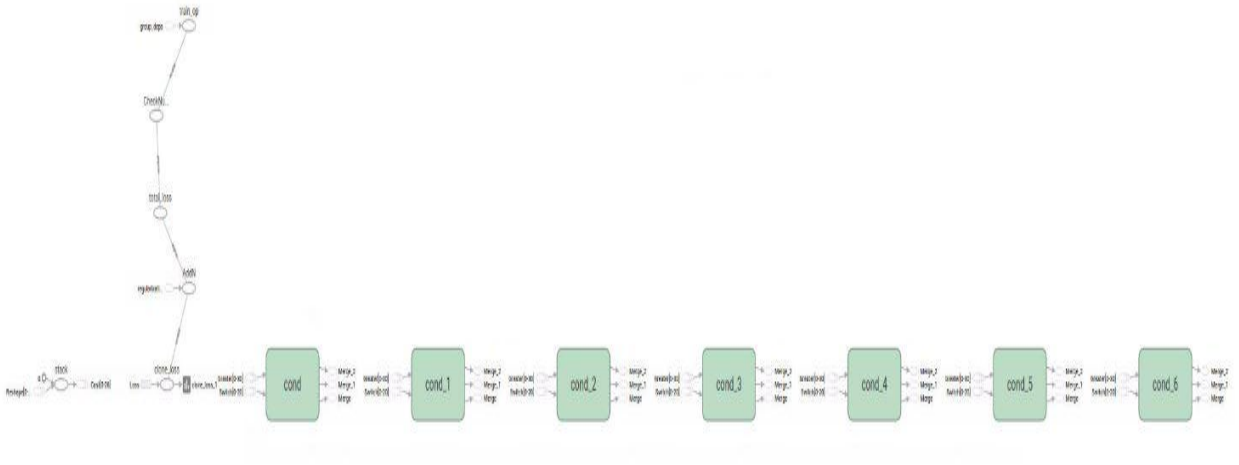


Fig.4 – Main graph diagram from Tensorboard

MobileNet models were trained in TensorFlow using RMSprop with asynchronous gradient descent similar to Inception V3. However, contrary to training large models we use less regularization and data augmentation techniques because small models have less trouble with overfitting. When training MobileNets we do not use side heads or label smoothing and additionally reduce the amount image of distortions by limiting the size of small crops that are used in large Inception training. Additionally, we found that it was important to put very little or no weight decay (l2 regularization) on the depth wise filters since there are so few parameters in them.

Benchmark

To create an initial benchmark for the classifier, I used the Resnet50[6] architecture using Keras with Tensorflow as the backend to classify various classes of images from the PASCAL VOC module. It achieved an accuracy of 80%.

Methodology

Data Preprocessing

The first step towards this project was to test the Mobilenet architecture which was trained with COCO data of 20 classes of images with the tensorflow object detector API. Here I modified the code and implemented OpenCV so that the classifier can also detect the objects from a webcam. I implemented the multiprocessing library of tensorflow to get a faster result with increased frames per second.

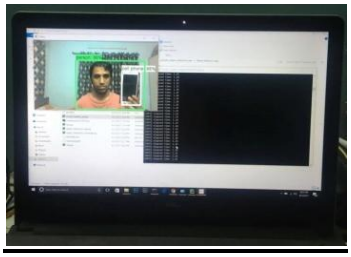


Fig.5 – Sample Object Detection Screen using Webcam

So as we can see, the basic philosophy behind the algorithm of this object detector is drawing a bounding box around the subjects and then making a prediction. For training my own dataset, I have scraped 150 images of Mickey Mouse from the web and manually hand labeled them using LabelImg which creates the annotation files with the bounding box's co-ordinates in form of a XML file. But tensorflow API uses the TFRecord file format. So I had to convert my data to TFRecord. To prepare the input file for the API I needed to consider two things. Firstly, I need an RGB image which is encoded as jpeg or png and secondly I need a list of bounding boxes (xmin, ymin, xmax, ymax) for the image and the class of the object in the bounding box. I used labelImg to create those bounding boxes. Now my next target was to create these annotations from XML to a CSV file and then split the csv file into two separate TfRecord files which will be fed to the neural network. I wrote a script to convert the files from xml to CSV and then the train test split was done in a 90%-10% ratio and converted into two separate tfrecord files. For creating the TfRecord files for both train and test data, I used a custom made script. I also took the visualization_utils.py file which has the labels detected and integrated pyttsx module in this file. I wrote 90 customized voice generated messages for the 90 classes it is being trained on. Now after the files are converted to TfRecord, we also resize the images to 300 x 300 pixels. Also to avoid skewness in data, we implement image augmentation via random horizontal flip and random crop.

Implementation

The project consists of multiple stages. So I will subdivide them into steps.

1. Collect a few hundred images that contain your object - The bare minimum would be about 100, ideally more like 500+, but, the more images you have, the more tedious step 2 is.
2. Annotate/label the images, ideally with a program. I personally used LabelImg. This process is basically drawing boxes around your object(s) in an image. The label program automatically will create an XML file that describes the object(s) in the pictures.
3. The Tensorflow Object Detection API uses Protobufs to configure model and training parameters. Before the framework can be used, the Protobuf libraries must be compiled. This should be done by running the following command from the tensorflow/models directory

   ```
   protoc object_detection/protos/*.proto --python_out=.
   ```

4. When running locally, the tensorflow/models/ and slim directories should be appended to PYTHONPATH. This can be done by running the following from tensorflow/models/:

   ```
   export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
   ```

5. Split this data into train/test samples in a 90:10 ratio.
6. Generate TF Records from these splits
7. Create a protobuf text file containing the label of the objects in a JSON format. Here as I am training just one class (Mickey Mouse), it will contain only one record.
8. Setup a .config file for the model of choice (you could train your own from scratch, but we'll be using transfer learning). This is the object detection pipeline in which we can control all the parameters like batch size, number of epochs, image resize, image augmentation, learning rate, weight decay, activation and regularizer of the neural network etc.
9. Train the model and at the same time validate the model with the test images. Log both the validation loss and the validation accuracy.
10. Plot the logged values.
11. Export graph from new trained model. This is the frozen graph we can use now in our object detection.
12. Detect custom objects in real time!

The loss function mainly consists of weighted sigmoid error. Large weights and biases of all four layers are penalized by using weight decay. The weights are added to the loss function after they are multiplied by their specific weight decay rates.

Refinement

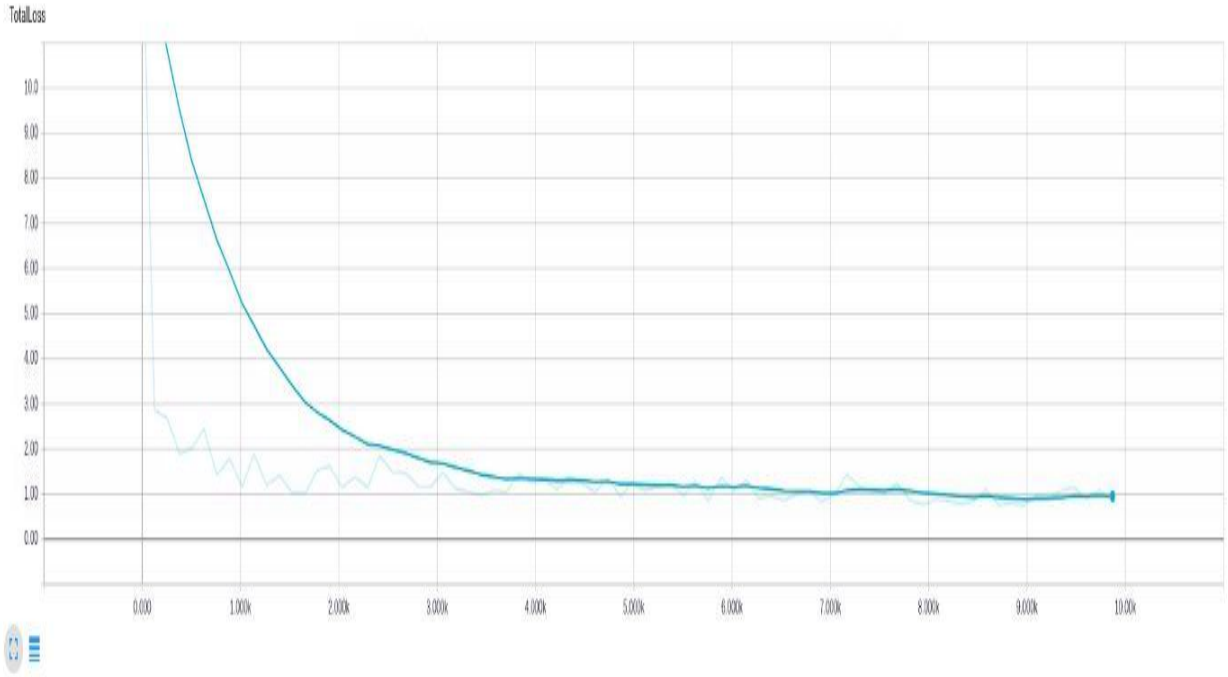


Fig.6 :Total loss/Validation loss

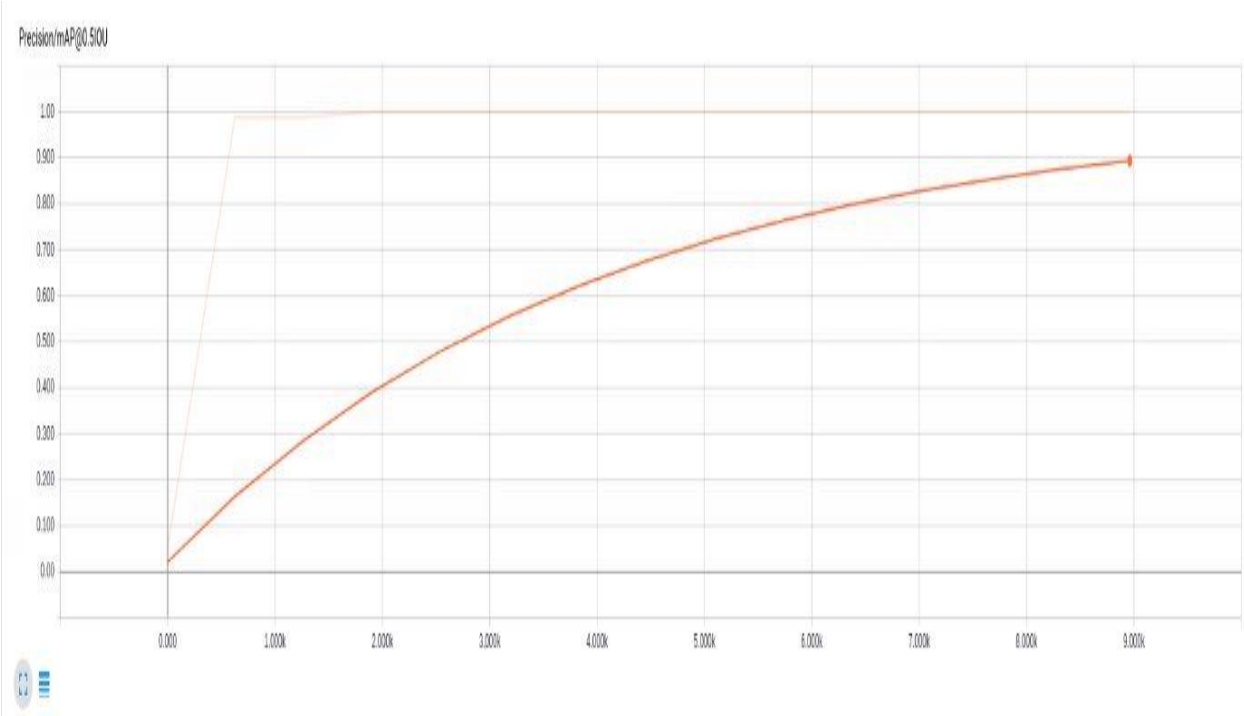


Fig.7: Accuracy

So as we see that the validation loss decrease quite quickly and steadily with the number of steps and the accuracy went up really high. I trained this model in cloud using a GPU. We see around 9000 steps, the loss gets steady around a value of 1 and the accuracy reaches a high value of 90%. As mentioned in the Benchmark section, the Resnet50 model trained with Pascal VOC data achieved a good accuracy of 80% and this model surpasses that.

In this model, some improvements were achieved mainly by the following points:

1. A decaying learning rate: Whenever the loss function stopped decreasing, a learning rate drop was applied.
2. Momentum, which takes into account the previous learning rate when applying the next one was fine-tuned.
3. Weight Decay: When overfitting was detected, i.e. the training and the validation loss diverged too much, the weight decay rate was increased.

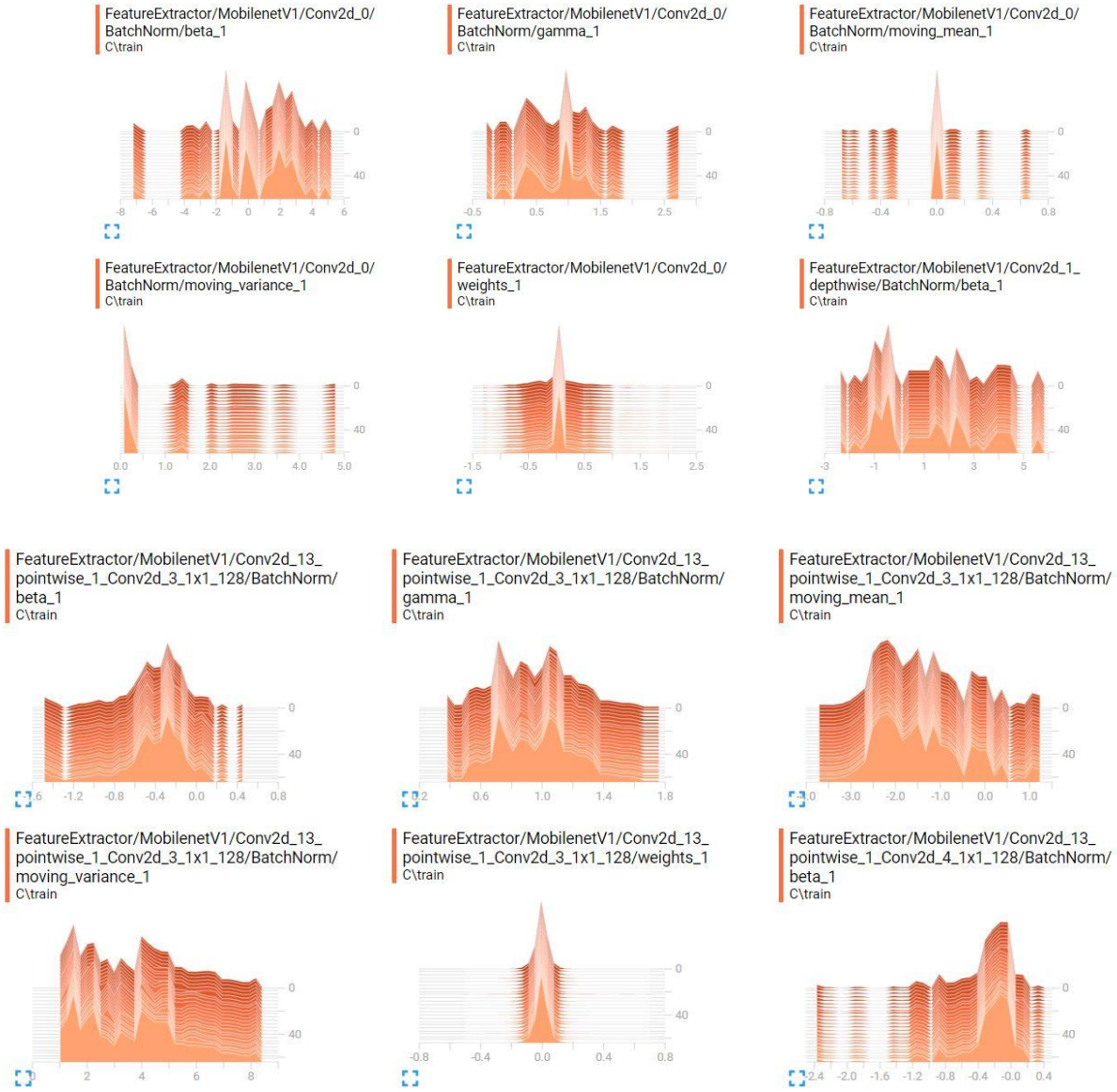


Fig.8 and 9: Divergence of the histogram in the Mobilenet architecture

Results

Model Evaluation and Validation

During the training process, a validation set was used to evaluate the model.

The final architecture was chosen as it performed better than the Resnet50 model, which was the benchmark model in this case. The Mobilenet model consists of 28 layers (Fig.3) and some salient analysis are as follows:

1. The shape of the filter for the convolutional layers was 3 x 3 and 1 x 1 alternately.
2. The depth of the filter for the first layer was 32, for the second layer, it was 64 and so on.
3. For each convolutional layer, a depth wise convolutional layer was added.
4. These resulted in shallow height and width but greater depth, which is great for spatial detection of objects.
5. A global average pooling layer and a fully connected layer was added near the end.
6. A softmax classifier is added in the end to give us the probability of the object detected.
7. The weights of the convolutional neural networks are initialized by sampling a truncated normal distribution with standard deviation of 0.03.
8. The learning rate is set to an exponentially decaying learning rate with an initial value of 0.004.
9. The training runs for 9000 iterations.

To verify the effectiveness and robustness of the final model, a test was conducted using a Mickey Mouse doll at different lighting conditions and also at various distances from the camera. It was observed, that on average, the camera could detect the Mickey Mouse figure with an accuracy of 82% with highest accuracy of over 90%.

Justification

Compared to our initial benchmark model, the mobilenet model performed significantly better. It achieved a high accuracy of 90% where around 140 images were used for training and 10 images for testing. The validation loss reached a low value of around 1 pretty quickly in 9000 iterations/steps. Also, the mobilenet model, even though it resembles the inception model, is much faster as it is much low duty and leaves little footsteps. So it will be an ideal model where speed is of utmost importance such as a mobile app.

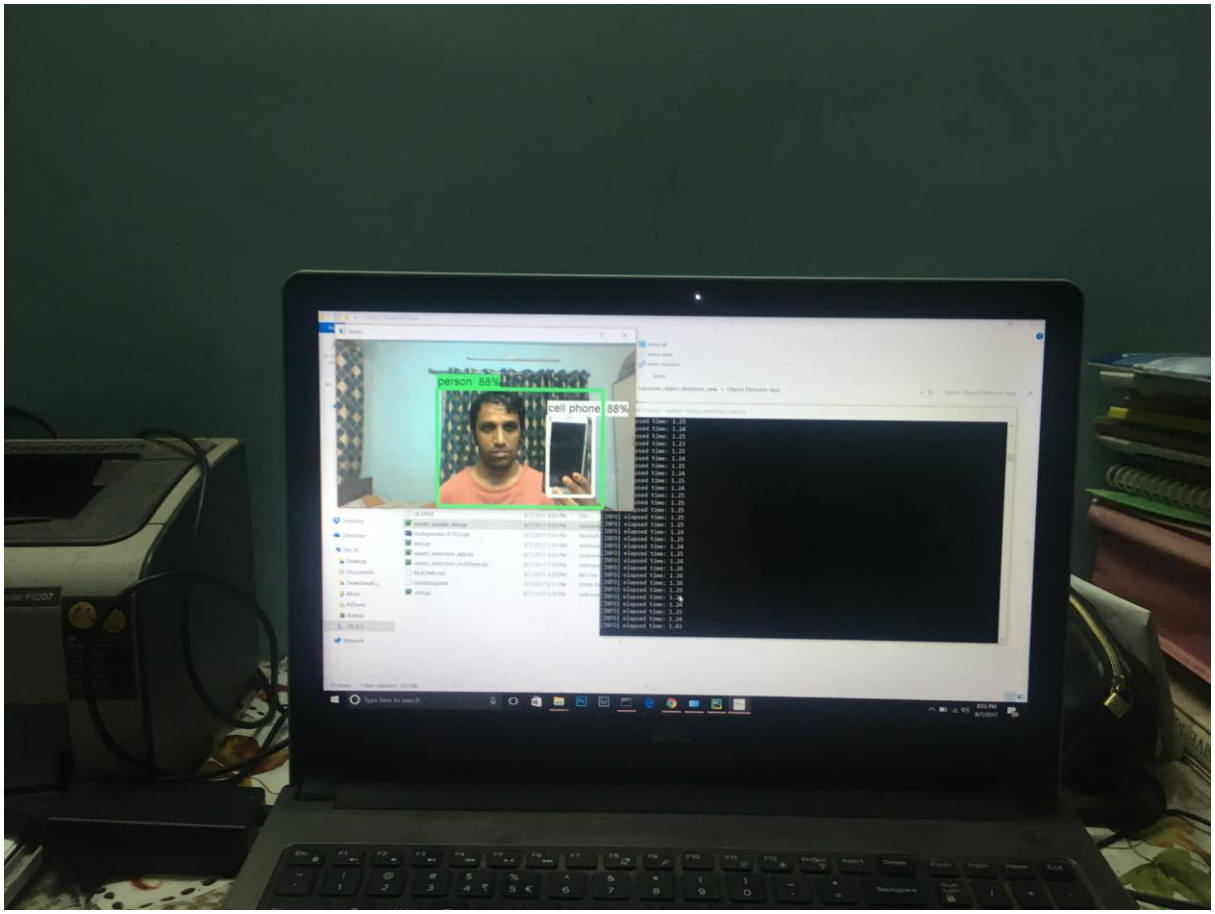
So in summary, this model can be a really good model for training a classifier for real time object detection and we can add to our model various classes of objects and train them.

Conclusion

Free form Visualizations

In the following sections, we can see both the object detection model trained on the MS-COCO dataset and the custom trained classifier results on the Mickey Mouse images. As we can see, it is able to successfully create the rectangular bounding box around the ROI and detect the objects with pretty high probability.


Mickey 82%


person 88%
cell phone 88%

Reflection

The process used for this project can be summarized using the following steps:

- An initial problem and relevant, public datasets were found.
- The data was downloaded and preprocessed.
- A benchmark was created for the classifier.
- The tensorflow object detection API was installed and implemented.
- The code was modified to change the object detection API from images to real time feed via the webcam.
- The classifier was trained using the data (multiple times, until a good set of parameters were found).
- The pyttsx module was implemented to enable a speech generation program to speak out loud the name of the objects detected.
- The model was tested with custom objects via the webcam.

I personally found step 5 to be pretty difficult as it was pretrained on images whereas my motive was to use it to detect objects real-time. So here, the code was significantly changed. Also the python multiprocessing library was used to speed up the computation and the frame rate.

Improvement

For further improvement of the model, more capable hardware is one of the requirements as that will result in faster training time and incorporation of more classes. Once that is achieved, we can take our custom trained model and improve on it by training it on many more custom objects and it will just add up to the model's existing knowledge of the range of objects it has knowledge of. We can also experiment with the object detection pipeline by changing the learning rates, batch size etc and see if we can improve on the accuracy and classification delay.

References:

1. Jaderberg, Max et al. "Reading text in the wild with convolutional neural networks." *International Journal of Computer Vision* 116.1 (2016): 1-20.

2. "BlindTool" < https://play.google.com/store/apps/details?id=the.blindtool&hl=en > (2016).

3. Devlin, Jacob et al. "Language models for image captioning: The quirks and what works." *a rXiv preprint arXiv:1505.01809* (2015).

4. Lin, Tsung-Yi et al. "Microsoft coco: Common objects in context." *Computer Vision–ECCV 2014* (2014): 740-755.

5. Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications."

6. Kaiming He et al. "Deep residual Learning for Image Recognition."

7. https://github.com/datitran/object_detector_app