Modified Dual Shot Face Detector

Peilu(Joy) Lu, Jianwei Li, Akshar Panchal, Ruchira Gothankar, Akash Budholia

Abstract

Detecting faces has historically been a challenging problem for computers. The advancements in Deep learning has made it possible to solve this efficiently. Convolutional Neural Network which is an algorithm of Deep learning has achieved state-of-the-art results. The crux of the issue is improving the performance when affected by the variability of non-linear effects like face alignment, of-angle faces, occlusions, low-resolution. This report will talk about a few existing powerful approaches of Convolutional Neural Network along with the approach that is selected for the study which is Dual Shot Face Detector(DSFD). The selected approach uses VGG-16 as the backbone. In this study, we present a CNN based on DSFD with backbone as EfcientNet-B0 EfcientNet-B1 - which is a new approach based on highly effective compound coefficient to scale up CNNs in a more structured manner. The report explains the implementation of the same and the results that we achieved by doing so along with the challenges faced.

1 Introduction

Face detection is a popular research area in computer vision. The current research focuses more on the issues like extreme illuminations and poses, occlusions, shadows etc. In this work, we apply Convolutional Neural Network to detect faces. A convolutional network consists of a sequence of layers, where each layer transforms one volume of activations to another through a function. It consists of input, output and multiple hidden layers.

The two greatest approaches to this day for face detection included Methods based on Regional Proposal Network and Single Shot methods such as Single Shot Detectors (SSD). We have selected the Dual Shot Face Detector approach which is based on SSD, to study and research. This approach includes better feature learning, progressive loss design and anchor assigned based data augmentation. The entire setup for the project is done on Amazon EC2. In this project, we have studied the working of DSFD with VGG16 as the backbone by training the model and testing.The backbone is initialised by pretrained network of VGG16.The model is trained on WIDER FACE dataset which consists of 393,703 annotated faces with large variations in scale, pose and occlusion in total 32,203 images. To experiment more on the baseline network, we have changed the baseline for the DSFD to EfficientNet-B0 and EfficientNet-B1 and have trained the model on the WIDER FACE dataset. This report summarises the entire study, implementation details and the results achieved.

A brief introduction on DSFD, VGG16 and EfficientNet is as follows:

1.1 Dual Shot Face Detector

DFSD introduces 3 major enhancements over previous Single Shot Detector model in terms of devising a new way to compute feature maps, an upgrade over the existing Loss function and improvement in the strategy to match the predictions.

1.1.1 Feature Enhanced Model:

It uses the same backbone network as the Single Shot Detector. The six feature maps are enhanced using a module called feature enhanced model(FEM), whose objective is to feed the object classifier and the bounding-box regressor with more robust features (*Refer Figure 1 and Figure 2*). The Feature Enhance Model brings in element-wise product of the current feature map and an upsampled version of the feature map at the next level. The resulting map is split into 3, each undergoes a series of dilated convolutions and being concatenated back to the original size of the feature map.

1.1.2 Progressive Anchor Loss:

This is the loss function employed by DSFD which uses two different kinds of anchors:

- The set of anchor ‘a’ is used to compute the Second Shot Loss(SSL)
- The set of smaller anchors ‘sa’ is used to compute First Shot Loss(FSL)

On average, the faces detected by original shots are smaller. To account for this loss, the authors use a smaller set of anchors for the first shot loss. The only shot used for all the anchors, 1:1.5, based on statistics of faces.

1.1.3 Improved Anchor Matching:

To overcome the problem of imbalance in the number of positive and negative bounding boxes, both the algorithms RPN/SSD, use the sampling the predicted boxes at a fixed positive/negative boxes ratio to data augmentation strategies to vary the relative scale of the objects in the input image. During training, with a probability of 40%,anchor-based sampling is applied to the input image.

1.2 VGG-16

VGG-16 is a convolutional neural network model developed by K. Simonyan and A. Zisserman in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The

model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG-16 was trained for weeks and was using NVIDIA Titan Black GPUs. (*Refer Figure 3*)

1.3 EfficientNet

EfficientNets are a family of image classification models, which achieve state-of-the-art accuracy, yet being an order-of-magnitude smaller and faster than previous models. EfficientNets was developed based on AutoML and Compound Scaling. In particular, it first use AutoML Mobile framework to develop a mobile-size baseline network, named as EfficientNet-B0; Then, it uses the compound scaling method to scale up this baseline to obtain EfficientNet-B1 to B7.The feature of EfficientNet is proposing a compound scaling method to balance all dimensions of network width/depth/resolution. EfficientNet proposes a new compound scaling method, which uses compound co-efficient φ to uniformly scale width, depth and resolution as follows:

$$\begin{aligned} \text{depth: } & d = \alpha^\phi \\ \text{width: } & w = \beta^\phi \\ \text{resolution: } & r = \gamma^\phi \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

Here α, β and ɣ are constants that can be determined by grid search, which imply how many more resources are available for scaling for network width, depth and resolution, Φ is user defined co-efficient that controls model scaling. It first calculate values of α, β and ɣ from grid search. If we take Φ = 1, assuming twice more resources are available. We find best values of α, β and ɣ under constraints mentioned above. After this we just need to scale up value of Φ to obtain Efficient Net B1 to B7.

2 Literature Survey

As a part of our literature survey, we analyzed 5 state of the art algorithms including DSFD (as explained above).

2.1 Selective Refinement Network

The author chose ResNet with 6 level feature pyramid as the backbone. A STC(used to filter out simple false negatives from lower layers to reduce the classification search space) selects the three lower pyramids to perform 2 step classification. The motivation behind this design is to sufficiently utilize the detailed features of large faces on the three higher pyramid levels to regress more accurate locations of bounding boxes and to make three lower pyramid levels pay more attention to the classification task. (*Refer Figure 6*)

2.2 **RetinaNet Architecture**:

RetinaNet is one stage object detection method. Its faster and simpler as compared to other two staged models which require extra processing. We will perform some post processing on the baseline model obtained from RetinaNet using two step classification and regression for detection. It consists of backbone: a feedforward ResNet structure which extracts feature maps, neck: a feature pyramid structure network which extracts and produces richer multi convolutional pyramid. We use IOU as the regression loss to minimize difference between predictions and ground rules. We revisit data augmentation using data anchoring samples for training. Using max out operation for fine classification. Lastly we test the model on WIDERFACE benchmark and achieve substantial state of the art average precision results. (*Refer Figure 5*)

2.3 **HyperFace** :

The proposed algorithm called HyperFace uses the intermediate layers of a deep CNN using a separate CNN followed by a multi-task learning algorithm that operates on the fused features and further involves generating class independent region proposals from the given image followed by CNN which classifies them as face or nonface. Following this is a postprocessing step which involves Iterative Region Proposals and Landmarks-based Non-Maximum Suppression (L-NMS) to boost the face detection score and improve the performance of individual tasks. (*Refer Figure 7*).

2.4 Modified Open Face:

So the author has proposed a new method to select appropriate hard-negatives for training using Triple Loss. The author incorporates the pairs, which otherwise would be discarded and helps increase overall efficiency of the model. Also, the introduction of Adaptive Moment Estimation algorithm mitigates the risk of early convergence, due to the introduction of hard negatives pairs. Open Face uses 2D Affine Transformation as its preprocessing, which sets the nose and eyes closer to the mean by cropping images to the edge of the landmarks produced by dlib face detector. The normalized images are fed in to an embedding. The reason hard triplets are chosen is that they improve the learning significantly. In openface, if a negative is found within a threshold, it is discarded. However, in this approach, we modify the triplet loss by choosing the pair that results in closest margin to **@.** It still needs to satisfy:

$$||f(xa) - f(xp)|| < ||f(xa) - f(xn)||$$

We choose Dual Show Face Detection to be our project reference because this algorithm indicates an efficient way to detect faces in different scales, occlusions, and appearance. In case, of modified open face the efficiency has been tested on LFW Database, which is very limited in the number of overall as well as distinct images.

3 Methodology and Implementation Details

So far, we have learned about DSFD's network, and now we need to work on the backbone network to extract the feature maps. As options, we have VGG-16, ResNet -51, ResNet-101, ResNet-151, EfficientNet-(B0~B7). As for now, we finished training using VGG-16, EfficientNet B0, and EfficientNet B1. Other than that, the training using Efficient B2 - B7 are still in progress. We will only focus on VGG-16, EfficientNet B0, and EfficientNet B1 in this report.

The backbone of DSFD here is mainly to extract the feature maps, so the key idea of changing the backbone is to change the feature extraction methods. For the VGG16 network, We select conv3_3, conv4_3, conv5_3, conv fc7, conv6_2 and conv7_2 as the layers to extract the 6 original feature maps. The sample code below shows the extraction of two original feature maps and the generation of two feature enhanced maps using VGG.

Extracting Two Original Feature Maps (Conv4_3, Fc 7)	Feature Enhancement of Two Original Feature Maps

# apply vgg up to conv4_3 relu 	
for k in range(16):
  x = self.vgg[k](x)
of1 = x
s = self.L2Normof1(of1)
pal1_sources.append(s)
 	
# apply vgg up to fc7 	
for k in range(16, 23):
  x = self.vgg[k](x)
of2 = x
s = self.L2Normof2(of2)
pal1_sources.append(s)
	x = F.relu(self.fpn_topdown[4](conv5), inplace=True)
	
	conv4 = F.relu(self._upsample_prod(
	  x, self.fpn_latlayer[3](of2)), inplace=True)
	ef1 = self.fpn_fem[0](conv3)
	
	x = F.relu(self.fpn_topdown[5](conv4), inplace=True)
	conv3 = F.relu(self._upsample_prod(
	  x, self.fpn_latlayer[4](of1)), inplace=True)
	ef2 = self.fpn_fem[1](conv4)

Then we generate the other six feature enhanced maps based on the original feature maps using *Feature Enhancement Model (FEM)*. Eventually we will have the following two shots to place the previous anchors and implement face detection.(*Refer Figure 1*)

Same for VGG, we need to choose the stages to extract the feature maps using EfficientNet. The base network of EfficientNet is found by performing a Neural Architecture Search that optimizes both accuracy and FLOPS. The networks are shown in figure 6. The MBConv here is Inverted Residual Block.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	28×28	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

In the beginning, we just choose stage 2 to 8 to extract the six original feature maps as the first shot. Then we work on feature enhancement using the original feature maps to generate the other six feature maps as the second shot. For EfficientNet B0 network, the parameters for width,

depth and resolution are 1.0, 1.0, and 1.0 respectively, and for EfficientNet B1 network, they are 1.0, 1.1, and 1.07. Actually, we have the following configuration for all the EfficientNets:

EfficientNet Parameters for Width, Depth, Resolution (*compared with 224x224*) and Dropout Rate

```
def efficientnet_params(model_name):
    """ Map EfficientNet model name to parameter coefficients. """
    params_dict = {
        # Coefficients:   width,depth,res,dropout
        'efficientnet-b0': (1.0, 1.0, 224, 0.2),
        'efficientnet-b1': (1.0, 1.1, 240, 0.2),
        'efficientnet-b2': (1.1, 1.2, 260, 0.3),
        'efficientnet-b3': (1.2, 1.4, 300, 0.3),
        'efficientnet-b4': (1.4, 1.8, 380, 0.4),
        'efficientnet-b5': (1.6, 2.2, 456, 0.4),
        'efficientnet-b6': (1.8, 2.6, 528, 0.5),
        'efficientnet-b7': (2.0, 3.1, 600, 0.5),
    }
    return params_dict[model_name]
```

The code below shows how we extract the feature maps from EfficientNet. We continuously tune the parameters here to change the layers we used to extract the feature maps in order to optimize the performance for small face detection.

Exposing All Feature Maps from EfficientNet

```
def extract_mutiple_features(self, inputs):
    """ Returns output of all convolution layers"""
    #stem
    x = self._swish(self._bn0(self._conv_stem(inputs)))
    mutiple_features = []

    #blocks
    for idx, block in enumerate(self._blocks):
      drop_connect_rate = self._global_params.drop_connect_rate
      if drop_connect_rate:
        drop_connect_rate *= float(idx) / len(self._blocks)
```

```
        x = block(x, drop_connect_rate = drop_connect_rate)
        mutiple_features.append(x)

    #head
     x = self._swish(self._bn1(self._conv_head(x)))
     mutiple_features.append(x)
     return mutiple_features
```

Continuously Changing the Final Layers Used to Extract Feature Maps

```
# layers used to extract feature maps
# layers index [4, 7, 15, 22], [10, 14, 15, 22] …...

features_maps = self.efficient.extract_mutiple_features(x)
of1 = self.L2Normof1(features_maps[4])
pal1_sources.append(of1)
of2 = self.L2Normof2(features_maps[7])
pal1_sources.append(of2)
of3 = self.L2Normof3(features_maps[15])
pal1_sources.append(of3)
of4 = features_maps[22]
pal1_sources.append(of4)
```

All of the training was done using the pretrained model. For VGG16, we trained on the weights without having any fully connected layers. For EfficientNet, Unfortunately, we didn't find the pre trained model without fully connection layer. Before using the pretrained model, we first need to remove the weights of FC.

Remove the Weights of FC

```
state_dict = model_zoo.load_url(url_map[model_name])
state_dict.pop('_fc.weight')
state_dict.pop('_fc.bias')
res = model.load_state_dict(state_dict, strict=False)
```

Due to limited GPU memory, we can't get 100 epoches done all at once. So this time, we changed our training strategy to save the intermediate weights of the model. And we also

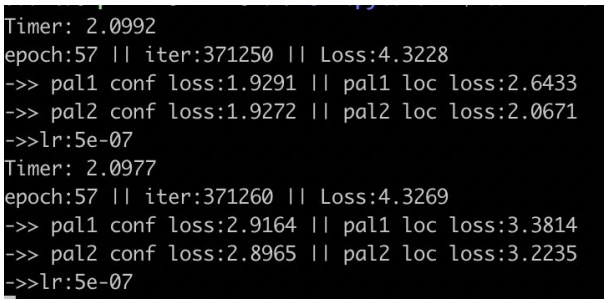
updated the code to help us resume the training process from the intermediate model. In this case, we do not need to worry about crashes caused by hardware problems. The new strategy is getting the weight for every 5000 iterations (Definition: *iterations = start_epoch * len(train_dataset) / batch_size*), and update the globally best model weights during testing when we get a smaller loss.

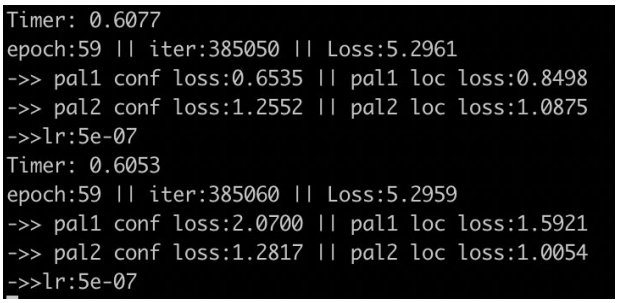
4 Results & Discussion

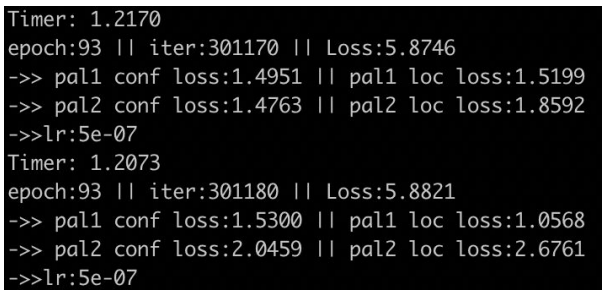
Because of the workload for training is very heavy and the machines are not enough, we trained multiple models at the same time, with limited efficiency. Eventually, we completed the training work for DSFD with VGG16, EfficientNet-B0 and EfficientNet-B1, we considerably good performance on both big and small faces using VGG16. The following evaluation accuracy on dataset AFW, PASCAL, FDDB is based on it.

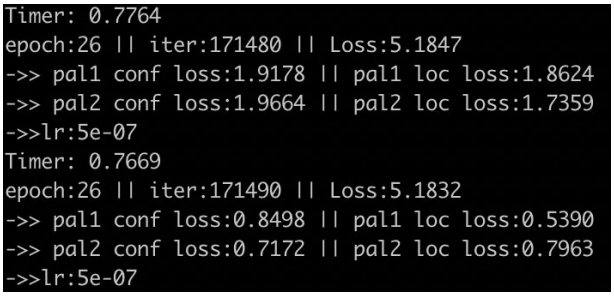
AFW	PASCAL	FDDB
99.89	99.11	98.3

For EfficientNet-B0 and EfficientNet-B1, the performance for small face detection is not as good as we expect. Here are some train logs of our work:









The above four figures are the logs for DSFD with VGG16, DSFD with EfficientNet-B0, DSFD with EfficientNet-B1 and DSFD with different feature maps from EfficientNet-B1 respectively. And the batch size for each network is 2, 2, 4, 2. We can see that when the epoch equals to 57,

the loss for DSFD with VGG-16 decreased to 4.3 However, for the EfficientNet-B0 and EfficientNet-B1, the loss has been hovering in the range of 5 and cannot be further reduced. And when we test it on the life image, the performance on the small face is not good enough compared with VGG-16.

So we rethink our network why the performance is not good on the small face for DSFD with EfficientNet? We focused on the extraction of feature maps. In the original design, we extract feature maps on some layers with low resolution. Since low resolution does not perform well for small face detection, maybe we can choose layers with higher resolution to perform feature extraction. We can see the loss has been 5.1 when the epoch equals 26 on this time. Higher number of epochs and training on higher number of images will result in a much better efficiency which we will cover as a future scope.

5 Conclusion

In this project, we focused on the DSFD for face detection. We implement the network with VGG16 and switched it to EfficientNet as the backbone. We get a good performance on both big and small face detection when using VGG-16. For EfficientNet-B0 and EfficientNet-B1, we only get a good performance on big faces. We trained again using feature maps from layers with higher resolutions and as a future scope we will keep optimizing our algorithm.

References

1. [2018] Jian Li, Yabiao Wang, Changan Wang, Ying Tai, Jianjun Qian, Jian Yang, Chengjie Wang, Jilin Li, Feiyue Huang, *DSFD: Dual Shot Face Detector*, arXiv:1810.10220
2. [2018] Cheng Chi, Shifeng Zhang, Junliang Xing, Zhen Lei, Stan Z. Li, Xudong Zou Selective Refinement Network for High Performance Face Detection , arXiv:1809.02693v1
3. [2017] Rajeev Ranjan, Vishal M. Patel, Rama Chellappa HyperFace: A Deep Multi-task Learning Framework for Face Detection, Landmark Localization, Pose Estimation, and Gender Recognition.
4. Faen Zhang, Xinyu Fan, Guo Ai, Jianfei Song, Yongqiang Qin, Jiahong Wu. Accurate Face Detection for High Performance.
5. [2018] Kevin Santoso, Gede Putra Kusuma: Face Recognition Using Modified OpenFace
6. https://github.com/yxlijun/DSFD.pytorch
7. https://neurohive.io/en/popular-networks/vgg16/
8. https://medium.com/@nainaakash012/efficientnet-rethinking-model-scaling-for-convolutional-neural-networks-92941c5bfb95
9. https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet
10. https://github.com/lukemelas/EfficientNet-PyTorch
11. https://arxiv.org/abs/1905.11946
12. https://arxiv.org/abs/1810.10220

Appendix

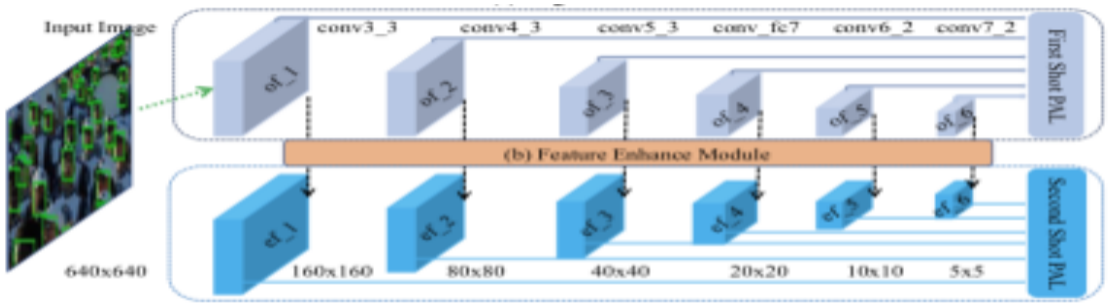


Figure 1

Figure 2

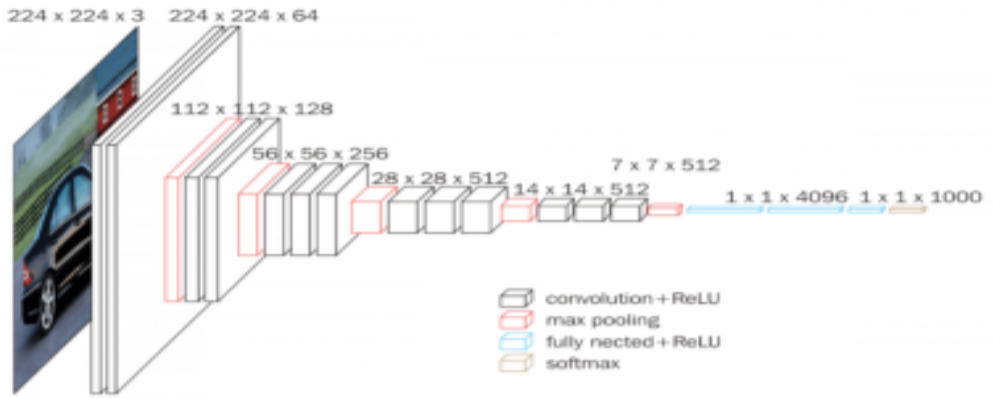


Figure 3

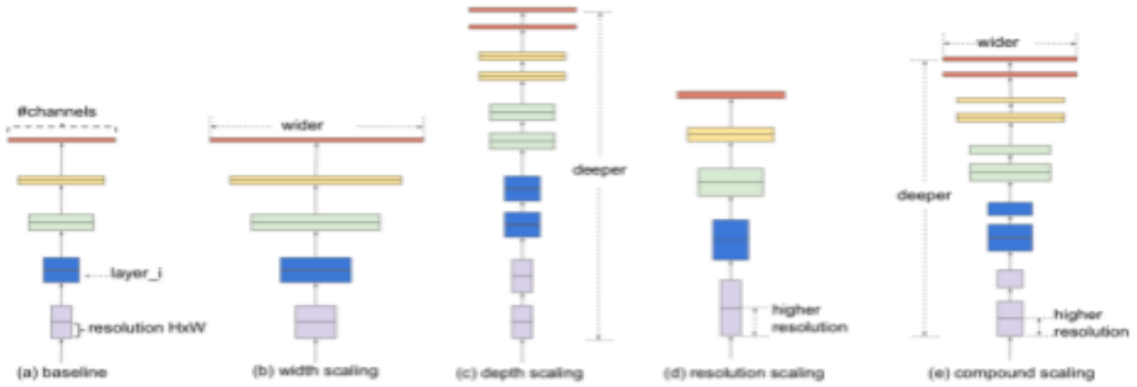


Figure 4

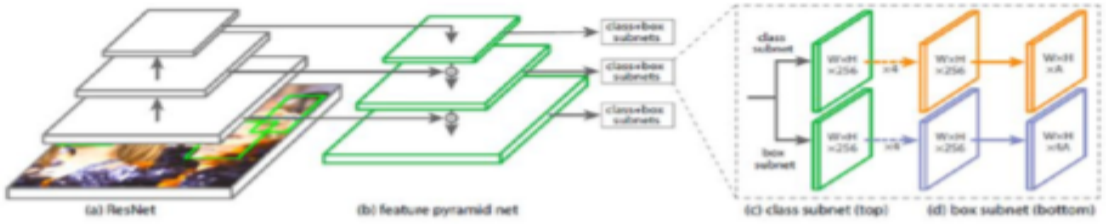


Figure 5

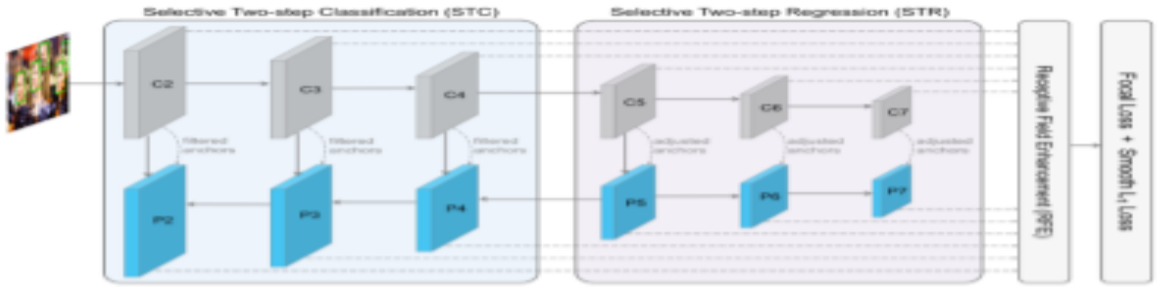


Figure 6

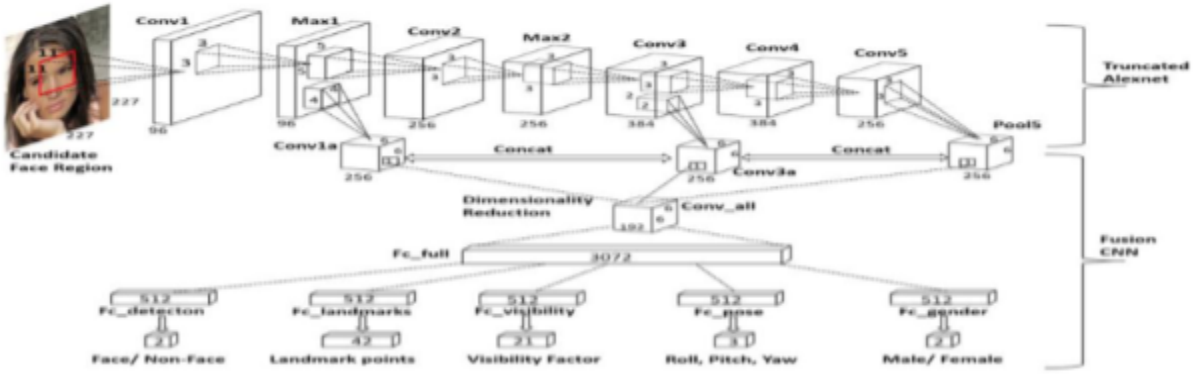


Figure 7

Acknowledgement:

- We would like to thank Amazon for providing us with AWS coupons.
- We would also like to thank Prof. Mashhour Solh for his timely inputs which helped us attain clarity on how to approach the project implementation