Face R-FCN Implementation

Vikram Karthikeyan * Ved Harish Valsangkar †

May 19, 2019

Abstract

This project is a re-implementation of the original paper Face R-FCN validated on WIDER FACE dataset. We have taken an existing code [1, 2] and attempted to tweak it to understand the internal workings of the R-FCN and its application in the domain of face detection. The base architecture was kept true to the original paper with minor additions in an attempt to improve the performance. Our training pipeline is one of the most efficient training pipelines which has been tested extensively using the Tesla P100 GPU as a baseline to achieve optimal CPU-GPU interactions. Several minor changes will be discussed in the paper which provides a strong theoretical basis for achieving potentially better results in the Face R-FCN architecture.

1 Introduction

As one of the bench-marking algorithms, Face R-FCN[3] achieved state-of-the-art performance in detecting faces and regressing the proposed bounding to fit the face. The paper was evaluated over 2 benchmark data sets, WIDER FACE[4] and FDDB[5]. Achieving better performance over Fast/Faster RCNN[6, 7], the algorithm made use of position sensitive pooling layers from the original R-FCN architecture[8]. The original paper made use of OHEM[9] technique to help achieve better performance by using loss from hard examples in order to force the network to learn the marginal and the border cases which would in turn enable the network to work better on easier data sets.

For our implementation we have evaluated on the WIDER FACE[4] data set only. This paper will provide an in-depth understanding of each layer involved in the R-FCN architecture[8] and how it could be tweaked to achieve better results for the face detection task. We have implemented each module from scratch in an attempt to understand the complex R-FCN architecture[8] and address the common issues faced during development which most of the papers do not address today.

2 Background

Face R-FCN[3] model utilizes the R-FCN[8] architecture with the ResNet-101[10] model as the backbone. Now there is only one object class to detect along with the background from the input images i.e. faces.

A broad procedure of the original paper can be given by the following algorithm.

*Person Number: 50289727 [vkarthik@buffalo.edu]
†Person Number: 52090388 [vedharis@buffalo.edu]

Figure 1: Faces with bounding box annotations in WIDER FACE [4] data set

Algorithm 1 Overview of the R-FCN pipeline

INPUT : Image

RESULT : Regions with faces

$features \leftarrow extractor(INPUT)$ ▷ Features @ RESNET Layer 3
$ROIs \leftarrow region\_proposal(features)$
$scores \leftarrow compute\_scores(features)$
$ps\_scores \leftarrow Conv(scores)$
$R \leftarrow$
for all $roi \in ROIs$ **do**
    for all $P \in PATCHs$ **do**
        $P_i \leftarrow argmax(ps\_scores[C * i : C * (i + 1)])$ ▷ C, i: Classes, position
    $Q \leftarrow vote(P)$
    $R \leftarrow R \cup regression[Q]$ ▷ Location and Dimensions for Q
return R

2.1 Face R-FCN[3]

It was one of the most recent advancements in achieving a fully learnable face detector which could be trained end-to-end. They have tried to utilize all of the best practices involved in object detection like OHEM[9] and weighted average pooling to try to improve the overall accuracy of the face detector. However, as we have observed, the training pipeline has several room for optimization both in terms of accuracy and training time. Our paper will explore in detail the areas of optimization with their respective algorithms for implementation.

2.2 YOLO[12]

Even though YOLO[12] primarily focused on improving inference speed for video processing, it introduced the idea of object detection as a regression problem for bounding boxes. This is essential for face detection as for faces, it is not always perfect squares or rectangles with a fixed aspect ratio. Faces usually extremely suffer from pose and rotational variance which needs to be taken into account during the training process. We have tried to define a comprehensive set of anchors in our implementation to help aid this process.

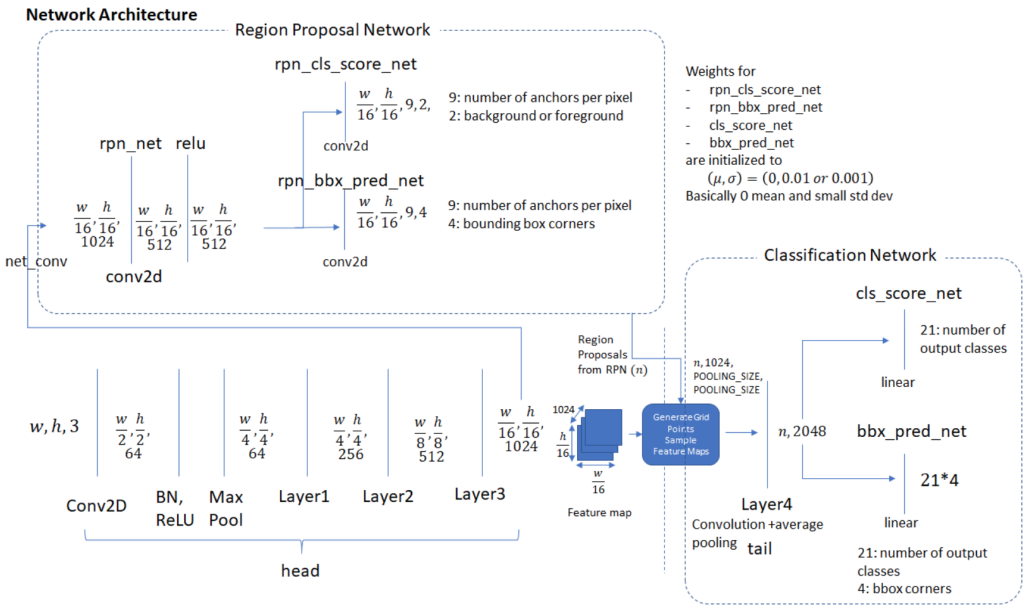

Figure 2: Detailed architecture of R-FCN [11]

2.3 Experiments done by GCN[13]

We have taken inspiration from GCN[13] which tries to achieve a task using only convolutional layers instead of intermediate fully connected layers to create a fully convolutional average pooling layer. It also helped us to conclude that smaller grid sizes and anchor sizes are essential to detect images of smaller scales in the WIDERFACE[4] data set.

2.4 Comparison with previous work

The region specific pooling in R-FCN[8] handles translation variance by using position sensitive score maps. We have improved upon this idea by introducing an additional BasicBlock before computing the score maps to have a more tailored network for face detection.

YOLO[12] uses only anchors of fixed ratios, 1:2, 1:1 and 2:1. Since, WIDERFACE[4] has face boxes of extremely varied dimensions, this poses a serious concern in terms of bounding box regression to appropriate ratios. In this paper, we have also used several anchor ratios for a more universal face representation by the RPN.

Face R-FCN[3] introduces the concept of using a separate weighted layer for replacing global average pooling with weighted average pooling. We have slightly modified this architecture by introducing a fully learnable convolution layer after PSROI pooling to achieve the same result with a fully convolutional architecture.

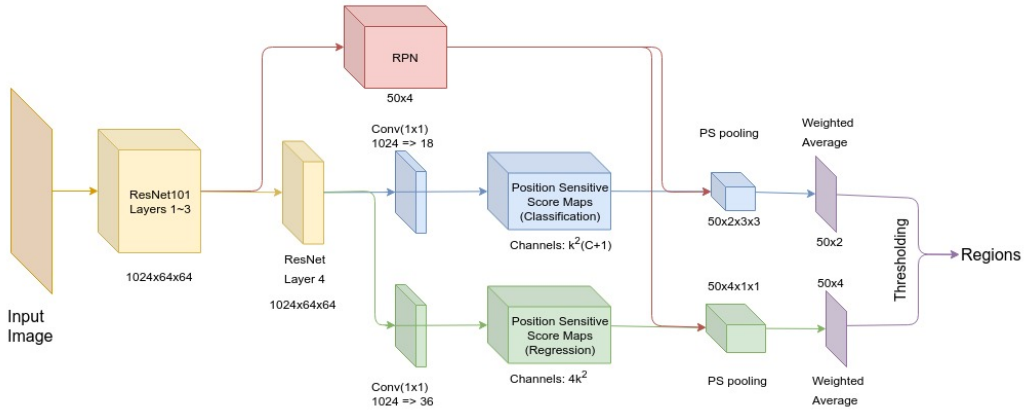

Figure 3: Our modified Face R-FCN architecture for the WIDERFACE dataset

3 Methodology

3.1 Components

3.1.1 Base Feature Extractor

The base feature extractor is a pre-trained ResNet[10] model. The model is pre-trained on ImageNet[14] and a publicly available weight snapshot is used. We utilize layers 1 through 3 for the extraction and layer 4 for further processing after passing the features to the Region Proposal Network. The resnet model is frozen and not trained/fine tuned further.

3.1.2 Region Proposal Network

The Region Proposal Network is responsible to generate regions which may contain faces as a part of the image foreground. These shall later be evaluated and pruned. The RPN has a Proposal layer which generates Regions of Interest (ROIs) and an Anchor Target Layer which helps evaluating these regions using loss functions.

3.1.3 Proposal Layer

The Proposal layer is responsible for generating the proposals for a given input image. The layer generates anchors for all positions in the features and keeps only the top 2000 anchors. These are the initial proposals. We apply Non Maximal Suppression on these proposals to get only a fraction of the proposals. The list is sorted and those above a preset threshold (≥ 0.7) are kept. Of these, only the top 200 are finalized.

3.1.4 Anchor Target Layer

The Anchor Target layer is responsible for generating targets for regression loss calculation. This layer generates anchors similar to the Proposal layer. Based on overlap with the ground truth boxes, hard positive (IOU ≥ 0.7) and hard negative (IOU ≤ 0.3) are passed on as positive and negative anchors for classification and regression loss calculations. All remaining positions are labeled IDC (I don't care) and not considered for losses.

Algorithm 2 Overview of the Proposal layer

INPUT : classification probabilities, bbox predictions
RESULT : Proposals

procedure PROPOSAL_LAYER(cls_prob, bbox_pred)
  for all $i_{h,w} \in cls\_prob$ **do**
    $anc = generate\_anchors(cls\_prob.shape)$
    $anc = anc + bbox\_pred$ ▷ bbox_pred : regression deltas
    $anc = clip(anc, 0, (H, W))$ ▷ H, W : Hieght and Width of feature space
    $anc\_list.append(anc)$
  $anc\_list = sorted(anc\_list)$
  $pre\_nms\_list = anc\_list[0 : 2000]$
  $nms\_list = pre\_nms\_list \geq 0.7$
  return $nms\_list[0 : 200]$

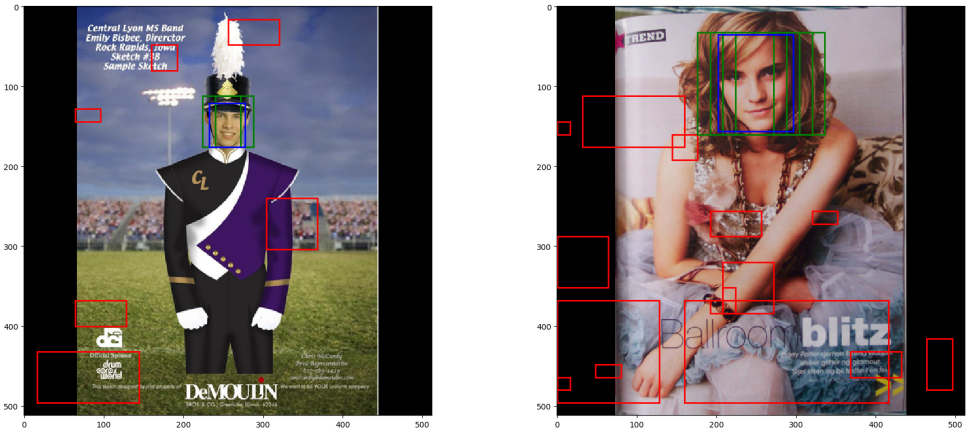

Figure 4: Output of the Anchor Target layer

Algorithm 3 Overview of the Anchor Target layer

INPUT : feature maps, ground truth boxes
RESULT : labels, targets

procedure PROPOSAL_LAYER(features, gt_box)
  for all $i_{h,w} \in features$ **do**
    $anc = generate\_anchors(cls\_prob.shape)$
    if $anc \notin [(0,0),(H,W)]$ **then**
      $remove\ anc$
    else
      $anc\_list.append(anc)$

$$labels = \begin{cases} 1 & where\ anc.IOU \geq 0.7 \\ 0 & where\ anc.IOU \leq 0.3 \\ -1 & otherwise \end{cases}$$

$$labels\_default = \begin{cases} 1 & where\ anc.IOU = max(anc[< all\ gt\_box >]) \\ -1 & otherwise \end{cases}$$

  if $length(labels) > THRESHOLD$ **then**
    $subsample(labels)$
  $targets = get\_targets(anc\_list[labels \neq -1])$
  return $labels,\ targets$

3.1.5 Proposal Target Layer

The proposal target layer is used while training the end-to-end model to help the model learn better ROI examples by adding the Ground Truth boxes along with the generated ROIs of the RPN. It also tries to maintain a more natural foreground-background ratio of 1:3 to help the model learn real-world scenarios better.

Algorithm 4 Overview of the Proposal Target layer

INPUT : ROIs, gt_boxes
RESULT : ROIs, bounding_box

procedure PROPOSAL_TARGET_LAYER(ROIs, gt_box)
  $all\_proposals = ROIs + gt\_box$
  $overlaps = all\_proposals \otimes gt\_box$ ▷ Cross product for IOUs
  $fg = overlaps[overlaps > 0.7]$ ▷ Positive anchors
  $bg = overlaps[overlaps < 0.3]$ ▷ Negetive anchors
  $bbox\_targets = get\_targets(fg)$
  return $fb, bg, bbox\_targets$

3.1.6 Position Sensitive Pooling Layer

The PS Pooling layer pools the Position Sensitive Score maps which are generated by the R-FCN network. Ideally, the network has two modules of $K^2 \times (C+1)$ channels where K represents the square grid dimension and C represents the number of classes other than the background. This is a very important consideration for our loss function which needs to be shear invariant as the ROIs are not always perfect squares.

The PS Pooling layer is the only module not developed from scratch in our implementation as we have implemented all of the other modules from scratch by optimizing the pipeline better. This component is a native C CUDA component that interacts with our Python model.

3.1.7 Average Pooling Layer

In most of the R-FCN architectures, only global average pooling as been implemented as they usually involve multiple classes to be predicted. This operation is flawed when faces are involved as different positions of the grid could contribute differently to the overall probability that a face exists in that image. Usually, the grid locations for the eyes are paid more attention to in face recognition tasks. Hence, we can apply weights to each region before getting the average for both the classification and regression Position Sensitive branches.

$$y_i = \frac{1}{N^2} \sum_{j=1}^{N^2} w_j x_{i,j}$$

The weighted average pooling layer can be replaced by a simple convolution layer with a 3×3 kernel with zero padding where the weights are learned as a part of the convolution back propagation. This produces a single score for each of the 200 ROIs generated by our model.

3.2 Vectorized Non-Max Suppression

NMS is a greedy algorithm that tries to remove overlapping anchor boxes generated if they both cover a significant portion of the same region. This algorithm is now performed in a traditional iterative manner in other implementations. Our implementation used a vectorized version of the NMS algorithm which was designed specifically for this implementation.

3.2.1 Improvements made by the vectorized implementation

3.3 Loss

There are two losses used in the network, one for classification of proposals into face or no face and regression loss for movement of bounding box to match the ground truth.

3.3.1 Classification Loss

Classification loss is calculated and the Binary Cross Entropy loss between the labels and the prediction scores obtained from the anchor target layer.

$$BCE\ Loss = -\Big(y \log(\hat{y}) + (1-y) \log(1-\hat{y})\Big)$$

3.3.2 Smooth L1 Loss

For regression, we have the x, y coordinates of the top left corner of the target and original bounding box denoted by T_x, T_y, O_x, O_y respectively and the width/height of the target and original bounding box by T_w, T_h, O_w, O_h respectively. Then, the regression targets (coefficients of the function that transform the original bounding box to the target box) are given as,

$$T_x = \frac{(T_x - O_x)}{O_w}$$
$$T_y = \frac{(T_y - O_y)}{O_h}$$
$$T_w = \log \frac{T_w}{O_w}$$
$$T_h = \log \frac{T_h}{O_h}$$

Using above targets, we calculate the loss as the output of the smooth L1 loss function or its variant, Huber loss function.

$$smooth\ L_1 = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if\ |(y - \hat{y})| < \delta \\ \delta(y - \hat{y}) - \frac{1}{2}\delta & otherwise \end{cases}$$

Our loss function needs to be shear invariant as the aspect ratio for actual face ROIs are not perfect squares but are converted to squares to ensure a uniform feature map. Natural Log is used so that it provides an easier and more natural inverse conversion with exponential while transforming the ROIs. Figure 5 highlights this point.

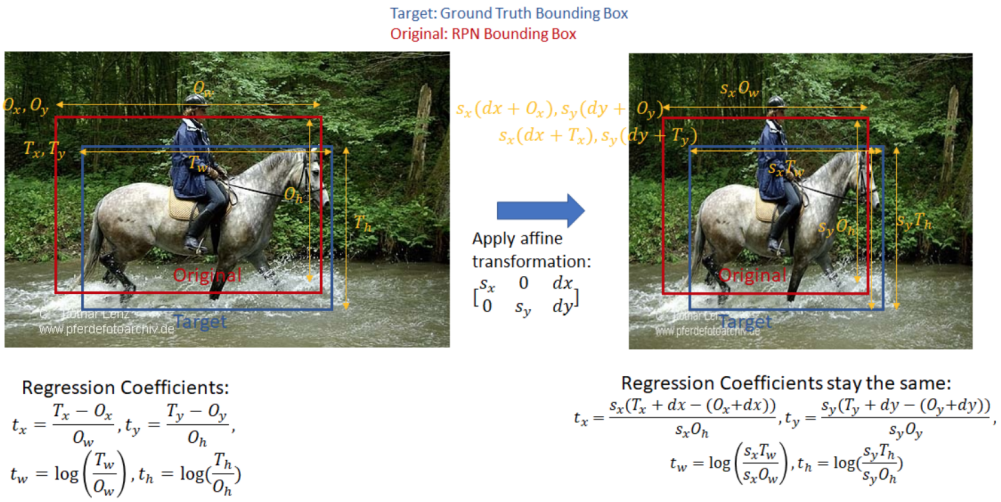

Figure 5: Shear Invariance

4 Experiments

4.1 Framework tweaks

4.1.1 Reusing Backbone Layers

We kept the base architecture close to original paper. However, we added another convolutional layer after the features are passed on to the RPN layer. This convolution layer is the layer 4 from the ResNet[10] backbone as seen in Figure 3.

This conversion proved to be absolutely essential as models trained using the ImageNet data set usually learn to be a very generic feature extractor for objects of multiple classes. Since, we freeze the feature extractor layer during training, we need an additional ResNet[10] BasicBlock to learn the feature conversions exclusively for faces.

4.1.2 Weighted Average Pooling

After acquiring the position sensitive scores for any ROI, we perform voting (Average Voting) on the output to determine the most probable class (face or no face) for a given region. We propose an additional 3×3 convolution to act as a learnable weight to increase performance.

4.2 Coding Considerations

We experimented with the following implementations to help decrease execution time. Starting with simple sequential programming, we attempted to vectorize our code to make use of the GPU where available or make use of standard optimized libraries. Table 3 highlights the runtimes observed for the configurations.

4.2.1 Naive sequential programming

This was the first attempt at completing the pipeline where multi-level nested loops were used. As the aim was to simply complete the pipeline, runtime for this phase was quite high.

4.2.2 Complete Torch implementation

After completion of a first draft of the pipeline, the code was vectorized using torch functions. This significantly improved the runtime of the program.

4.2.3 Complete Numpy implementation

We attempted binding all the deterministic and repetitive logic in numpy which significantly increased performance. However, a fundamental flaw in this approach was that the torch linkages were no longer maintained and back-propogation was not possible and PyTorch could not see the flow of data through numpy.

4.2.4 Complementary Torch-Numpy implementation

To combat the flaw mentioned above, we found a solution in other implementations wherein the tensor is detached from the flow and a copy is made. This copy can be interacted using any library as we see fit without damaging the flow of the network. Once we have performed the necessary calculations, we may simply insert this data in the original copy of the tensor and processed on.

Style	**Runtime**
Naive Sequential	$\geq$ 10 sec. / image
Torch	4.5 - 5 sec. / image
Numpy	2 - 2.5 sec. / image
Torch+Numpy	1.3 - 1.5 sec. / image

Table 1: Implementations tried and their observed runtimes per input image

4.3 Challenges

4.3.1 GPU Memory Issues

One of the major issues we faced was memory issue. We trained the architecture on a Google Cloud Platform instance with 100GB disk space and 14.4GB memory. The GPU rented was a Nvidia P100 with 16 GB memory. While the disk space and RAM were adequate, we kept running into run time errors of CUDA memory overflow as the default garbage collector failed to clear the cache memory in time, leading to execution end in a little over 2 epochs.We circumvented that issue by saving the model after every epoch and reloading the model to clear out the memory used by the previous epoch.

4.3.2 Pure Python Implementation

We were unable to create a pure python implementation as the references available online re-used CUDA C implementation. We had to reuse their code as the Position Senstive Pooling layer required a lot of attention to be implemented from scratch similar to the RPN module.

4.3.3 End to end BackProp

The entire Face R-FCN architecture switches back and forth between deterministic and stochastic components which involves a lot of tensor conversion to numpy and back. During this conversion, the gradient BackProp flow gets broken as numpy does not support variable based computational graphs. This implementation retains tensors when necessary for backprop and uses numpy primarily for loss calculation as numpy computation clearly is far more efficient than tensor computation.

5 Hyper parameters

5.1 Image dimensions and pre-processing

Due to a very large in-memory model size for BackProp, we were unable to train the images in their original 1024*1024 dimensions. We reduced the image scales to 256*256 which reduces the number of weights by 4 and enabled our model to fit into the GPU space provided.

5.2 Optimizer

Parameter	Value
Learning Rate	0.0001
Momentum	0.9
Weight Decay	0.0005

Table 2: Parameters involved for the SGD Optimizer

5.3 RPN

We have also implemented uniform example weighting as a normalization measure to ensure that the model learns enough from the foreground face examples. Weight initialization was also done on all of the learnable layers of the model with a mean=0 and stddev=0.01.

Config Setting	Value
ROIs generated Post-NMS	200
Pre-NMS ROIs	2000
NMS Positive Overlap Threshold	0.9
NMS Negative Overlap Threshold	0.3
FG Fraction for Anchor Target Layer	1/3

Table 3: Parameters involved for the SGD Optimizer

6 Results

6.1 WIDERFACE Easy set

Since we resampled images to a very small scale of 256*256, we trained the images only on large faces in the WIDERFACE dataset. These are faces which have a minimum dimension of 300*300.

6.2 Validation results

The model still suffers from GPU memory errors and hence only a few epochs of the model had run which was not enough for computing a solid AP score. However, the model did learn the face representations for a certain size of faces based on the anchors we gave. We believe that our model would perform with sufficient training time.

6.3 Losses generated

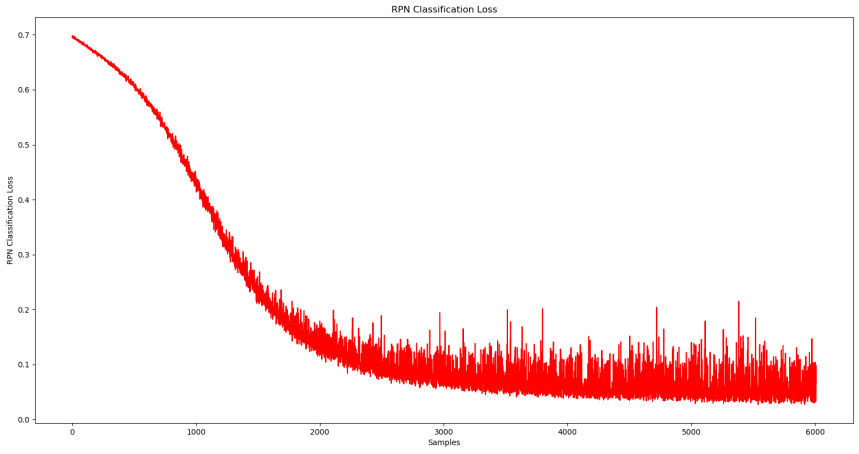

Figure 6: RPN Classification Loss

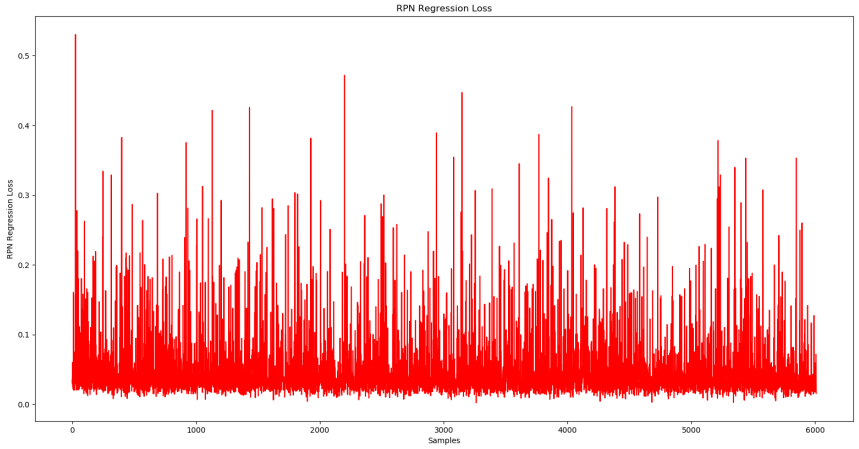

Figure 7: RPN Regression Loss

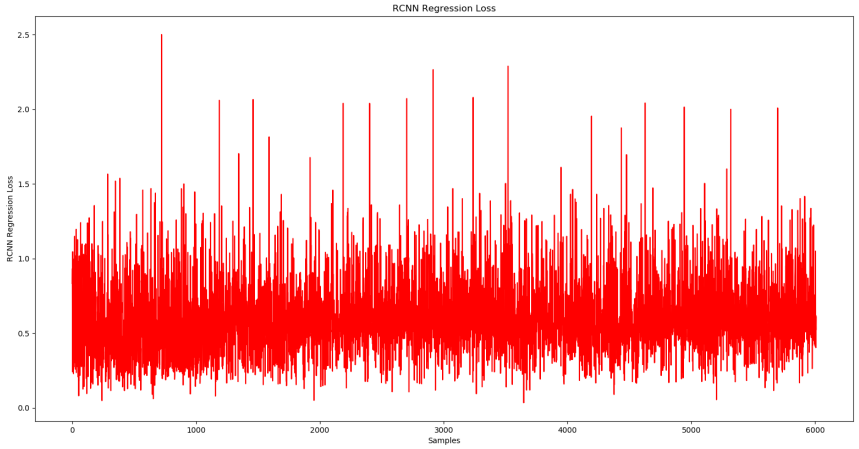

Figure 8: RCNN Regression Loss

6.4 Output images

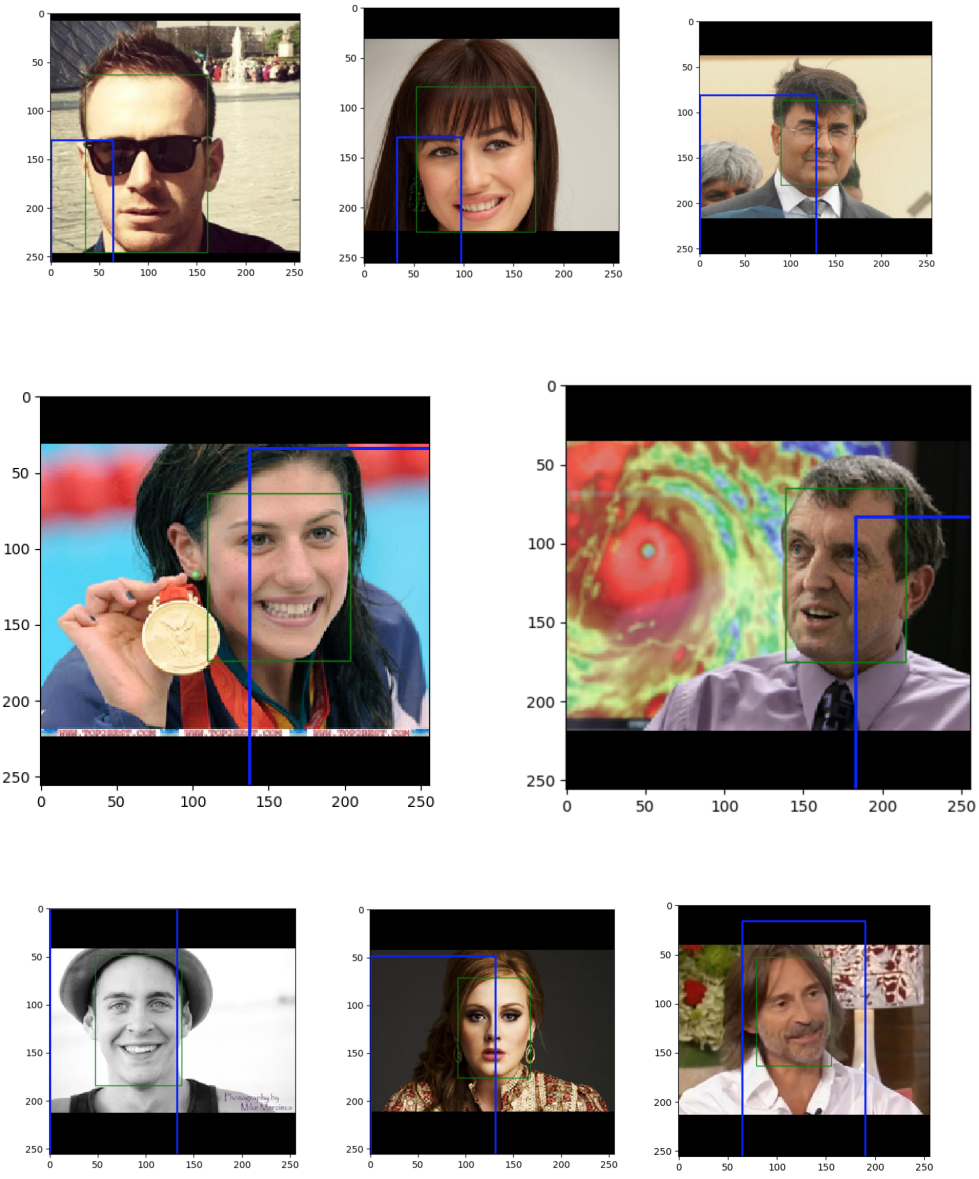

Figure 11: Results from training of the model.

The model is now able to produce bounding boxes that encapsulate a much bigger portion of the image along with the face. This is mainly due to the model learning the classification quicker when compared to bounding box regression as there are just two classes for the model to distinguish.

6.5 Inference Algorithm

Since the model predicts 200 candidate ROIs we perform three steps to obtain the faces: 1. Apply the class specific bounding box transformations on the ROIs 2. Perform NMS on the 200 candidate ROIs to filter repetitive bounding boxes 3. Filter those ROIs that have Face probability ¿ 0.5 4. Plot the ROIs on the final image by upscaling the box

Algorithm 5 Steps for Inference

1: Apply the class specific bounding box transformations on the ROIs
2: Perform NMS on the 200 candidate ROIs to filter repetitive bounding boxes
3: Filter those ROIs that have Face probability > 0.5
4: Plot the ROIs on the final image by upscaling the box

7 Conclusion

We presented a modified implementation of the Face R-FCN that shows significant promise in terms of training time (0.3 seconds per image for a 256*256 image) and understanding face representations better which need to be shear invariant due to the restriction of the square grid size in the PSROI pooling layer. We have also provided with a detailed understanding of how each layer contributes to the overall output, debunking the common myth that a fully learnable network is a black box. The architecture can also be used to train smaller faces with a GPU with higher capacity as the learnable parameters increase by the order of millions for bigger feature maps.

Our modifications proposed can also be further extended with recent advancements like SNIPER[15] and Deformable Convolutions[16] to explore the possiblity of understanding faces that are scale, pose and rotation invariant which are usually the false-negatives in the outputs currently.

References

[1] https://github.com/jmerkow/R FCN.rsna2018, "A pytorch implementation of r-fcn/couplenet."

[2] https://github.com/arvindmvepa/R FCN.pytorch, "A pytorch implementation of r-fcn/couplenet."

[3] Y. Wang, X. Ji, Z. Zhou, H. Wang, and Z. Li, "Detecting faces using region-based fully convolutional networks," *CoRR*, vol. abs/1709.05256, 2017.

[4] S. Yang, P. Luo, C. C. Loy, and X. Tang, "Wider face: A face detection benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] V. Jain and E. Learned-Miller, "Fddb: A benchmark for face detection in unconstrained settings," Tech. Rep. UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[6] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015.

[7] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[8] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: object detection via region-based fully convolutional networks," *CoRR*, vol. abs/1605.06409, 2016.

[9] A. Shrivastava, A. Gupta, and R. B. Girshick, "Training region-based object detectors with online hard example mining," *CoRR*, vol. abs/1604.03540, 2016.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[11] http://www.telesens.co/2018/03/11/object-detection-and-classification-using-r cnns/, "Object detection and classification using r-cnns."

[12] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[13] C. Peng, X. Zhang, G. Yu, G. Luo, and J. Sun, "Large kernel matters - improve semantic segmentation by global convolutional network," *CoRR*, vol. abs/1703.02719, 2017.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[15] B. Singh, M. Najibi, and L. S. Davis, "SNIPER: efficient multi-scale training," *CoRR*, vol. abs/1805.09300, 2018.

[16] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," *CoRR*, vol. abs/1703.06211, 2017.