DSFD with VGG-16 / EfficientNet-B0, B1

Group C

Approaches

- R-CNN
  Selective search for regions. Huge time to train network.

- Fast R-CNN
  Feature map generation. Faster than R-CNN.

- DSFD
  Robust using FEM. Anchor matching and experiments.

Introduction

- Face detection a popular research area in Computer Vision.
- CNNs yields better results
- Regional face detection and Single Shot Detectors
- Our focus would be on Dual Shot Detectors

Datasets

- Face Detection Dataset and Benchmark
  Approx. 2845 images with face images > 5000.
  Achieves state of art and bounding boxes for Face anchors.

- WIDERFACE
  Consists of 393,703 images.
  Large variations.
  Training set.

Backbones

- EfficientNet

  - Huge advantage for scaling

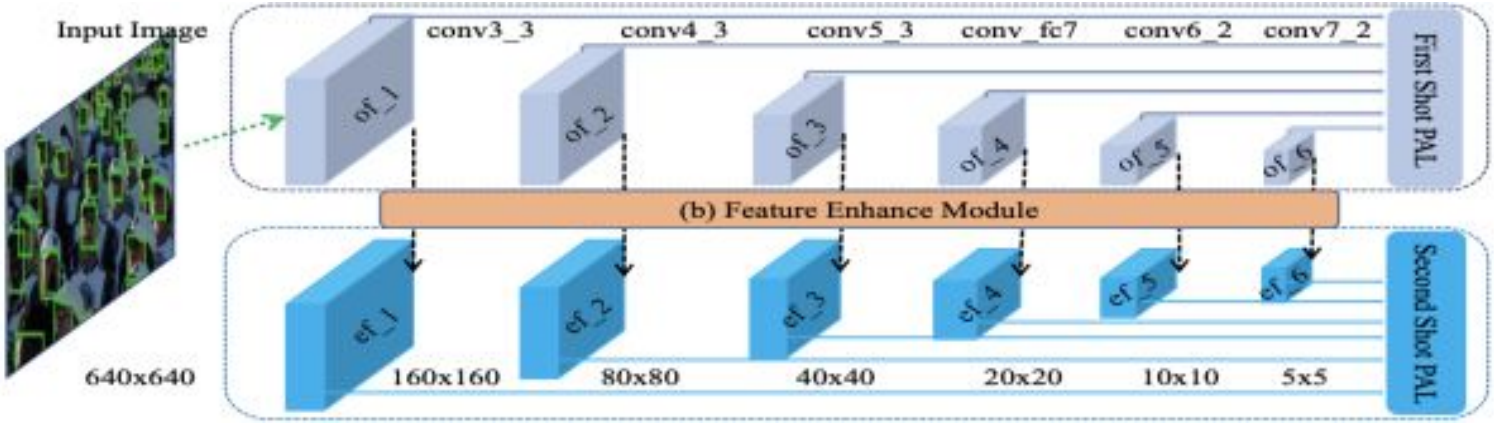
  - Can increase efficiency in some cases

  - Compound scaling for increased accuracy

Dual Shot Face Detection

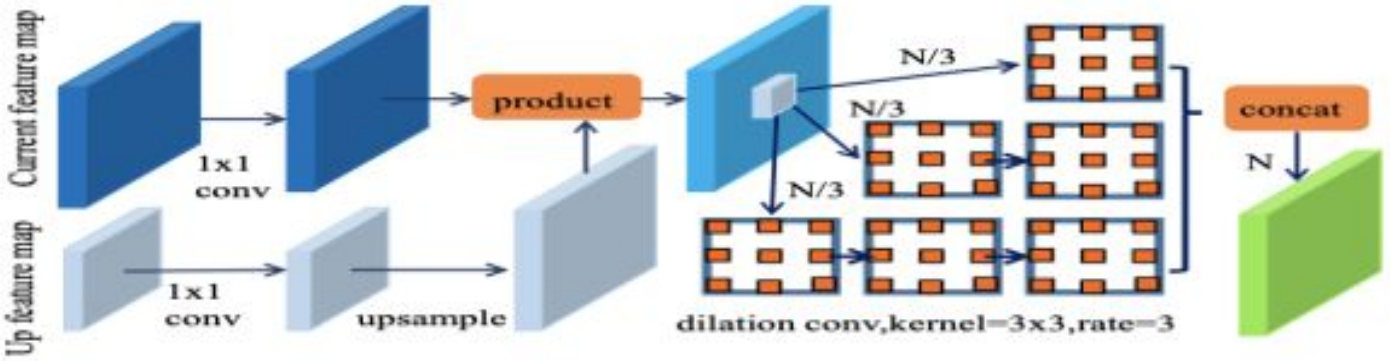
Feature Enhance Module (FEM)

- Enhance the original feature maps more robust and discriminable
- Extend single shot detector to dual shot detector
- Utilizes different dimension information



Dual Shot Face Detector - FEM

- Use 1 x 1 convolutional kernel to normalize the feature maps
- Up-sample upper feature maps to do element-wise product with the current ones
- Split the feature maps to three parts, followed by three sub-networks containing different numbers of dilation convolutional layers



Dual Shot Face Detector - PAL

Progressive Anchor Loss (PAL)

$$\mathcal{L}_{SSL}(p_i, p_i^*, t_i, g_i, a_i) = \frac{1}{N_{conf}}(\Sigma_i L_{conf}(p_i, p_i^*) + \frac{\beta}{N_{loc}}\Sigma_i p_i^* L_{loc}(t_i, g_i, a_i))$$

$$\mathcal{L}_{FSL}(p_i, p_i^*, t_i, g_i, sa_i) = \frac{1}{N_{conf}}\Sigma_i L_{conf}(p_i, p_i^*) + \frac{\beta}{N_{loc}}\Sigma_i p_i^* L_{loc}(t_i, g_i, sa_i)$$

$$\mathcal{L}_{PAL} = \mathcal{L}_{FSL}(sa) + \lambda\mathcal{L}_{SSL}(a).$$

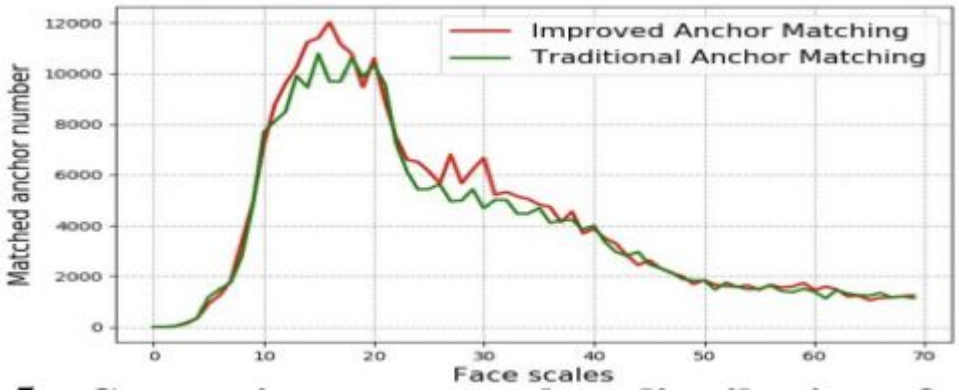
Improved Anchor Matching

- Remaining Probability (60%)
- Data augmentation strategy similar to the one used in SSD
  - Random crop of the input image (eventually with a minimum overlap with an object). If an object is present, the crop is centered around it.
  - Resizing to a fixed size.
  - Random horizontal flip.
  - Photo-metric distortions.
- Match anchors and ground truth faces as far as possible to provide better initialization for the regressor

Improved Anchor Matching

Improved Anchor Matching (IAM)

- IAM targets on addressing the contradiction between the discrete anchor scales and continuous face scales
- Augmented by $S_{input} * S_{face}/S_{anchor}$



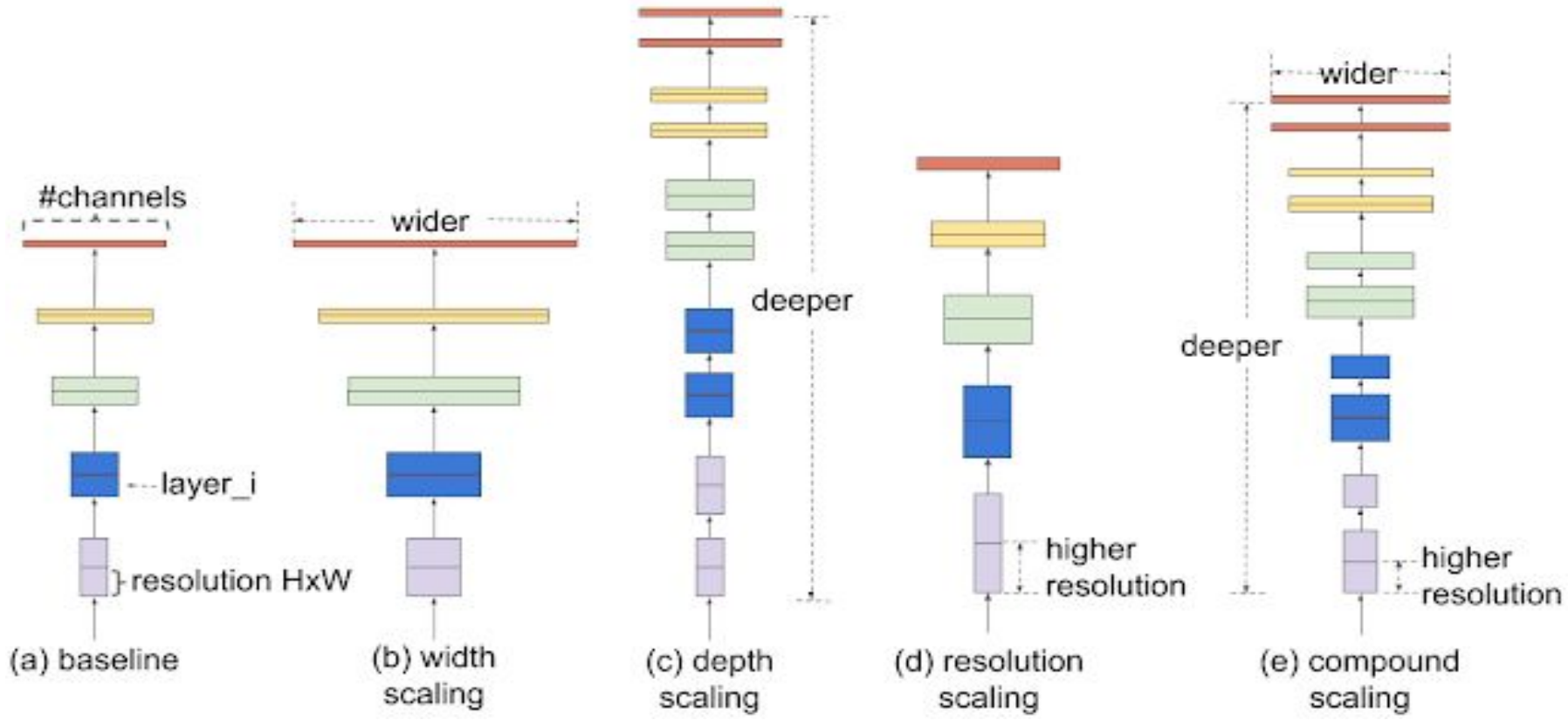
EfficientNet

- A CNN architecture which uses a principled method to scale up a CNN to obtain better accuracy and efficiency.

Idea behind Model Scaling:

- Change model depth
- Change model width
- Input image resolution

Scaling up any dimension of network width, depth, or resolution improves accuracy, but the accuracy gain diminishes for bigger models.
In order to pursue better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution during ConvNet scaling.


#channels
layer_i
resolution HxW
(a) baseline
wider
(b) width scaling
deeper
(c) depth scaling
higher resolution
(d) resolution scaling
wider
deeper
higher resolution
(e) compound scaling

Compound Scaling

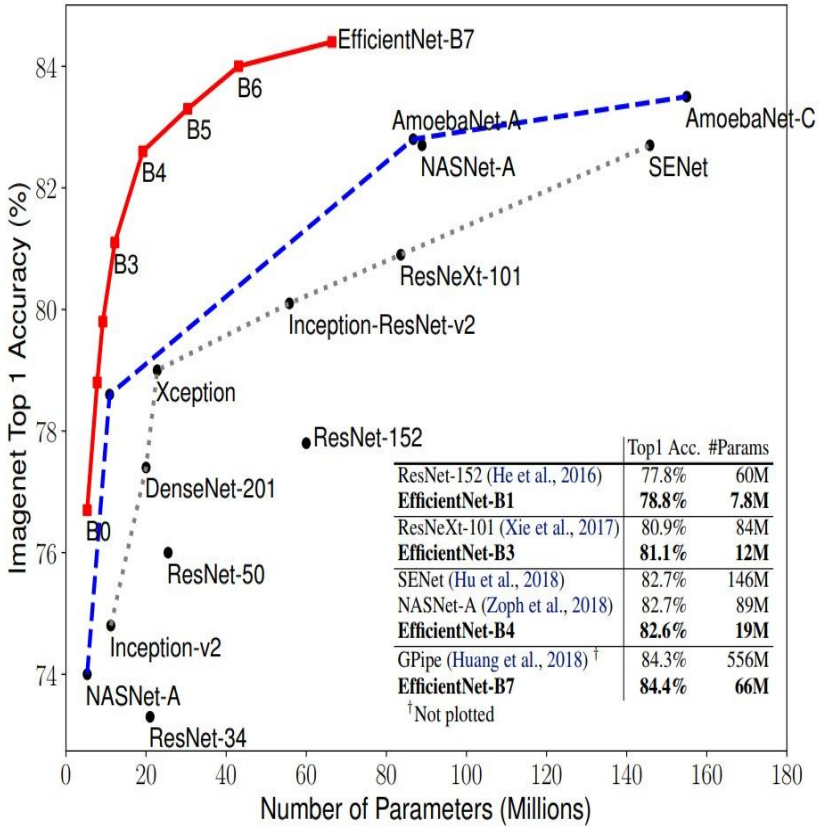
$$\text{depth: } d = \alpha^\phi$$
$$\text{width: } w = \beta^\phi$$
$$\text{resolution: } r = \gamma^\phi$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

The FLOPS consumed in a convolutional operation is proportional to d, w^2, and r^2, and this fact is reflected in the above equation. The authors restrict $\alpha \cdot \beta^2 \cdot \gamma^2$ to 2 so that every new ϕ, the FLOPs needed goes by up 2^ϕ

Φ is the compound coefficient that is user specified and controls the resources available

α, β, and γ distribute the resources to depth, width, and resolution respectively.

1. Fix $\phi = 1$.
2. Assume the resource available at any step of scaling is twice of the resource at the previous step.
3. Do a small grid search over α, β, and γ such that the constraint in the above equation is not violated.
4. The authors found the parameters $\alpha = 1.2$, $\beta = 1.1$, $\gamma = 1.15$ to work the best.
5. Fix α, β, and γ as constants and scale up EfficientNet-B0 with different ϕ to obtain new scaled networks EfficientNet-B1 to B7.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**78.8%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.1%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.6%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

Implementation Details

- VGG-16, ResNet-51, ResNet-101, ResNet151, EfficientNet (B0 - B7)
- Done with VGG-16, **EfficientNet B0, and EfficientNet B1**
- EfficientNet B2 - B7 Training in Progress

Implementation Details - VGG

- Backbone: Feature maps extraction
- VGG: Conv3_3, Conv4_3,Conv5_3, fc7, Conv6_2, Conv7_2
- Generate six feature enhanced maps

Implementation Details - 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)

Switching to EfficientNet

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

Switching to EfficientNet

- Stage 2 to 8 to extract the six original feature maps as the first shot
- Generate the six feature enhanced maps as the second shot
- EfficientNet B0 Parameters (Width, Depth, Resolution):
  - 1.0, 1.0, 1.0
- EfficientNet B1 Parameters (Width, Depth, Resolution):
  - 1.0, 1.1, 1.07

EfficientNet Configuration

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]
```

EfficientNet - Feature Extraction

- Continuously tune parameters to change layers that we used to extract 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)
```

Training

- Pre-trained Model
- VGG: Trained on the weights without any fully connection layers
- EfficientNet: No proper tre-trained model found
  - 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)

Challenges

- GPU Memory: 100 epoches can’t be done all at once
- New strategy
  - Save intermediate weights of the model
  - Update code to help us resuming the training process from the intermediate model
  - No more crashes caused by hardware
  - Getting the weight for every 5000 iterations (iterations = start_epoch * len(train_dataset) / atch_size), and update the globally best model wrights during testing when we get a smaller loss

Results & Discussion

- VGG: Works well for both big and small face detection
- EfficientNet: Works well for big face detection, but not as good as we expect for small face detection

Results & Discussion

AFW	PASCAL	FDDB
99.89	99.11	98.3

Training Log

```
Timer: 2.0992
epoch:57 || iter:371250 || Loss:4.3228
->> pal1 conf loss:1.9291 || pal1 loc loss:2.6433
->> pal2 conf loss:1.9272 || pal2 loc loss:2.0671
->>lr:5e-07
Timer: 2.0977
epoch:57 || iter:371260 || Loss:4.3269
->> pal1 conf loss:2.9164 || pal1 loc loss:3.3814
->> pal2 conf loss:2.8965 || pal2 loc loss:3.2235
->>lr:5e-07
```

```
Timer: 0.6077
epoch:59 || iter:385050 || Loss:5.2961
->> pal1 conf loss:0.6535 || pal1 loc loss:0.8498
->> pal2 conf loss:1.2552 || pal2 loc loss:1.0875
->>lr:5e-07
Timer: 0.6053
epoch:59 || iter:385060 || Loss:5.2959
->> pal1 conf loss:2.0700 || pal1 loc loss:1.5921
->> pal2 conf loss:1.2817 || pal2 loc loss:1.0054
->>lr:5e-07
```

```
Timer: 1.2170
epoch:93 || iter:301170 || Loss:5.8746
->> pal1 conf loss:1.4951 || pal1 loc loss:1.5199
->> pal2 conf loss:1.4763 || pal2 loc loss:1.8592
->>lr:5e-07
Timer: 1.2073
epoch:93 || iter:301180 || Loss:5.8821
->> pal1 conf loss:1.5300 || pal1 loc loss:1.0568
->> pal2 conf loss:2.0459 || pal2 loc loss:2.6761
->>lr:5e-07
```

```
Timer: 0.7764
epoch:26 || iter:171480 || Loss:5.1847
->> pal1 conf loss:1.9178 || pal1 loc loss:1.8624
->> pal2 conf loss:1.9664 || pal2 loc loss:1.7359
->>lr:5e-07
Timer: 0.7669
epoch:26 || iter:171490 || Loss:5.1832
->> pal1 conf loss:0.8498 || pal1 loc loss:0.5390
->> pal2 conf loss:0.7172 || pal2 loc loss:0.7963
->>lr:5e-07
```

Result for DSFD using VGG16 as backbone

Result for DSFD using EfficientNet B1 as backbone

Demo

Causes & Solutions

- Original Design of Feature extraction
- Tried layers with higher resolution
  - loss changed to 5.1 when epoch equals to 26
- Pre-ancher strategy?

Github

https://github.com/mexiQQ/DSFD-VGG16-EfficientNet

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.