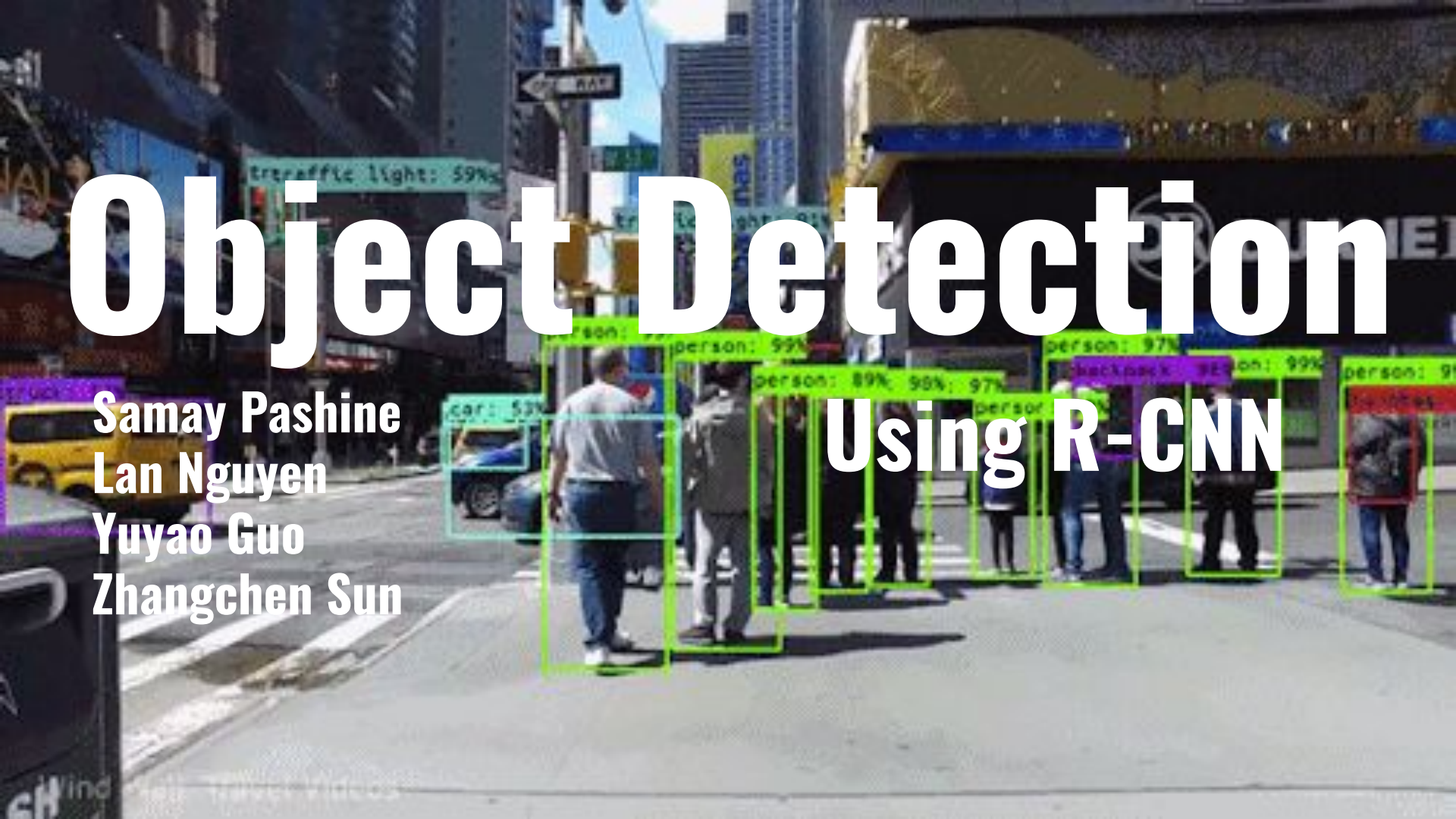


Object Detection Using R-CNN

Samay Pashine
Lan Nguyen
Yuyao Guo
Zhangchen Sun

Agenda

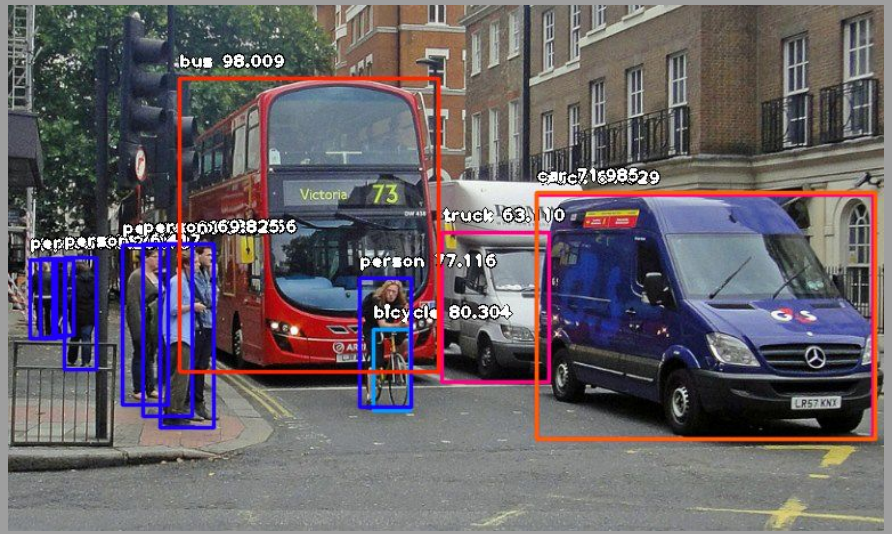
What is Object Detection ?

R-CNN

Implementations

Conclusion

What is Object Detection?



The Object detection is a computer vision technique that allows us to identify and locate objects in an image or video.

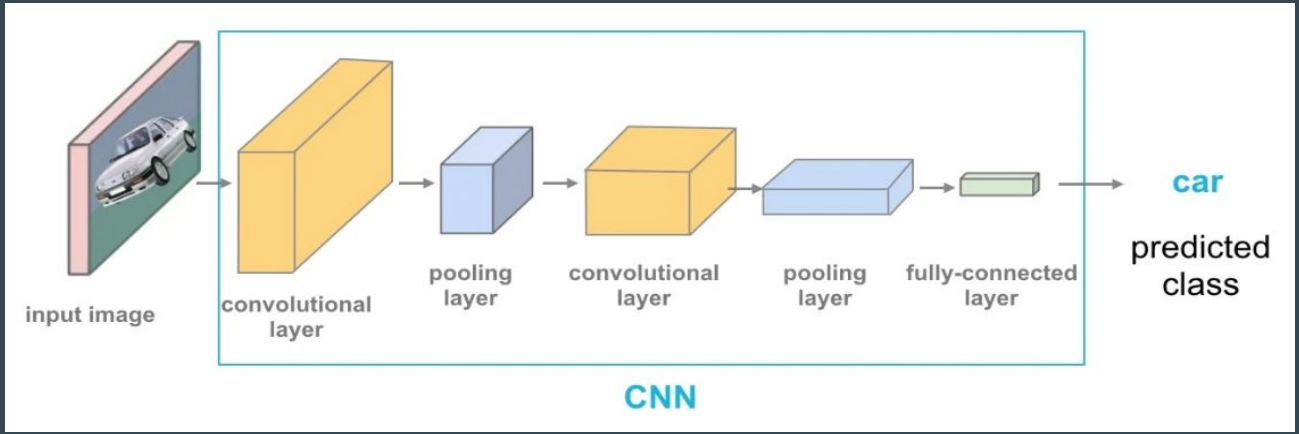
With this kind of identification and localization, object detection can be used to count objects in a scene, determine and track their precise locations, all while accurately labeling them.

What is a **Convolutional Neural Network** (CNN)?

A Convolutional Neural Network (CNN) is a type of artificial neural network used in image recognition and processing that is optimized to process pixel data. Therefore, Convolutional Neural Networks are the fundamental and basic building blocks for computer vision.

- **Convolutional layer**
- **Pooling layer**
- **Fully connected layer**

What is a **Convolutional Neural Network** (CNN)?



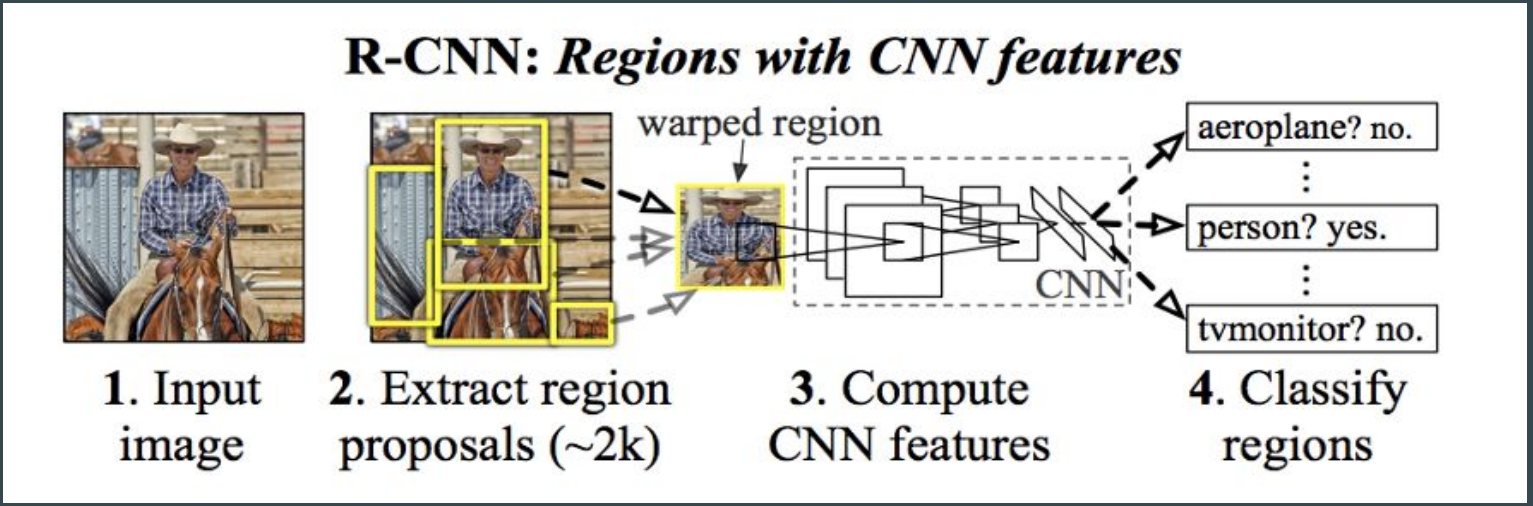
Combining these layers of a CNN enables the designed neural network to learn how to identify and recognize the object of interest in an image. Simple Convolutional Neural Networks are built for image classification and object detection with a single object in the image.

What is R-CNN?

R-CNN or RCNN, stands for Region-Based Convolutional Neural Network, it is a type of machine learning model that is used for computer vision tasks, specifically for object detection.

How does R-CNN work?

The **R-CNN** utilizes bounding boxes across the object regions, which then evaluates convolutional networks independently on all the Regions of Interest (ROI) to classify multiple image regions into the proposed class. The R-CNN architecture was designed to solve image detection tasks.



Rich feature hierarchies for accurate object detection and semantic segmentation, CVPR14

R-CNN for object detection

1. Region proposal generation
2. Bounding box Unification
3. Feature extraction from Region Proposals (OpenCV)
4. VGG for object classification

1. Region Proposal Generation

The first step, we use selective search to generate region proposals

Greedy algorithm:

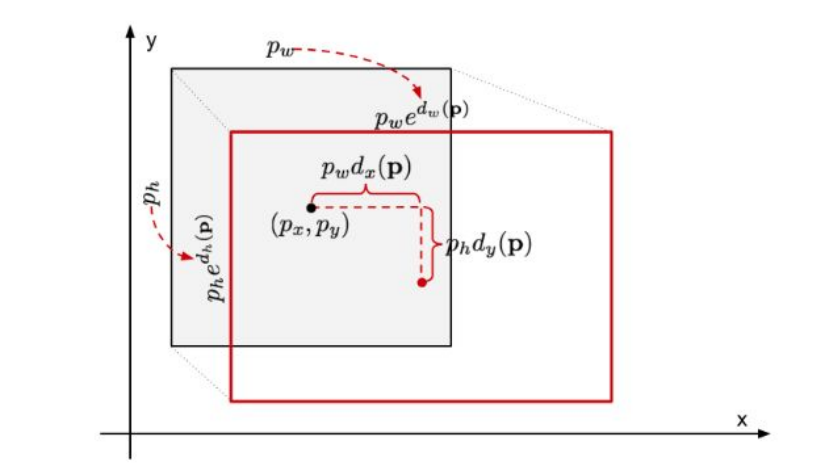
1. From set of regions, choose two that are most similar.
2. Combine them into a single, larger region.
3. Repeat until only one region remains.

```
self.sed_detector.setBaseImage(img)
self.sed_detector.switchToSelectiveSearchFast()
results = self.sed_detector.process()
```

2. Bounding Box Unifications (IoU)

In order to improve localization performance, we can use bounding-box regression step to correct the predicted object (airplane) location. For this step, we have used the **OpenCV** segment detector tool to get the bounding boxes.

Once we have all the bounding boxes, we find **Intersection of Union (IoU)** to get one prime box encapsulating the object.



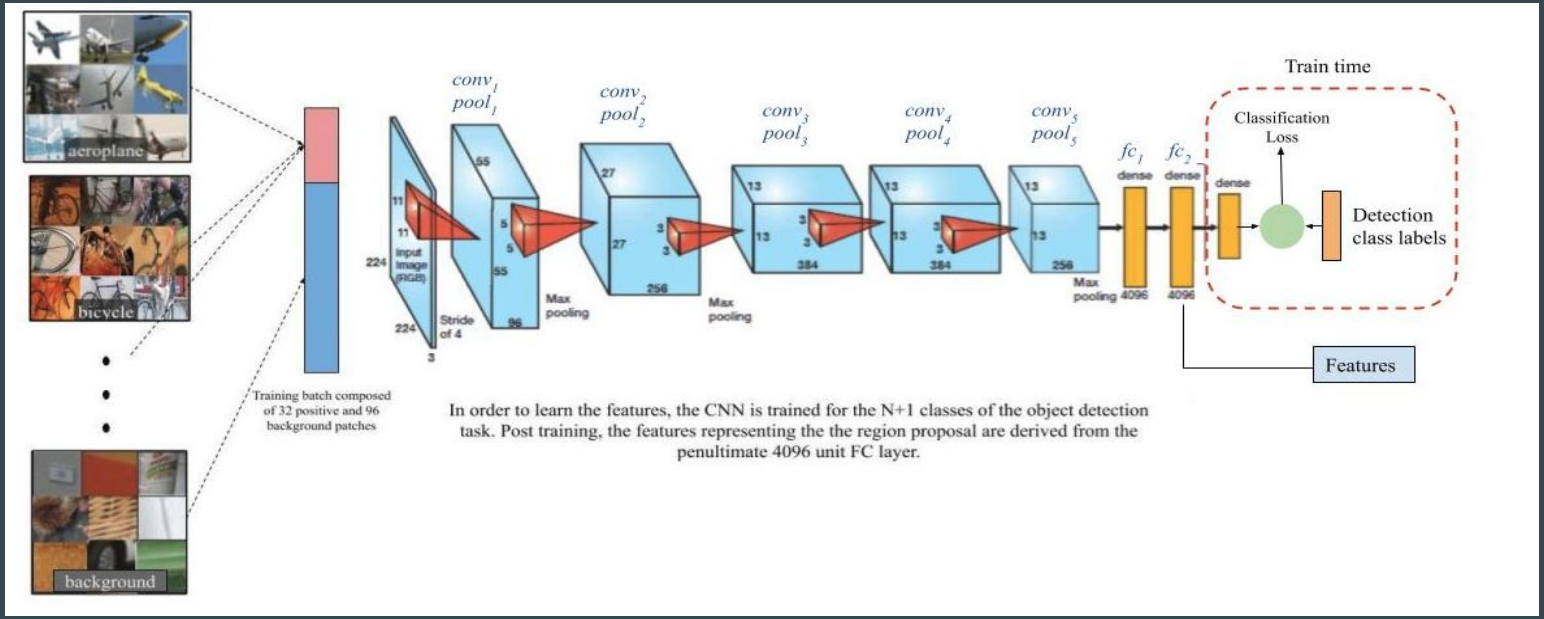
3. Feature extraction from Region Proposals

```
for e,result in enumerate(detector_results):                    ⚠ 2 ⚠ 32 ✕ 10
    if e < 2000 and flag == 0:
        for bb_value in bb_values:

            x,y,w,h = result
            iou = IOU(bb_value, {"x1": x, "x2": x + w, "y1": y, "y2": y + h})

            if counter < 30:
                if iou > 0.70:
                    timage = imout[y: y + h, x: x + w]
                    resized = cv2.resize(timage, (224, 224), interpolation = cv2.INTER_AREA)
                    train_images.append(resized)
                    train_labels.append(1)
                    counter += 1
            else:
                fflag =1

            if falsecounter < 30:
                if iou < 0.3:
                    timage = imout[y:y+h,x:x+w]
                    resized = cv2.resize(timage, (224, 224), interpolation = cv2.INTER_AREA)
                    train_images.append(resized)
                    train_labels.append(0)
                    falsecounter += 1
            else:
                bflag = 1


        if fflag == 1 and bflag == 1:
            # print("inside")
            flag = 1
```

The first step of the R-CNN pipeline is the generation of ‘region proposals’ in an image that could belong to a particular object.

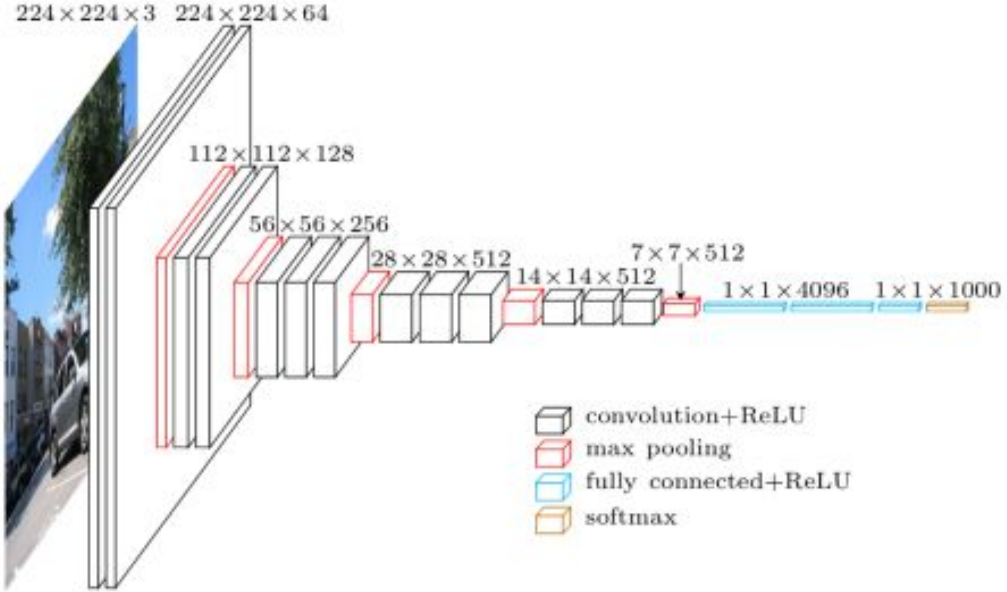
At the end of this step in the pipeline, we can generate a 4096 dimensional feature vector from each of the 2000 region proposals for every image using a Convolutional Neural Network (CNN).

3. Feature extraction from Region Proposals



The final output of Step 2: After training, the final classification layer is removed and a 4096 dimensional feature vector is obtained from the penultimate layer of the CNN for each of the 2000 region proposals.

4. VGG for object classification




```
# Base model and additional fully-connected layer.
self.base_model = VGG16(weights='imagenet', include_top=True)
self.freeze_layer(base_layer_train_flag)
print("\n[INFO]. VGG-16 Model Summary : ")
self.base_model.summary()

X = self.base_model.layers[-2].output
```

VGG models are a type of CNN Architecture proposed by Karen Simonyan & Andrew Zisserman of Visual Geometry Group (VGG), Oxford University. VGG16 is one of the most popular models which is able to classify 1000 different categories with 92.7% accuracy.

For this part we are only working with airplane object (i.e. binary classification) but we can also use this with COCO dataset.

Training Data

```
Epoch 3: val_loss improved from 0.11203 to 0.05846, saving model to ieeercnn_vgg16_1.h5
10/10 [==============================] - 216s 22s/step - loss: 0.1832 - accuracy: 0.9283 - val_loss: 0.0585 - val_accuracy: 1.0000
Epoch 4/10
10/10 [==============================] - ETA: 0s - loss: 0.1877 - accuracy: 0.9469
Epoch 4: val_loss did not improve from 0.05846
10/10 [==============================] - 226s 22s/step - loss: 0.1877 - accuracy: 0.9469 - val_loss: 0.2207 - val_accuracy: 0.9375
Epoch 5/10
10/10 [==============================] - ETA: 0s - loss: 0.2473 - accuracy: 0.9156
Epoch 5: val_loss did not improve from 0.05846
10/10 [==============================] - 220s 22s/step - loss: 0.2473 - accuracy: 0.9156 - val_loss: 0.2800 - val_accuracy: 0.8438
Epoch 6/10
10/10 [==============================] - ETA: 0s - loss: 0.1937 - accuracy: 0.9219
Epoch 6: val_loss did not improve from 0.05846
10/10 [==============================] - 222s 22s/step - loss: 0.1937 - accuracy: 0.9219 - val_loss: 0.1538 - val_accuracy: 0.9688
Epoch 7/10
10/10 [==============================] - ETA: 0s - loss: 0.1357 - accuracy: 0.9500
Epoch 7: val_loss did not improve from 0.05846
10/10 [==============================] - 222s 23s/step - loss: 0.1357 - accuracy: 0.9500 - val_loss: 0.1380 - val_accuracy: 0.9688
Epoch 8/10
10/10 [==============================] - ETA: 0s - loss: 0.1745 - accuracy: 0.9531
Epoch 8: val_loss did not improve from 0.05846
10/10 [==============================] - 219s 22s/step - loss: 0.1745 - accuracy: 0.9531 - val_loss: 0.0796 - val_accuracy: 0.9688
Epoch 9/10
10/10 [==============================] - ETA: 0s - loss: 0.0787 - accuracy: 0.9761
Epoch 9: val_loss did not improve from 0.05846
10/10 [==============================] - 204s 21s/step - loss: 0.0787 - accuracy: 0.9761 - val_loss: 0.0876 - val_accuracy: 0.9844
Epoch 10/10
10/10 [==============================] - ETA: 0s - loss: 0.1353 - accuracy: 0.9625
Epoch 10: val_loss did not improve from 0.05846
10/10 [==============================] - 219s 22s/step - loss: 0.1353 - accuracy: 0.9625 - val_loss: 0.1413 - val_accuracy: 0.9688
```

Graphs and Examples

Figure : Loss graph of the model with the training and validation dataset.

Figure : Sample inference image to check the functioning of the model.

Attention layer

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 224, 224, 3)]	0	[]
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792	['input_1[0][0]']
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928	['block1_conv1[0][0]']
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0	['block1_conv2[0][0]']
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856	['block1_pool[0][0]']
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584	['block2_conv1[0][0]']
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0	['block2_conv2[0][0]']
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168	['block2_pool[0][0]']
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080	['block3_conv1[0][0]']
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080	['block3_conv2[0][0]']
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0	['block3_conv3[0][0]']
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160	['block3_pool[0][0]']
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808	['block4_conv1[0][0]']
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808	['block4_conv2[0][0]']
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0	['block4_conv3[0][0]']
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808	['block4_pool[0][0]']
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808	['block5_conv1[0][0]']
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808	['block5_conv2[0][0]']
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0	['block5_conv3[0][0]']
tf.reshape (TFOpLambda)	(None, 49, 512)	0	['block5_pool[0][0]']
attention (Attention)	(None, 49, 512)	0	['tf.reshape[0][0]', 'tf.reshape[0][0]', 'tf.reshape[0][0]']
tf.math.reduce_mean (TFOpLambda)	(None, 512)	0	['attention[0][0]']
dense (Dense)	(None, 2)	1026	['tf.math.reduce_mean[0][0]']

Figure : Architecture with attention layer

```
attention (Attention)          (None, 49, 512)      0           ['tf.reshape[0][0]',
                                                                 'tf.reshape[0][0]',
                                                                 'tf.reshape[0][0]']

tf.math.reduce_mean (TFOpLambd  (None, 512)         0           ['attention[0][0]']
a)

dense (Dense)                  (None, 2)            1026        ['tf.math.reduce_mean[0][0]']

==================================================================================================
Total params: 14,715,714
Trainable params: 7,080,450
Non-trainable params: 7,635,264
__________________________________________________________________________________________________
[INFO]. Starting the Training.
Epoch 1/10
10/10 [==============================] - ETA: 0s - loss: 2.0572 - accuracy: 0.5414
Epoch 1: val_loss improved from inf to 1.02860, saving model to ./model_weights.h5
10/10 [==============================] - 75s 7s/step - loss: 2.0572 - accuracy: 0.5414 - val_loss: 1.0286 - val_accuracy: 0.7872
Epoch 2/10
10/10 [==============================] - ETA: 0s - loss: 0.9243 - accuracy: 0.7906
Epoch 2: val_loss improved from 1.02860 to 0.30398, saving model to ./model_weights.h5
10/10 [==============================] - 75s 8s/step - loss: 0.9243 - accuracy: 0.7906 - val_loss: 0.3040 - val_accuracy: 0.8511
Epoch 3/10
10/10 [==============================] - ETA: 0s - loss: 0.8316 - accuracy: 0.8276
Epoch 3: val_loss did not improve from 0.30398
10/10 [==============================] - 68s 7s/step - loss: 0.8316 - accuracy: 0.8276 - val_loss: 0.3833 - val_accuracy: 0.8936
Epoch 4/10
10/10 [==============================] - ETA: 0s - loss: 0.7344 - accuracy: 0.8594
Epoch 4: val_loss improved from 0.30398 to 0.30229, saving model to ./model_weights.h5
10/10 [==============================] - 74s 7s/step - loss: 0.7344 - accuracy: 0.8594 - val_loss: 0.3023 - val_accuracy: 0.9149
Epoch 5/10
10/10 [==============================] - ETA: 0s - loss: 0.6044 - accuracy: 0.8621
Epoch 5: val_loss improved from 0.30229 to 0.24330, saving model to ./model_weights.h5
10/10 [==============================] - 68s 7s/step - loss: 0.6044 - accuracy: 0.8621 - val_loss: 0.2433 - val_accuracy: 0.8936
Epoch 6/10
10/10 [==============================] - ETA: 0s - loss: 0.5078 - accuracy: 0.8500
Epoch 6: val_loss did not improve from 0.24330
10/10 [==============================] - 74s 7s/step - loss: 0.5078 - accuracy: 0.8500 - val_loss: 0.5461 - val_accuracy: 0.8085
Epoch 7/10
10/10 [==============================] - ETA: 0s - loss: 0.3327 - accuracy: 0.8813
Epoch 7: val_loss did not improve from 0.24330
10/10 [==============================] - 74s 7s/step - loss: 0.3327 - accuracy: 0.8813 - val_loss: 0.4181 - val_accuracy: 0.8511
Epoch 8/10
10/10 [==============================] - ETA: 0s - loss: 0.4379 - accuracy: 0.8862
Epoch 8: val_loss improved from 0.24330 to 0.22781, saving model to ./model_weights.h5
10/10 [==============================] - 66s 7s/step - loss: 0.4379 - accuracy: 0.8862 - val_loss: 0.2278 - val_accuracy: 0.9362
Epoch 9/10
10/10 [==============================] - ETA: 0s - loss: 0.4273 - accuracy: 0.8621
Epoch 9: val_loss did not improve from 0.22781
10/10 [==============================] - 70s 8s/step - loss: 0.4273 - accuracy: 0.8621 - val_loss: 0.3300 - val_accuracy: 0.9149
Epoch 10/10
10/10 [==============================] - ETA: 0s - loss: 0.2914 - accuracy: 0.9241
Epoch 10: val_loss did not improve from 0.22781
10/10 [==============================] - 72s 7s/step - loss: 0.2914 - accuracy: 0.9241 - val_loss: 0.2756 - val_accuracy: 0.9574
[INFO]. Plotting the loss graph.
```

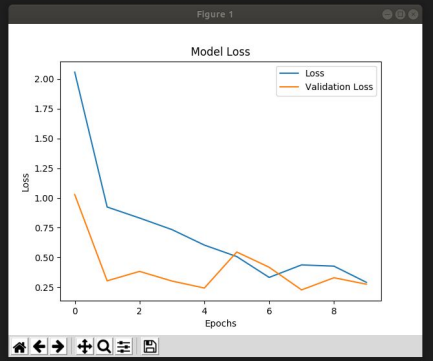


Figure : Loss graph of the model with the training and validation dataset after adding attention layer

Conclusion

- In this project, we studied R-CNN architecture and build this model by python from scratch using airplane dataset.

- We extensively studied about the OpenCV functionalities to develop the major parts of the project.

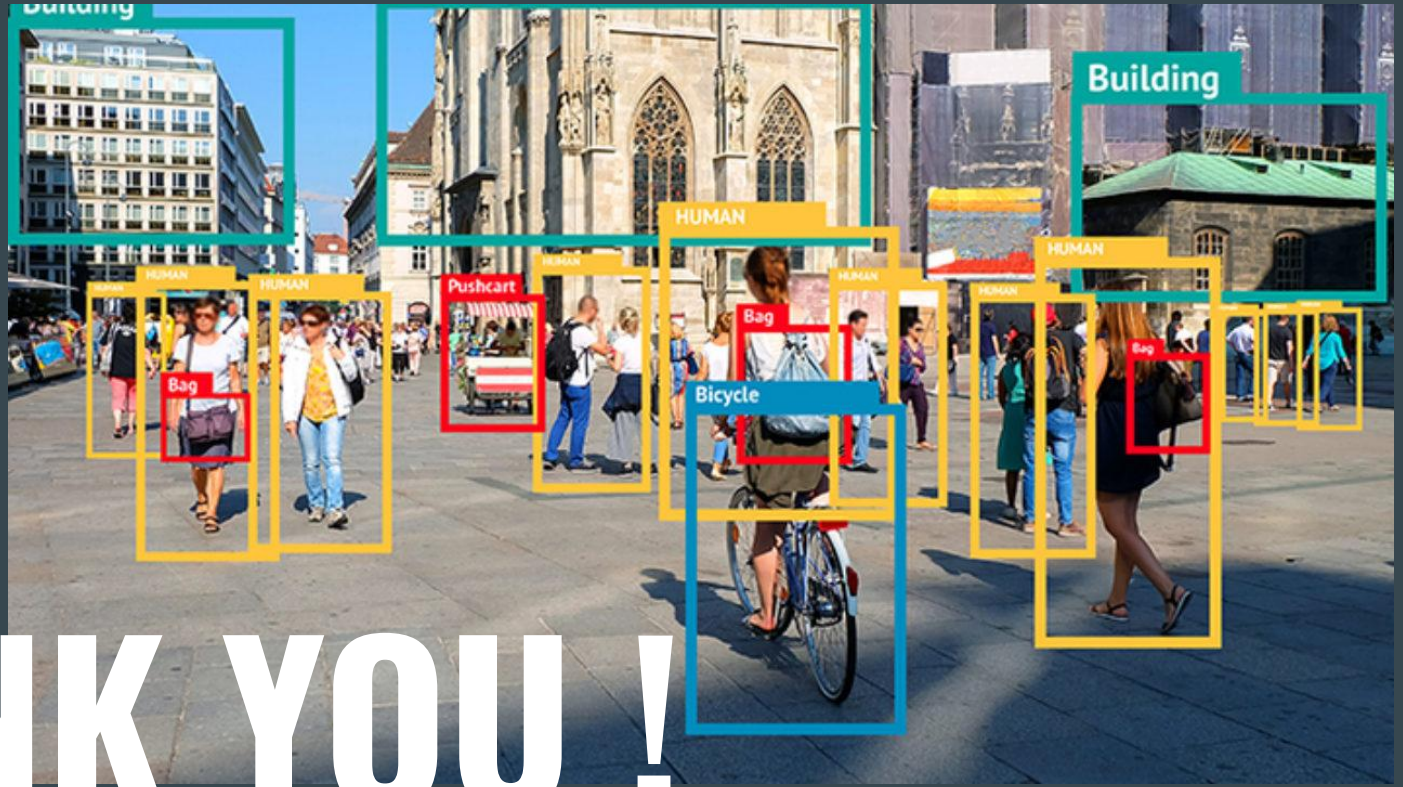
- We leverage self-attention layer to improve the model prediction. Self-attention layer computes a weighted sum of features vectors in feature map for each feature vector to focus on discriminative locations in an image crop (self-attention module from keras tensorflow). Result = 92.4%

- We also explored the tensorflow and keras library and also explored the more advanced algorithm for object detection like YOLO v4, etc.

- Our experiment on airplane model dataset demonstrated the effectiveness of our proposed method.

References

- Mask R-CNN: A Beginner's Guide —— Elisha Odemakinde
- How to use a pre-trained model (VGG) for image classification —— Dr. Saptarsi Goswami
- R-CNN for object detection —— Shilpa Ananth
- R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.
- https://towardsdatascience.com/how-to-use-a-pre-trained-model-vgg-for-image-classification-8dd7c4a4a517
- https://towardsdatascience.com/step-by-step-r-cnn-implementation-from-scratch-in-python-e97101ccde55
- https://towardsdatascience.com/r-cnn-for-object-detection-a-technical-summary-9e7bfa8a557c



THANK YOU !