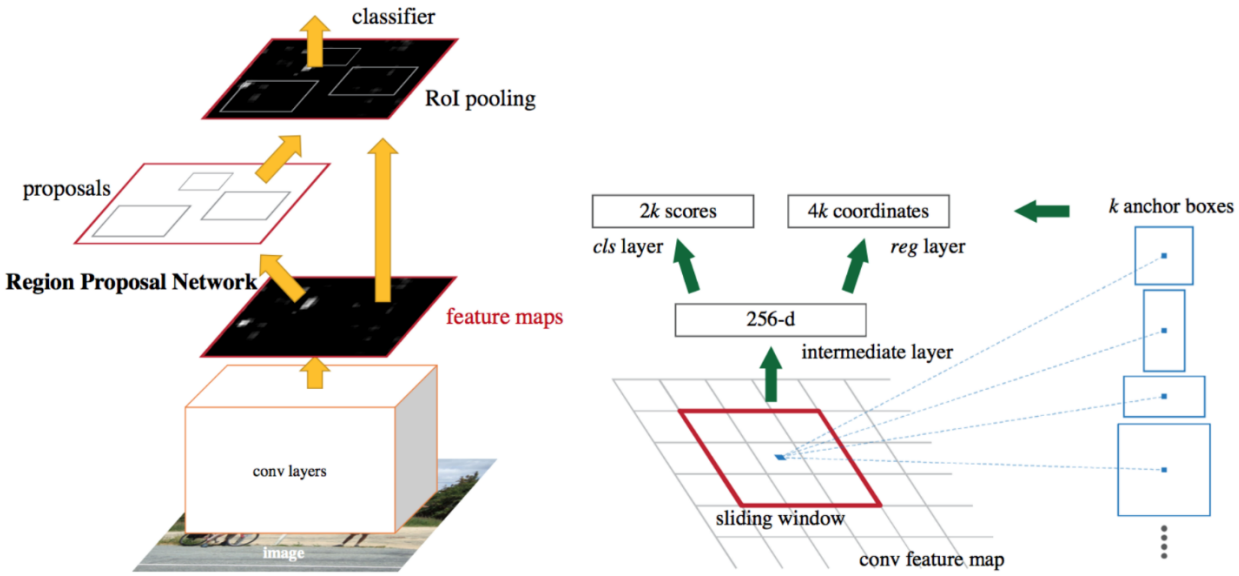
Faster R-CNN

Faster R-CNN is an advanced version of Fast RCNN architecture, which is 250x faster than the R-CNN model as well as smaller in size. This network introduces Region Proposal Network (RPN) that shares full-image convolutional features with the detection network, which is its key contribution. The RPN module generates anchor boxes that propose numbers of regions at multiple scales and aspect ratios for a specific pixel location. Therefore, this model does not require any additional algorithms like selective search for region proposals compared to its predecessor Fast RCNN and RCNN. Apart from the region proposal parts, this architecture is pretty much similar to Fast RCNN. The main paper and following blogs will give us more details about this model its architectural pattern.



- Main paper
- Blog-1
- Blog-2
- Blog-3
- Blog-4
- Blog-5

Anchor Box

Anchor boxes are bounding boxes which are placed over the whole feature map and serve as locations at which the RPN is going to search for objects. To place those anchor boxes, the feature map is divided into a regular grid first. Next, we place a number of anchor boxes, centered on the center of the cell, into each of these grid cells. It is up to us how many of those anchor boxes we choose and of what size and aspect ratio they are. Example, we can decide that we want an anchor box of size 16x16, with two types of aspect ratio: 1:1, 1:2, and 2:1, and with 2 types of scale: x1, and x2. Which means we end up with 6 anchor boxes per region.

Blog-1: https://www.mathworks.com/help/vision/ug/anchor-boxes-for-object-detection.html

Blog-2: https://www.thinkautonomous.ai/blog/anchor-boxes/

Implementation

To successfully implement the Faster RCNN model, we need to develop its three core components including a ***backbone***, a ***region proposal network (RPN)***, and a ***RoI head***. The backbone extracts relevant information from the image and generate feature maps, these feature maps are then fed into the RPN module. The RPN module generate anchor boxes from the feature maps and convert these boxes into proposals using box regression predictions and objectiveness scores. These proposals are then pass into the RoI head, where roi pooling is applied on these proposals and pass these features to both classification and regression layer. This approach effectively classifies and localize objects in the image.

- Initially, we need to develop a custom data loader that will return the images and their corresponding ground truth boxes and labels.
- After that, we will develop our RPN module that generates anchor boxes for region proposals. We need another helper function to convert those anchor boxes into proposals.
- The RPN module takes input image, its feature maps and target as input. This module has several steps. Initially, it predicts class scores and bounding box prediction (transformation) and generates anchor boxes based on defined scales and aspect ratios. After that it utilizes bounding box regression prediction and anchor boxes to generate proposals.
- These proposals are filtered with using a function that takes proposals, class scores and image shape as input and perform several steps including pre-NMS, coordinates validation, NMS, and post NMS top-k filtering. In the training phase, it assigns labels to each proposal (foreground & background) and apply sampling method to differentiate certain number of positive and negative samples. Finally, this module trains with a multi-task loss for classification (cross-entropy) and regression (smooth-L1-loss). Classification is employed to determine the objectiveness of a proposal for being foreground or background.
- Following that, we will develop a RoI head module that take proposals generated from the RPN as input. This network contains a RoI pooling layer, fully connected layer (in our case two), classification layer and a regression layer.
- The proposals are initially passed through a function that assign labels to each proposal based on IoU scores between them and ground truth. In this stage, we also assign labels for background and ignored proposals based on thresholds that completely ignored during the training time. We also sample these proposals based on some pre-defined thresholds and transforms the proposals target to work with regression model.
- After that, these samples proposals are fed into the roi layer. The output from the roi layer goes through the fully connected layers (linear), and finally the output from the FC layers passed through the classification layer and regression layer separately for object localization and classification.
- During the training RoI uses multitask loss for classification and localization to optimize the model performance.
- Finally, we need to integrate these RPN module and RoI head alongside a backbone to develop Faster RCNN model. This model takes image and targets as input to predict the classification score as well as the bounding boxes.
- Initially, Faster RCNN pass the input image to the backbone for feature extraction. The image should be preprocessed before passing through the model (i.e., resize, normalization, augmentation etc.).

- The extracted features from the backbone, image and targets are then fed to the RPN module that will provide us with the proposals.
- The extracted features from the backbone, proposals and targets are then passed through the RoI head for final prediction.
- We can utilize any optimizer like Adam, SGD with custom parameters for training the overall framework.

Inference

- To perform inference, we need to load the trained model and define as evaluation mode. Here, we also need to move the model into the existing device to tackle device mismatch error.
- After that, we will pass an input image through the model that will provide us prediction output, in our case two (it can be different based on implementation). In our implementation, model gives us RPN output and FRCNN output. The RPN output contains proposals and objectiveness scores, whereas the FRCNN provides predicted bounding boxes, labels and scores alongside the ground truth bounding box.
- The NMS and other necessary transformation has been implemented inside the model architecture, therefore, we do not need to apply those post processing steps during the inference.
- We require a function to draw the predicted bounding boxes on the original image to create final output.
- To evaluate overall model performance, we can use built in libraires or employ function to compute mean average precision.

Observation

- We have developed this model from scratch that provide us around 0.45 mAP (for IoU=0.3) on the PASCAL VOC test data.
- Further experiments with proper fine-tuning may increase its performance.
- This model takes around 1.2-1.4 seconds for each prediction.
- It can effectively generate region of proposals using proposals within a single stage pipeline.

Faster R-CNN (Pre-trained Version)

PyTorch has built-in Faster RCNN model, which trained on the MS COCO datasets that contains 80 different objects. We can utilize this pre-trained model on our custom dataset using transfer learning. Usually, a scratch model needs higher training time and proper parametrization to get optimal performance as their weights are initialized randomly before training. These models are usually work best for specific datasets, because most of them are trained on a specific needs or smaller datasets. However, transfer learning facilitates us to adopt prior knowledge from another model that was trained on a large dataset and obtained a benchmarked performance. To achieve this, we only some of the layers of these models and train on our custom datasets based on our needs. This approach reduces training time and also can provide good results with proper fine-tuning, because they already have prior knowledge to predict something. For object detection task, we can employ this model to train our custom dataset with some modifications. The official module of Faster RCNN in Pytorch can be found here. A sample implementation of this model is here and the official source code are these (docs, github).

All pre-trained models in Pytorch can be found in these links (models, with docs).

To implement a model with their pre-trained weights, we should follow several steps, including choosing dataset, necessary modifications in model architecture, training and evaluation.

N.B: Pre-train Faster RCNN takes input bounding as PASCAL VOC format.

Implementation

We selected Vehicle detection dataset to implement pre-trained Faster RCNN model. This dataset was downloaded from Kaggle and it contains 8 types of vehicles including auto, bus, car, lcv (Light Motor Vehicle), motorcycle, multiaxle, tractor and truck. The overall implementation steps are as follows:

- First, we investigated our dataset and found there are total 8218 images and same number of text files with their corresponding bounding box annotation and class label.
- We carefully checked and ensure every image contains its annotation before splitting the data.
- We found that the bounding box annotations of these are images in YOLO format, but Faster RCNN expects bounding box in VOC format. Therefore, we converted these annotations into VOC format and validated it by visualizing images with bounding boxes to ensure.
- As this dataset does not contain separate training and validation set, we manually split the dataset into 85:15 ratio to create train and test dataset by script.
- Before splitting data, we perform some data processing by creating a pickle file for keeping their information including image path, bounding boxes and their labels. And using a function, we read all the images, applied these transformations and moved to the train and test folder based on the splitting ratio.
- The dataset contains 8 classes, but typically in the Faster RCNN requires an additional class as 'background' to learn more robust object representation.
- We developed a function to normalize and perform transformation to the dataset and a custom data loader class to load data batch-wise.
- After the data processing part, we need to load the pre-trained model. We can load pre-trained Faster RCNN in several ways.
- We can utilize the fully pr-trained network by only modifying the input and shape (by default it is 3x600x1000) and number of class (in our case it is 9).
- We can also define each module within the model architecture including backbone, roi alignment, anchor generator etc.
- This pre-trained model gives us the flexibility to fine-tune with the necessary parameters within the model architecture.
- After initializing the pre-trained model, we need to define optimizer and perform training with proper optimization.

Inference

- Initially, we need to load the trained model and define this as evaluation mode for the inference.
- inference, we just need to pass the input images to the trained model and it will gives us a list of dictionary prediction that contains predicted boxes, labels and scores.
- We don't need to perform any post processing steps like NMS, because the model itself take care of this.

- After getting the prediction output, we need to separate the values to draw predicted bounding boxes on the input images.

Observation

- It requires less training time than the scratch model.
- For a single input image prediction inference, it takes around 0.7 to 0.8 seconds.
- We can utilize this model any custom dataset for object detection.

Codes

Built-in Faster RCNN: https://github.com/phil-hoang/general-object-detector/tree/main

CVGL-Stanford: https://cvgl.stanford.edu/research.html

3D Vision: https://viso.ai/computer-vision/3d-computer-vision/

Pathway: https://courses.thinkautonomous.ai/obstacle-tracking?ref=thinkautonomous.ai