Object Detection and Classification using R-CNNs

In this post, I'll describe in detail how R-CNN (Regions with CNN features), a recently introduced deep learning based object detection and classification method works. R-CNN's have proved highly effective in detecting and classifying objects in natural images, achieving mAP scores far higher than previous techniques. The R-CNN method is described in the following series of papers by Ross Girshick et al.

1. R-CNN (Girshick et al. 2013)*
2. Fast R-CNN (Girshick 2015)*
3. Faster R-CNN (Ren et al. 2015)*

This post describes the final version of the R-CNN method described in the last paper. I considered at first to describe the evolution of the method from its first introduction to the final version, however that turned out to be a very ambitious undertaking. I settled on describing the final version in detail.

Fortunately, there are many implementations of the R-CNN algorithm available on the web in TensorFlow, PyTorch and other machine learning libraries. I used the following implementation:

https://github.com/ruotianluo/pytorch-faster-rcnn

Much of the terminology used in this post (for example the names of different layers) follows the terminology used in the code. Understanding the information presented in this post should make it much easier to follow the PyTorch implementation and make your own modifications.

Post Organization

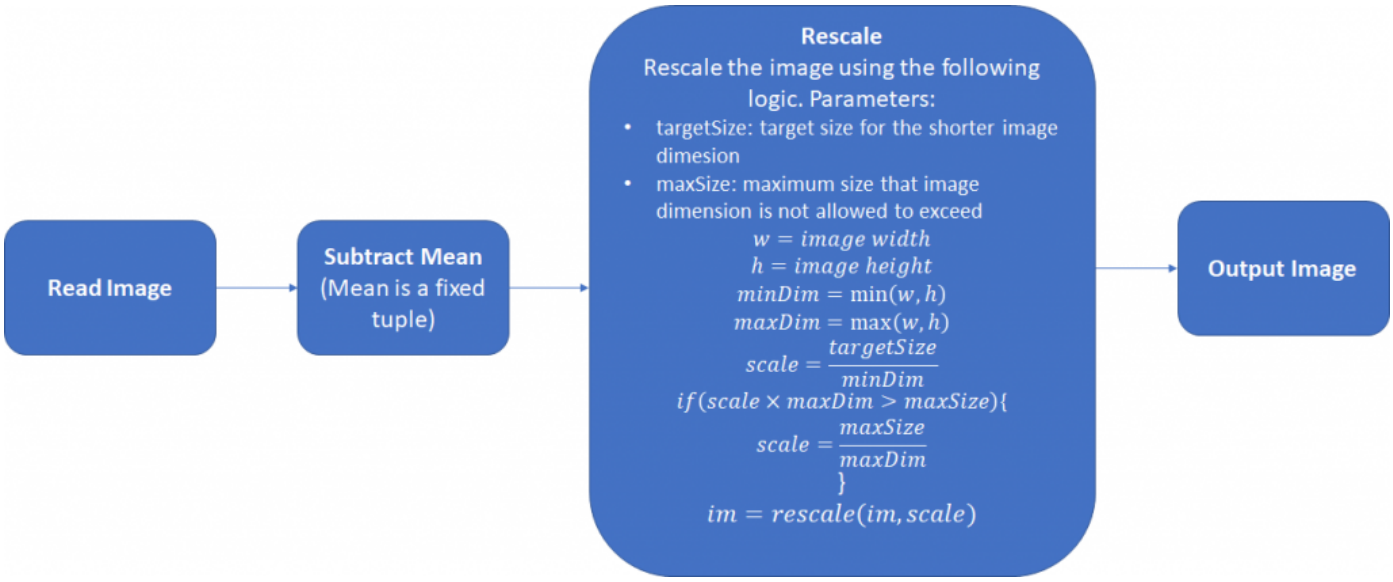
- **Section 1 – Image Pre-Processing:** In this section, we'll describe the pre-processing steps that are applied to an input image. These steps include subtracting a mean pixel value and scaling the

image. The pre-processing steps must be identical between training and inference

- **Section 2 – Network Organization:** In this section, we'll describe the three main components of the network – the "head" network, the region proposal network (RPN) and the classification network.
- **Section 3 – Implementation Details (Training):** This is the longest section of the post and describes in detail the steps involved in training a R-CNN network
- **Section 4 – Implementation Details (Inference):** In this section, we'll describe the steps involved during inference – i.e., using the trained R-CNN network to identify promising regions and classify the objects in those regions.
- **Appendix:** Here we'll cover the details of some of the frequently used algorithms during the operation of a R-CNN such as non-maximum suppression and the details of the Resnet 50 architecture.

Image Pre-Processing

The following pre-processing steps are applied to an image before it is sent through the network. These steps must be identical for both training and inference. The mean vector (3×1, one number corresponding to each color channel) is not the mean of the pixel values in the current image but a configuration value that is identical across all training and test images.



The default values for $targetSize$ and $maxSize$ parameters are 600 and 1000 respectively.

Network Organization

A R-CNN uses neural networks to solve two main problems:

- Identify promising regions (Region of Interest – ROI) in an input image that are likely to contain foreground objects

- Compute the object class probability distribution of each ROI – i.e., compute the probability that the ROI contains an object of a certain class. The user can then select the object class with the highest probability as the classification result.

R-CNNs consist of three main types of networks:

1. Head
2. Region Proposal Network (RPN)
3. Classification Network

R-CNNs use the first few layers of a pre-trained network such as ResNet 50 to identify promising features from an input image. Using a network trained on one dataset on a different problem is possible because neural networks exhibit "transfer learning" (Yosinski et al. 2014)*. The first few layers of the network learn to detect general features such as edges and color blobs that are good discriminating features across many different problems. The features learnt by the later layers are higher level, more problem specific features. These layers can either be removed or the weights for these layers can be fine-tuned during back-propagation. The first few layers that are initialized from a pre-trained network constitute the "head" network. The convolutional feature maps produced by the head network are then passed through the Region Proposal Network (RPN) which uses a series of convolutional and fully connected layers to produce promising ROIs that are likely to contain a foreground object (problem 1 mentioned above). These promising ROIs are then used to crop out corresponding regions from the feature maps produced by the head network. This is called "Crop Pooling". The regions produced by crop pooling are then passed through a classification network which learns to classify the object contained in each ROI.

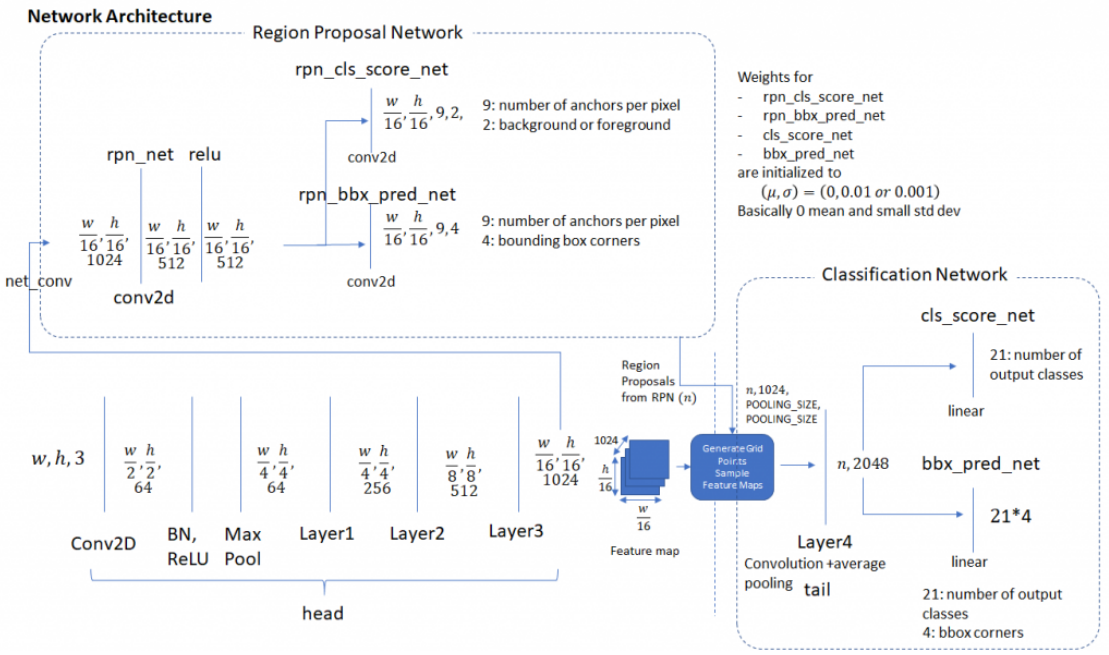
As an aside, you may notice that weights for a ResNet are initialized in a curious way:

```
n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
m.weight.data.normal_(0, math.sqrt(2. / n))
```

If you are interested in learning more about why this method works, read my post about initializing weights for convolutional and fully connected layers.

Network Architecture

The diagram below shows the individual components of the three network types described above. We show the dimensions of the input and output of each network layer which assists in understanding how data is transformed by each layer of the network. w and h represent the width and height of the input image (after pre-processing).



Implementation Details: Training

In this section, we'll describe in detail the steps involved in training a R-CNN. Once you understand how training works, understanding inference is a lot easier as it simply uses a subset of the steps involved in training. The goal of training is to adjust the weights in the RPN and Classification network and fine-tune the weights of the head network (these weights are initialized from a pre-trained network such as ResNet). Recall that the job of the RPN network is to produce promising ROIs and the job of the classification network to assign object class scores to each ROI. Therefore, to train these networks, we need the corresponding ground truth i.e., the coordinates of the bounding boxes around the objects present in an image and the class of those objects. This ground truth comes from free to use image databases that come with an annotation file for each image. This annotation file contains the coordinates of the bounding box and the object class label for each object present in the image (the object classes are from a list of pre-defined object classes). These image databases have been used to support a variety of object classification and detection challenges. Two commonly used databases are:

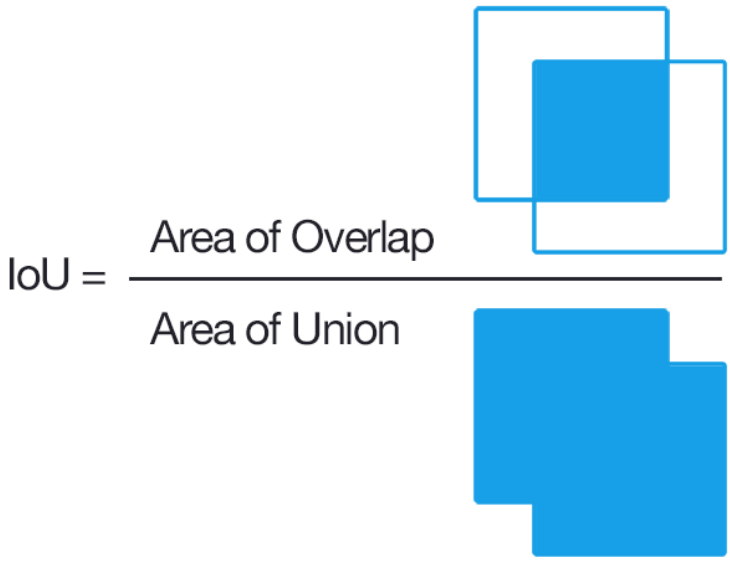
- PASCAL VOC: The VOC 2007 database contains 9963 training/validation/test images with 24,640 annotations for 20 object classes.
  - *Person:* person
  - *Animal:* bird, cat, cow, dog, horse, sheep
  - *Vehicle:* aeroplane, bicycle, boat, bus, car, motorbike, train
  - *Indoor:* bottle, chair, dining table, potted plant, sofa, tv/monitor

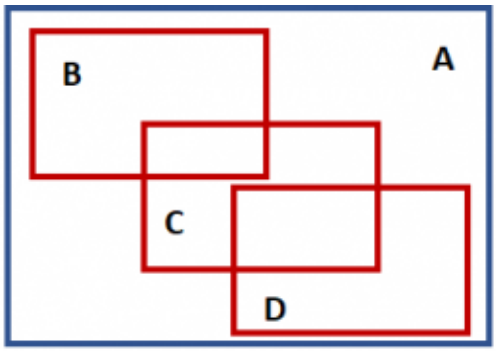
- COCO (Common Objects in Context): The COCO dataset is much larger. It contains > 200K labelled images with 90 object categories.

I used the smaller PASCAL VOC 2007 dataset for my training. R-CNN is able to train both the region proposal network and the classification network in the same step.

Let's take a moment to go over the concepts of "bounding box regression coefficients" and "bounding box overlap" that are used extensively in the remainder of this post.

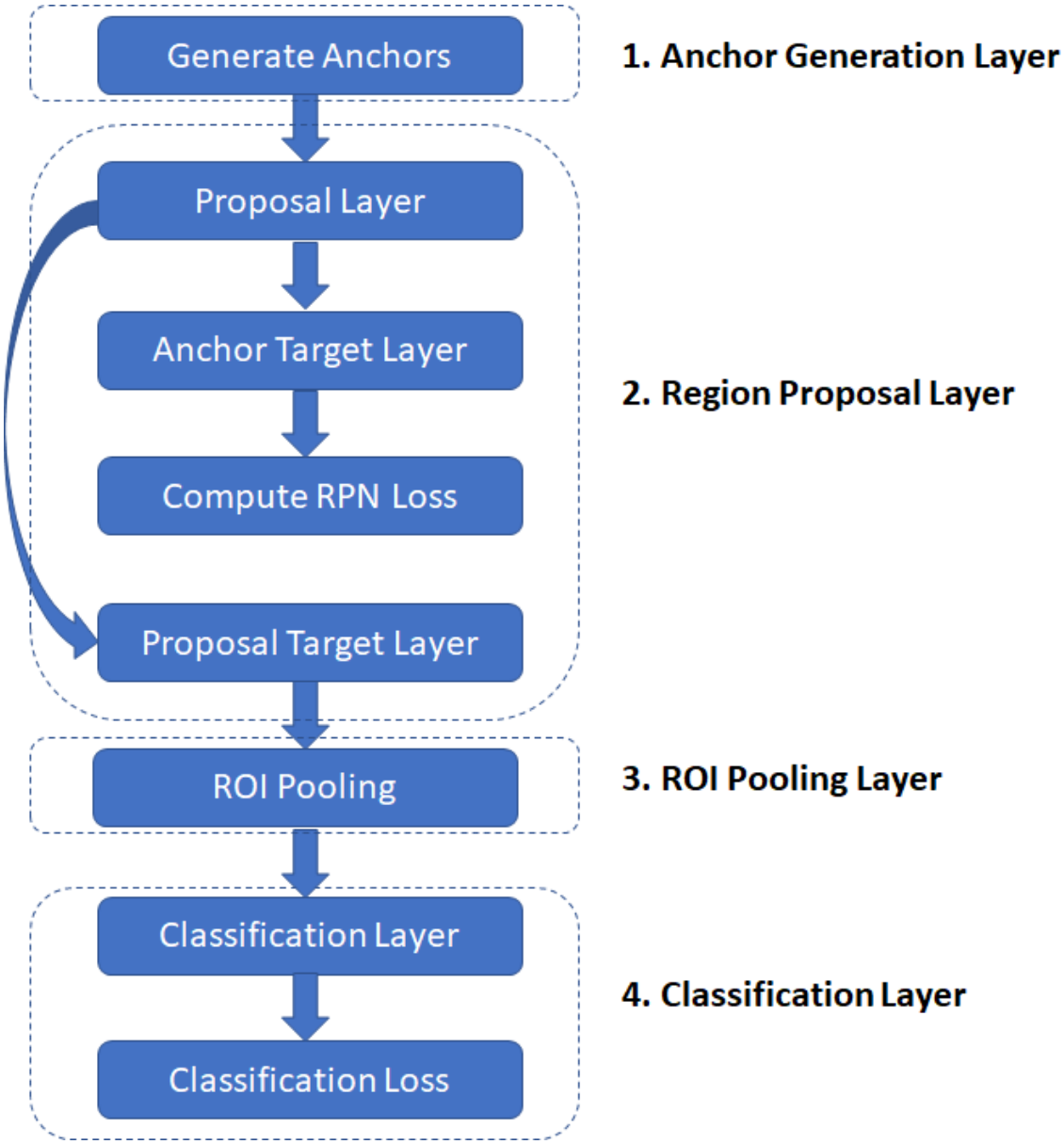
- **Bounding Box Regression Coefficients** (also referred to as "regression coefficients" and "regression targets"): One of the goals of R-CNN is to produce good bounding boxes that closely fit object boundaries. R-CNN produces these bounding boxes by taking a given bounding box (defined by the coordinates of the top left corner, width and height) and tweaking its top left corner, width and height by applying a set of "regression coefficients". These coefficients are computed as follows (Appendix C of (Anon. 2014)*. Let the x, y coordinates of the top left corner of the target and original bounding box be 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:
  - $t_x = \frac{(T_x - O_x)}{O_x}, t_y = \frac{(T_y - O_y)}{O_y}, t_w = log(\frac{T_w}{O_w}), t_h = log(\frac{T_h}{O_h})$. This function is readily invertible, i.e., given the regression coefficients and coordinates of the top left corner and the width and height of the original bounding box, the top left corner and width and height of the target box can be easily calculated. Note that the shape of the bounding box is not changed – i.e., a rectangle remains a rectangle under this transformation.
- **Intersection over Union (IoU) Overlap:** We need some measure of how close a given bounding box is to another bounding box that is independent of the units used (pixels etc) to measure the dimensions of a bounding box. This measure should be intuitive (two coincident bounding boxes should have an overlap of 1 and two non-overlapping boxes should have an overlap of 0) and fast and easy to calculate. A commonly used overlap measure is the "Intersection over Union (IoU) overlap, calculated as shown below.





boxes B, C, D are all contained within box A and have the same area. Then IoU overlap of B, C, D with A is the same

With these preliminaries out of the way, lets now dive into the implementation details for training a R-CNN. In the software implementation, R-CNN execution is broken down into several layers, as shown below. A layer encapsulates a sequence of logical steps that can involve running data through one of the neural networks and other steps such as comparing overlap between bounding boxes, performing non-maxima suppression etc.



- **Anchor Generation Layer:** This layer generates a fixed number of "anchors" (bounding boxes) by first generating 9 anchors of different scales and aspect ratios and then replicating these anchors by translating them across uniformly spaced grid points spanning the input image.
- **Proposal Layer:** Transform the anchors according to the bounding box regression coefficients to generate transformed anchors. Then prune the number of anchors by applying non-maximum suppression (see Appendix) using the probability of an anchor being a foreground region
- **Anchor Target Layer**: The goal of the anchor target layer is to produce a set of "good" anchors and the corresponding foreground/background labels and target regression coefficients to train the Region Proposal Network. The output of this layer is only used to train the RPN network and is not used by the classification layer. Given a set of anchors (produced by the anchor generation layer, the anchor target layer identifies promising foreground and background anchors. Promising foreground anchors are those whose overlap with some ground truth box is higher than a threshold. Background boxes are those whose overlap with any ground truth box is lower than a threshold. The anchor target layer also outputs a set of bounding box regressors i.e., a measure of

how far each anchor target is from the closest bounding box. These regressors only make sense for the foreground boxes as there is no notion of "closest bounding box" for a background box.

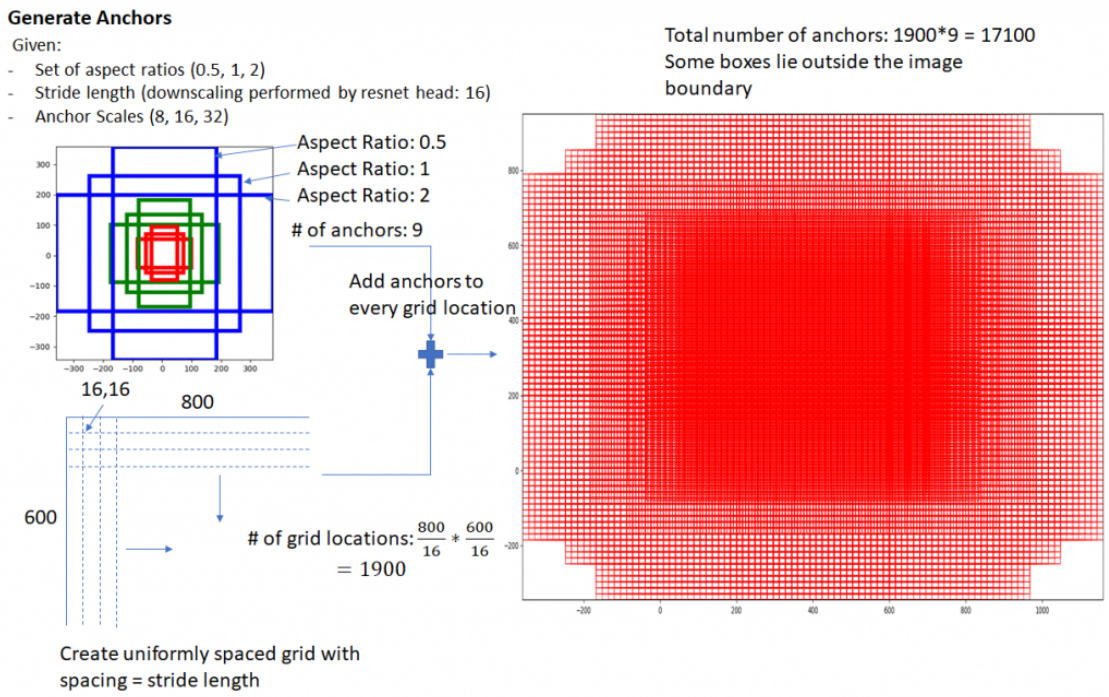
- **RPN Loss:** The RPN loss function is the metric that is minimized during optimization to train the RPN network. The loss function is a combination of:
  - The proportion of bounding boxes produced by RPN that are correctly classified as foreground/background
  - Some distance measure between the predicted and target regression coefficients.
- **Proposal Target Layer:** The goal of the proposal target layer is to prune the list of anchors produced by the proposal layer and produce *class specific* bounding box regression targets that can be used to train the classification layer to produce good class labels and regression targets
- **ROI Pooling Layer:** Implements a spatial transformation network that samples the input feature map given the bounding box coordinates of the region proposals produced by the proposal target layer. These coordinates will generally not lie on integer boundaries, thus interpolation based sampling is required.
- **Classification Layer:** The classification layer takes the output feature maps produced by the ROI Pooling Layer and passes them through a series of convolutional layers. The output is fed through two fully connected layers. The first layer produces the class probability distribution for each region proposal and the second layer produces a set of class specific bounding box regressors.
- **Classification Loss:** Similar to RPN loss, classification loss is the metric that is minimized during optimization to train the classification network. During back propagation, the error gradients flow to the RPN network as well, so training the classification layer modifies the weights of the RPN network as well. We'll have more to say about this point later. The classification loss is a combination of:
  - The proportion of bounding boxes produced by RPN that are correctly classified (as the correct object class)
  - Some distance measure between the predicted and target regression coefficients.

We'll now go through each of these layers in detail.

Anchor Generation Layer

The anchor generation layer produces a set of bounding boxes (called "anchor boxes") of varying sizes and aspect ratios spread all over the input image. These bounding boxes are the same for all images i.e., they are agnostic of the content of an image. Some of these bounding boxes will enclose foreground objects while most won't. The goal of the RPN network is to learn to identify which of these boxes are good boxes – i.e., likely to contain a foreground object and to produce target regression coefficients, which when applied to an anchor box turns the anchor box into a better bounding box (fits the enclosed foreground object more closely).

The diagram below demonstrates how these anchor boxes are generated.



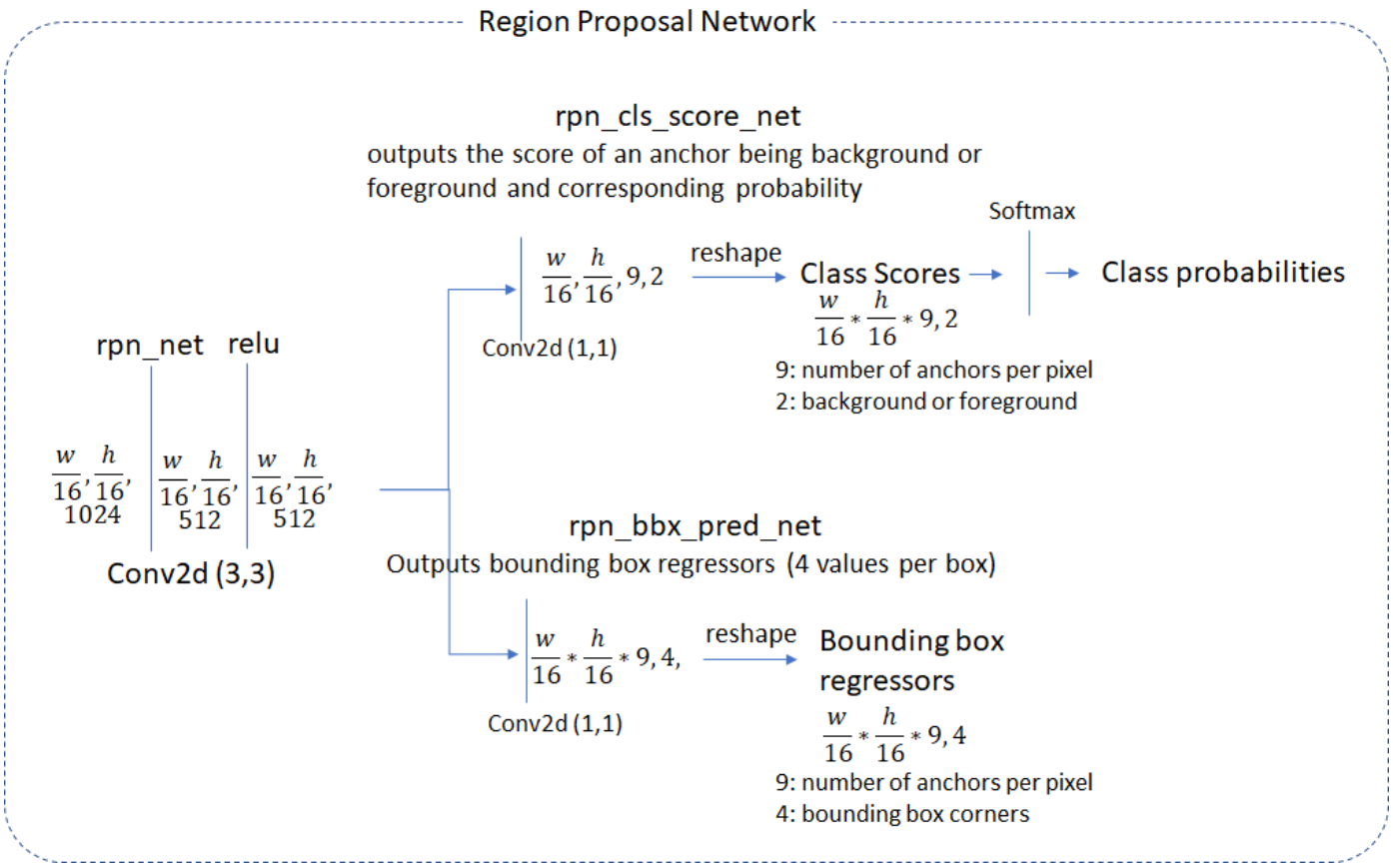
Region Proposal Layer

Object detection methods need as input a "region proposal system" that produces a set of sparse (for example selective search (Anon.)*) or a dense (for example features used in deformable part models (Anon.)*) set of features. The first version of the R-CNN system used the selective search method for generating region proposal. In the current version (known as "Faster R-CNN"), a "sliding window" based technique (described in the previous section) is used to generate a set of dense candidate regions and then a neural network driven region proposal network is used to rank region proposals according to the probability of a region containing a foreground object. The region proposal layer has two goals:

- From a list of anchors, identify background and foreground anchors
- Modify the position, width and height of the anchors by applying a set of "regression coefficients" to improve the quality of the anchors (for example, make them fit the boundaries of objects better)

The region proposal layer consists of a Region Proposal *Network* and three layers – Proposal Layer, Anchor Target Layer and Proposal Target Layer. These three layers are described in detail in the following sections.

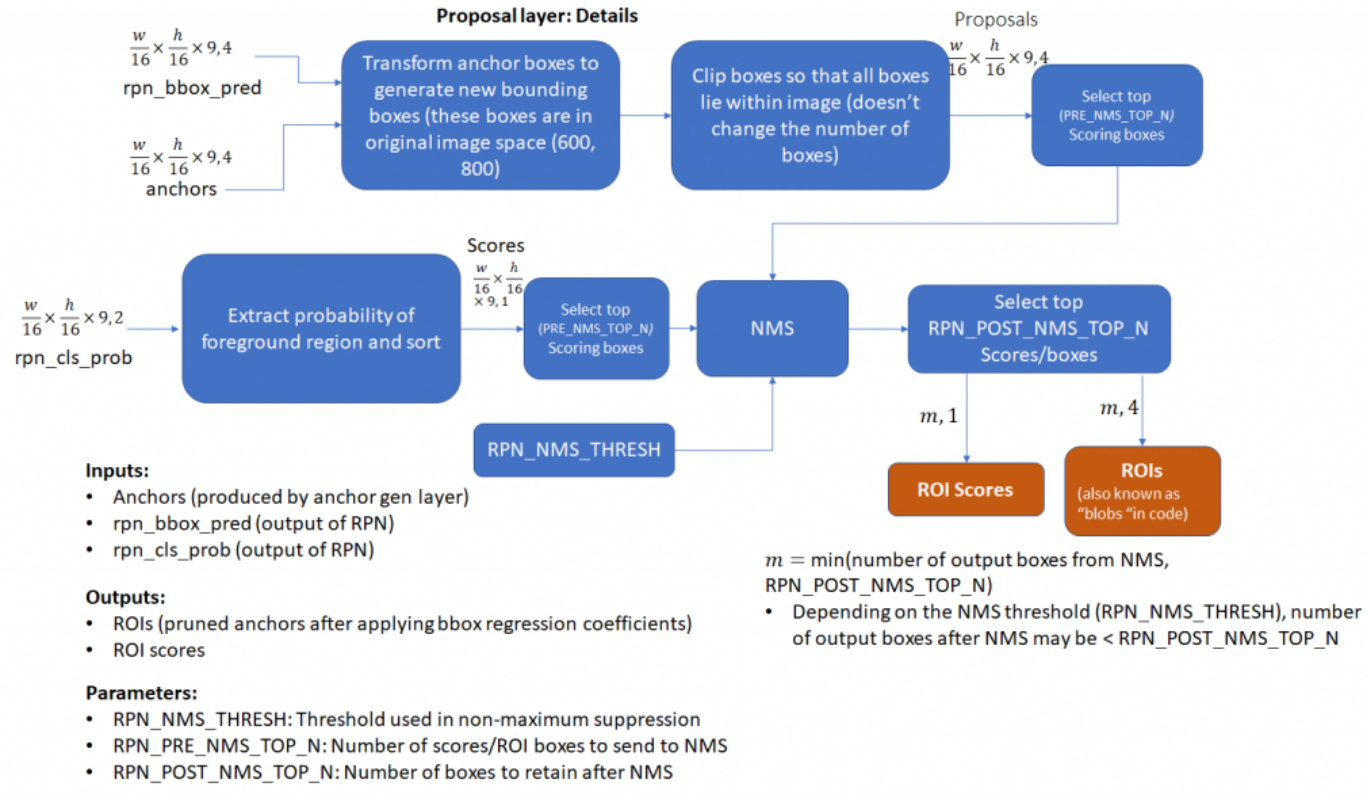
Region Proposal Network



The region proposal layer runs feature maps produced by the head network through a convolutional layer (called rpn_net in code) followed by RELU. The output of rpn_net is run through two (1,1) kernel convolutional layers to produce background/foreground class scores and probabilities and corresponding bounding box regression coefficients. The stride length of the head network matches the stride used while generating the anchors, so the number of anchor boxes are in 1-1 correspondence with the information produced by the region proposal network (number of anchor boxes = number of class scores = number of bounding box regression coefficients = $\frac{w}{16} \times \frac{h}{16} \times 9$)

Proposal Layer

The proposal layer takes the anchor boxes produced by the anchor generation layer and prunes the number of boxes by applying non-maximum suppression based on the foreground scores (see appendix for details). It also generates transformed bounding boxes by applying the regression coefficients generated by the RPN to the corresponding anchor boxes.



Anchor Target Layer

The goal of the anchor target layer is to select promising anchors that can be used to train the RPN network to:

1. distinguish between foreground and background regions and
2. generate good bounding box regression coefficients for the foreground boxes.

It is useful to first look at how the RPN Loss is calculated. This will reveal the information needed to calculate the RPN loss which makes it easy to follow the operation of the Anchor Target Layer.

Calculating RPN Loss

Remember the goal of the RPN layer is to generate good bounding boxes. To do so from a set of anchor boxes, the RPN layer must learn to classify an anchor box as background or foreground and calculate the regression coefficients to modify the position, width and height of a foreground anchor box to make it a "better" foreground box (fit a foreground object more closely). RPN Loss is formulated in such a way to encourage the network to learn this behaviour.

RPN loss is a sum of the classification loss and bounding box regression loss. The classification loss uses cross entropy loss to penalize incorrectly classified boxes and the regression loss uses a function of the distance between the true regression coefficients (calculated using the closest matching ground truth box for a foreground anchor box) and the regression coefficients predicted by the network (see rpn_bbx_pred_net in the RPN network architecture diagram).

$$RPNLoss = \text{Classification Loss} + \text{Bounding Box Regression Loss}$$

Classification Loss:

cross_entropy(predicted _class, actual_class)

Bounding Box Regression Loss:

$$L_{loc} = \sum_{u \in \text{all foreground anchors}} l_u$$

Sum over the regression losses for all foreground anchors. Doing this for background anchors doesn't make sense as there is no associated ground truth box for a background anchor

$$l_u = \sum_{i \in x,y,w,h} smooth_{L1}(u_i(predicted) - u_i(target))$$

This shows how the regression loss for a given foreground anchor is calculated. We take the difference between the predicted (by the RPN) and target (calculated using the closest ground truth box to the anchor box) regression coefficients. There are four components – corresponding to the coordinates of the top left corner and the width/height of the bounding box. The smooth L1 function is defined as follows:

$$smooth_{L1}(x) = \begin{cases} \frac{\sigma^2 x^2}{2} & \|x\| < \frac{1}{\sigma^2} \\ \|x\| - \frac{0.5}{\sigma^2} & otherwise \end{cases}$$

Here σ is chosen arbitrarily (set to 3 in my code). Note that in the python implementation, a mask array for the foreground anchors (called "bbox_inside_weights") is used to calculate the loss as a vector operation and avoid for-if loops.

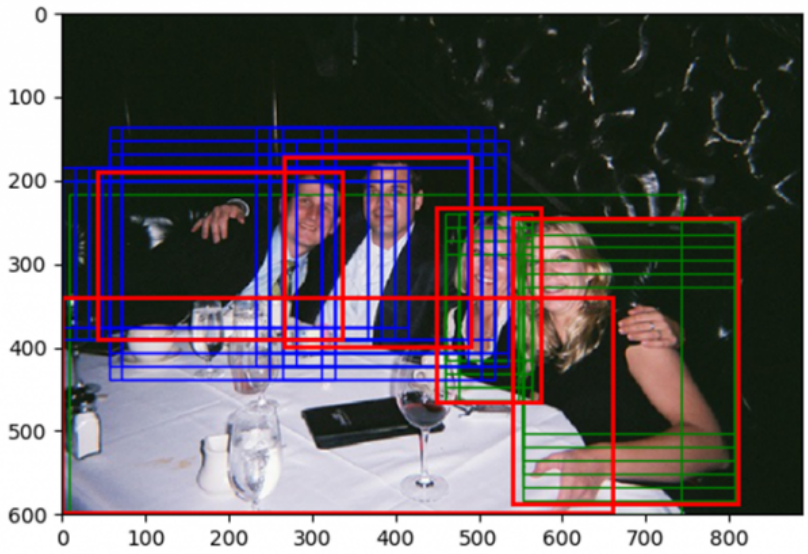
Thus, to calculate the loss we need to calculate the following quantities:

1. Class labels (background or foreground) and scores for the anchor boxes
2. Target regression coefficients for the foreground anchor boxes

We'll now follow the implementation of the anchor target layer to see how these quantities are calculated. We first select the anchor boxes that lie within the image extent. Then, good foreground boxes are selected by first computing the IoU (Intersection over Union) overlap of all anchor boxes (within the image) with all ground truth boxes. Using this overlap information, two types of boxes are marked as foreground:

1. **type A:** For each ground truth box, all foreground boxes that have the max IoU overlap with the ground truth box
2. **type B:** Anchor boxes whose maximum overlap with some ground truth box exceeds a threshold

these boxes are shown in the image below:



Note that only anchor boxes whose overlap with some ground truth box exceeds a threshold are selected as foreground boxes. This is done to avoid presenting the RPN with the "hopeless learning task" of learning the regression coefficients of boxes that are too far from the best match ground truth box. Similarly, boxes whose overlap are less than a negative threshold are labeled background boxes. Not all boxes that are not foreground boxes are labeled background. Boxes that are neither foreground or background are labeled "don't care". These boxes are not included in the calculation of RPN loss.

There are two additional thresholds related to the total number of background and foreground boxes we want to achieve and the fraction of this number that should be foreground. If the number of foreground boxes that pass the test exceeds the threshold, we randomly mark the excess foreground boxes to "don't care". Similar logic is applied to the background boxes.

Next, we compute bounding box regression coefficients between the foreground boxes and the corresponding ground truth box with maximum overlap. This is easy and one just needs to follow the formula to calculate the regression coefficients.

This concludes our discussion of the anchor target layer. To recap, let's list the parameters and input/output for this layer:

Parameters:

- TRAIN.RPN_POSITIVE_OVERLAP: Threshold used to select if an anchor box is a good foreground box (Default: 0.7)
- TRAIN.RPN_NEGATIVE_OVERLAP: If the max overlap of a anchor from a ground truth box is lower than this thershold, it is marked as background. Boxes whose overlap is > than RPN_NEGATIVE_OVERLAP but < RPN_POSITIVE_OVERLAP are marked "don't care".(Default: 0.3)
- TRAIN.RPN_BATCHSIZE: Total number of background and foreground anchors (default: 256)
- TRAIN.RPN_FG_FRACTION: fraction of the batch size that is foreground anchors (default: 0.5). If the number of foreground anchors found is larger than TRAIN.RPN_BATCHSIZE×

TRAIN.RPN_FG_FRACTION, the excess (indices are selected randomly) is marked “don’t care”.

Input:

- RPN Network Outputs (predicted foreground/background class labels, regression coefficients)
- Anchor boxes (generated by the anchor generation layer)
- Ground truth boxes

Output

- Good foreground/background boxes and associated class labels
- Target regression coefficients

The other layers, proposal target layer, ROI Pooling layer and classification layer are meant to generate the information needed to calculate classification loss. Just as we did for the anchor target layer, let’s first look at how classification loss is calculated and what information is needed to calculate it

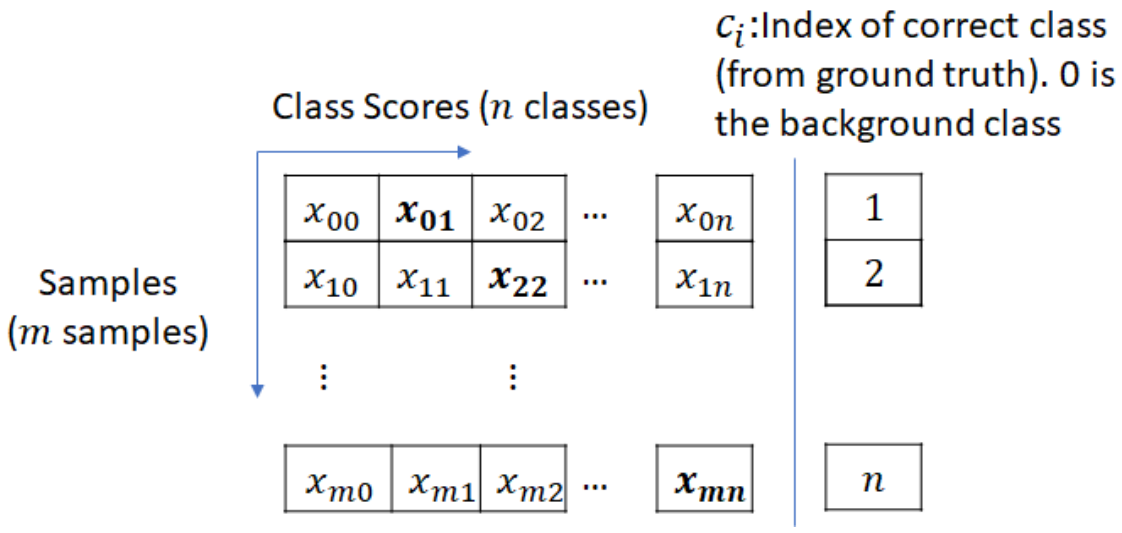
Calculating Classification Layer Loss

Similar to the RPN Loss, classification layer loss has two components – classification loss and bounding box regression loss

$$\text{Classification Layer Loss} = \text{Classification Loss} + \text{Bounding Box Regression Loss}$$

The key difference between the RPN layer and the classification layer is that while the RPN layer dealt with just two classes – foreground and background, the classification layer deals with all the object classes (plus background) that our network is being trained to classify.

The classification loss is the cross entropy loss with the true object class and predicted class score as the parameters. It is calculated as shown below.



$$Cross\ entropy\ loss\ = \frac{-1}{Num\ Samples}\left(\sum_{\substack{i \in Anchor \\ Box\ Samples}} \log\left(\frac{e^{x[i][c_i]}}{\sum_j e^{x[i][j]}}\right)\right)$$

The bounding box regression loss is also calculated similar to the RPN except now the regression coefficients are class specific. The network calculates regression coefficients for each object class. The target regression coefficients are obviously only available for the correct class which is the object class of the ground truth bounding box that has the maximum overlap with a given anchor box. While calculating the loss, a mask array which marks the correct object class for each anchor box is used. The regression coefficients for the incorrect object classes are ignored. This mask array allows the computation of loss to be a matrix multiplication as opposed to requiring a for-each loop.

Thus the following quantities are needed to calculate classification layer loss:

1. Predicted class labels and bounding box regression coefficients (these are outputs of the classification network)
2. class labels for each anchor box
3. Target bounding box regression coefficients

Let's now look at how these quantities are calculated in the proposal target and classification layers.

Proposal Target Layer

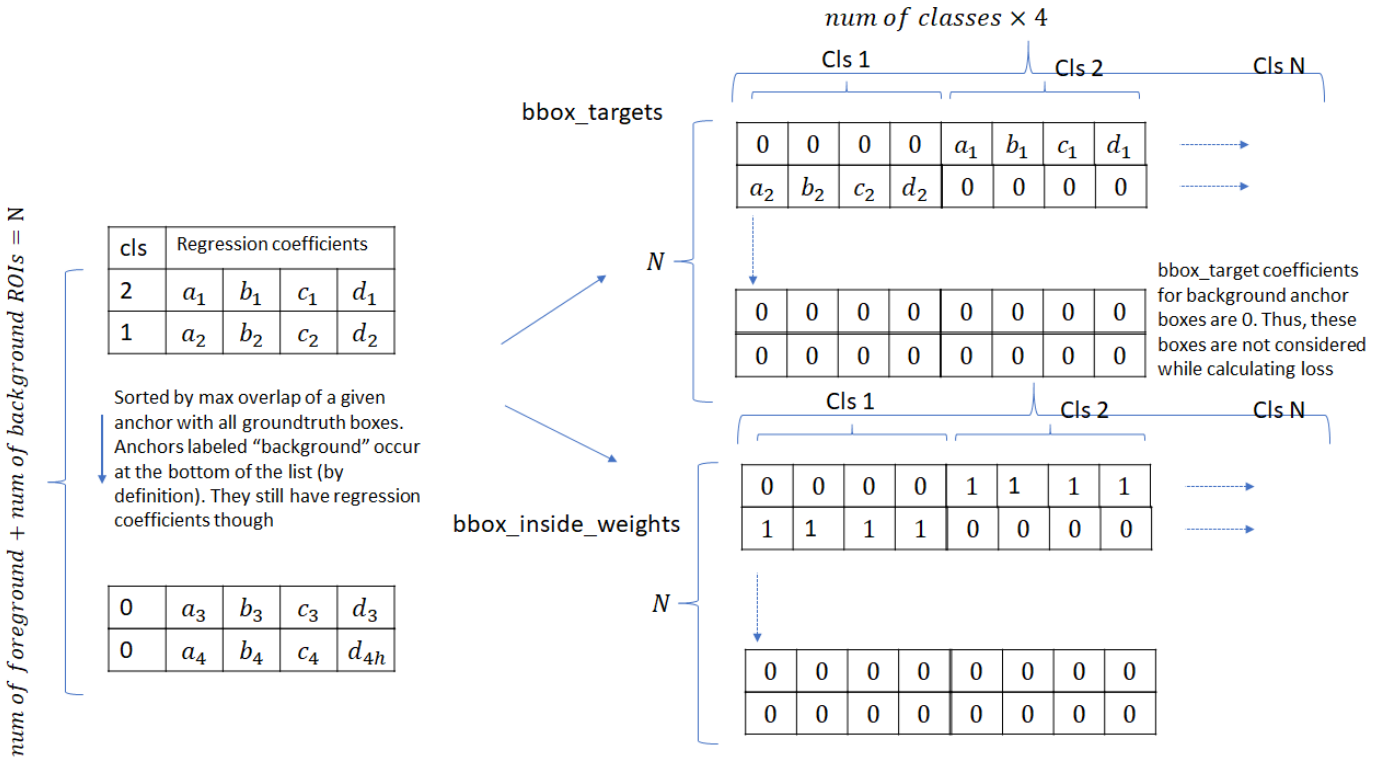
The goal of the proposal target layer is to select promising ROIs from the list of ROIs output by the proposal layer. These promising ROIs will be used to perform crop pooling from the feature maps produced by the head layer and passed to the rest of the network (head_to_tail) that calculates predicted class scores and box regression coefficients.

Similar to the anchor target layer, it is important to select good proposals (those that have significant overlap with gt boxes) to pass on to the classification layer. Otherwise, we'll be asking the classification layer to learn a "hopeless learning task".

The proposal target layer starts with the ROIs computed by the proposal layer. Using the max overlap of each ROI with all ground truth boxes, it categorizes the ROIs into background and foreground ROIs. Foreground ROIs are those for which max overlap exceeds a threshold (TRAIN.FG_THRESH, default: 0.5). Background ROIs are those whose max overlap falls between TRAIN.BG_THRESH_LO and TRAIN.BG_THRESH_HI (default 0.1, 0.5 respectively). This is an example of "hard negative mining" used to present difficult background examples to the classifier.

There is some additional logic that tries to make sure that the total number of foreground and background region is constant. In case too few background regions are found, it tries to fill in the batch by randomly repeating some background indices to make up for the shortfall.

Next, bounding box target regression targets are computed between each ROI and the closest matching ground truth box (this includes the background ROIs also, as an overlapping ground truth box exists for these ROIs also). These regression targets are expanded for all classes as shown in the figure below.



the bbox_inside_weights array acts as a mask. It is 1 only for the correct class for each foreground ROI. It is zero for the background ROIs as well. Thus, while computing the bounding box regression component of the classification layer loss, only the regression coefficients for the foreground regions are taken into account. This is not the case for the classification loss – the background ROIs are included as well as they belong to the "background" class.

Input:

- ROIs produced by the proposal layer
- ground truth information

Output:

- Selected foreground and background ROIs that meet overlap criteria.
- Class specific target regression coefficients for the ROIs

Parameters:

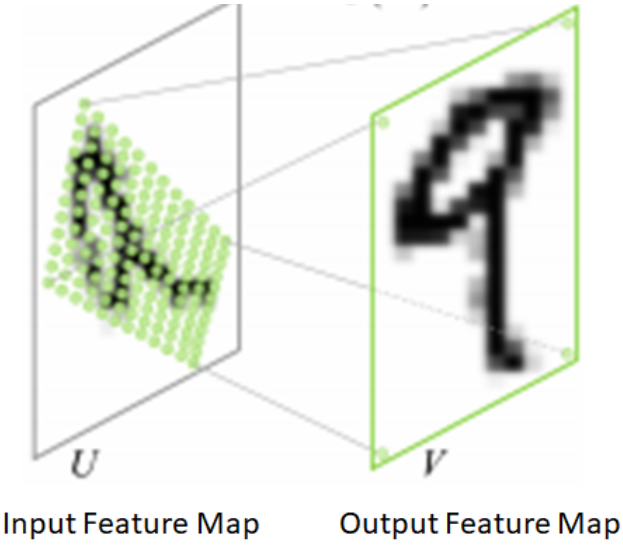
- TRAIN.FG_THRESH: (default: 0.5) Used to select foreground ROIs. ROIs whose max overlap with a ground truth box exceeds FG_THRESH are marked foreground

- TRAIN.BG_THRESH_HI: (default 0.5)
- TRAIN.BG_THRESH_LO: (default 0.1) These two thresholds are used to select background ROIs. ROIs whose max overlap falls between BG_THRESH_HI and BG_THRESH_LO are marked background
- TRAIN.BATCH_SIZE: (default 128) Maximum number of foreground and background boxes selected.
- TRAIN.FG_FRACTION: (default 0.25). Number of foreground boxes can't exceed BATCH_SIZE*FG_FRACTION

Crop Pooling

Proposal target layer produces promising ROIs for us to classify along with the associated class labels and regression coefficients that are used during training. The next step is to extract the regions corresponding to these ROIs from the convolutional feature maps produced by the head network. The extracted feature maps are then run through the rest of the network ("tail" in the network diagram shown above) to produce object class probability distribution and regression coefficients for each ROI. The job of the Crop Pooling layer is to perform region extraction from the convolutional feature maps.

The key ideas behind crop pooling are described in the paper on "Spatial Transformation Networks" (Anon. 2016)*. The goal is to apply a warping function (described by a 2×3 affine transformation matrix) to an input feature map to output a warped feature map. This is shown in the figure below



There are two steps involved in crop pooling:

1. For a set of target coordinates, apply the given affine transformation to produce a grid of source coordinates.
   $\begin{bmatrix} x_i^s \\ y_i^s \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{bmatrix} x_i^t \\ y_i^t \\ 1 \end{bmatrix}$. Here $x_i^s, y_i^s, x_i^t, y_i^t$ are height/width normalized coordinates (similar to the texture coordinates used in graphics), so $-1 \leq x_i^s, y_i^s, x_i^t, y_i^t \leq 1$.
2. In the second step, the input (source) map is sampled at the source coordinates to produce the output (destination) map. In this step, each (x_i^s, y_i^s) coordinate defines the spatial location in the

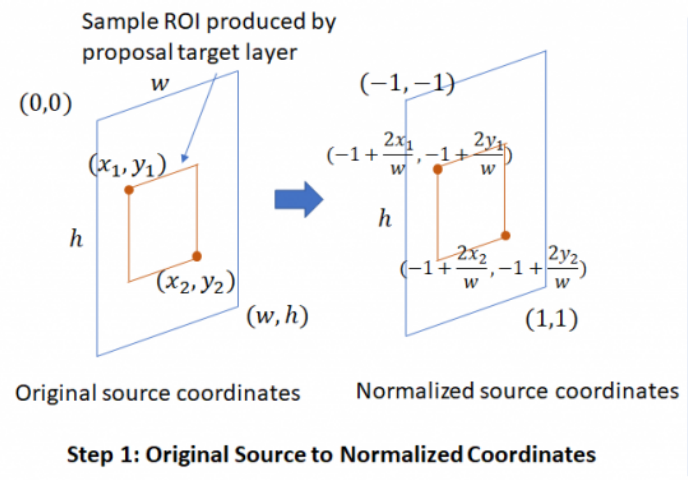
input where a sampling kernel (for example bi-linear sampling kernel) is applied to get the value at a particular pixel in the output feature map.

The sampling methodology described in the spatial transformation gives a differentiable sampling mechanism allowing for loss gradients to flow back to the input feature map and the sampling grid coordinates.

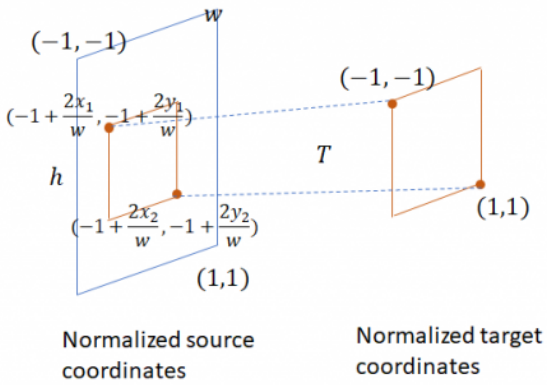
Fortunately, crop pooling is implementated in PyTorch and the API consists of two functions that mirror these two steps. `torch.nn.functional.affine_grid` takes an affine transformation matrix and produces a set of sampling coordinates and `torch.nn.functional.grid_sample` samples the grid at those coordinates. Back-propagating gradients during the backward step is handled automatically by pyTorch.

To use crop pooling, we need to do the following:

1. Divide the ROI coordinates by the stride length of the "head" network. The coordinates of the ROIs produced by the proposal target layer are in the original image space (! 800 $\times$ 600). To bring these coordinates into the space of the output feature maps produced by "head", we must divide them by the stride length (16 in the current implementation).
2. To use the API shown above, we need the affine transformation matrix. This affine transformation matrix is computed as shown below
3. We also need the number of points in the x and y dimensions on the target feature map. This is provided by the configuration parameter cfg.POOLING_SIZE (default 7). Thus, during crop pooling, non-square ROIs are used to crop out regions from the convolution feature map which are warped to square windows of constant size. This warping must be done as the output of crop pooling is passed to further convolutional and fully connected layers which need input of a fixed dimension.



Step 1: Original Source to Normalized Coordinates



Step 2: Normalized Target to Source Transformation

Thus, we are looking for a transformation T that maps the normalized target to normalized source coordinates

$$T\begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} x_s \\ y_s \end{bmatrix}$$

Since we are interested in a transformation without skew, our T has four parameters:

$$\begin{bmatrix} a & 0 & t_x \\ 0 & b & t_y \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} x_s \\ y_s \end{bmatrix}$$

We already know the coordinates of the two corner points for the source and target maps, we can use that information to solve for T

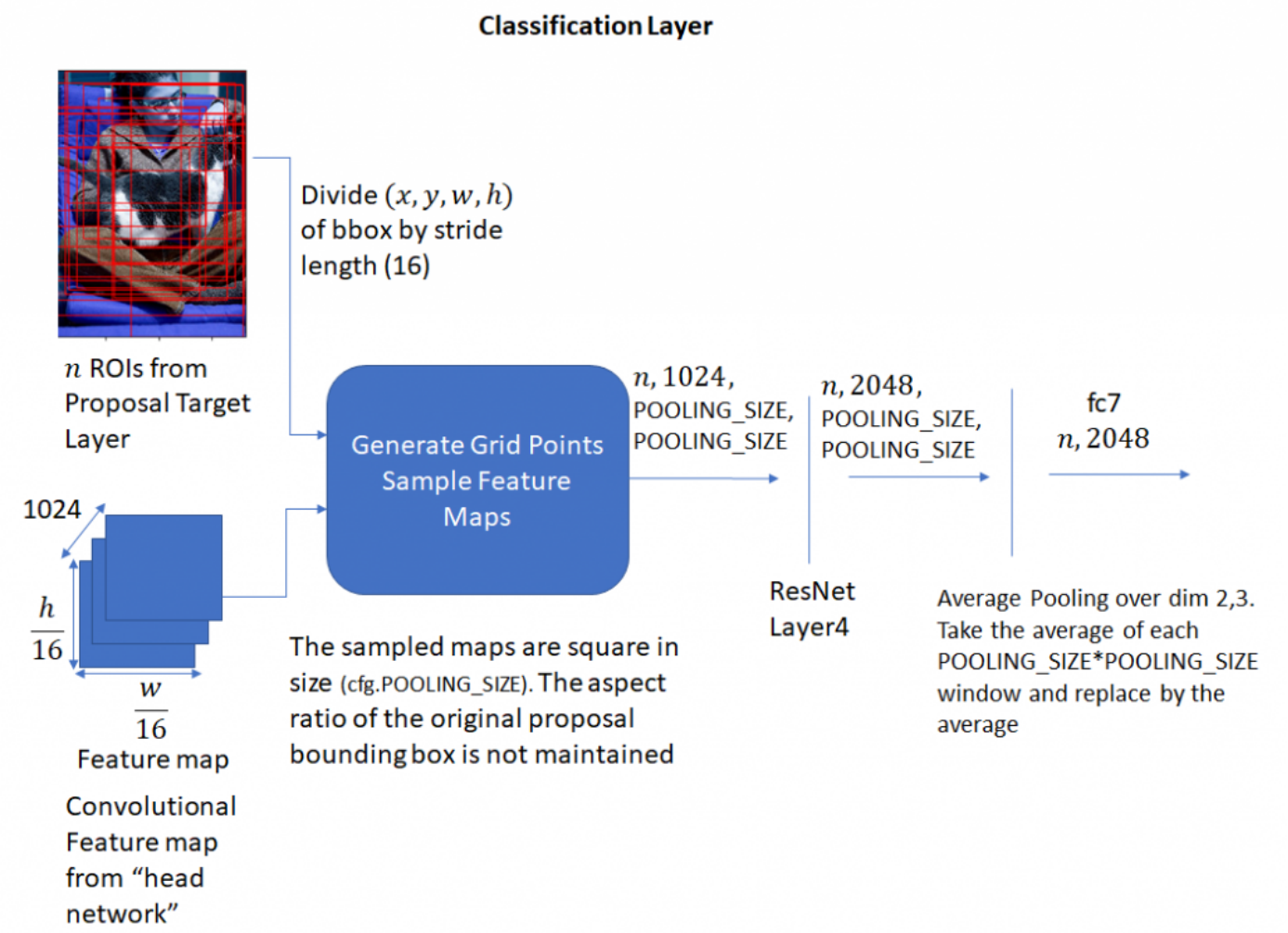
$$\begin{bmatrix} a & 0 & t_x \\ 0 & b & t_y \end{bmatrix} \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} -1+\frac{2x_1}{w} & -1+\frac{2x_2}{w} \\ -1+\frac{2y_1}{w} & -1+\frac{2y_2}{w} \end{bmatrix}$$

Solving, we get

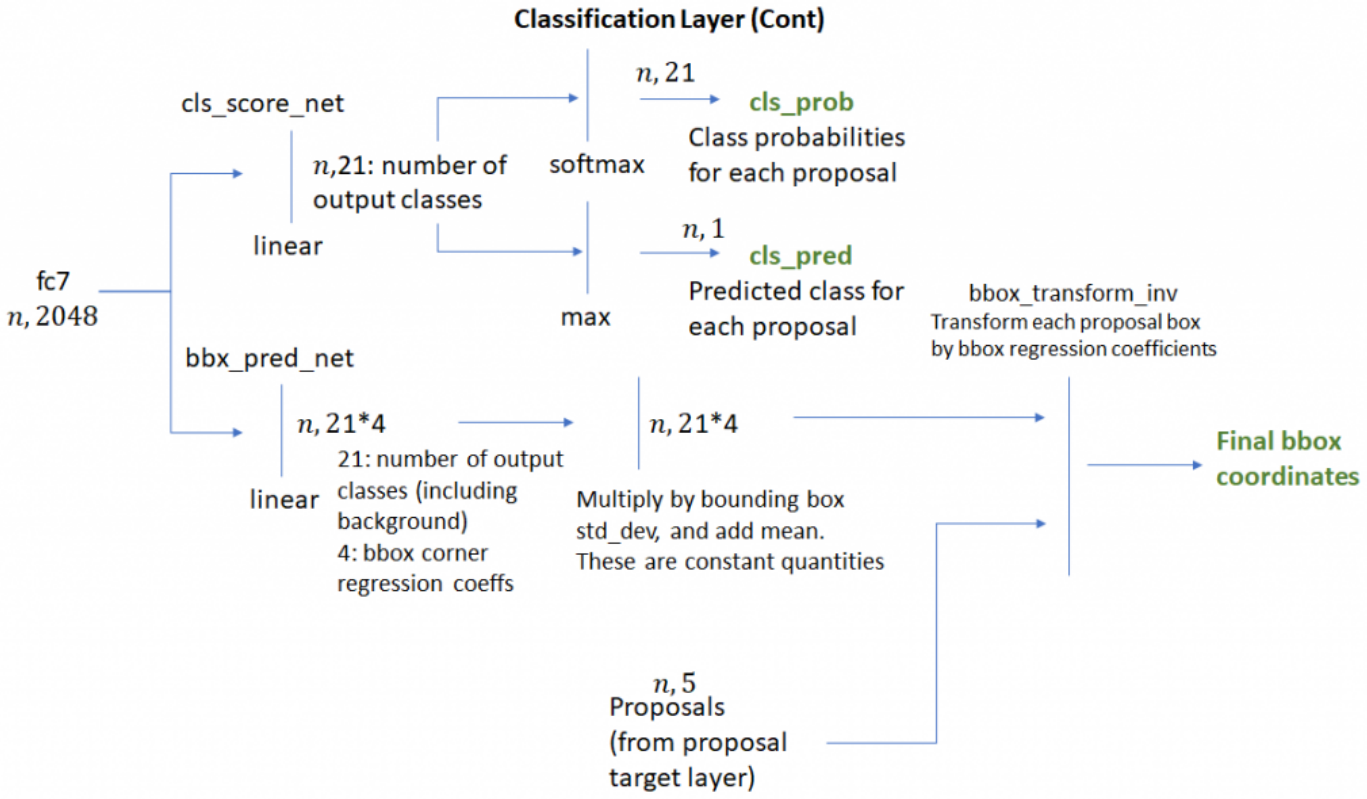
$$\begin{bmatrix} a & 0 & t_x \\ 0 & b & t_y \end{bmatrix} = \begin{bmatrix} \frac{x_2-x_1}{w} & 0 & -1+\frac{x_2+x_1}{w} \\ 0 & \frac{y_2-y_1}{h} & -1+\frac{y_2+y_1}{h} \end{bmatrix}$$

Classification Layer

The crop pooling layer takes the ROI boxes output by the proposal target layer and the convolutional feature maps output by the "head" network and outputs square feature maps. The feature maps are then passed through layer 4 of ResNet following by average pooling along the spatial dimensions. The result (called "fc7" in code) is a one-dimensional feature vector for each ROI. This process is shown below.



The feature vector is then passed through two fully connected layers – bbox_pred_net and cls_score_net. The cls_score_net layer produces the class scores for each bounding box (which can be converted into probabilities by applying softmax). The bbox_pred_net layer produces the class specific bounding box regression coefficients which are combined with the original bounding box coordinates produced by the proposal target layer to produce the final bounding boxes. These steps are shown below.



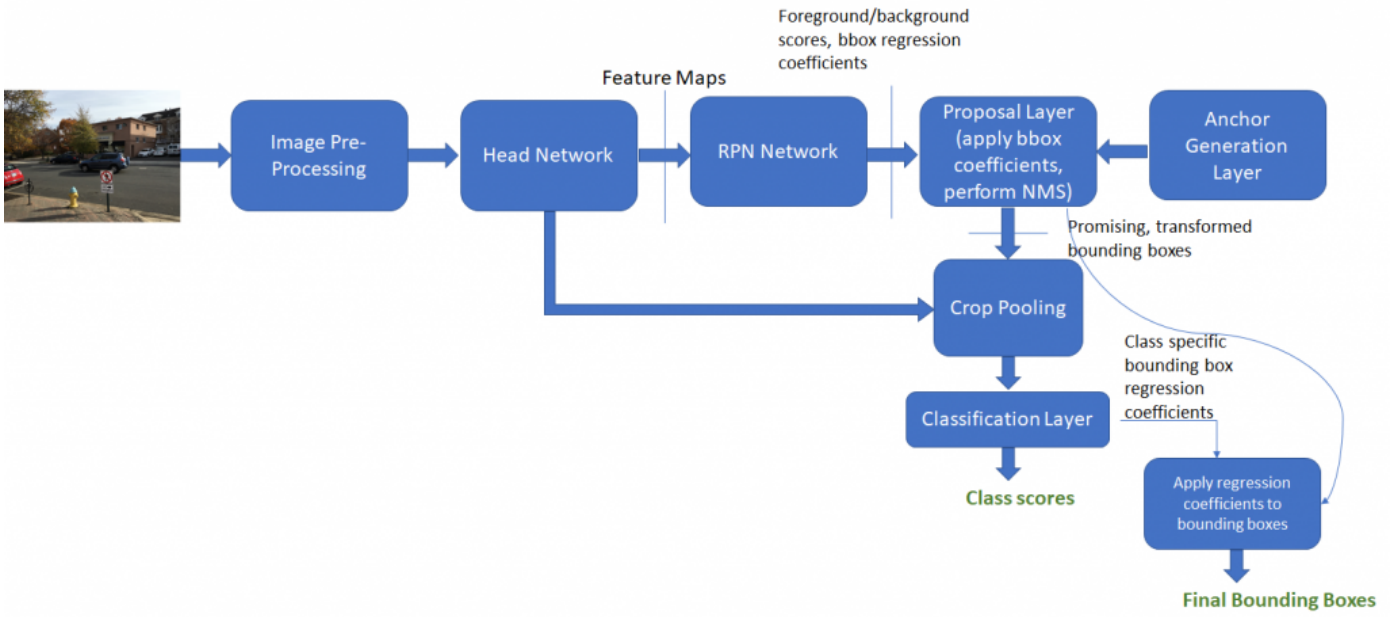
It's good to recall the difference between the two sets of bounding box regression coefficients – one set produced by the RPN network and the second set produced by the classification network. The first set is used to train the RPN layer to produce good foreground bounding boxes (that fit more tightly around object boundaries). The target regression coefficients i.e., the coefficients needed to align a ROI box with its closest matching ground truth bounding box are generated by the anchor target layer. It is difficult to identify precisely how this learning takes place, but I'd imagine the RPN convolutional and fully connected layers learn how to interpret the various image features generated by the neural network into deciphering good object bounding boxes. When we consider Inference in the next section, we'll see how these regression coefficients are used.

The second set of bounding box coefficients is generated by the classification layer. These coefficients are class specific, i.e., one set of coefficients are generated per object class for each ROI box. The target regression coefficients for these are generated by the proposal target layer.

It is interesting to note that while training the classification layer, the error gradients propagate to the RPN network as well. This is because the ROI box coordinates used during crop pooling are themselves network outputs as they are a result of applying the regression coefficients generated by the RPN network to the anchor boxes. During back-propagation, the error gradients will propagate back through the crop-pooling layer to the RPN layer. Calculating and applying these gradients would be quite tricky to implement, however thankfully the crop pooling API is provided by PyTorch as a built-in module and the details of calculating and applying the gradients are handled internally. This point is discussed in Section 3.2 (iii) of the Faster RCNN paper (Ren et al. 2015)*.

Implementation Details: Inference

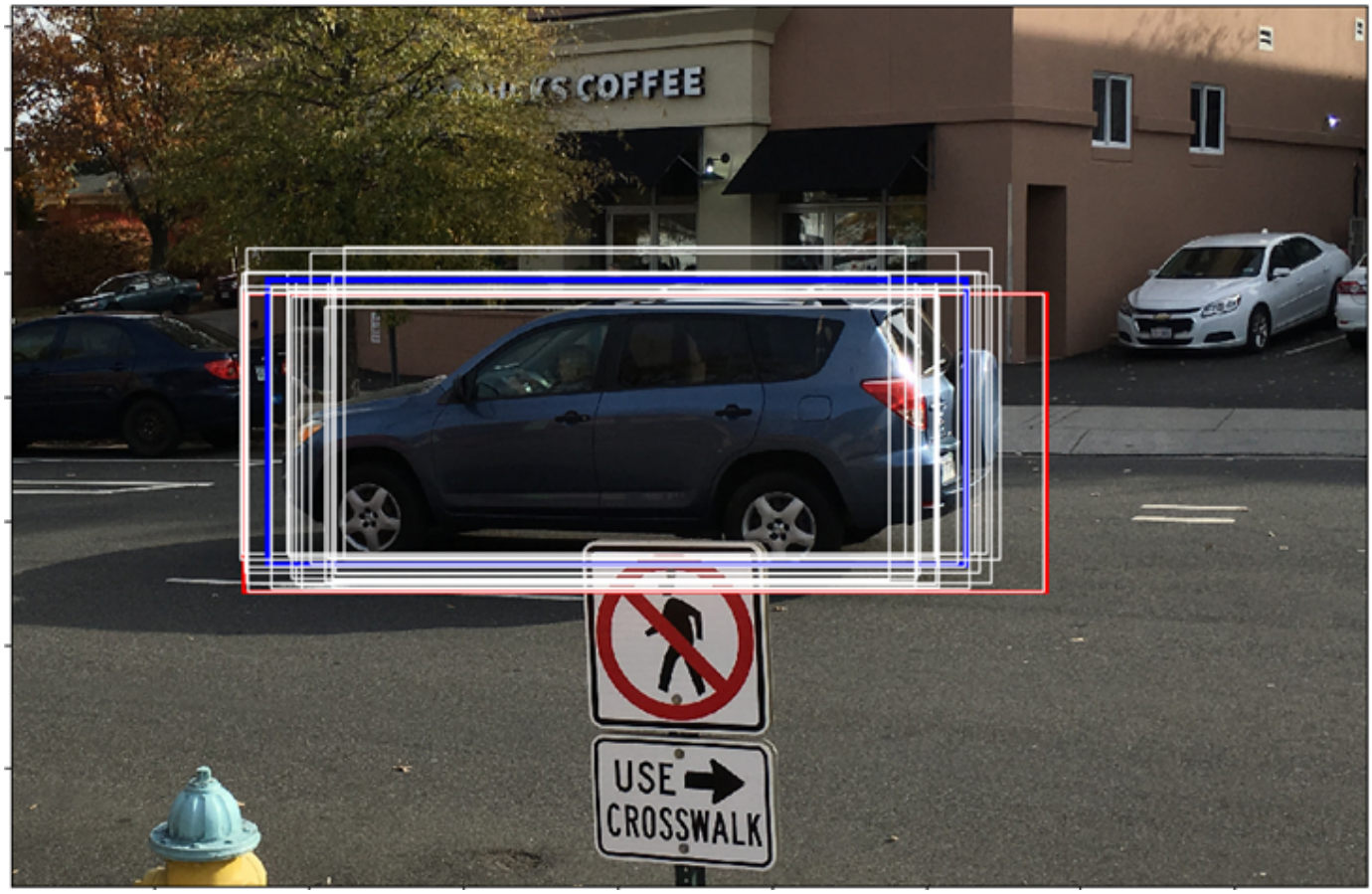
The steps carried out during inference are shown below



Anchor target and proposal target layers are not used. The RPN network is supposed to have learnt how to classify the anchor boxes into background and foreground boxes and generate good bounding box coefficients. The proposal layer simply applies the bounding box coefficients to the top ranking anchor boxes and performs NMS to eliminate boxes with a large amount of overlap. The output of these steps are shown below for additional clarity. The resulting boxes are sent to the classification layer where class scores and class specific bounding box regression coefficients are generated.

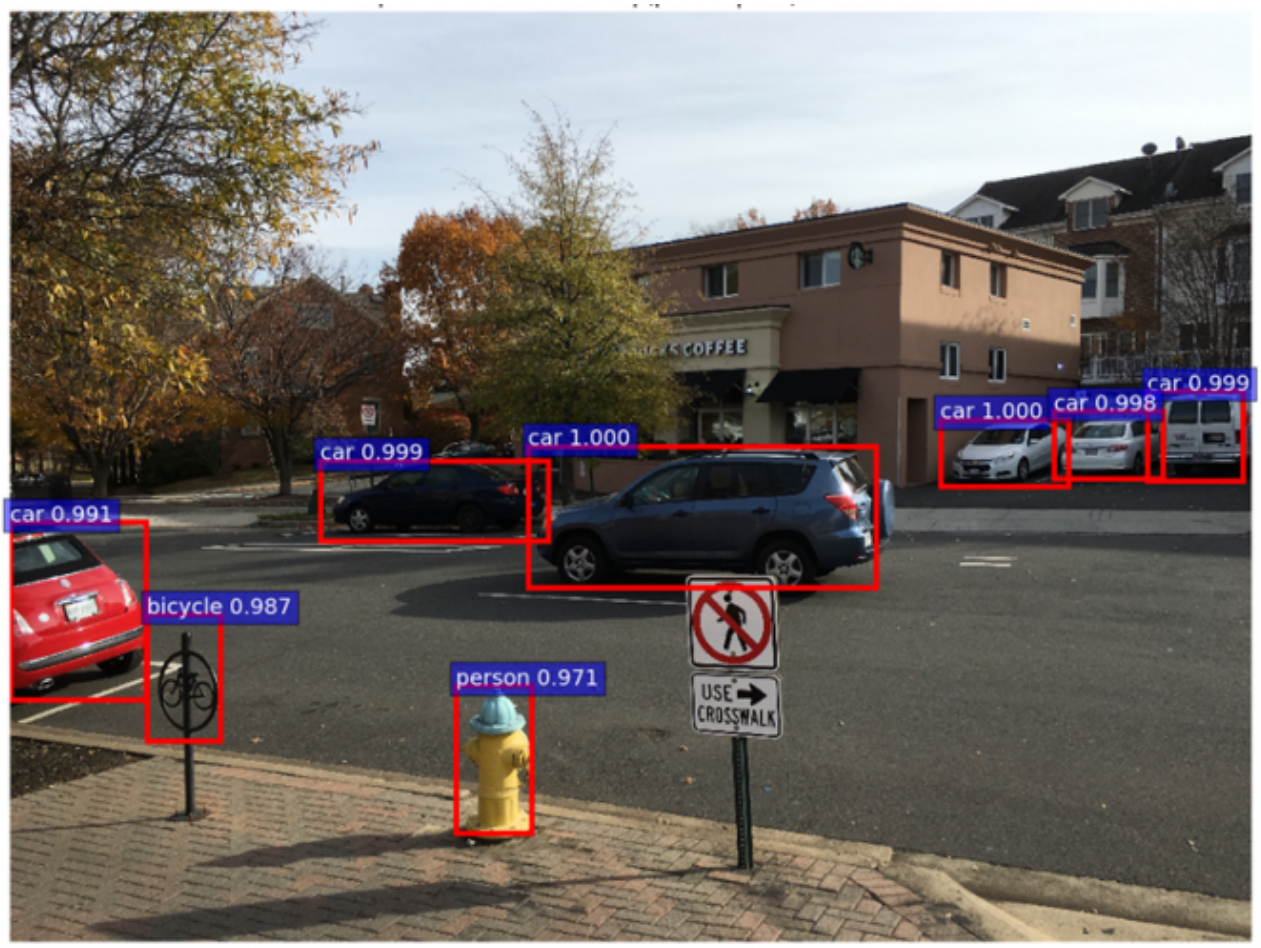
The red boxes show the top 6 anchors ranked by score. Green boxes show the anchor boxes after applying the regression parameters computed by the RPN network. The green boxes appear to fit the underlying object more tightly. Note that after applying the regression parameters, a rectangle remains a rectangle, i.e., there is no shear. Also note the significant overlap between rectangles. This redundancy is addressed by applying non-maxima suppression

Red boxes show the top 5 bounding boxes before NMS, green boxes show the top 5 boxes after NMS. By suppressing overlapping boxes, other boxes (lower in the scores list) get a chance to move up



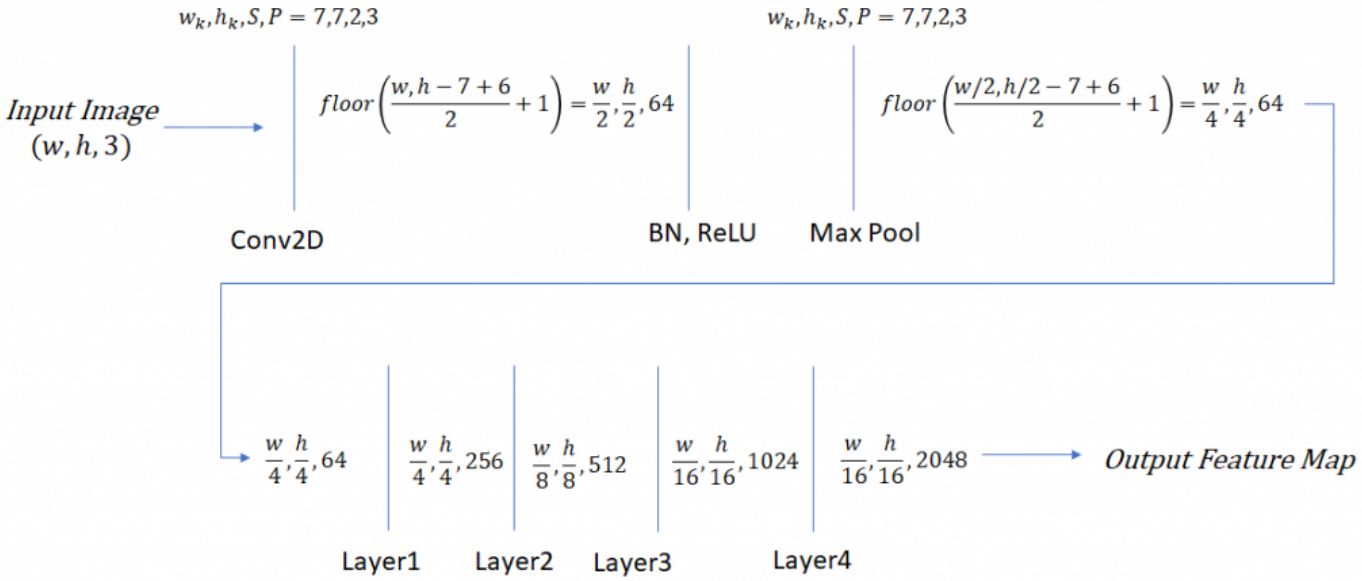
From the final classification scores array (dim: n, 21), we select the column corresponding to a certain foreground object, say car. Then, we select the row corresponding to the max score in this array. This row corresponds to the proposal that is most likely to be a car. Let the index of this row be car_score_max_idx Now, let the array of final bounding box coordinates (after applying the regression coefficients) be bboxes (dim: n,21*4). From this array, we select the row corresponding to car_score_max_idx. We expect that the bounding box corresponding to the car column should fit the car in the test image better than the other bounding boxes (which correspond to the wrong object classes). This is indeed the case. The red box corresponds to the original proposal box, the blue box is the calculated bounding box for the car class and the white boxes correspond to the other (incorrect) foreground classes. It can be seen that the blue box fits the actual car better than the other boxes.

For showing the final classification results, we apply another round of NMS and apply an object detection threshold to the class scores. We then draw all transformed bounding boxes corresponding to the ROIs that meet the detection threshold. The result is shown below.



Appendix

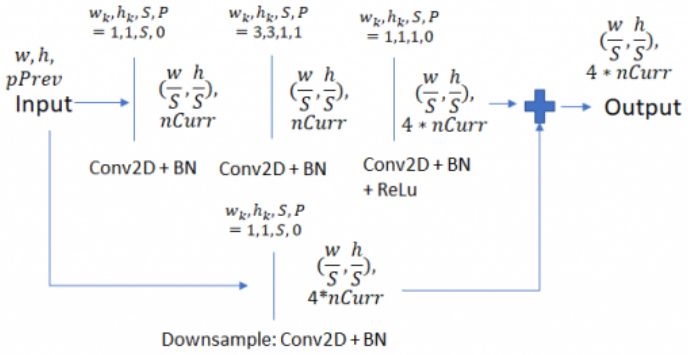
ResNet 50 Network Architecture



For a conv2d layer with filter parameters (windows size, stride length, padding) w_k, h_k, S, P, if input dimension = w, h, output dimension = $\frac{(w,h)-(w_k,h_k)+2P}{S}+1$

Each layer consists of several "bottlenecks". A layer accepts the following parameters:

- $nPrev$: number of output planes in the previous layer
- $nCurr$: number of output planes for the current layer. Somewhat confusingly, the actual number of output planes will be $4*nCurr$
- Stride S
- Number of "blocks" in a bottleneck



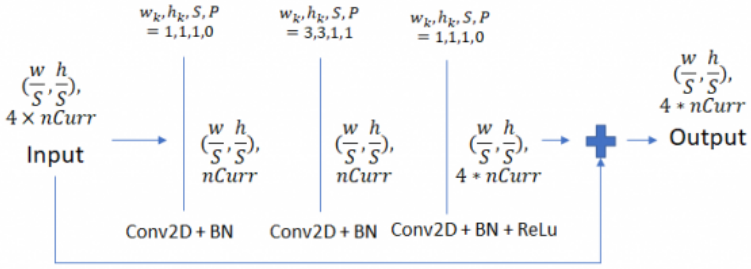
The first block consists of a "Downsample" operation if:

- $S \neq 1$
- $pPrev \neq 4 \times nCurr$

This is needed because a resnet adds the input to the output of a block and if the dimensions of the input (width, height and number of planes) don't match the output, a transformation must be added. The Downsample is this transformation

For the subsequent blocks, $S = 1, pPrev = 4 \times nCurr$, so the downsample operation is not needed.

The final output of the layer has the following dimensions: $(\frac{w,h}{S}, 4 \times nCurr)$



Non-Maximum Suppression (NMS)

Non-maximum suppression is a technique used to reduce the number of candidate boxes by eliminating boxes that overlap by an amount larger than a threhold. The boxes are first sorted by some criteria (usually the y coordinate of the bottom right corner). We then go through the list of boxes and suppress those boxes whose IoU overlap with the box under consideration exceeds a threshold. Sorting the boxes by the y coordinate results in the lowest box among a set of overlapping boxes being retained. This may

not always be the desired outcome. NMS used in R-CNN sorts the boxes by the foreground score. This results in the box with the highest score among a set of overlapping boxes being retained. The figures below show the difference between the two approaches. The numbers in black are the foreground scores for each box. The image on the right shows the result of applying NMS to the image on left. The first figure uses standard NMS (boxes are ranked by y coordinate of bottom right corner). This results in the box with a lower score being retained. The second figure uses modified NMS (boxes are ranked by foreground scores). This results in the box with the highest foreground score being retained, which is more desirable. In both cases, the overlap between the boxes is assumed to be higher than the NMS overlap threhold.

Standard NMS. Boxes are ranked according to y coordinate of lower right corner

Modified NMS. Boxes are ranked by their foreground ROI score

Bibliography

Anon. 2014. . October 23. https://arxiv.org/pdf/1311.2524.pdf.

Anon. 2016. . February 5. https://arxiv.org/pdf/1506.02025.pdf.
Anon. . http://link.springer.com/article/10.1007/s11263-013-0620-5.
Anon. Object Detection with Discriminatively Trained Part-Based Models - IEEE Journals & Magazine. https://doi.org/10.1109/TPAMI.2009.167.
Girshick, Ross. 2015. Fast R-CNN. *arXiv.org*. April 30. https://arxiv.org/abs/1504.08083.
Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2013. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv.org*. November 11. https://arxiv.org/abs/1311.2524.
Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv.org*. June 4. https://arxiv.org/abs/1506.01497.
Yosinski, Jason, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? *arXiv.org*. November 6. https://arxiv.org/abs/1411.1792.

13 COMMENTS

ylxie

APRIL 4, 2018 AT 4:45 PM

this is an extraordinary blog! Never seen such a detailed dig in of the faster rcnn

REPLY

ankur ☆

APRIL 4, 2018 AT 5:37 PM

Thanks! Glad you found it useful.

REPLY

Chris

JUNE 9, 2018 AT 1:47 AM

After reading your excellent job, I just wondering how to visualize the middle stage of the model, like how to generate the images in your blogs' Inference part. And how to project rois to the original image. Thanks a lot and I am so willing to hearing from your reply!

REPLY

Suneel Marthi

JUNE 27, 2018 AT 9:30 PM

Excellent blog Ankur – mind if I were to borrow some of ur visuals and graphs for my work ? 🙂

REPLY

ankur6ue ☆

JUNE 27, 2018 AT 9:54 PM

Glad you found it useful. You are welcome to use the visuals/graphs. If possible, would be great if you can mention the blog in your references 🙂

REPLY

Babatunde Ishola

JULY 24, 2018 AT 3:37 PM

I think this is the best blog I have seen on Faster RCNN. I especially love the mathematical details. Great write-up!

REPLY

ankur6ue ☆

JULY 24, 2018 AT 3:49 PM

Thank you very much. I have benefitted from blogs that others have written and want to contribute back to the AI community. Glad you found it useful.

REPLY

Aruni RC

AUGUST 6, 2018 AT 4:36 PM

Excellent post. A minor correction: in Bounding Box Regression Coefficients section, the P_w and P_h are not defined prior to usage.

REPLY

ankur6ue ☆

AUGUST 6, 2018 AT 4:52 PM

Thank you very much for pointing this out. I corrected the typo.

REPLY

zorrocai

AUGUST 13, 2018 AT 8:55 AM

Appreciate your excellent job! This is the best blog about Faster RCNN.
And I have two puzzles that may help improve the quality of the blog.
The first one is about the training of faster rcnn. In the original paper, it wrote that there are four steps in training phase:
1.train RPN, initialized with ImgeNet pre-trained model;
2.train a separate detection network by fast rcnn using proposals generated by step1 RPN, initialized by ImageNet pre-trained model;
3.fix conv layer, fine-tune unique layers to RPN, initialized by detector network in step2;
4.fix conv layer, fine-tune fc-layers of fast rcnn.
While the blog writes that "R-CNN is able to train both the region proposal network and the classification network in the same step.". So, what is the difference between those two methods?
The second puzzle is regarding Proposal layer. The Proposal layer prunes the number of anchors by applying NMS using the probability of an anchor being a foreground region. So, where does the probability come from?

REPLY

ankur6ue ☆

AUGUST 13, 2018 AT 7:49 PM

Thank you for your puzzles. I will let other readers answer the puzzles if they wish.

REPLY

JordiG

AUGUST 14, 2018 AT 3:48 PM

Excellent blog!
What is the relation between anchor scales and anchor sizes. In your example, you set anchor scales (8, 16, 32), that means that the anchor sizes are regions with 8^2, 16^2 and 32^2 pixels?
Thank you very much!

REPLY

ankur6ue ☆

AUGUST 16, 2018 AT 1:20 PM

The process of anchor box generation (see generate_anchors.py) proceeds as follows:

1. start with an anchor box of default size (16, 16) – corners: 0, 0, 15, 15, center: 7.5, 7.5
2. Generate three different aspect ratios of w, h: (23, 12), (16, 16), (11, 22). See function _ratio_enum. So now for each square anchor box, there are two rectangular boxes and one square box
3. For each of these boxes, generate scaled anchors (in the original image space). Default scales are 8, 16, 32, so for the first box with w, h = (23, 12), you'll get three scaled boxes with dimensions (184, 96), (369, 192) etc. You'll end up with 9 boxes in total.
4. Each of these 9 boxes are then translated across the image with a stride = feat_stride, see generate_anchor_pre in snippets.py.

Easiest way to see all this is to set breakpoints in the code and look at the contents of the various arrays. Hope this helps.

REPLY

Leave a Reply

Your email address will not be published.

Comment

Name *

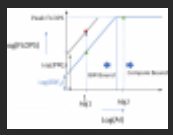
Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

OTHER MACHINE LEARNING POSTS

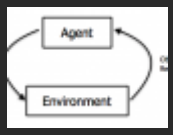


Understanding Roofline Charts

July 26, 2018 0

Data Augmentation in SSD (Single Shot Detector)

June 28, 2018 0



Efficiently Computing the Fisher Vector Product in TRPO

June 9, 2018 3

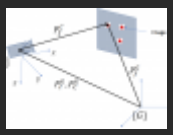
Initializing Weights for the Convolutional and Fully Connected Layers

April 9, 2018 0

Object Detection and Classification using R-CNNs

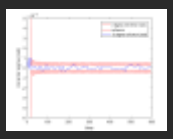
March 11, 2018 13

OTHER SENSOR FUSION RELATED POSTS



Sensor Fusion: Part 4

May 7, 2017 0



Sensor Fusion – Part 3: Implementation of Gyro-Accel Sensor Fusion

May 2, 2017 0

Sensor Fusion: Part 2 (combining Gyro-Accel data)

April 30, 2017 0

Sensor Fusion: Part 1

April 27, 2017 0