CPSC 8810 - Deep Learning
Deep Learning Model to Detect Cyberbully Actions in Images

Submitted By:
Vivek Koodli Udupa - (C12768888)
Shashi Shivaraju - (C88650674)

Clemson University

May 1, 2019

Abstract

This report explains the process involved in implementing a deep CNN and Faster-RCNN model in order to classify various cyberbully actions in an image and to detect the predator and victim in the same image.

1 Introduction

This report considers the problem of detection and classification of cyberbully actions and to identify the predator and victim in a bullying image using Deep learning models.

Cyberbullying is bullying that takes place over digital devices like cell phones, computers, and tablets. Cyberbullying can occur through SMS, Text, and apps, or online in social media, forums, or gaming where people can view, participate in, or share content. Cyberbullying includes sending, posting, or sharing negative, harmful, false, or mean content about someone else. Some cyberbullying crosses the line into unlawful or criminal behavior. With the prevalence of social media and digital forums, comments, photos, posts, and content shared by individuals can often be viewed by strangers as well as acquaintances. The content an individual shares online – both their personal content as well as any negative, mean, or hurtful content – creates a kind of permanent public record of their views, activities, and behavior. This public record can be thought of as an online reputation, which may be accessible to schools, employers, colleges, clubs, and others who may be researching an individual now or in the future. Cyberbullying can harm the online reputations of everyone involved – not just the person being bullied, but those doing the bullying or participating in it.[1]

Deep learning is a subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data. The term 'deep learning' because the neural networks have various (deep) layers that enable learning. Deep learning allows machines to solve complex problems even when using a data set that is very diverse, unstructured and inter-connected. The more deep learning algorithms learn, the better they perform. [2]

This report describes modeling of a convolutional neural network for detecting and classifying cyberbully actions for a given image along with a RCNN model inorder to identify predator and victim present in that image. The cyberbullying actions considered in this project are laughing, pulling-hair, quarrel, slapping, punching, stabbing, gossiping, strangle and isolation. The CNN is trained using the provided image dataset which contain above mentioned 9 categories of cyberbully actions in them. The proposed RCNN is trained using provided images with ground truth bounding boxes for predator and victim classes.

2 Methods

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.[3] A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, ReLU layer i.e. activation function, pooling layers, fully connected layers and normalization layers[4], which have been described in detain in section 2.1.

The implemented CNN model classifies the given input image into one of the nine categories of cyberbully actions. In order to further identify the predator and victim in the classified cyberbully image, an object detection model must be used. This can be achieved by using a Region based Convolutional Neural Network(RCNN).

In R-CNN, the CNN is forced to focus on a single region at a time because it minimizes the interference and it is expected that only a single object of interest will dominate in a given region. These regions in the R-CNN are detected by selective search algorithm followed by resizing so that the regions are of equal size before they are fed to a CNN for classification and bounding box regression. The drawback of RCNN is that selective search algorithm is computation expensive. Thus in order to overcome this drawback a Faster RCNN(F-RCNN) model is proposed.

F-RCNN approach is similar to the R-CNN algorithm but instead of feeding the region proposals to the CNN, the whole input image is fed to the CNN to generate a convolutional feature map. From the convolutional feature map, region of proposals are identified and warped into squares(Bounding Boxes) and by using a RoI(Region of Interest) pooling layer, region proposals are reshaped into fixed sizes so that it can be fed into a fully connected layer. Class of the proposed region and the offset of the bounding boxes are calculated by performing softmax on the RoI feature vector(Final layer of fully connected network)[6]

The implementation of our models based on CNN and FRCNN algorithm using an open source machine learning library PyTorch is described in the below sections.

2.1 Implementation of CNN Model

Our CNN model for classification consists of the following layers:

1. **Convolutional Layer:** The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels). During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

2. **ReLU layer:** ReLU is the abbreviation of rectified linear unit, which applies the non-saturating activation function

$$f(x) = max(0, x) \tag{1}$$

   It effectively removes negative values from an activation map by setting them to zero. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.

3. **Max Pooling:** Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. Max pooling is the most common non-linear function for down-sampling. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum.

4. **Fully Connected Layer:** Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular (non-convolutional) artificial neural networks. Their activations can thus be computed as an affine transformation, with matrix multiplication followed by a bias offset.

5. **Dropout Layer:** A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training. This is called dropout and offers a very computationally cheap and remarkably effective regularization method to reduce over fitting and generalization error in deep neural networks.[5]

The implementation details of our model is as follows:

1. **Image Pre-Processing:** The given image is resized to (256 x 256)pixels. Then the images are randomly flipped or rotated for the purpose of data augmentation. Then it is converted to a PyTorch tensor image and its values are normalized with a mean of 0.5 and Standard Deviation of 0.5.

2. **Convolution Layer 1_1:** The input to this layer is a preprocessed 3 channel tensor image from the previous layer. This layer performs 2D convolution using a 3x3 kernel with stride set to 1 and padding enabled to produce an output which is a 64 channel feature map. Xavier initialization is used to initialize the weights of this layer.

3. **ReLU Layer 1:** This layer applies a relu activation function to the 16 channel feature map.

4. **Convolution Layer 1_2 and ReLU:** The input to this layer is a 64 channel feature map from the previous layer. This layer performs 2D convolution using a 3x3 kernel with stride set to 1 and padding enabled to produce an output which is a 64 channel feature map. The output is normalized using ReLU.

5. **Batch Normalization:** This layer performs batch normalization inorder to avoid overfitting.

6. **Max Pooling Layer 1:** This layer down-samples the 256 x 256 64 channel feature map to 128 x 128 64 channel feature map.

7. **Convolution Layer 2_1 and ReLU:** The input to this layer is the 64 channel 128 x 128 feature map from the previous layer. This layer performs 2D convolution using a 3 x 3 kernel with stride set to 1 and padding enabled to produce an output which is a 128 channel feature map. ReLU activation is used to normalize the outputs.

8. **Convolution Layer 2_2 and ReLU:** The input to this layer is the 128 channel 128 x 128 feature map from the previous layer. This layer performs 2D convolution using a 3 x 3 kernel with stride set to 1 and padding enabled to produce an output which is a 32 channel feature map. ReLU activation is used to normalize the outputs.

9. **Batch Normalization:** This layer performs batch normalization inorder to avoid overfitting.

10. **Max Pooling Layer 2:** This layer down-samples the 128 x 128 128 channel feature map to 64 x 64 128 channel feature map.

11. **Convolution 3 with ReLU:** Similar to above convolution layers, Convolution 3 has 3 convolution layers, Convolution Layer 3_1, Convolution Layer 3_2 and Convolution Layer 3_3. The output of this layer is 64 x 64 512 channel feature map.

12. **Max Pooling Layer 3:** This layer down-samples the 64 x 64 512 channel feature map to 32 x 32 512 channel feature map.

13. **Convolution 4 and Convolution 5 with ReLU:** These two layers have similar configurations and structure as the Convolution 3 layer. The final output is a 8 x 8 512 channel feature map.

14. **Flattening Layer:** This layer flattens the 2D feature map to 1D feature map.

15. **Dropout Layer:** This layer randomly zeros some of the element of the input tensor with probability 0.4.

16. **Fully Connected Layer 1 and Relu layer 3:** This layer maps the 1D feature map into 4096 neurons.

17. **Fully Connected Layer 2 and Relu layer 4:** This layer maps the 4096 neurons to 1000 neurons.

18. **Fully Connected Layer 3 and softmax:** This layer maps 1000 neurons into 10 categories of classification and softmax for normalization.

2.2 Training the CNN Model

To train a deep learning model, the following parameters are considered:

1. **Epoch:** An epoch describes the number of times the algorithm sees the entire data set. So, each time the algorithm has seen all samples in the dataset, an epoch has completed.

2. **Batch Size:** The total number of training examples present in a single batch, wherein a batch is a subset of the entire data set.

3. **Iteration:** The number of batches needed to complete one Epoch.

4. **Learning Rate:** The learning rate or step size in machine learning is hyper-parameter which determines to what extent newly acquired information overrides old information.

The implemented model is trained using the below mentioned configuration:

1. Epoch = 100
2. Batch Size = 10
3. Learning Rate = 0.001

The model is trained with the given training dataset as per the below mentioned algorithm:

1. Initialize the CNN model with default parameters.
2. Create an instance of Adam optimizer for setting the learning rate
3. Create an instance of cross entropy loss
4. Initialize the optimizer with zero gradients
5. Feed a training input image from the current batch to the model to perform forward propagation
6. After the completion of forward propagation, calculate the cross entropy loss
7. Perform back propagation to minimize the loss
8. Update gradients
9. Iterate through step 4 for all the batches in the training dataset
10. Repeat the above steps for the given number of epochs
11. Save the trained model for testing purpose

2.3 Implementation of FRCNN Model

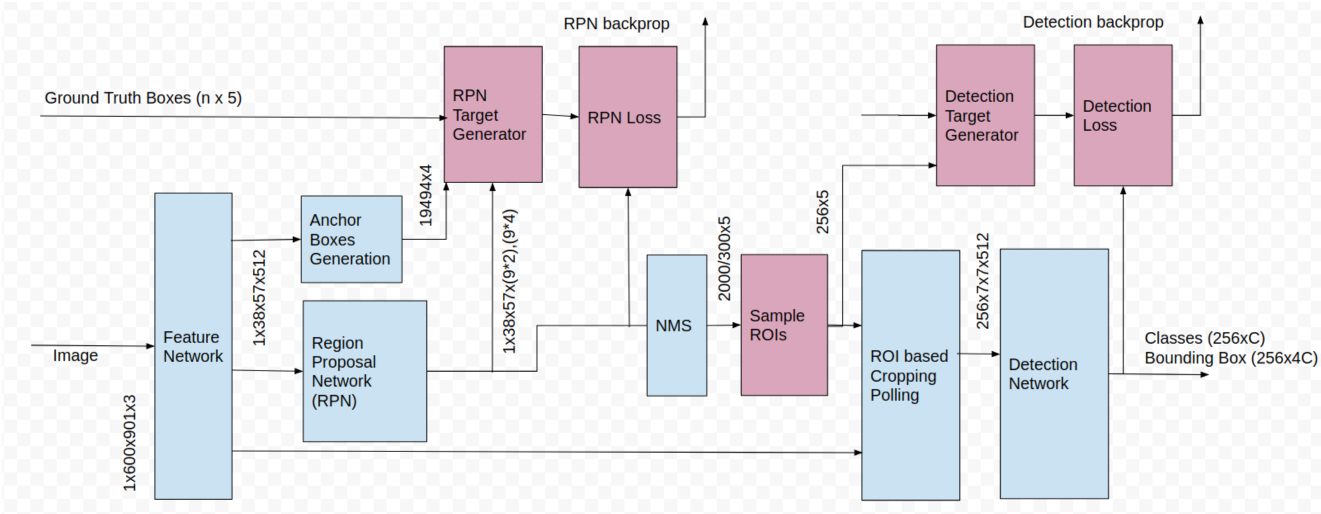

Figure 1: Faster-RCNN block diagram. The magenta colored blocks are active only during training. The numbers indicate size of the tensors[7].

The architecture for the object detection model, FRCNN is shown in Figure 1. To detect objects in the given image, the Faster RCNN uses two models which are RPN for generating region proposals and another detection model which uses generated proposals to identify objects[10]. The basic building blocks of FRCNN are explained in detail below:

Region Proposal Network(RPN):

1. In step 1, the input image goes through a CNN which will output a set of convolutional feature maps.

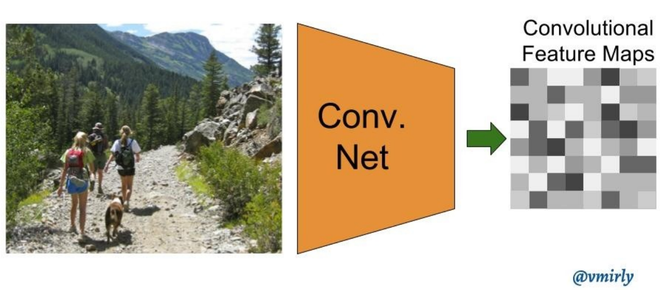

Figure 2: CNN layer with the feature map output

2. In step 2, a sliding window is run spatially on these feature maps. The size of sliding window is n x n (generally 3x3). For each sliding window, a set of 9 anchors are generated which all

have the same center (x_a, y_a) but with 3 different aspect ratios and 3 different scales as shown below. Note that all these coordinates are computed with respect to the original image.

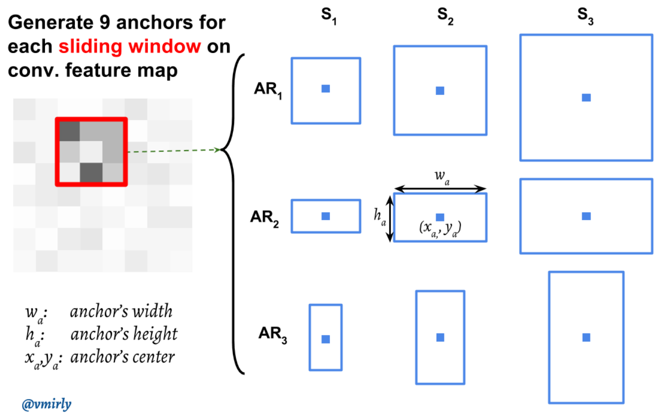

Figure 3: Anchors

For each anchor, a value p^* is computed which indicates how much the anchor overlaps with the groundtruth boxes. p^* is 1 if IoU > 0.7, -1 if IoU < 0.3 and 0 otherwise. where IoU is the intersection over union which is defined as: $IoU = \frac{\text{Anchor} \cap \text{GT Box}}{\text{Anchor} \cup \text{GT Box}}$

3. In step 3, Finally, the 3×3 spatial features extracted from those convolution feature maps (shown above within red box) are fed to a smaller network which has two tasks: classification (cls) and regression (reg). The output of regressor determines a predicted bounding-box (x, y, w, h), The output of classification sub-network is a probability p indicating whether the predicted box contains an object (1) or it is from background (0 for no object).

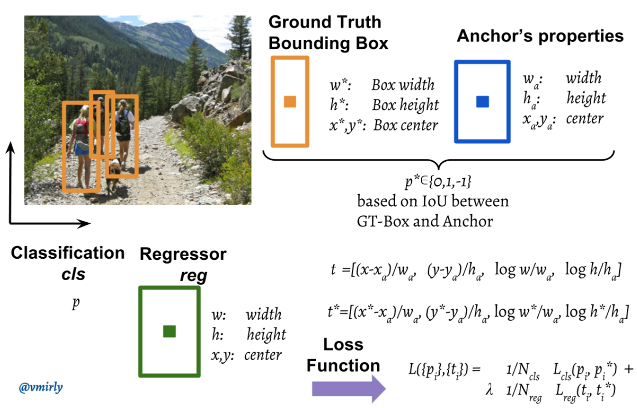

Figure 4: Classification and Regression

Non Maximum Supression(NMS): It is the process in which we remove/merge extremely highly overlapping bounding boxes. The general idea of non-maximum suppression is to reduce the number of detections in a frame to the actual number of objects present. If the object in the frame is fairly large and more than 2000 object proposals have been generated, it is quite likely that some of these will have significant overlap with each other and the object. The pseudo code to implement NMS is given below:

```
- Take all the roi boxes [roi_array]
- Find the areas of all the boxes [roi_area]
- Take the indexes of order the probability score in descending order [order_array]
keep = []
while order_array.size > 0:
  - take the first element in order_array and append that to keep
  - Find the area with all other boxes
  - Find the index of all the boxes which have high overlap with this box
  - Remove them from order array
  - Iterate this till we get the order_size to zero (while loop)
- Ouput the keep variable which tells what indexes to consider.
```

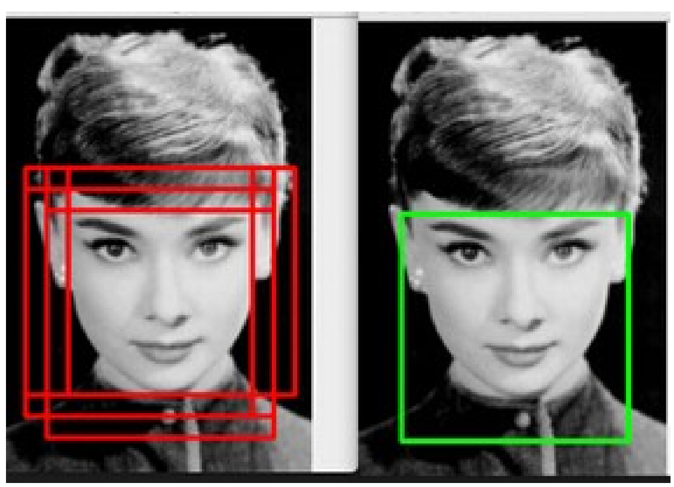

Figure 5: An example of NMS

RoI Pooling: Region of interest pooling (also known as RoI pooling) purpose is to perform max pooling on inputs of non-uniform sizes to obtain fixed-size feature maps. RoI Pooling is done in three steps:

1. Dividing the region proposal into equal-sized sections (the number of which is the same as the dimension of the output)

2. Finding the largest value in each section

3. Copying these max values to the output buffer

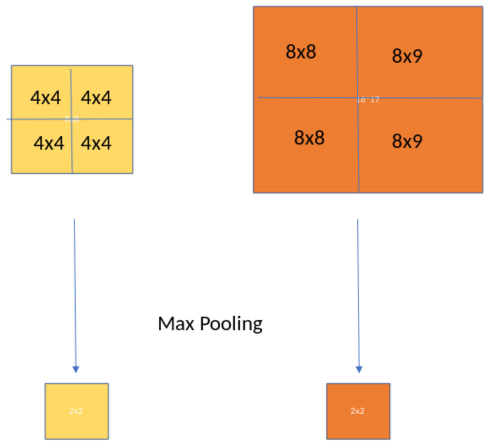

Figure 6: An example of RoI pooling

Detection Network: Using the region proposals generated by the RPN network, Fast R-CNN detection network is used to classify and regresses the bounding boxes. Here, ROI pooling is performed first and then the pooled area goes through CNN and two FC branches for class softmax and bounding box regressor[11].

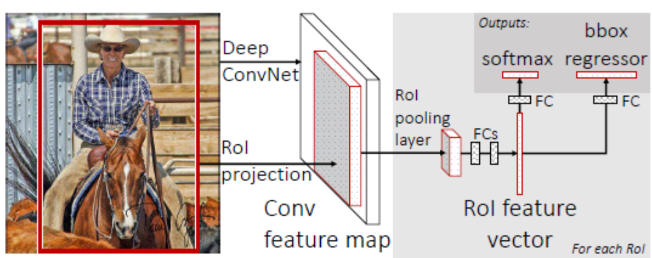

Figure 7: Fast R-CNN detection Network

2.4 Training the FRCNN

The steps taken to develop FRCNN is as follows[8]:

1. Pre-train a CNN network on image classification tasks as described in section 2.2

2. Fine-tune the RPN (Region Proposal Network) end-to-end for the region proposal task, which is initialized by the pre-train image classifier. Positive samples have IoU (Intersection-over-Union) > 0.7, while negative samples have IoU < 0.3.

   (a) Slide a small n x n spatial window over the convolution feature map of the entire image.

   (b) At the center of each sliding window, we predict multiple regions of various scales and ratios simultaneously. An anchor is a combination of (sliding window center, scale, ratio).

3. Train a Fast R-CNN object detection model using the proposals generated by the current RPN

4. Then use the Fast R-CNN network to initialize RPN training. While keeping the shared convolutional layers, only fine-tune the RPN-specific layers. At this stage, RPN and the detection network have shared convolutional layers.

5. Finally fine-tune the unique layers of Fast R-CNN

6. Step 4-5 can be repeated to train RPN and Fast R-CNN alternatively if needed.

Loss functions for the FRCNN is calculated as follows:

$$L_{box}(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i,p_i^*) + \frac{\lambda}{N_{box}}\sum_i p_i^* \cdot L_i^{smoott}(t_i - t_i^*) \tag{2}$$

$$L_{cls}(p_i,p_i^*) = -p_i^* log p_i - (1-p_i^*) log(1-p_i) \tag{3}$$

$$L = L_{cls} + L_{box} \tag{4}$$

where

$$\begin{aligned}
p_i &= \text{Predicted probability of anchor i being an object}\\
p_i* &= \text{Ground truth label (binary) of whether anchor i is an object}\\
t_i &= \text{Predicted four parameterized coordinates}\\
t_i* &= \text{Ground truth coordinates}\\
N_{cls} &= \text{Normalization term, set to be mini-batch size (256) in the paper}\\
N_{box} &= \text{Normalization term, set to the number of anchor locations (2400) in the paper}\\
\lambda &= \text{A balancing parameter, set to be 10 in the paper}
\end{aligned}$$

Please refer the appendix for the python implementation of the above described model.

3 Expected Results

Upon implementation and training of the model described in Section 2.3 the following results are expected.

Figure 8: Test image from slapping category

Figure 9: Result with bounding boxes for predator and victim

The test image shown in Figure 8 will be classified as cyberbully action : slapping with predator identified with red bounding box and the victim identified with green bounding box as shown in Figure 9.

Figure 10: Test image from slapping category

Figure 11: Result with bounding boxes for predator and victim

The test image shown in Figure 10 will be classified as cyberbully action : punching with predator identified with red bounding box and the victim identified with green bounding box as shown in Figure 11.

4 Conclusion

The goal of this project was to develop a deep neural network that takes an image as input and categorizes it into one of the 10 below mentioned categories of bullying. Further it was desired that the identification network would identify the predator and victim in the images classified as cyberbullyig.
1) Gossiping 2) Isolation 3) Laughing 4) Pulling hair 5) Punching 6) Quarrel 7) Strangle 8) Slapping 9) Stabbing 10) Non bullying
The neural network was developed using PyTorch. The training dataset for the classification network was made up of 2494 images belonging to the 10 above mentioned categories.

In order to avoid overfitting in the CNN model for classification network, batch normalization and image augmentation strategies were implemented.

Unfortunately, due to time constraints, we were unable to complete the implementation of the researched FRCNN model. Please refer to the appendix for the partial implementation code of FRCNN model. We believe that the above researched model would provide satisfactory result for the given project problem statement as shown in [9].

5 References

[1] https://www.stopbullying.gov/cyberbullying/what-is-it/index.html

[2] https://www.forbes.com/sites/bernardmarr/2018/10/01/what-is-deep-learning-ai-a-simple-guide-with-8-practical-examples/#434cffaa8d4b

[3] https://en.wikipedia.org/wiki/Convolutional_neural_network#Convolutional

[4] https://cs231n.github.io/convolutional-networks/

[5] https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

[6] https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-al

[7] https://medium.com/@whatdhack/a-deeper-look-at-how-faster-rcnn-works-84081284e1cd

[8] https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3 html#fast-r-cnn

[9] https://arxiv.org/abs/1506.01497

[10] https://www.quora.com/How-does-the-region-proposal-network-RPN-in-Faster-R-CNN-work

[11] https://towardsdatascience.com/review-faster-r-cnn-object-detection-f5685cb30202

6 Appendix

6.1 Bully Detection CNN Model

```
# Importing libraries
import torch
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import sys # For command Line arguments

#Defining the CNN
class CNNModel(nn.Module):
    """ A CNN Model for image classification """

    def __init__(self,image_size, op_size):
        """ CNN layer to process the image"""
        super(CNNModel, self).__init__() # Super is used to refer to the base c

        # Convolution Layer 1
        self.cnn1_1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,

        # Xavier Initialization
        nn.init.xavier_uniform_(self.cnn1_1.weight)

        self.cnn1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,

        # Batch Normalization
        self.cnnBN1 = nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,

        # Max Pooling 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        self.dropout = nn.Dropout(p=0.8)

        # Convolution Layer 2
        self.cnn2_1 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3
        self.cnn2_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=
        self.cnnBN2 = nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,

        # Max Pooling 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

```

```
        self.dropout = nn.Dropout(p=0.8)

        # Convolution Layer 3
        self.cnn3_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=
        self.cnn3_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=
        self.cnn3_3 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=
        self.cnnBN3 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,

        # Max Pooling 3
        self.maxpool3 = nn.MaxPool2d(kernel_size=2)

        # Dropout Regularization
        self.dropout = nn.Dropout(p=0.8)

        # Convolution Layer 4
        self.cnn4_1 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=
        self.cnn4_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
        self.cnn4_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
        self.cnnBN4 = nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,

        # Max Pooling 4
        self.maxpool4 = nn.MaxPool2d(kernel_size=2)

        # Dropout Regularization
        self.dropout = nn.Dropout(p=0.5)

        # Convolution Layer 5
        self.cnn5_1 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
        self.cnn5_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
        self.cnn5_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
        self.cnnBN5 = nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,

        # Max Pooling 5
        self.maxpool5 = nn.MaxPool2d(kernel_size=2)

        # Dropout Regularization
        self.dropout = nn.Dropout(p=0.8)
#-------------------------------------------------
        # Fully connected linear layer
        #self.fc1 = nn.Linear(32*75*75 , 9)  #32 channels, 75x75 final image si
        self.fc1 = nn.Linear(512*image_size*image_size, 4096)
#32 channels, 7x7 final image size
        self.relu4 = nn.ReLU()

        self.fc2 = nn.Linear(4096, 1000)  #32 channels, 7x7 final image size
        self.relu5 = nn.ReLU()

```

```
        self.fc3 = nn.Linear(1000, 10)  #32 channels, 7x7 final image size



    #Image size = 28x28 -> 13x13 after first pooling
    #14x14 after padding = 1
    #7x7 after second pooling

    def forward(self, x):
        """ Forward Propogation for classification """

        #CNN layer 1
        out = self.cnn1_1(x)
        out = F.relu(out)
        out = self.cnn1_2(out)
        out = F.relu(out)
        out = self.cnnBN1(out)
        out = self.maxpool1(out)

        #out = self.dropout(out)

        #CNN layer 2
        out = self.cnn2_1(out)
        out = F.relu(out)
        out = self.cnn2_2(out)
        out = F.relu(out)
        out = self.cnnBN2(out)
        out = self.maxpool2(out)

        #out = self.dropout(out)

        #CNN layer 3
        out = self.cnn3_1(out)
        out = F.relu(out)
        out = self.cnn3_2(out)
        out = F.relu(out)
        out = self.cnn3_3(out)
        out = F.relu(out)
        out = self.cnnBN3(out)
        out = self.maxpool3(out)

        #out = self.dropout(out)

        #CNN layer 4
        out = self.cnn4_1(out)
        out = F.relu(out)
        out = self.cnn4_2(out)
```

```
        out = F.relu(out)
        out = self.cnn4_3(out)
        out = F.relu(out)
        out = self.cnnBN4(out)
        out = self.maxpool4(out)

        #out = self.dropout(out)

        #CNN layer 5
        out = self.cnn5_1(out)
        out = F.relu(out)
        out = self.cnn5_2(out)
        out = F.relu(out)
        out = self.cnn5_3(out)
        out = F.relu(out)
        out = self.cnnBN5(out)
        out = self.maxpool5(out)

        out = self.dropout(out)

        # Resize the tensor, -1 decides the best dimension automatically
        #out = out.view(out.size(0), -1)
        out = out.view(out.size(0), -1)

        # Dropout
        #out = self.dropout(out)

        # Fully connected 1
        out = self.fc1(out)
        out = F.relu(out)

        out = self.fc2(out)
        out = F.relu(out)

        out = self.fc3(out)

        out = F.log_softmax(out, dim=0)  #Softmax along Row
        # Return
        return out
```

6.2 Training Code

```
# Importing libraries
import torch
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets
```

```python
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import sys # For command Line arguments
import os
from shutil import copyfile
from detection_Model import CNNModel

# Hyperparameter initialization
n_epoch         = 100
n_class         = 10
batch_size      = 10
learning_rate   = 0.0001

# check if GPU is available
print(torch.cuda.current_device())
print(torch.cuda.device(0))
print(torch.cuda.device_count())
print(torch.cuda.get_device_name(0))

#To run on GPU
device = torch.device("cuda:0")
dtype = torch.float
# Sorting out the data

# Image parameters
n_cnn = 5 #Number of CNN layer
img_size = (256,256)
conv_size = int( img_size[0]/ (2**(n_cnn)) ) # image_size / 8 for 3 cnn layer.
train_img = "../TrainingData"
Model = "./Model"

# Define the transformation
transform = transforms.Compose( [transforms.Resize(img_size),
                                 transforms.RandomRotation((90, 360)),
                                 transforms.RandomVerticalFlip(),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ColorJitter(),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5
                                 ])

# Training dataset
train_dataset = datasets.ImageFolder(root=train_img, transform=transform)

# Placing data into dataloader for better accessibility
# Shuffle training dataset to eleminate bias
```

```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=ba

# Instance creation
#model = CNNModel(conv_size, n_class).cuda()
model = nn.DataParallel(CNNModel(conv_size, n_class))
model = model.to(device)
# Create instance of loss
criterion = nn.CrossEntropyLoss()

# Create instance of optimizer (Adam)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Model Training

n_iteration = 0

for epoch in range(n_epoch):
    total = 0
    correct = 0
    for i, (images, labels) in enumerate(train_loader):
        # Wrap into Variable
        #images = Variable(images).cuda()
        #labels = Variable(labels).cuda()
        images = Variable(images).to(device)
        labels = Variable(labels).to(device)

        # Clear the gradients
        optimizer.zero_grad()

        # Forward propogation
        #outputs = model(images).cuda()
        outputs = model(images)

        # Loss calculation ( softmax )
        loss = criterion(outputs, labels)

        # Backpropogation
        loss.backward()

        # Update Gradients
        optimizer.step()

        n_iteration += 1

        # Total number of labels
        total += labels.size(0)

```

```
        # obtain prediction from max value
        _, predicted = torch.max(outputs.data, 1)

        # Calculate the number of right answers
        correct += (predicted == labels).sum().item()

        # Prit loss and accuracy
        if (i + 1) % 10 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%

# Saving the trained model
if not os.path.exists(Model):
    os.makedirs(Model)
torch.save(model.state_dict(), "./Model/model.pth")
print("Model saved at ./Model/model.pth")
```

6.3 Test Code

```
# Importing libraries
import torch
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import sys # For command Line arguments
import os
from shutil import copyfile
from detection_Model import CNNModel
from PIL import Image

# loading the input test image
if(len(sys.argv)<2):
    sys.exit("Please specify an image to test")
else:
    test_img_filename = sys.argv[1]

img_size = (256,256)
n_cnn = 3
conv_size = int(img_size[0] /(2**(n_cnn + 1)) )
test_img  = "./TestData/test/"
test_img1  = "./TestData"
Model = "./Model"

device = torch.device("cuda:0")

```

```
# Hyperparameter initialization
batch_size       = 1

# Define the transformation
transform = transforms.Compose( [transforms.Resize(img_size),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5
                                 ])


# Testing dataset
test_dataset = Image.open(test_img_filename)
test_loader = transform(test_dataset)


# Image parameters
n_class          = 10

model = nn.DataParallel(CNNModel(conv_size, n_class))
model = model.to(device)
model.load_state_dict(torch.load('./Model/model.pth'))
model.eval().to(device)

def ten_to_str(x):
    """ Function to convert tensor label to a string """
    str_label = ["gossiping", "isolation", "laughing", "nonbullying", "pullingh
    return str_label[x]

# Testing the model
with torch.no_grad():
    images = Variable(test_loader, requires_grad=True)
    images = images.unsqueeze(0)
    images = images.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)
    predicted = predicted.item()
    print("{}".format(ten_to_str(predicted)))
```

6.4 FRCNN

```
# Imports
import torch
import torchvision
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torchvision import transforms
```

```
from PIL import Image

# Debug flag, set to 1 to get debug messages
__DEBUG__ = 0

image_size = (256, 256)
# Sample Black background image
image = torch.zeros((1, 3, image_size[0], image_size[1])).float()

if __DEBUG__:
    print("size of image tensor: %s " %(image.size()))

# Generate Sample Bounding Box
bbox = torch.FloatTensor([[20, 30, 200, 150],[150, 200, 220, 250]]) #[Ymin, Xmi
labels = torch.LongTensor([6, 8])
sub_sample = 16

# Squeeze to remove the first dimention of tensor (convert from 4d to 3d I thin
pil_image = transforms.ToPILImage()(image.squeeze())

#plt.imshow(pil_image)
if __DEBUG__:
    pil_image.show()

# Create a dummy image
dummy_img = torch.zeros((1, 3, image_size[0], image_size[1])).float()

#print(dummy_img)

# Load vgg16
model = torchvision.models.vgg16(pretrained=True)

# List out all the features
fetrs = list(model.features)

# Pass dummy image through layers to check for layer whose output matches the r
req_fetrs = []
# Clone dummy image and pass it through all layers and check for layer output s
clone_img = dummy_img.clone()
for lyr in fetrs:
    clone_img = lyr(clone_img)
    if clone_img.size()[2] < 256//16:
        break
    req_fetrs.append(lyr)
    out_channels = clone_img.size()[1]

if __DEBUG__:
```

```
    print("Length of required features: ", len(req_fetrs))
    print("Number of Out channels: ", out_channels)

# Convert required features into sequential module
frcnn_fe = nn.Sequential(*req_fetrs)

# Using frcnn as backend compute features for dummy image
out_map = frcnn_fe(image)

if __DEBUG__:
    print("Out map size is: ", out_map)

# Creating Anchors
# ----------------------
# Define 3 ratio and scales that we will be using
ratio = [0.5, 1, 2]
anchor_scales = [8, 16, 32]

# Number of Ratios and anchor scales
n_ratio = len(ratio)
n_scales = len(anchor_scales)

# Base for the anchor
anchor_base = np.zeros((n_ratio * n_scales, 4), dtype=np.float32)

if __DEBUG__:
    print("anchor base is: \n", anchor_base)

# Define center for base anchor
center_y = sub_sample / 2.
center_x = sub_sample / 2.

if __DEBUG__:
    print("Center for base anchor is: (%s, %s) " %(center_x, center_y))

# Generating Anchos for first feature map pixel
# Iterate through all ratios and scales
for i in range(n_ratio):
    for j in range(n_scales):
        h = sub_sample * anchor_scales[j] * np.sqrt(ratio[i])
        w = sub_sample * anchor_scales[j] * np.sqrt(1. / ratio[i])

        index = i * n_scales + j

        anchor_base[index, 0] = center_y - h / 2. #y_min
        anchor_base[index, 1] = center_x - w / 2. #x_min
        anchor_base[index, 2] = center_y + h / 2. #y_max
```

```
        anchor_base[index, 3] = center_x + w / 2. #x_max

if __DEBUG__:
    print("anchor bases: \n", anchor_base)
    print("Negative anchors represent the ones that are out of the image bounda

# Generationg anchors for all feature map pixels
feature_size = (image_size[0] // sub_sample)
# 16 sub_samples in feature map where each has dimension 16*16
center_x = np.arange(sub_sample, (feature_size +  1) * 16, 16)
center_y = np.arange(sub_sample, (feature_size +  1) * 16, 16)

# Generation Centers
center = np.zeros((len(center_x) * len(center_y), 2))
index = 0
for x in range(len(center_x)):
    for y in range(len(center_y)):
        center[index, 1] = center_x[x] - int(sub_sample / 2)
        center[index, 0] = center_y[y] - int(sub_sample / 2)
        index += 1

# Generating anchors for above generated centers
num_anchors_per_pixel = n_ratio * n_scales
anchors = np.zeros(((feature_size * feature_size * num_anchors_per_pixel), 4))

index = 0
for c in center:
    center_y, center_x = c
    for i in range(n_ratio):
        for j in range(n_scales):
            h = sub_sample * anchor_scales[j] * np.sqrt(ratio[i])
            w = sub_sample * anchor_scales[j] * np.sqrt(1. / ratio[i])

            anchors[index, 0] = center_y - h / 2.
            anchors[index, 1] = center_x - w / 2.
            anchors[index, 2] = center_y + h / 2.
            anchors[index, 3] = center_x + w / 2.
            index += 1

if __DEBUG__:
    print("Total anchors size is: ", anchors.shape)

# Labeling the anchors
#[Ymin, Xmin, Ymax, Xmax] format
bbox = np.asarray([[20, 30, 200, 150],[150, 200, 220, 250]], dtype=np.float32)
labels = np.asarray([6, 8])

```

```
# Find the index of the anchors that are inside the image boundary
index_inside = np.where(
                (anchors[:, 0] >= 0) &
                (anchors[:, 1] >= 0) &
                (anchors[:, 2] <= image_size[1]) &
                (anchors[:, 3] <= image_size[0])
                )[0]

if __DEBUG__:
    print("anchors that are insdie the image are: \n", index_inside)
    print("\nNumber of anchors inside the image boundary: ", index_inside.shape

# Make a label array and fill it with -1
label = np.empty((len(index_inside), ), dtype=np.int32)
label.fill(-1)

if __DEBUG__:
    print("Created Label size is %s and index inside size is %s" %(label.size,

# Array with valid anchor boxes
anchor_valid = anchors[index_inside]

if __DEBUG__:
    print("Valid anchor box shape is: ", anchor_valid.shape)

# Calculate IoU for valid anchor boxes
ious = np.empty((len(anchor_valid), 2), dtype=np.float32)
if __DEBUG__:
    print("Bounding Boxes are : \n", bbox)

for num1, i in enumerate(anchor_valid):
    # ymin, xmin, ymax, xmax format for anchors
    ya1, xa1, ya2, xa2 = i
    # anchor area = height * width
    area_anchor = (ya2 - ya1) * (xa2 - xa1)
    for num2, j in enumerate(bbox):
        yb1, xb1, yb2, xb2 = j
        area_box = (yb2 - yb1) * (xb2 - xb1)

        intersection_x1 = max([xb1, xa1])
        intersection_y1 = max([yb1, ya1])
        intersection_x2 = min([xb2, xa2])
        intersection_y2 = min([yb2, ya2])

        # Check for intersection
        if (intersection_x1 < intersection_x2) and (intersection_y1 < intersect
            area_intersection = (intersection_y2 - intersection_y1) * (intersec
```

```
            #intersection over union
            iou = area_intersection / (area_anchor + area_box - area_intersecti
        else:
            # In case of No overlap/ intersection
            iou = 0.

        ious[num1, num2] = iou

if __DEBUG__:
    print("all the iou count: ", ious.shape)

# Case-1
# Highest IoU for each gt and corrosponding anchor
# Location of max Iou
gt_argmax_ious = ious.argmax(axis=0)
if __DEBUG__:
    print("Indices of MAX IoU: ", gt_argmax_ious)

# Value of Max IoU
gt_max_ious = ious[gt_argmax_ious, np.arange(ious.shape[1])]
if __DEBUG__:
    print("Values of MAX IoU: ", gt_max_ious)

# Case-2
# Highest Iou In between every anchor
argmax_ious = ious.argmax(axis=1)
if __DEBUG__:
    print("shape of argmax_ious: ", argmax_ious.shape)
    print("MAX Iou indices for every anchor: ", argmax_ious)

max_ious = ious[np.arange(len(index_inside)), argmax_ious]
if __DEBUG__:
    print("MAX IoU values: ", max_ious)

# Anchor Box that has the HIGHEST IoU with GT
gt_argmax_ious = np.where(ious == gt_max_ious)[0]
if __DEBUG__:
    print("Ultimate MAX IoU: ", gt_argmax_ious)

# Assigning Labels which helps to compute Loss
# Defining thresholds
pos_thres = 0.7
neg_thres = 0.3

# Assign negative label (0) to all anchors that have IoU < 0.3
label[max_ious < neg_thres] = 0

```

```
# Assign positive label (1) to anchor boxes with  highest IoU with GT
label[gt_argmax_ious] = 1

# Assign positive label (1) to anchor boxes with IoU > 0.7
label[max_ious >= pos_thres] = 1


# =====================================================================================
#                               TRAINING RPN
# =====================================================================================

# Define positive and negative anchor sample parameters
num_samples = 128
pos_ratio = 0.5
num_pos = pos_ratio * num_samples

# Picking Positive Samples
pos_index = np.where(label == 1)[0]

if len(pos_index) > num_pos:
    disable_index = np.random.choice(pos_index, size=(len(pos_index) - num_pos)
    label[disable_index] = -1

# Picking Negative Samples
num_neg = num_samples * np.sum(label == 1)
neg_index = np.where(label == 0)[0]

if len(neg_index) > num_neg:
    disable_index = np.random.choice(neg_index, size=(len(neg_index) - num_neg)
    label[disable_index] = -1

# GT with MAX IoU for each anchor
max_iou_bbox = bbox[argmax_ious]

# Convert [ymin, xmin, ymax, xmax] format to [center_y, center_x, h, w] format

height = anchor_valid[:, 2] - anchor_valid[:, 0]
width  = anchor_valid[:, 3] - anchor_valid[:, 1]
center_y  = anchor_valid[:, 0] + 0.5 * height
center_x  = anchor_valid[:, 1] + 0.5 * width

base_height = max_iou_bbox[:, 2] - max_iou_bbox[:, 0]
base_width = max_iou_bbox[:, 3] - max_iou_bbox[:, 1]
base_center_y = max_iou_bbox[:, 0] + 0.5 * base_height
base_center_x = max_iou_bbox[:, 1] + 0.5 * base_width

# Find the locations
```

```
eps = np.finfo(height.dtype).eps
height = np.maximum(height, eps)
width = np.maximum(width, eps)

dy = (base_center_y - center_y) / height
dx = (base_center_x - center_x) / width
dh = np.log(base_height / height)
dw = np.log(base_width / width)

anchor_locs = np.vstack((dy, dx, dh, dw)).transpose()
if __DEBUG__:
    print("Shape of anchor locations", anchor_locs.shape)
    print("Anchor locations", anchor_locs)

# Final Labels
anchor_labels = np.empty((len(anchors),), dtype=label.dtype)
anchor_labels.fill(-1)
anchor_labels[index_inside] = label

# Final Locations
anchor_locations = np.empty((len(anchors),) + anchors.shape[1:], dtype=anchor_l
anchor_locations.fill(0)
anchor_locations[index_inside, :] = anchor_locs

# ==============================================================================
#       Region Proposal Network
# ==============================================================================

mid_channels = 512
in_channels = 512
n_anchor = 9

conv1 = nn.Conv2d(in_channels, mid_channels, 3, 1, 1)

# Bounding Box Regressor network
reg_layer = nn.Conv2d(mid_channels, n_anchor * 4, 1, 1, 0)

# Classifier network
cls_layer = nn.Conv2d(mid_channels, n_anchor * 2, 1, 1, 0)

# Initialization
# convolution sliding layer
conv1.weight.data.normal_(0, 0.01)
conv1.bias.data.zero_()

# Regression layer
reg_layer.weight.data.normal_(0, 0.01)
```

```
reg_layer.bias.data.zero_()

# classification layer
cls_layer.weight.data.normal_(0, 0.01)
cls_layer.bias.data.zero_()

# Training
# ---------------

x = conv1(out_map)
pred_anchor_locs = reg_layer(x)
pred_cls_scores = cls_layer(x)

if __DEBUG__:
    print("predicted class score shape: ", pred_cls_scores.shape)
    print("predicted anchor location shape: ", pred_anchor_locs.shape)

# Rearrange the tensors to align with anchor targets
pred_anchor_locs = pred_anchor_locs.permute(0, 2, 3, 1).contiguous().view(1, -1
if __DEBUG__:
    print("Rearranged anchor location shape: ", pred_anchor_locs.shape)

pred_cls_scores = pred_cls_scores.permute(0, 2, 3, 1).contiguous()
if __DEBUG__:
    print("Rearranged class score shape: ", pred_cls_scores.shape)

# Calculate the Objectness score
objectness_score = pred_cls_scores.view(1, 16, 16, 9, 2)[:, :, :, :, 1].contigu
if __DEBUG__:
    print("Objectness Score shape: ", objectness_score.shape)

pred_cls_scores  = pred_cls_scores.view(1, -1, 2)
if __DEBUG__:
    print("predicted class score shape final: ", pred_cls_scores.shape)

# ========================================================
# Generationg Proposals
# ========================================================

# Define parameters for training and testing

nms_thresh = 0.7 # Non- Maximum Supression Threshold
n_train_pre_nms = 12000 # number of bboxes before nms during training
n_train_post_nms = 2000 # number of bboxes after nms during training
n_test_pre_nms = 6000 # number of bboxes before nms during testing
n_test_post_nms = 300 # number of bboxes after nms during testing
min_size = 16 # minimum height of the object required to create a proposal
```

```

# Convert anchors from [ymin, xmin, ymax, xmax] to [center_Y, center_x, h, w] f
anc_height = anchors[:, 2] - anchors[:, 0]
anc_width = anchors[:, 3] - anchors[:, 1]
anc_ctr_y = anchors[:, 0] + 0.5 * anc_height
anc_ctr_x = anchors[:, 1] + 0.5 * anc_width

# Convert prediction locations and objectness score to numpy array
pred_anchor_locs_numpy = pred_anchor_locs[0].data.numpy()
objectness_score_numpy = objectness_score[0].data.numpy()

dy = pred_anchor_locs_numpy[:, 0::4]
dx = pred_anchor_locs_numpy[:, 1::4]
dh = pred_anchor_locs_numpy[:, 2::4]
dw = pred_anchor_locs_numpy[:, 3::4]

ctr_y = dy * anc_height[:, np.newaxis] + anc_ctr_y[:, np.newaxis]
ctr_x = dx * anc_width[:, np.newaxis] + anc_ctr_x[:, np.newaxis]
h = np.exp(dh) * anc_height[:, np.newaxis]
w = np.exp(dw) * anc_width[:, np.newaxis]

# Region of Interest
roi = np.zeros(pred_anchor_locs_numpy.shape, dtype=pred_anchor_locs_numpy.dtype
roi[:, 0::4] = ctr_y - 0.5 * h
roi[:, 1::4] = ctr_x - 0.5 * w
roi[:, 2::4] = ctr_y + 0.5 * h
roi[:, 3::4] = ctr_x + 0.5 * w

# Clip the predicted boxes to the image size
roi[:, slice(0, 4, 2)] = np.clip(roi[:, slice(0, 4, 2)], 0, image_size[0])
roi[:, slice(1, 4, 2)] = np.clip(roi[:, slice(1, 4, 2)], 0, image_size[1])

if __DEBUG__:
    print("Region of interest: ", roi)

# Remove predicted boxes with either height of width < threshold
hs = roi[:, 2] - roi[:, 0]
ws = roi[:, 3] - roi[:, 1]
keep = np.where((hs >= min_size) & (ws >= min_size))[0]
roi = roi[keep, :]
score = objectness_score_numpy[keep]

# Sort (proposal, score) in descending order
order = score.ravel().argsort()[::-1]

# Take top values?
order = order[:n_train_pre_nms]
```

```
roi = roi[order, :]

# Calculate Region Proposals
y1 = roi[:, 0]
x1 = roi[:, 1]
y2 = roi[:, 2]
x2 = roi[:, 3]

area = (x2 - x1 + 1) * (y2 - y1 + 1)
order = score.argsort()[::-1]

keep = []

while order.size > 0:
    i = order[0]
    keep.append(i)
    xx1 = np.maximum(x1[i], x1[order[1:]])
    yy1 = np.maximum(y1[i], y1[order[1:]])
    xx2 = np.minimum(x2[i], x2[order[1:]])
    yy2 = np.minimum(y2[i], y2[order[1:]])

    w = np.maximum(0.0, xx2 - xx1 + 1)
    h = np.maximum(0.0, yy2 - yy1 + 1)
    inter = w * h

    ovr = inter / (area[i] + area[order[1:]] - inter)

    inds = np.where(ovr <= nms_thresh)[0]
    order = order[inds + 1]

keep = keep[:n_train_post_nms]
roi = roi[keep]

if __DEBUG__:
    print("Final region proposal count: ", roi.shape)

# Proposal Targets
n_sample = 128
pos_ratio = 0.25
pos_iou_thresh = 0.5
neg_iou_thresh_hi = 0.5
neg_iou_thresh_lo = 0.0

# Calculate IoU
ious = np.empty((len(roi), 2), dtype=np.float32)
ious.fill(0)
for num1, i in enumerate(roi):
```

```
    ya1, xa1, ya2, xa2 = i
    anchor_area = (ya2 - ya1) * (xa2 - xa1)
    for num2, j in enumerate(bbox):
        yb1, xb1, yb2, xb2 = j
        box_area = (yb2 - yb1) * (xb2 - xb1)

        inter_x1 = max([xb1, xa1])
        inter_y1 = max([yb1, ya1])
        inter_x2 = min([xb2, xa2])
        inter_y2 = min([yb2, ya2])

        if(inter_x1 < inter_x2) and (inter_y1 < inter_y2):
            inter_area = (inter_y2 - inter_y1) * (inter_x2 - inter_x1)
            iou = inter_area / (anchor_area + box_area - inter_area)
        else:
            iou = 0.

        ious[num1, num2] = iou

if __DEBUG__:
    print("Proposal Targets: ", ious.shape)

# GT with max IoU for each region
gt_assignment = ious.argmax(axis=1)
max_iou = ious.max(axis=1)

if __DEBUG__:
    print("GT location with max IoU for each region and max IoU's")
    print(gt_assignment)
    print(max_iou)

# Assign label to each proposal
gt_roi_label = labels[gt_assignment]

if __DEBUG__:
    print("GT labels: ", gt_roi_label)
```