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 Faster-RCNN model in order to detect and classify various cyberbully actions in an image.

1 Introduction

This report considers the problem of detection and classification of cyberbully actions captured in an 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 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]

The implemented CNN model classifies the given input image into one of the nine categories of cyberbully actions. In order to further detect the predator and victim in the classified cyberbully image, an object detection model must be used. This can be achieved 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 that way interference is minimized because it is expected that only a single object of interest will dominate in a given region. The 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. In order to overcome this drawback Faster RCNN(F-RCNN) is proposed.

F-RCNN approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, the 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 model based on CNN using an open source machine learning library PyTorch is described in the section below.

2.1 Implementation of CNN Model

Our CNN model for classification is implemented using 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

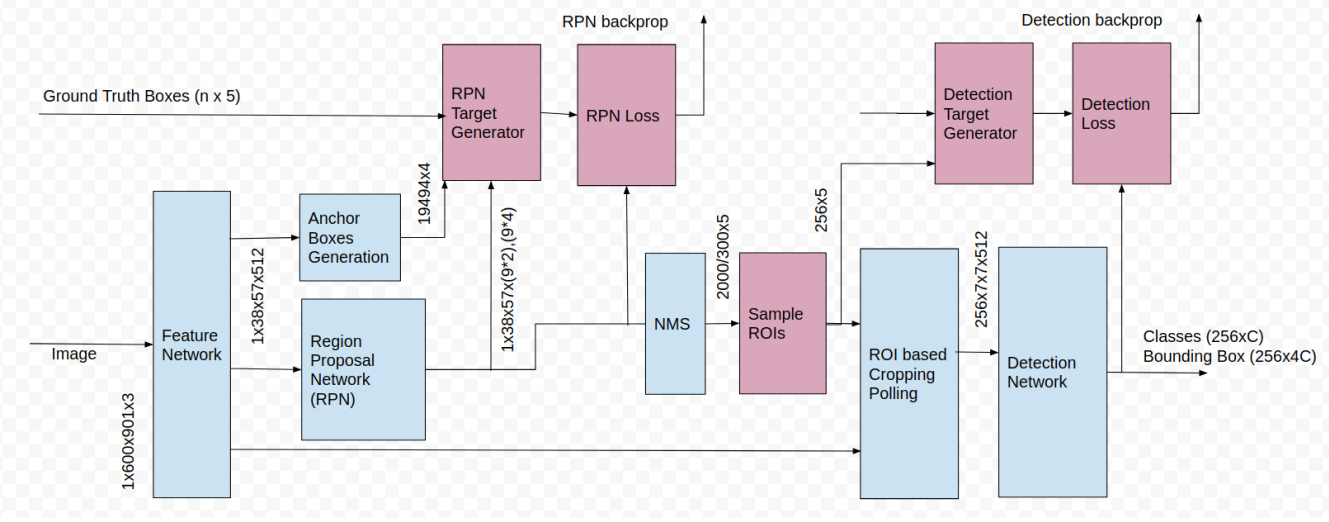

Figure 1: Faster-RCNN block diagram. The magenta colored blocks are active only during training. The numbers indicate size of the tensors[7].

The steps taken to develop FRCNN is as follows:

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

p_i = Predicted probability of anchor i being an object

p_i* = Ground truth label (binary) of whether anchor i is an object

t_i = Predicted four parameterized coordinates

t_i* = Ground truth coordinates

N_{cls} = Normalization term, set to be mini-batch size (256) in the paper

N_{box} = Normalization term, set to the number of anchor locations (2400) in the paper

λ = A balancing parameter, set to be 10 in the paper

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

3 Results

The network described in section 2.1 was trained using 2494 images split into 10 categories. For a batch size of 1, 25 epochs and a learning rate of 0.0001, the following results were observed.

The training accuracy(single execution) was found to be 98.08% .

The network was tested using the following images.

The image for the first test is pulled out from the training dataset. Figure 2 is an image of two women involved in a heated argument and thus the image belongs to the 'quarrel' category. The purpose of this test is to check if the network can recognize an image from the training dataset.

Figure 2: Test image from the 'quarrel' category

The network classified Figure 2 as 'quarrel' category.

The images used for the further tests are images that the network has never seen during the training phase. The purpose of these tests is to check if the network has learned any features based upon which it can categorize images into its respective categories.

A Human being would classify Figure 3 as two girls in the foreground gossiping about the third girl in the background. Thus this image falls under the 'gossiping' category.

Figure 3: Test image of two girls gossiping in the foreground

The network classified Figure 3 to be in the ‘strangle’ category.

Upon inspection, Figure 4 appears to comprise of two girls in the background gossiping about the girl in the foreground. Again this is an instance of ‘gossiping’

Figure 4: Test image of two girls gossiping in the background

The network classified Figure 4 to be in the ‘isolation’ category.

Figure 5 is quiet straightforward. This is a picture of a girl being excluded from a group of friends. Thus this image falls under the ‘isolation’ category.

Figure 5: Test image of isolation

The network classified Figure 5 as ‘punching’ category.

4 Conclusion

The goal of this project is 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.
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 network was made up of 2494 images belonging to the 10 above mentioned categories. The training accuracy for a single run was observed to be 98%. Four testing examples are mentioned in this report.

When the network was presented with an image from the training dataset, it had no problem in predicting the category of the image. The image for this test was taken from the 'quarrel' category(Figure 2) and the network predicted it correctly. This confirms that the network actually recognizes the training images and it is not just randomly predicting the image category.

For the next three tests, the network was given images that it had never seen before. Here the network is expected to recognize a set of features(in the test image) that it has learnt through the training phase to predict the category of bullying.

When the network was tested with Figure 3, which is an instance of 'gossiping' the network wrongly classified it as 'strangle'. Since the subject in Figure 3 seem to be very close to each other, the network confuses it with 'strangling'. The reason for this wrong prediction could be that the network has not learnt enough features to clearly distinguish between strangling and gossiping. A deeper network with more complex structure could help the network extract more features.

Figure 4 is a complicated image compared to the other test images. This image could either be gossiping or isolation. The girl in the foreground appears to be isolated from the group in the background who are gossiping. The network classified this image to be of the 'isolation' category. This is probably because the network weighs the foreground subject more than the background. Having a bounding box label would probably help the network prioritize the subject more accurately.

Figure 5 clearly falls under the 'isolation' category. But the network classified it as punching.

Over-fitting could be the cause for this result. Adding a better dropout layer might refrain the network from over-fitting.

In conclusion the model defined in section 2.1 is shallow. Such a basic CNN struggles to deal with images that does not directly fall under a single category of bullying. Having an object detection model such as the YOLO or the RCNN would help in better image classification. Having a pre-trained model like the Resnet or the VGG-16 would improve the test accuracy significantly. The type of image labeling used also plays a significant role. Having boundary box labeling might aid the network in more accurate subject prioritization.

5 Future Work

Based on the Results, the following changes are planned to improve the network accuracy.

1. Implement YOLO and RCNN for object detection
2. Implement a deeper network structure for more feature extraction
3. Label the images using Boundary box labeling
4. Increase the training image dataset

6 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

7 Appendix

7.1 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)

```

```
43          self.dropout = nn.Dropout(p=0.8)
44
45          # Convolution Layer 3
46          self.cnn3_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=
47          self.cnn3_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=
48          self.cnn3_3 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=
49          self.cnnBN3 = nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
50
51          # Max Pooling 3
52          self.maxpool3 = nn.MaxPool2d(kernel_size=2)
53
54          # Dropout Regularization
55          self.dropout = nn.Dropout(p=0.8)
56
57          # Convolution Layer 4
58          self.cnn4_1 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=
59          self.cnn4_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
60          self.cnn4_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
61          self.cnnBN4 = nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
62
63          # Max Pooling 4
64          self.maxpool4 = nn.MaxPool2d(kernel_size=2)
65
66          # Dropout Regularization
67          self.dropout = nn.Dropout(p=0.5)
68
69          # Convolution Layer 5
70          self.cnn5_1 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
71          self.cnn5_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
72          self.cnn5_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=
73          self.cnnBN5 = nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
74
75          # Max Pooling 5
76          self.maxpool5 = nn.MaxPool2d(kernel_size=2)
77
78          # Dropout Regularization
79          self.dropout = nn.Dropout(p=0.8)
80  #-----------------------------------------------
81          # Fully connected linear layer
82          #self.fc1 = nn.Linear(32*75*75 , 9)  #32 channels, 75x75 final image si
83          self.fc1 = nn.Linear(512*image_size*image_size, 4096)
    #32 channels, 7x7 final image size
84          self.relu4 = nn.ReLU()
85
86          self.fc2 = nn.Linear(4096, 1000)  #32 channels, 7x7 final image size
87          self.relu5 = nn.ReLU()
88
```

```
89          self.fc3 = nn.Linear(1000, 10)  #32 channels, 7x7 final image size
90
91
92
93      #Image size = 28x28 -> 13x13 after first pooling
94      #14x14 after padding = 1
95      #7x7 after second pooling
96
97      def forward(self, x):
98          """ Forward Propogation for classification """
99
100         #CNN layer 1
101         out = self.cnn1_1(x)
102         out = F.relu(out)
103         out = self.cnn1_2(out)
104         out = F.relu(out)
105         out = self.cnnBN1(out)
106         out = self.maxpool1(out)
107
108         #out = self.dropout(out)
109
110         #CNN layer 2
111         out = self.cnn2_1(out)
112         out = F.relu(out)
113         out = self.cnn2_2(out)
114         out = F.relu(out)
115         out = self.cnnBN2(out)
116         out = self.maxpool2(out)
117
118         #out = self.dropout(out)
119
120         #CNN layer 3
121         out = self.cnn3_1(out)
122         out = F.relu(out)
123         out = self.cnn3_2(out)
124         out = F.relu(out)
125         out = self.cnn3_3(out)
126         out = F.relu(out)
127         out = self.cnnBN3(out)
128         out = self.maxpool3(out)
129
130         #out = self.dropout(out)
131
132         #CNN layer 4
133         out = self.cnn4_1(out)
134         out = F.relu(out)
135         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
```

7.2 Training Code

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

```
6  import torch.nn as nn
7  import torch.nn.functional as F
8  from torch.autograd import Variable
9  import sys # For command Line arguments
10 import os
11 from shutil import copyfile
12 from detection_Model import CNNModel
13
14 # Hyperparameter initialization
15 n_epoch         = 100
16 n_class         = 10
17 batch_size      = 10
18 learning_rate   = 0.0001
19
20 # check if GPU is available
21 print(torch.cuda.current_device())
22 print(torch.cuda.device(0))
23 print(torch.cuda.device_count())
24 print(torch.cuda.get_device_name(0))
25
26 #To run on GPU
27 device = torch.device("cuda:0")
28 dtype = torch.float
29 # Sorting out the data
30
31 # Image parameters
32 n_cnn = 5 #Number of CNN layer
33 img_size = (256,256)
34 conv_size = int( img_size[0]/ (2**(n_cnn)) ) # image_size / 8 for 3 cnn layer.
35 train_img = "../TrainingData"
36 Model = "./Model"
37
38 # Define the transformation
39 transform = transforms.Compose( [transforms.Resize(img_size),
40                                  transforms.RandomRotation((90, 360)),
41                                  transforms.RandomVerticalFlip(),
42                                  transforms.RandomHorizontalFlip(),
43                                  transforms.ColorJitter(),
44                                  transforms.ToTensor(),
45                                  transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5
46                                  ])
47
48 # Training dataset
49 train_dataset = datasets.ImageFolder(root=train_img, transform=transform)
50
51 # Placing data into dataloader for better accessibility
52 # 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")
```

7.3 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)))
```