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

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

Clemson University

March 8, 2019

Abstract

This report explains the process involved in a Convolutional Neural Network 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 deep learning network based on the concept of convolutional neural network for detecting and classifying cyberbully actions for a given image. The cyberbullying actions considered in this project are laughing, pulling-hair, quarrel, slapping, punching, stabbing, gossiping, strangle and isolation. The designed model is trained using the provided image dataset which contain above mentioned 9 categories of cyberbully actions in them.

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]

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

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 model 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 input image is converted into mono-channeled, gray scale image of size 256 x 256. 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:** The input to this layer is a preprocessed 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 16 channel feature map.

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

4. **Max Pooling Layer 1:** This layer down-samples the 256 x 256 16 channel feature map to 128 x 128 16 channel feature map.

5. **Convolution Layer 2:** The input to this layer is the 16 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.

6. **ReLU Layer 2:** This layer applies a relu activation function to the 32 channel feature map.

7. **Max Pooling Layer 2:** This layer down-samples the 128 x 128 32 channel feature map to 64 x 64 32 channel feature map.

8. **Convolution Layer 3:** The input to this layer is the 32 channel 64 x 64 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.

9. **ReLU Layer 3:** This layer applies a relu activation function to the 32 channel feature map.

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

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

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

13. **Fully Connected Layer 1 and Relu layer 3:** This layer maps the 1D feature map into 5000 neurons.

14. **Fully Connected Layer 2 and Relu layer 4:** This layer maps the 5000 neurons to 500 neurons.

15. **Fully Connected Layer 3 and Relu layer 5:** This layer maps 500 neurons into 250 neurons.

16. **Fully Connected Layer 4 and Relu layer 6:** This layer maps 250 neurons into 100 neurons.

17. **Fully Connected Layer 5 and softmax:** This layer maps 100 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 = 25
2. Batch Size = 1
3. Learning Rate = 0.0001

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

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 1 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 1: Test image from the 'quarrel' category

The network classified Figure 1 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 2 as two girls in the foreground gossiping about the third girl in the background. Thus this image falls under the 'gossiping' category.

Figure 2: Test image of two girls gossiping in the foreground

The network classified Figure 2 to be in the ‘strangle’ category.

Upon inspection, Figure 3 appears to comprise of two girls in the background gossiping about the girl in the foreground. Again this is an instance of ‘gossiping’

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

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

Figure 4 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 4: Test image of isolation

The network classified Figure 4 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 1) 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 2, which is an instance of 'gossiping' the network wrongly classified it as 'strangle'. Since the subject in Figure 2 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 3 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 4 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/

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 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, st
        self.relu1 = nn.ReLU()

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

        # Convolution Layer 2
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, s
        self.relu2 = nn.ReLU()

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

        # Convolution Layer 3
        self.cnn3 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, s
        self.relu3 = nn.ReLU()

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

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

```

```
43          # Fully connected linear layer
44          #self.fc1 = nn.Linear(32*75*75 , 9)  #32 channels, 75x75 final image si
45          self.fc1 = nn.Linear(32*image_size*image_size, 5000)
#32 channels, 7x7 final image size
46          self.relu4 = nn.ReLU()
47
48          self.fc2 = nn.Linear(5000, 500)  #32 channels, 7x7 final image size
49          self.relu5 = nn.ReLU()
50
51          self.fc3 = nn.Linear(500, 250)  #32 channels, 7x7 final image size
52          self.relu6 = nn.ReLU()
53
54          self.fc4 = nn.Linear(250, 100)  #32 channels, 7x7 final image size
55          self.relu7 = nn.ReLU()
56
57          self.fc5 = nn.Linear(100, 10)  #32 channels, 7x7 final image size
58
59      #Image size = 28x28 -> 13x13 after first pooling
60      #14x14 after padding = 1
61      #7x7 after second pooling
62
63      def forward(self, x):
64          """ Forward Propogation for classification """
65
66          # Convolution 1
67          out = self.cnn1(x)
68          out = self.relu1(out)
69
70          # Max pool 1
71          out = self.maxpool1(out)
72
73          # Convolution 2
74          out = self.cnn2(out)
75          out = self.relu2(out)
76
77          # Max pool 2
78          out = self.maxpool2(out)
79
80          # Convolution 3
81          out = self.cnn3(out)
82          out = self.relu3(out)
83
84          # Max pool 2
85          out = self.maxpool3(out)
86
87          # Resize the tensor, -1 decides the best dimension automatically
88          #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 = self.relu4(out)

        out = self.fc2(out)
        out = self.relu5(out)

        out = self.fc3(out)
        out = self.relu5(out)

        out = self.fc4(out)
        out = self.relu6(out)

        out = self.fc5(out)

        # Return
        return out
```

7.2 Training 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

# Hyperparameter initialization
n_epoch         = 20
n_class         = 10
batch_size      = 1
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:1")
dtype = torch.float
# Sorting out the data

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

# Define the transformation
transform = transforms.Compose( [transforms.Resize(img_size),
                                 transforms.Grayscale(num_output_channels=1),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5
                                 #transforms.Normalize((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()

# 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()

        # Clear the gradients
        optimizer.zero_grad()

        # Forward propogation
        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) % 100 == 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

# loading the input test image
test_img_filename = sys.argv[1]

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

# Hyperparameter initialization
batch_size      = 1

if not os.path.exists(test_img):
    os.makedirs(test_img)

# clear the contents of the directory
for the_file in os.listdir(test_img):
    file_path = os.path.join(test_img, the_file)
    try:
        if os.path.isfile(file_path):
            os.unlink(file_path)
        #elif os.path.isdir(file_path): shutil.rmtree(file_path)
    except Exception as e:
        print(e)


copyfile(test_img_filename, test_img+test_img_filename)

# Define the transformation
transform = transforms.Compose( [transforms.Resize(img_size),
                                 transforms.Grayscale(num_output_channels=1),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5
                                 #transforms.Normalize((0.5),(0.5))
                                 ])


# Testing dataset
```

```
test_dataset = datasets.ImageFolder(root=test_img1, transform=transform)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batc

# Image parameters
n_class         = 10

model = CNNModel(conv_size, n_class).cuda()
model.load_state_dict(torch.load('./Model/model.pth'))
model.eval().cuda()

def ten_to_str(x):
    """ Function to convert tensor label to a string """
    value = x.data[0] #Convert to data
    str_label = ["gossiping", "isolation", "laughing", "nonbullying", "pullingh
    return str_label[value]

# Testing the model
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()
        outputs = model(images).cuda()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        #if (predicted!=labels):
        #   print("predicted: {} | Actual: {}, total: {} ".format(ten_to_str(pr
        print("{}".format(ten_to_str(predicted)))


#print('Test Accuracy of the model on the {} test images: {} %'.format(len(test
```