Darius Bailey

DATS 6203 - 11

Machine Learning II

*Individual Final Project Report - Group 3*

**Introduction**

Computer vision, and more specifically, facial recognition presents a powerful use case for machine learning. For this project, I was interested in better understanding that application of deep learning as well as one of the commonly used frameworks in industry.

Our project goal was to create a facial expression recognition model trained on the Yale Face dataset of images. Our initial investigation of tagged facial recognition datasets was challenged because access was restricted for many of the facial expression datasets, requiring permission from the data provider, which took time. We adjusted our approach to train our model on the Yale Face images and attempt to later test that model on another set of images. The outline of the shared work is as follows:

* Data Collection
* Code for Data Ingestion/Treatment
* Code for Network Creation
* Code for Model Training & Evaluation

**Description of your Individual Work**.

* Data Collection
  + This work consisted purely of downloading the Yale Face[[1]](#footnote-1) and AffectNet[[2]](#footnote-2) datasets. AffectNet was approved for download a couple weeks after it was requested, leaving less time to work on integrating this dataset
* Code for Data Ingestion/Treatment
  + I wrote the class to import and read labels from the YaleFace images. I found code to leverage FaceCascade[[3]](#footnote-3), a tool used to crop the faces from images and combined that with code I found that reads in the image, convert to numpy array, detect the face, and append faces to images & labels.
  + I also wrote code to split into train and test datasets and to call the created models
  + There is code that we leveraged to create logs for Tensorboard [[4]](#footnote-4)that we eventually did not use due to difficulty interpreting output
* Code for Network Creation
  + I wrote the code to create the network models that we compared using the existing function get\_convnet\_output\_size[[5]](#footnote-5) to troubleshoot and help find the appropriate output sizing for each layer.

The Convolutional Layer class was written to create a single convoluational layer with max pooling to reduce dimensionality and dropout to help generalizability. The general structure of the network code was adapted from Convolution networks found online.

class ConvLayer(nn.Module):

def \_\_init\_\_(self, in\_c, out\_c, kernel\_size, max\_pool\_stride=2,

dropout\_ratio=0.5):

super(ConvLayer, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_c, out\_c, kernel\_size=kernel\_size)

self.max\_pool2d = nn.MaxPool2d(max\_pool\_stride)

self.relu = nn.ReLU()

self.dropout = nn.Dropout2d(p=dropout\_ratio)

def forward(self, x):

x = self.relu(self.conv1(x))

return self.max\_pool2d(x)

return self.dropout(self.max\_pool2d(x))

The first network leveraged the convolutional layer plus two fully connected layers. Output tensor of the convolutional layer was set with following the equation:

\[ O = \frac{I - K + 2P}{S} + 1 \],

where O is size of output image,

I is size of input image,

K is size of kernels,

N is number of kernels,

S is stride,

and P is padding

Size of the fully connected layer connected to the convolutional layer was set equal to the number of neurons.

class FaceModule(nn.Module):

def \_\_init\_\_(self):

super(FaceModule, self).\_\_init\_\_()

self.conv1 = ConvLayer(1, 16, kernel\_size=5)

conv\_output\_size, \_ = get\_convnet\_output\_size(self.conv1, 128)

self.fully\_connected1 = nn.Linear(conv\_output\_size, 1024)

self.fully\_connected2 = nn.Linear(1024, 11)

def forward(self, x):

x = self.conv1(x)

x = x.view(x.size(0), -1)

x = self.fully\_connected1(x)

x = nn.functional.log\_softmax(self.fully\_connected2(x))

return x

The second network adds a second convolutional layer and two fully connected layers, with output sizing determined by the equations listed above.

class FaceModule2(nn.Module):

"""Some Information about FaceModule"""

def \_\_init\_\_(self):

super(FaceModule2, self).\_\_init\_\_()

self.convs = []

self.conv1 = ConvLayer(1, 32, kernel\_size=3)

self.conv2 = ConvLayer(32, 64, kernel\_size=3)

self.fully\_connected1 = nn.Linear(57600, 1024)

self.fully\_connected2 = nn.Linear(1024, 11)

def forward(self, x):

x = self.conv1(x)

x = self.conv2(x)

x = x.view(x.size(0), -1)

x = self.fully\_connected1(x)

x = nn.functional.log\_softmax(self.fully\_connected2(x))

return x

The third layer adds batch normalization to try to help generalizability, particularly for leveraging the network to identify images from other datasets with other properties.

class FaceModule3(nn.Module):

def \_\_init\_\_(self):

super(FaceModule3, self).\_\_init\_\_()

self.convs = []

self.conv1 = ConvLayer(1, 32, kernel\_size=3)

self.batchnorm1 = nn.BatchNorm2d(32)

self.conv2 = ConvLayer(32, 64, kernel\_size=3)

self.fully\_connected1 = nn.Linear(57600, 1024)

self.batchnorm2 = nn.BatchNorm1d(1024)

self.fully\_connected2 = nn.Linear(1024, 11)

def forward(self, x):

x = self.conv1(x)

x = self.batchnorm1(x)

x = self.conv2(x)

x = x.view(x.size(0), -1)

x = self.fully\_connected1(x)

x = self.batchnorm2(x)

x = nn.functional.log\_softmax(self.fully\_connected2(x))

return x

The fourth network adds a third convolutional layer to try to assess whether a deeper network with more inputs helps performance.

class FaceModule4(nn.Module):

def \_\_init\_\_(self):

super(FaceModule4, self).\_\_init\_\_()

self.conv1 = ConvLayer(1, 16, kernel\_size=3)

self.batchnorm1 = nn.BatchNorm2d(16)

self.conv2 = ConvLayer(16, 32, kernel\_size=3)

self.batchnorm2 = nn.BatchNorm2d(32)

self.conv3 = ConvLayer(32, 64, kernel\_size=3)

self.batchnorm3 = nn.BatchNorm2d(64)

self.fully\_connected1 = nn.Linear(12544, 1024)

self.batchnorm4 = nn.BatchNorm1d(1024)

self.fully\_connected2 = nn.Linear(1024, 11)

def forward(self, x):

x = self.conv1(x)

x = self.batchnorm1(x)

x = self.conv2(x)

x = self.batchnorm2(x)

x = self.conv3(x)

x = self.batchnorm3(x)

x = x.view(x.size(0), -1)

x = self.fully\_connected1(x)

x = self.batchnorm4(x)

x = nn.functional.log\_softmax(self.fully\_connected2(x))

return x

* Code for Model Training & Evaluation
  + I wrote the code to evaluate model performance across epochs using training loss and confusion matrix performance. I ran into an error getting this to display test error loss by epoch so was unable to show that comparison.

for epoch in range(10): # loop over the dataset multiple times

running\_loss = 0.0

for i, data in enumerate(train\_loader, 0):

# get the inputs

inputs, labels = data

if CUDA:

inputs = Variable(inputs.cuda())

labels = Variable(labels.cuda())

# zero the parameter gradients

optimizer.zero\_grad()

# forward + backward + optimize

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % 5 == 4: # print every 2000 mini-batches

print('[%d, %5d] loss: %.3f' %

(epoch + 1, i + 1, running\_loss / 50))

running\_loss = 0.0

print('Finished Training')

print('Now Evaluating')

net.eval()

test\_inputs, labels = next(iter(test\_loader))

if CUDA:

test\_inputs = test\_inputs.cuda()

labels = labels.cuda()

output = net(test\_inputs)

pred = output.data.max(1, keepdim=True)[1]

correct = pred.eq(labels.data.view\_as(pred)).cpu().sum()

print("Accuracy = {}%".format(100. \* correct / len(test\_loader.dataset)))

y\_pred = pred.cpu().numpy().flatten()

y\_actual = labels.cpu().numpy().reshape((33,1))

y\_probs = torch.nn.functional.softmax(output, dim=1)

## Plot ROC AUC Curve

skplt.metrics.plot\_roc(y\_actual, y\_probs.cpu().detach().numpy())

plt.show()

**Results**. Describe the results of your experiments, using figures and tables wherever possible. Include all results (including all figures and tables) in the main body of the report, not in appendices. Provide an explanation of each figure and table that you include. Your discussions in this section will be the most important part of the report.

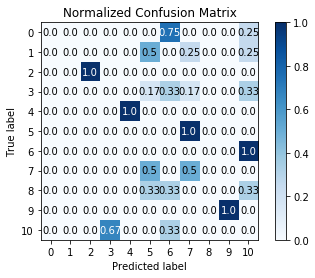
We unfortunately ran out of time before we could properly adapt the code to another set of images, so the conclusions will be based on performance on the relatively small YaleFace dataset. We were, however able to measure test accuracy across all classes. We found Networks 3 and 4 to be most accurate, in aggregate. The figure below illustrates test accuracy aggregated across all ten classes of images.

|  |  |  |  |
| --- | --- | --- | --- |
| FaceModule1 | FaceModule2 | FaceModule3 | FaceModule4 |
| 30% | 3% | 57% | 57% |

We generally found training over 10 epochs to be sufficient without resulting in significant overfit, so all network comparisons were based on that number of epochs, a learn rate of 0.001 and a batch size of 16.

Network 1

Loss by epoch decreased consistently, resulting in a loss of 0.013. Validation on the test subset showed that the model predicted Left Light (class 2), Right Light (class 4), and Center Light (class 9) perfectly. It confused Sad (class 5). expressions for Surprised (class 7) and Sleepy (class. 6) with No Glasses (class 10).



[1, 5] loss: 3.169

[2, 5] loss: 2.105

[3, 5] loss: 0.408

[4, 5] loss: 0.263

[5, 5] loss: 0.138

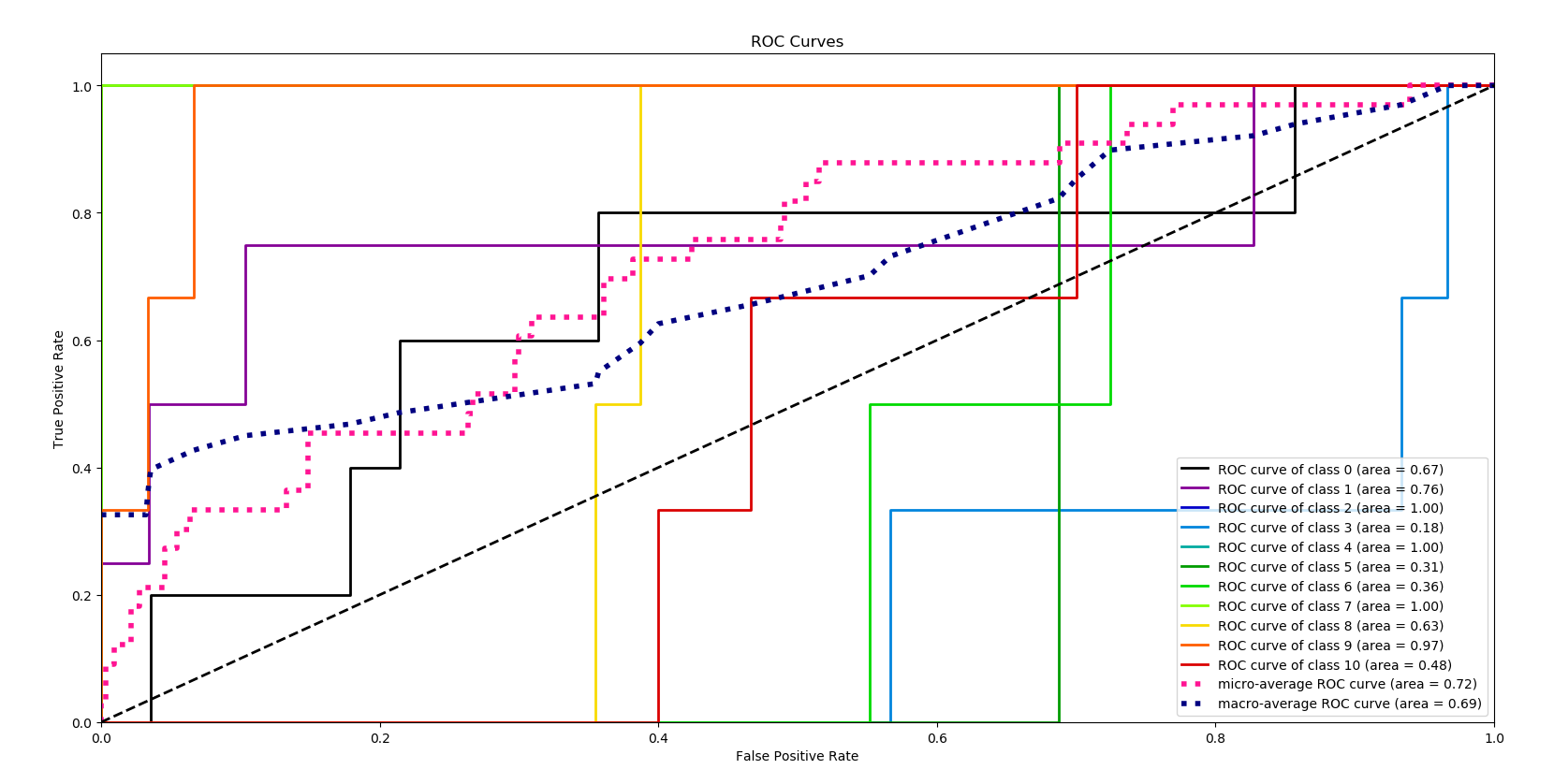
[6, 5] loss: 0.084

[7, 5] loss: 0.055

[8, 5] loss: 0.036

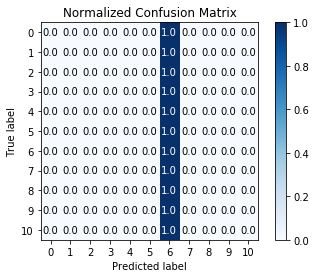
[9, 5] loss: 0.024

[10, 5] loss: 0.013



Network 2

While this network showed better overall performance, loss did not decrease over iterations, which was worrying. Validation showed that every prediction was Sad (class 6), indicating that the network was mis-specified and/or overfit to the characteristics of that expression, potentially driven by the addition of the 2nd convolutional layer.

[1, 5] loss: 1.033

[2, 5] loss: 0.249

[3, 5] loss: 0.240

[4, 5] loss: 0.240

[5, 5] loss: 0.240

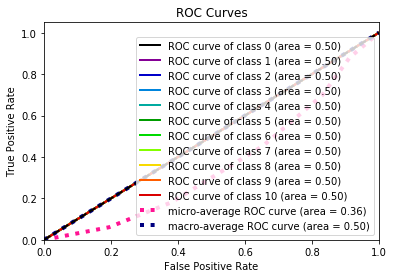
[6, 5] loss: 0.240

[7, 5] loss: 0.240

[8, 5] loss: 0.239

[9, 5] loss: 0.239

[10, 5] loss: 0.239



Network 3

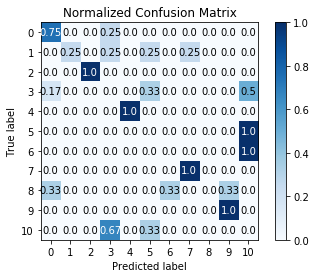
In addition to the second convolutional layer, network 3 adds batch normalization, which appears to improve performance. Now, we are able to reliably predict Glasses, Left Light, Right Light, Center Light, and Surprised. Expressions that are neutral or reliant upon smaller characteristics such as mouth and eyes were still not being picked up accurately, though. We tried to investigate kernel size, but that had no meaningful effect on the performance.

[1, 5] loss: 0.317

[2, 5] loss: 0.117

[3, 5] loss: 0.096

[4, 5] loss: 0.065

[5, 5] loss: 0.033

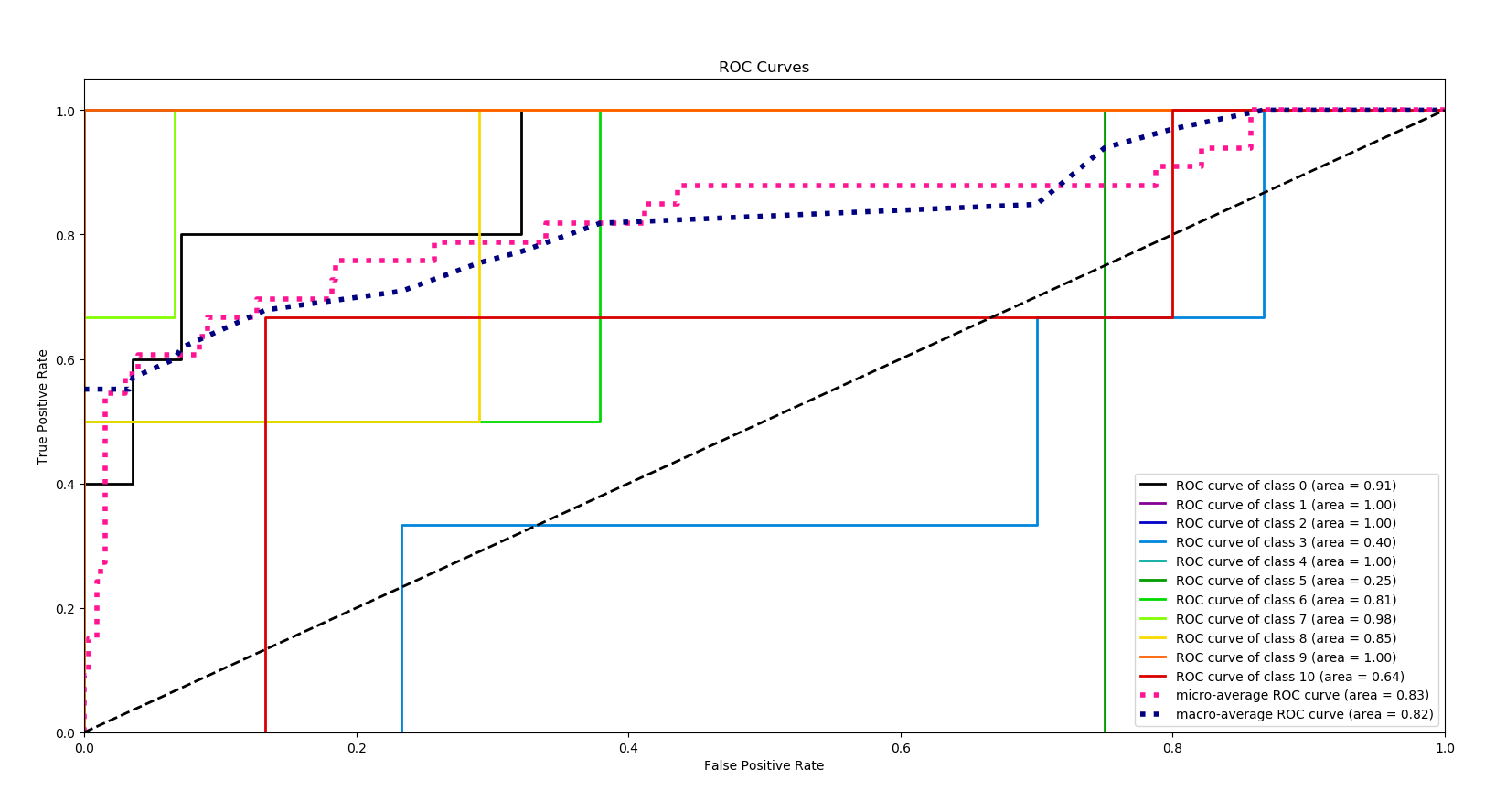
[6, 5] loss: 0.020

[7, 5] loss: 0.012

[8, 5] loss: 0.007

[9, 5] loss: 0.005

[10, 5] loss: 0.003



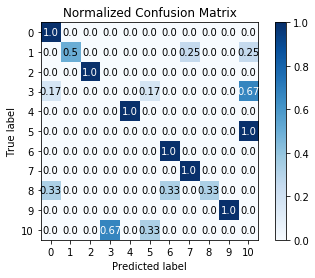
Network 4

This network builds on Network 3 by adding a third convolutional layer and a second batch normalization layer between the final fully connected layers. In training, loss is reduced at a similar rate to what is seen in Network 3. These additions help the network accurately classify Glasses, Left, Right, and Center Light, Sleepy, and Surprised. The additional complexity of the network helped pick up the smaller traits associated with Glasses and the closed eyes associated with Sleepy.

Performance on Happy improved but was also confused with Surprised and No Glasses.

Performance on Normal was unimproved, with the model confused with Sad and No Glasses.

The model struggled to differentiate Wink from Sleepy and Glasses, signaling that the kernels used to distinguish features around the eyes could still use improvement.



[1, 5] loss: 0.323

[2, 5] loss: 0.071

[3, 5] loss: 0.049

[4, 5] loss: 0.026

[5, 5] loss: 0.013

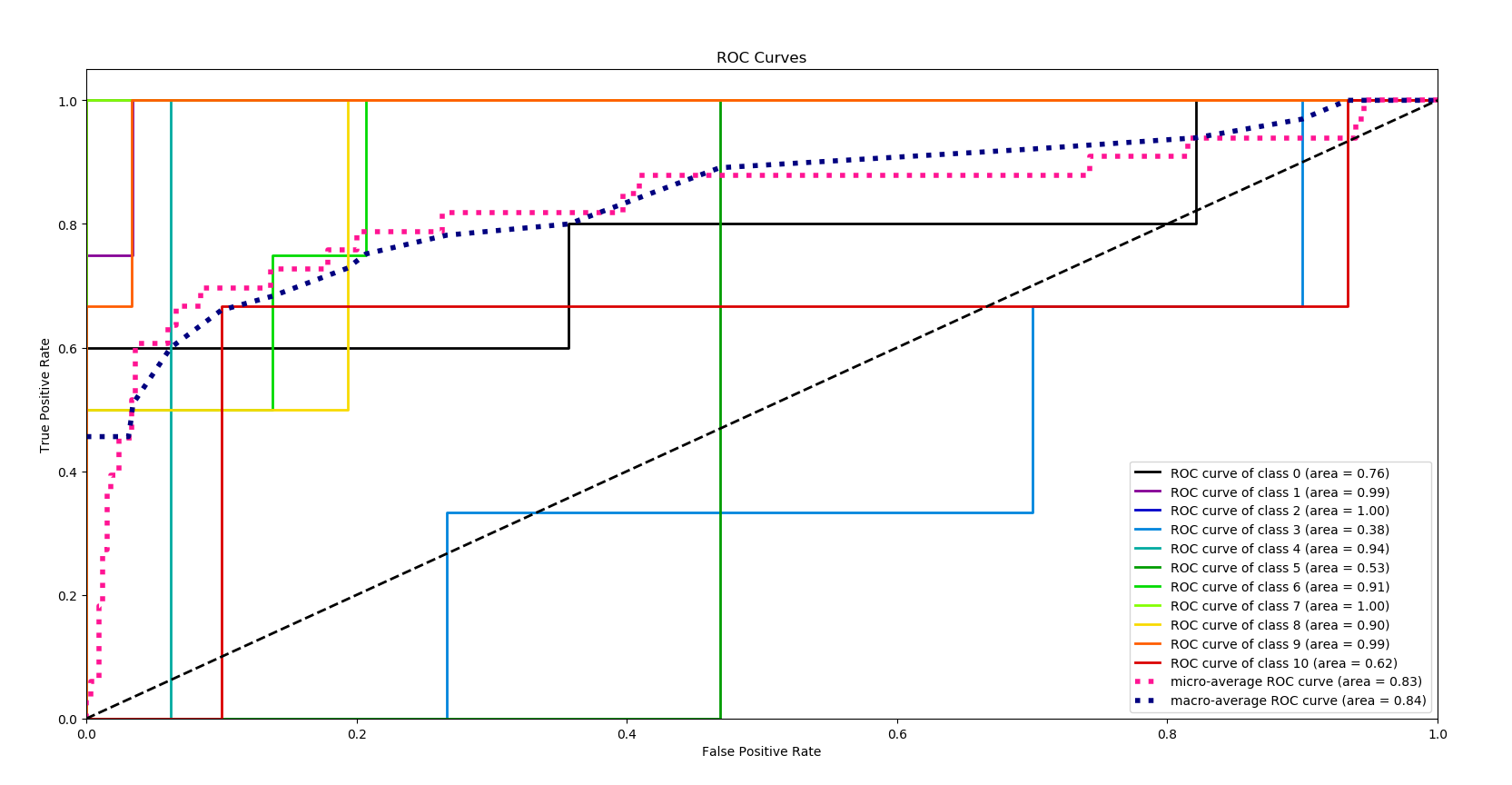
[6, 5] loss: 0.007

[7, 5] loss: 0.005

[8, 5] loss: 0.004

[9, 5] loss: 0.003

[10, 5] loss: 0.002



**Summary and Conclusions**

As our networks became more complex, we saw an improved ability to correctly classify more detailed distinguishing characteristics. With more time, I would have liked to investigate deeper networks ability to continue to improve model performance, while controlling overfit.

As with all data science projects, the majority of our time was spent on data preparation and wrangling. The rest of our time was spent creating networks that ran successfully – both time consuming exercises. We ultimately developed a network that achieved decent performance for the most distinguishable expressions, but through the process gained an appreciation for the level of complexity needed to capture more detailed distinguishable characteristics. By extension, I also have gained an understanding of the level of complexity needed for an application such as facial recognition.

**Code Created vs. Borrowed**

All references to borrowed code was commented in the code itself. My estimate of the proportion of code self generated vs. adapted or copied is based on 354 total lines of code (with spaces), roughly 57 borrowed lines of code, 130 lines of modified code, resulting in 16% of code directly borrowed, 37% modified, and 47% individually contributed.

**References**

<http://vision.ucsd.edu/content/yale-face-database>

<http://mohammadmahoor.com/affectnet/>

<http://hanzratech.in/2015/02/03/face-recognition-using-opencv.html>

<https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/04-utils/tensorboard>

<https://github.com/apsdehal/Face-Recognition/blob/master/model.py>

<https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/>

<https://github.com/Airconaaron/blog_post_visualizing_pytorch_cnn/blob/master/Visualizing%20Learned%20Filters%20in%20PyTorch.ipynb>

1. <http://vision.ucsd.edu/content/yale-face-database> [↑](#footnote-ref-1)
2. <http://mohammadmahoor.com/affectnet/> [↑](#footnote-ref-2)
3. http://hanzratech.in/2015/02/03/face-recognition-using-opencv.html [↑](#footnote-ref-3)
4. <https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/04-utils/tensorboard> [↑](#footnote-ref-4)
5. https://github.com/apsdehal/Face-Recognition/blob/master/model.py [↑](#footnote-ref-5)