DEEP LEARNING PROJECT

Emotion Detection with Facial Images

Abstract: In this project, I used a VGG-liked model to identify the key seven emotions: anger, disgust, fear, happy, sad, surprise and neutral from static facial images. The dataset used for training and testing is Real-world Affective Faces (RAF) Database. The best accuracy achieved is 79% which has outperformed many other previous models which achieved 60-75% accuracy.

1. Introduction

Communication is used in everyday life. Communication is not only expressed by words but also by other things: the way we talk, the way we stand, how we make eye-contact with others, etc. Perhaps not surprisingly, non-verbal behaviors account for two third of our communication. It's not the wording but the way non-verbal behaviors (like gestures, facial expressions, tone of voice, posture, eye movement) are communicated that determines the effectiveness of the messages. In addition, non-verbal indicators can help to reveal hidden messages and emotions that are not communicated verbally. Another interesting thing about non-verbal communication is that most are consistent among people despite their differences in backgrounds, cultures, genders. Therefore, non-verbal behaviors are reliable indicators for researchers to study about human feelings and emotions.

Among non-verbal indicators, facial expressions rank the highest (Mehrabian, 2009) People can infers different emotions like joy, sadness, anger, etc. and information such as ages and genders just by looking at face pictures. People tends to unconciously judge the competence of strangers by the face in a matter of 1/10 sec. In a study conducted by Todorov in 2006, the psychologist has shown that only by using these snap judgements of political candidates' face, he can predict the winner with 70% accuracy (Boutin, 2007).

With the development of deep learning and artifical intelligence, interest in automatic facial emotion recognition (FER) to encode expressions has also increased recently. The majority of basic FER are start with six fundemental emotions: anger, disgust, fear, happiness, sadness, and surprise that are perceived in the same ways accross cultures (Ekman & Friesen, 1971). Data inputs can be images (static image FER) or videos (dynamic sequence FER) collected from sensors like (electromyograph (EMG), electrocardiogram (ECG), electroencephalograph (EEG) and primarily from camera.

This project tackles the problem of detecting emotions based on facial expressions using convolutional neural network CNN models. For the scope of the project, this deep learning model is built from scratch and the data used for training are static face images.

2. Software and Hardware

Google Colab offers free 12.72GB GPU memory and 23.81 GB storage for the purpose of Deep Learning, so I decided to use it. Data is stored on Google Drive and the scripts are run on Google Colab. Nevertheless, I still struggle with processing many datasets, for example:

+ Extended Cohn-Kanade Dataset (http://www.consortium.ri.cmu.edu/ckagree/) with 593 image sequences (640* 490): Google Colab exceeds its limit in already when doing data processing so this dataset is not selected.

+ AffectNet (http://mohammadmahoor.com/affectnet/) is a face database that has around 1M samples but the size(120+Gb) is too big for Google Drive. Therefore, I cannot utilize this dataset for training this model. However, if resources allow, this is a prominent dataset to train the network on.

+ Kaggle dataset FER2013 (https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data) contains 35887 images labeled with 7 emotions. Images have been converted into digits and combined into 1 excel file. However, Google Colab cannot handle this large csv file so I'm skipping this as well.

+ RAF Database (http://whdeng.cn/RAF/model1.html) contains roughly 15k images and I have been able to process data successfully so I choose to train my model with it. Running models sometimes I still ran into the problem of RAM limitation.

3. Dataset

Data is retrieved from Real-world Affective Faces (RAF) Database (URL: http://whdeng.cn/RAF/model1.html). Dataset contains 15339 face images, out of which 12271 images are training data and the rest 3068 images are for testing purpose. All the images are labeled with one of 7 classes - 1: 'surprise', 2: 'fear', 3: 'disgust', 4: 'happy', 5: 'sad', 6: 'angry', 7: 'neutral'. Some example of the images are:

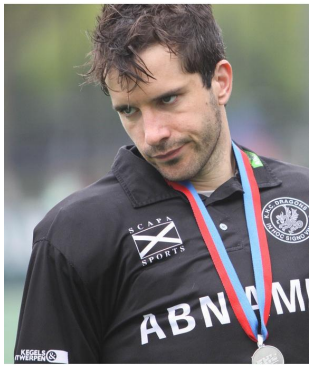
Sad

Happy

Surprise

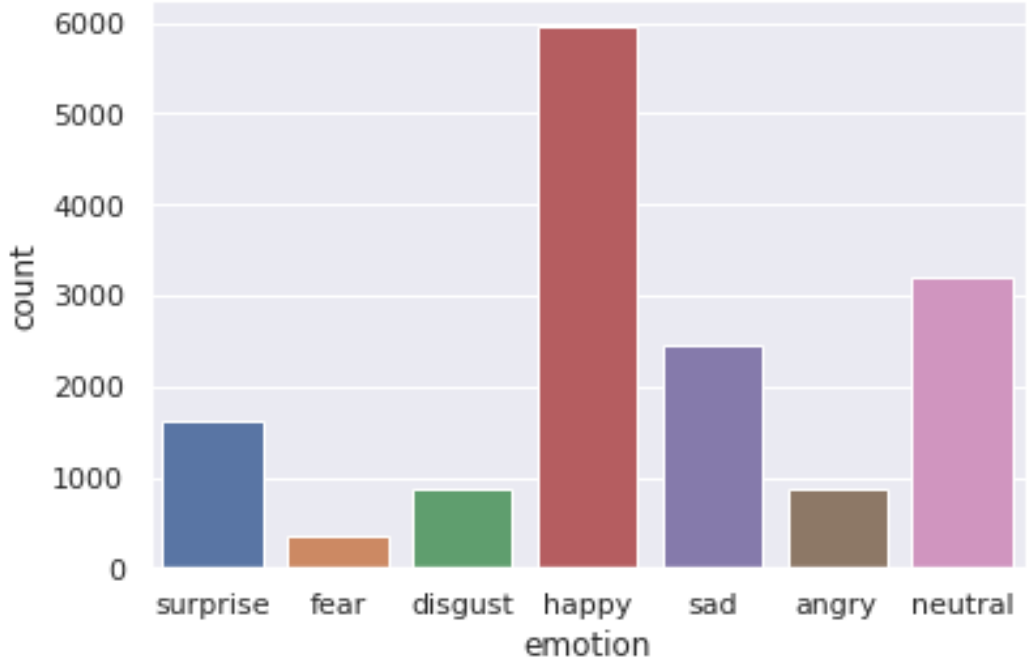
Fear

Anger



Disgust

The distribution of each class in RAF dataset is as followed:



Classes are not balance. The majority of the pictures tend to fall into “happy” or “neutral”, “fear” is the most underrepresented class. Although “Happy” and “Neutral” accounted for the majority of the dataset, similar characteristics also appear in other datasets, for example below is the count for AffectNet dataset and data distribution of FER2013. Therefore, for the scope of this assignment, I won’t go into tackling the issue of class imbalance yet.

Expression	Number
Neutral	80,276
Happy	146,198
Sad	29,487
Surprise	16,288
Fear	8,191
Disgust	5,264
Anger	28,130

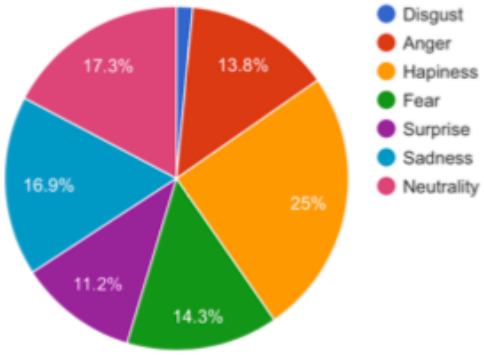


Figure 1. AffectNet images count for selected labels (Ali & Mohammad, 2017) and FER2013 data distribution (Alexandru & James, 2017)

Images varies in dimensions, some of them are colored images and some are gray-scaled images. Converting to Tensor, each image data will have the format [height, width, 3] with 3 denoting the value of 3 color channels.

4. Methods

4.1 First Attempt

Working with original data

In order to use the module dataset of Pytorch, images need to be rearranged into folders corresponding to test/train data and with the folder name indicating its labels. For example, training images with label “surprise” needs to be put into the folder link ‘/train/surprise’. The below code creates these directories and put images to the corresponding folders.

```
#IMAGE DATA
#Put image data into subfolders by each classes
emotion = {1: 'surprise', 2: 'fear', 3: 'disgust', 4: 'happy', 5: 'sad', 6: 'angry', 7: 'neutral'}
folderpath = '/content/gdrive/My Drive/DLProject/Images'

#Create new folder for each label
for i in ['train', 'test']:
  for j in emotion.keys():
    os.makedirs(os.path.join(folderpath, i, emotion[j])


#Iterate over zipfile, extract images and save to corresponding label folders

image_dir = os.path.join(data_dir, 'Image')
imagefile = 'original.zip'

with zipfile.ZipFile(os.path.join(image_dir, imagefile)) as archive:
    for name in archive.namelist():
        if name != 'original/':
          imgname = os.path.basename(name)
          label = labels[labels['filename']==imgname]['emotion'] #Match label of the image
          img = Image.open(name)
          if 'test' in imgname:
            img.save(os.path.join(folderpath, 'test', str(emotion[int(label)]), imgname))
          else:
            img.save(os.path.join(folderpath, 'train', str(emotion[int(label)]), imgname))
```

Since the original dataset varies in size and resolutions, I first need to preprocess the image data. Since I intended to use VGG16, the standard input image size that I aim for is 224x224.

In the first try, I use the most basic data transformation techniques: resize and cropping at the center to obtain image size 224x224.Then the images are transformed to Tensor and normalized.

```
TRAIN_DATA_PATH = "/content/gdrive/My Drive/DLProject/Images/train"
TEST_DATA_PATH = "/content/gdrive/My Drive/DLProject/Images/test"

TRANSFORM_IMG = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,), (0.5))
    ])

train_data = torchvision.datasets.ImageFolder(root=TRAIN_DATA_PATH, transform=TRANSFORM_IMG)
train_loader = data.DataLoader(train_data, batch_size=32, shuffle=True, num_workers=2)
test_data = torchvision.datasets.ImageFolder(root=TEST_DATA_PATH, transform=TRANSFORM_IMG)
test_loader  = data.DataLoader(test_data, batch_size=5, shuffle=False, num_workers=2)
```

VGG16

VGGNet is one of the top convolutional network architecture which has achieved 2nd position in ILSVRC 2014 competition. VGGNet has been discussed in the lecture, so I'm not going in details here. I decided to use VGG16 since it is one of the most popular architectures of VGGNet. VGG16 contains 6 building blocks which in total has 13 layers of convolutional layers with filter size 3x3, max-pooling at the end of each building block and 3 linear layers at the end (Figure 2). My first attempt to is to build a VGG16 net and train the model from scratch.

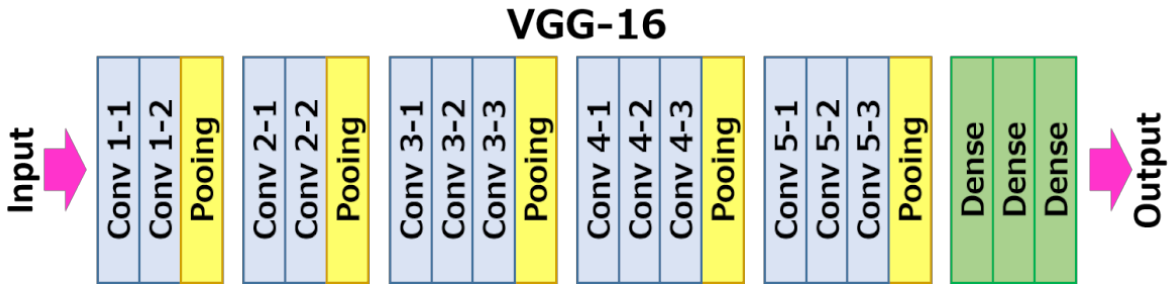


Figure 2. VGG16 Architecture (Neurohive 2018)

VGG16, however, built for classification task with 1000 classes. To adapt the model to 7 classes, I added another Linear Layer at the end with input 1000 and output 7. In addition, I use Batch Normalization after each layers to normalize the outputs. This technique helps each layer maintain some degree of independece and increase the stability of the network.

```python
class VGG16Net(nn.Module):
    def __init__(self, n_channels=64):
        """
        Args:
          n_channels (int): Number of channels in the first convolutional layer. The number of channels in the
                            following layers are the multipliers of n_channels.
        """
        super(VGG16Net, self).__init__()
        n = n_channels
        #Block 1
        self.conv11 = nn.Conv2d(3, n, 3, padding = 1)
        self.conv11_bn = nn.BatchNorm2d(n)
        self.conv12 = nn.Conv2d(n, n, 3, padding = 1)
        self.conv12_bn = nn.BatchNorm2d(n)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 2
        self.conv21 = nn.Conv2d(n, 2*n, 3, padding = 1)
        self.conv21_bn = nn.BatchNorm2d(2*n)
        self.conv22 = nn.Conv2d(2*n, 2*n, 3, padding = 1)
        self.conv22_bn = nn.BatchNorm2d(2*n)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 3
        self.conv31 = nn.Conv2d(2*n, 4*n, 3, padding = 1)
        self.conv31_bn = nn.BatchNorm2d(4*n)
        self.conv32 = nn.Conv2d(4*n, 4*n, 3, padding = 1)
        self.conv32_bn = nn.BatchNorm2d(4*n)
        self.conv33 = nn.Conv2d(4*n, 4*n, 3, padding = 1)
        self.conv33_bn = nn.BatchNorm2d(4*n)

        self.maxpool3 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 4
        self.conv41 = nn.Conv2d(4*n, 8*n, 1)
        self.conv41_bn = nn.BatchNorm2d(8*n)
        self.conv42 = nn.Conv2d(8*n, 8*n, 1)
        self.conv42_bn = nn.BatchNorm2d(8*n)
        self.conv43 = nn.Conv2d(8*n, 8*n, 1)
        self.conv43_bn = nn.BatchNorm2d(8*n)

        self.maxpool4 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 5
        self.conv51 = nn.Conv2d(8*n, 8*n, 1)
        self.conv51_bn = nn.BatchNorm2d(8*n)
        self.conv52 = nn.Conv2d(8*n, 8*n, 1)
        self.conv52_bn = nn.BatchNorm2d(8*n)
        self.conv53 = nn.Conv2d(8*n, 8*n, 1)
        self.conv53_bn = nn.BatchNorm2d(8*n)

        self.maxpool5 = nn.MaxPool2d(kernel_size=2, stride = 2)


        #self.avgpool = nn.AvgPool2d(kernel_size = 2, stride = 1)

        self.fc1 = nn.Linear(7*7*512, 1*1*4096)
        self.fc2 = nn.Linear(1*1*4096, 1*1*4096)
        self.fc3 = nn.Linear(1*1*4096, 1*1*1000)
        self.fc4 = nn.Linear(1*1*1000, 1*1*7)
```

```
def forward(self, x, verbose=False):
    """You can (optionally) print the shapes of the intermediate variables with verbose=True."""
    if verbose: print(x.shape)
    x = F.relu(self.conv11_bn(self.conv11(x)))
    x = F.relu(self.conv12_bn(self.conv12(x)))

    if verbose: print("conv1", x.shape)
    x = self.maxpool1(x)
    if verbose: print('maxpool1:', x.shape)

    x = F.relu(self.conv21_bn(self.conv21(x)))
    x = F.relu(self.conv22_bn(self.conv22(x)))
    if verbose: print('conv2', x.shape)
    x = self.maxpool2(x)
    if verbose: print('maxpool2:', x.shape)


    x = F.relu(self.conv31_bn(self.conv31(x)))
    x = F.relu(self.conv32_bn(self.conv32(x)))
    x = F.relu(self.conv33_bn(self.conv33(x)))
    if verbose: print('conv3', x.shape)
    x = self.maxpool3(x)
    if verbose: print('maxpool3:', x.shape)


    x = F.relu(self.conv41_bn(self.conv41(x)))
    x = F.relu(self.conv42_bn(self.conv42(x)))
    x = F.relu(self.conv43_bn(self.conv43(x)))
    if verbose: print('conv4',x.shape)
    x = self.maxpool4(x)
    if verbose: print('maxpool4:', x.shape)

    x = F.relu(self.conv51_bn(self.conv51(x)))
    x = F.relu(self.conv52_bn(self.conv52(x)))
    x = F.relu(self.conv53_bn(self.conv53(x)))
    if verbose: print('conv5',x.shape)
    x = self.maxpool5(x)
    if verbose: print('maxpool5:', x.shape)

    #x = self.avgpool(x)
    #if verbose: print('avgpool:', x.shape)


    x = x.view(x.size(0), -1)
    if verbose: print('x flatten:', x.shape)

    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.relu(self.fc3(x))

    x = self.fc4(x)

    if verbose: print('out :', x.shape)

    return x
```

Result

The network converge to a very low test set accuracy with 38%. Taking a closer look at the result, the network classified all the images into one category **Happy** only which is not something we want. I also try tweaking some parameters and apply other transformations to the images but no improvements to the result. This could probably be due to VGG16 has too many parameters (138M) compare to the amount of available training data, which leads to overfitting. Another reason is that the image data, after

transformation, are not standardize enough for the network to learn features needed for classification work.

4.2 Second Attempt

Working with modified dataset

RAF database also contains another dataset that are derived from the original set. Images from original data are put through a similarity transformation to by using landmark location of two eyes and centered of the mouth. With this method, facial images are "standardized" and the model could identify the features better. These images comes in size 100x100. Some examples are:


Surprise


Sad

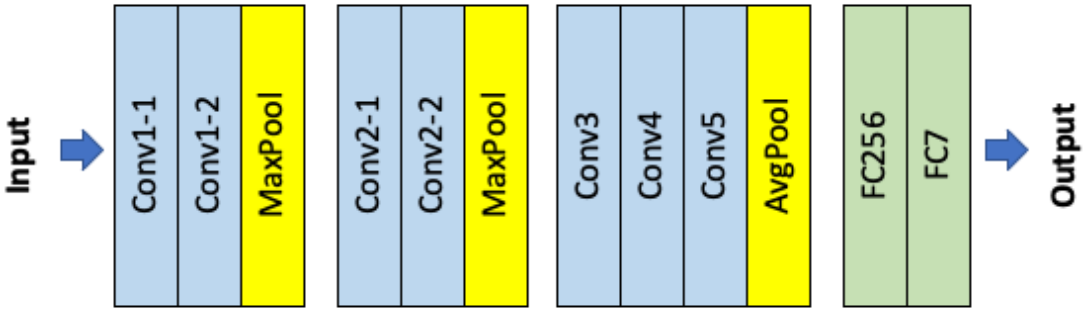

Neutral


Happy

This aligned dataset could possibly be more suitable for training purpose. Therefore, I reran the existing codes with this new dataset.

VGG-liked network

With this aligned dataset, image resolution is 100x100. Therefore, the original VGG16 model no longer work for this dataset. Instead, I use a similar version of the VGG-liked network in Assignment 3 for processing this dataset. The network architecture is as followed:

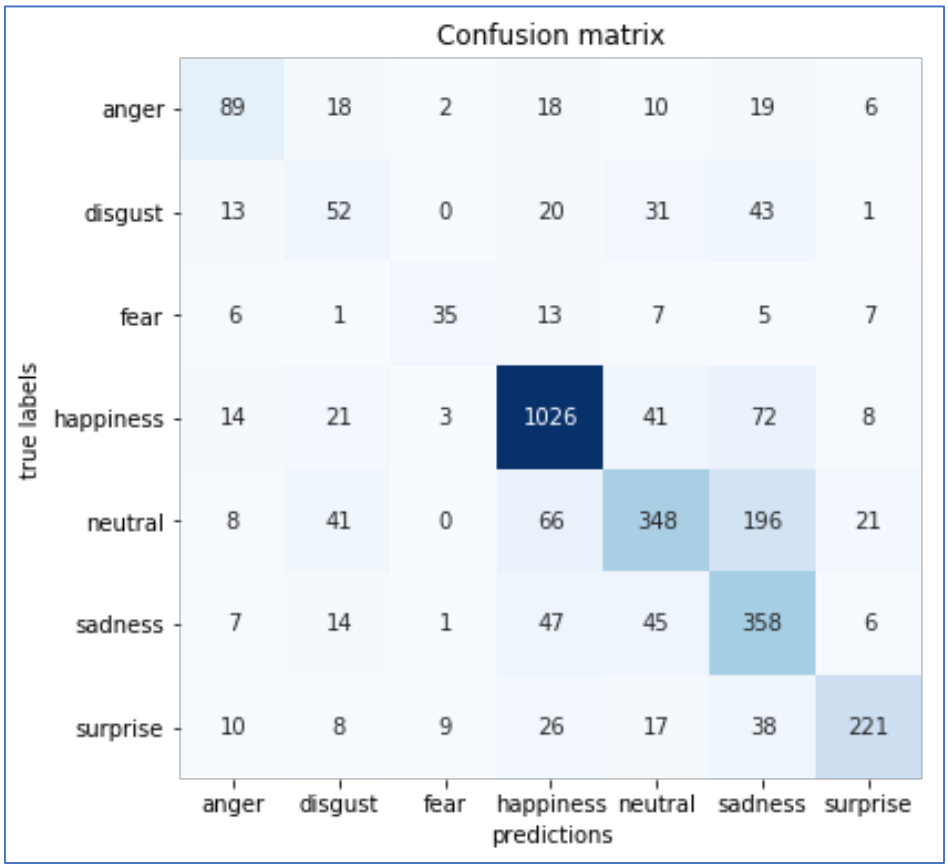


The codes:

```python
class VGG(nn.Module):
    def __init__(self, n_channels=32):
        """
        Args:
          n_channels (int): Number of channels in the first convolutional layer. The number
                            following layers are the multipliers of n_channels.
        """
        super(VGG, self).__init__()
        n = n_channels
        #Block 1
        self.conv11 = nn.Conv2d(3, n, 3, padding = 1) #Change 1 to 3
        self.conv11_bn = nn.BatchNorm2d(n)
        self.conv12 = nn.Conv2d(n, n, 3, padding = 1)
        self.conv12_bn = nn.BatchNorm2d(n)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride = 2)
        # Add dropout


        #Block 2
        self.conv21 = nn.Conv2d(n, 2*n, 3, padding = 1)
        self.conv21_bn = nn.BatchNorm2d(2*n)
        self.conv22 = nn.Conv2d(2*n, 2*n, 3, padding = 1)
        self.conv22_bn = nn.BatchNorm2d(2*n)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride = 2)

        self.conv3 = nn.Conv2d(2*n, 3*n, 3)
        self.conv3_bn = nn.BatchNorm2d(3*n)

        self.conv4 = nn.Conv2d(3*n, 2*n, 1)
        self.conv4_bn = nn.BatchNorm2d(2*n)

        self.conv5 = nn.Conv2d(2*n, n, 1)
        self.conv5_bn = nn.BatchNorm2d(n)

        self.avgpool = nn.AvgPool2d(kernel_size = 5)

        self.fc1 = nn.Linear(512, 256)
        self.fc2 = nn.Linear(256, 7)
def forward(self, x, verbose=False):
    """You can (optionally) print the shapes of the intermediate variables with verbose=
    if verbose: print(x.shape)
    x = F.relu(self.conv11_bn(self.conv11(x)))
    x = F.relu(self.conv12_bn(self.conv12(x)))

    if verbose: print("conv1", x.shape)
    x = self.maxpool1(x)
    if verbose: print('maxpool1:', x.shape)

    x = F.relu(self.conv21_bn(self.conv21(x)))
    x = F.relu(self.conv22_bn(self.conv22(x)))
    if verbose: print('conv2', x.shape)
    x = self.maxpool2(x)
    if verbose: print('maxpool2:', x.shape)

    x = F.relu(self.conv3_bn(self.conv3(x)))
    if verbose: print('conv3', x.shape)

    x = F.relu(self.conv4_bn(self.conv4(x)))
    if verbose: print('conv4',x.shape)

    x = F.relu(self.conv5_bn(self.conv5(x)))
    if verbose: print('conv5',x.shape)

    x = self.avgpool(x)
    if verbose: print('avgpool:', x.shape)

    x = x.view(x.size(0), -1)
    if verbose: print('x flatten:', x.shape)

    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    if verbose: print('out :', x.shape)

    return x
```

Result

This model generates some good results: it achieves 69-72% accuracy on the test set. The confusion matrix (below) shows that it performs well on all the classes although neutral and disgust could be confused by the network as sadness.



4.3 Model Tuning

To further finetune the model performance, the following paramaters have been adjusted:

+ Change the **optimizer** from Adam's method to stochastic gradient descent SGD. There are some sources that suggest that SGD optimizer may achieve better performance results compare to Adam optimizer. This pushes the accuracy slightly up to 73%.

+ Introducing weight decay as a regulization method for the model to reduce overfit

+ Using grid search to train different models with different hyperparameters (learning rate, weight decay, n_epoches, optimizers) to identify the hyperparameters that give better results.

```python
def grid_search(*iterables):

    return itertools.product(*iterables)


def train(learning_rate, optimizer, number_epochs, wd):
  net_tunning = VGG()
  net_tunning.to(device)

  criterion = nn.CrossEntropyLoss()
  optimizer = optimizer(net_tunning.parameters(), lr=learning_rate, weight_decay = wd)
  n_epochs = number_epochs

  test_accuracy_history = []
  test_errors = []  # Keep track of the test error

  net_tunning.train()

  for epoch in range(n_epochs):
      running_loss = 0.0
      print_every = 200
      for i, (inputs, labels) in enumerate(train_loader, 0):
          # Transfer to GPU
          inputs, labels = inputs.to(device), labels.to(device)

          # zero the parameter gradients
          optimizer.zero_grad()

          # forward + backward + optimize
          outputs = net_tunning(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()
          running_loss += loss.item()

          if (i % print_every) == (print_every-1):
            print('[%d, %5d] loss: %.3f' % (epoch+1, i+1, running_loss/print_every))
            running_loss = 0.0

          if skip_training:
            break
      # Print accuracy after every epoch
      accuracy = compute_accuracy(net_tunning, test_loader)
      print('Accuracy of the network on the 3068 test images: %d %%' % (100 * accuracy))
      test_accuracy_history.append(accuracy)

  print('Finished Training')
  return net_tunning, test_accuracy_history

```

For the first test, I only try SGD optimizer and 50 epoches. Since training takes a lot of time and often crash, I haven't able to try additional hyperparameters.

lrate_range	optimizers	n_epochs	weightdecay	accuracy
0.005	<class 'torch.optim.sgd.SGD'>	50	0.001	0.763
0.005	<class 'torch.optim.sgd.SGD'>	50	0.010	0.760
0.010	<class 'torch.optim.sgd.SGD'>	50	0.010	0.750
0.010	<class 'torch.optim.sgd.SGD'>	50	0.001	0.741
0.050	<class 'torch.optim.sgd.SGD'>	50	0.001	0.734
0.050	<class 'torch.optim.sgd.SGD'>	50	0.010	0.616

+ Data Augmentation: Data Augmentation techniques introduce variants and potentially new data to the training set by adjusting colors, flipping, adjusting brightness, crop, etc. For the facial dataset, I apply random horizontal flip and color jitter to introduce more noise to the data. This helps somewhat to increase the accuracy to 77%.

```
TRAIN_DATA_PATH = "/content/gdrive/My Drive/DLProject/train"
TEST_DATA_PATH = "/content/gdrive/My Drive/DLProject/test"
TRANSFORM_IMG_TRAIN = transforms.Compose([
    transforms.Resize(100,100),
    transforms.RandomHorizontalFlip( p = 0.2),
    transforms.ColorJitter(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,), (0.5,))
    ])

TRANSFORM_IMG_TEST = transforms.Compose([
    transforms.Resize(100,100),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,), (0.5,))
    ])



train_data = torchvision.datasets.ImageFolder(root=TRAIN_DATA_PATH, transform=TRANSFORM_IMG_TRAIN)
train_loader = data.DataLoader(train_data, batch_size=32, shuffle=True)
test_data = torchvision.datasets.ImageFolder(root=TEST_DATA_PATH, transform=TRANSFORM_IMG_TEST)
test_loader  = data.DataLoader(test_data, batch_size=5, shuffle=False)

```

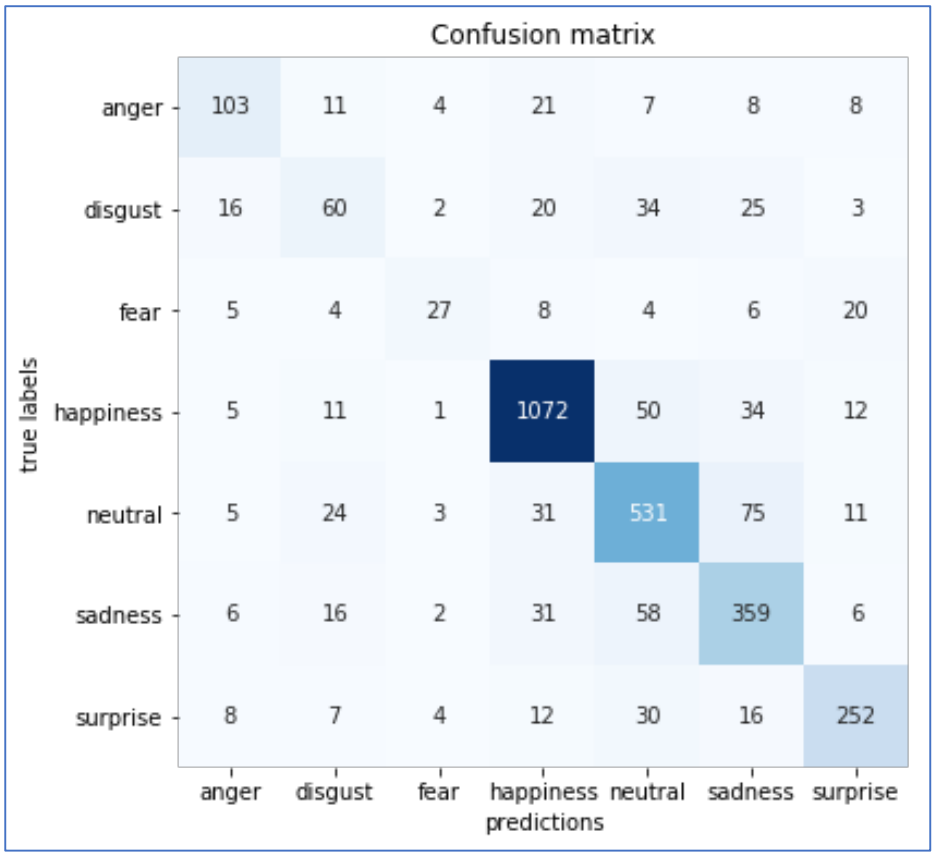
```
Accuracy of the network on the 3068 test images: 76 %
[35,   200] loss: 0.346
Accuracy of the network on the 3068 test images: 77 %
[36,   200] loss: 0.318
Accuracy of the network on the 3068 test images: 77 %
[37,   200] loss: 0.311
Accuracy of the network on the 3068 test images: 76 %
[38,   200] loss: 0.301
Accuracy of the network on the 3068 test images: 75 %
[39,   200] loss: 0.293
Accuracy of the network on the 3068 test images: 77 %
[40,   200] loss: 0.280
Accuracy of the network on the 3068 test images: 75 %
[41,   200] loss: 0.261
Accuracy of the network on the 3068 test images: 77 %
[42,   200] loss: 0.262
Accuracy of the network on the 3068 test images: 75 %
[43,   200] loss: 0.247
Accuracy of the network on the 3068 test images: 77 %
[44,   200] loss: 0.239
```

+ Adjusting number of outputs in convolutional layers of VGG also help to increase the accuracy up to 79%. The new VGG architecture is using the following layers:

```
class VGG2(nn.Module):
    def __init__(self, n_channels=32):

        super(VGG2, self).__init__()
        n = n_channels
        #Block 1
        self.conv11 = nn.Conv2d(3, n, 3, padding = 1) #Change 1 to 3
        self.conv11_bn = nn.BatchNorm2d(n)
        self.conv12 = nn.Conv2d(n, n, 3, padding = 1)
        self.conv12_bn = nn.BatchNorm2d(n)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 2
        self.conv21 = nn.Conv2d(n, 2*n, 3, padding = 1)
        self.conv21_bn = nn.BatchNorm2d(2*n)
        self.conv22 = nn.Conv2d(2*n, 2*n, 3, padding = 1)
        self.conv22_bn = nn.BatchNorm2d(2*n)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride = 2)

        #Block 3
        self.conv31 = nn.Conv2d(2*n, 4*n, 3, padding = 1)
        self.conv31_bn = nn.BatchNorm2d(4*n)

        #Block 4
        self.conv41 = nn.Conv2d(4*n, 8*n, 1)
        self.conv41_bn = nn.BatchNorm2d(8*n)

        #Block 5
        self.conv51 = nn.Conv2d(8*n, 8*n, 1)
        self.conv51_bn = nn.BatchNorm2d(8*n)

        self.avgpool = nn.AvgPool2d(kernel_size = 5)

        self.fc1 = nn.Linear(6400, 256)
        self.fc2 = nn.Linear(256, 7)
```

Final Result

After adding additional regularization and tweaking hyperparameters, the best accuracy I achieved so far is 79% the test set. Comparing the confusion matrix with the confusion matrix result in the second attempt, we can see that the model has improved much more, especially in smaller classes.



Displaying some images and comparing their ground truth labels and predictions:

```
classes = ['anger', 'disgust', 'fear', 'happiness', 'neutral', 'sadness', 'surprise']


# Display random images from the test set, the ground truth labels and the network's predictions
net.eval()
with torch.no_grad():
        dataiter = iter(test_loader)
        images, labels = dataiter.next()
        outputs = net(images)
        out = torchvision.utils.make_grid(images)
        predicted = torch.argmax(outputs.data, 1)
        imshow(out)
        print('Ground truth labels: ', ' '.join('%10s' % classes[labels[j]] for j in range(5)))
        print('Predictions:         ', ' '.join('%10s' % classes[j] for j in predicted))
```

```
Ground truth labels:       anger      anger      anger      anger      anger
Predictions:               anger      anger      anger   surprise      anger
```

```
        images, labels = dataiter.next()
        outputs = net(images)
        out = torchvision.utils.make_grid(images)
        predicted = torch.argmax(outputs.data, 1)
        imshow(out)
        print('Ground truth labels: ', ' '.join('%10s' % classes[labels[j]] for j in range(5)))
        print('Predictions:         ', ' '.join('%10s' % classes[j] for j in predicted))
```

```
Ground truth labels:       anger      anger    disgust    disgust    disgust
Predictions:                fear      anger      anger    disgust      anger
```

5. Model Evaluation

Most of the papers used accuracy as the metric to evaluate emotion detection networks. Accuracy measures the number of correctly classified images. In this project, I also use accuracy to compare different models.

The table below compile and compare accuracy results of different Emotion Detector network from many sources:

Models	URL	Training Data	Accuracy
AlexNet	https://arxiv.org/pdf/1708.03985.pdf	AffectNet	58-72%

CNN	https://github.com/priya-dwivedi/face_and_emotion_detection	FER2013	61.3%
Xception	https://appliedmachinelearning.blog/2018/11/28/demonstration-of-facial-emotion-recognition-on-real-time-video-using-cnn-python-keras/	FER2013	65-66%
CNN + LSTM	https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5856145/	MMI	63-78%
CNN	https://medium.com/neurohive-computer-vision/state-of-the-art-facial-expression-recognition-model-introducing-of-covariances-9718c3cca996	RAF	85-87%
Transfer-Learning CNN	https://arxiv.org/pdf/1806.04957.pdf	RAF	85%
DLP-CNN	http://ice.dlut.edu.cn/valse2018/ppt/WeihongDeng_VALSE2018.pdf	RAF	74.2%
AlexNet/VGG16	https://www.researchgate.net/publication/326681943_Multi-Region_Ensemble_Convolutional_Neural_Network_for_Facial_Expression_Recognition	RAF	74-76%
CNN - CAKE	http://bmvc2018.org/contents/workshops/iahfar2018/0037.pdf	RAF	69-72%
Transfer-Learning CNN	http://cs231n.stanford.edu/reports/2017/pdfs/224.pdf?fbclid=IwAR2jvJQllH_1bE2Vl59GSqdgRakjmBKvljKNOkz8-BygQ6z5Csq_SiC3vKU	RAF	67.2%
My proposed model		RAF	79%

Compare with the average models which has achieved the accuracy of 58-75%, this proposed model with 79% accuracy has outperformed many of them. However, there are still rooms for improvement to get to the state-of-art level of 85+% accuracy. Some approaches that has been proven effective are transfer learning (Shreyank & Abhinav, 2018) and introducing covariance pooling and manifold network (Koidan, 2018).

6. Unimplemented Approach

- Other convolutional network architecture: For this image classification problem, ones can apply variants of convolutional network architectures like ResNet, DenseNet, Inception, etc. to see how these network perform on the dataset. The accuracy could be improved by a few percentage.

- Another improvement can be done is train the network with more data, however there needs to be more computational resources and memory for this task. As I stated in the first part of the paper, working with Google Colab has many limitation on how big the dataset can be.

- Hyperparameter tuning: I have only been able to test the network with SGD optimizer and 50 epoches. Other learning rate, optimizers and number of epoches can be tried out to see whether the accuracy can be improved.

- Adversarial Training (GAN network) is also an interesting approach to train this model with limited amount of data and may also generate good results on the test set. Unfortunately I didn't have time to try with this.

- Transfer Learning is a technique which used a larger dataset to first trained the model. The model will learn specific weights and features from the first dataset and these learnings can be transferred when using this model to finetune/retrain on the target dataset. This approach is very common in deep learning since most of the time the dataset is not big enough to train a network from scratch. With RAF database, I was lucky that despite a limited training samples, the network still able to generate good result. However, this technique has been used in one paper to boost the accuracy to 85.77% (Shreyank & Abhinav, 2018)

7. Conclusion

In this project, I have explored VGG-16 and VGG-liked network on the dataset RAF for emotion detection with facial images. Original image data does not work initially and requires alignment techniques to standardize facial image. The model is then finetuned with different hyperparameters. I was able to get to 79% accuracy with the test set which has outperformed many previous results. However, the model can still be improved further to reach better accuracy (stage-of-art models have achieved 85-87% accuracy).

Reference

Alexandru & James, 2017. Recognizing Facial Expressions Using Deep Learning. URL: http://cs231n.stanford.edu/reports/2017/pdfs/224.pdf?fbclid=IwAR2jvJQllH_1bE2Vl59GSqdgRakjmBKvljKNOkz8-BygQ6z5Csq_SiC3vKU

Ali, M. & Mohammad, H. 2017. AffectNet: A Database for Facial Expression, Valence, and Arousal Computing in the Wild. URL https://arxiv.org/pdf/1708.03985.pdf

Boutin, C. 2007. To determine election outcomes, study says snap judgments are sufficient. URL: https://www.princeton.edu/news/2007/10/22/determine-election-outcomes-study-says-snap-judgments-are-sufficient

Hassan, M. 2018. VGG16 – Convolutional Network for Classification and Detection. URL: https://neurohive.io/en/popular-networks/vgg16/

Koidan, K. 2018. https://medium.com/neurohive-computer-vision/state-of-the-art-facial-expression-recognition-model-introducing-of-covariances-9718c3cca996

Mehrabian, Albert (2009). ""Silent Messages" – A Wealth of Information About Nonverbal Communication (Body Language)". Personality & Emotion Tests & Software: Psychological Books & Articles of Popular Interest. *Los Angeles*: self-published.

P. Ekman and W. V. Friesen, "Constants across cultures in the face and emotion." Journal of personality and social psychology, vol. 17, no. 2, pp. 124–129, 1971.

Shreyank & Abhinav, 2018. Expression Empowered ResiDen Network for Facial Action Unit Detection. URL: https://arxiv.org/pdf/1806.04957.pdf