**Real-Time Face & Emotion Recognition using face images and Deep Learning**

**251 Final Project, Summer, 2021**

**Team:**

Diana Chacon, Jerico Johns, Josh Jonte, Sudhrity Mondal

**Prof(s):**

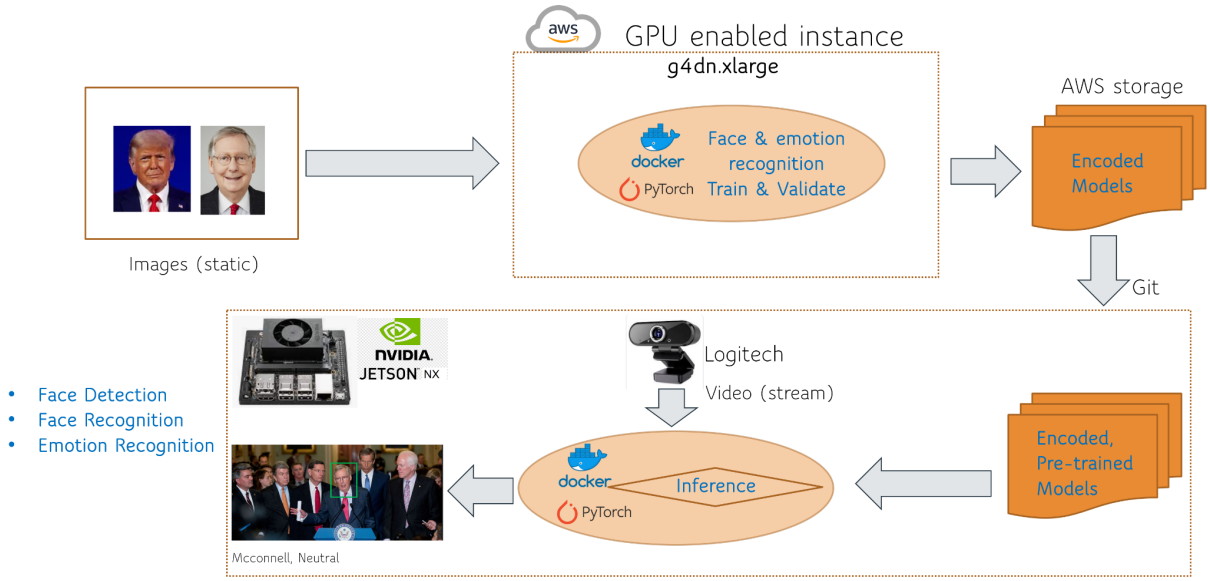
Darragh Hanley & Brad DesAulniers

# 1. Introduction

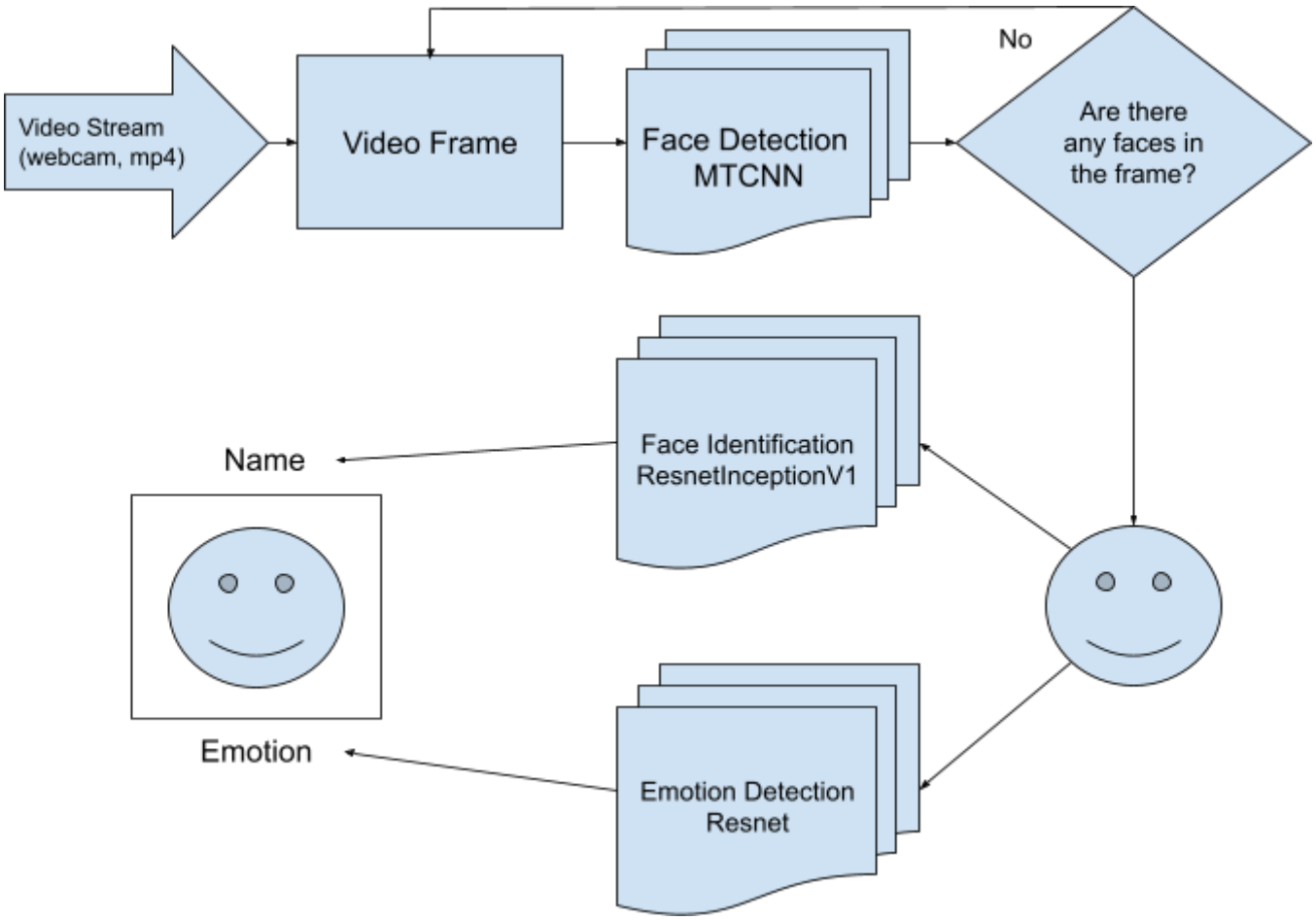
Our team put together a system for recognizing face and basic emotional state using a video feed and the Nvidia Jetson Xavier NX Developer Kit. The video feed can be from a camera attached to the Jetson or a video file.

The system could have multiple commercial uses such as detecting classroom or meeting attendance and engagement, improving dining experiences, and gauging an audiences' feelings towards something they are viewing.

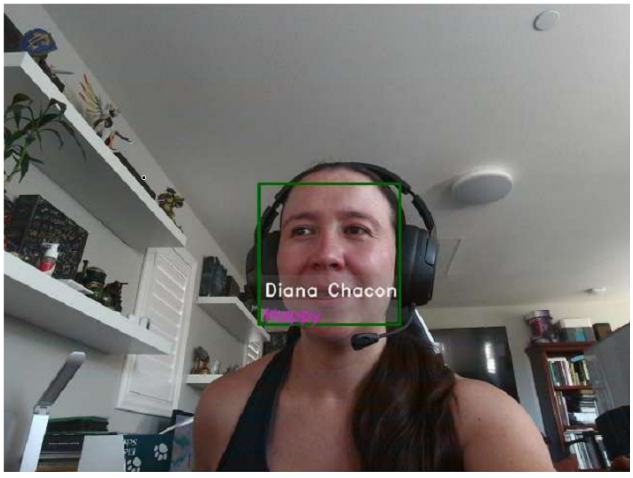
# 1.1 High Level Architecture



# 2. System Architecture




The system is able to process multiple faces in a single frame. This flow chart describes the processing of a single face. The end-state for a single face is a box drawn around the face and the system's estimate of the identity and emotional state of the face drawn at the bottom of the drawn box:



The system consists of three separate neural networks; face detection, face identification, and emotion recognition.

## 2.1 Face Detection

In early proof-of-concepts of the system we used OpenCV's CascadeClassifier with the Haar Cascade Frontal Face detector. While this was adequate for our purposes, we explored if we could train a Deep Neural Network to detect faces because this was, after all, a Deep Learning class. We came across a type of neural network called a Multi-Task Convolutional Neural Network, MTCNN, that is supposed to be superior to Haar for facial detection.

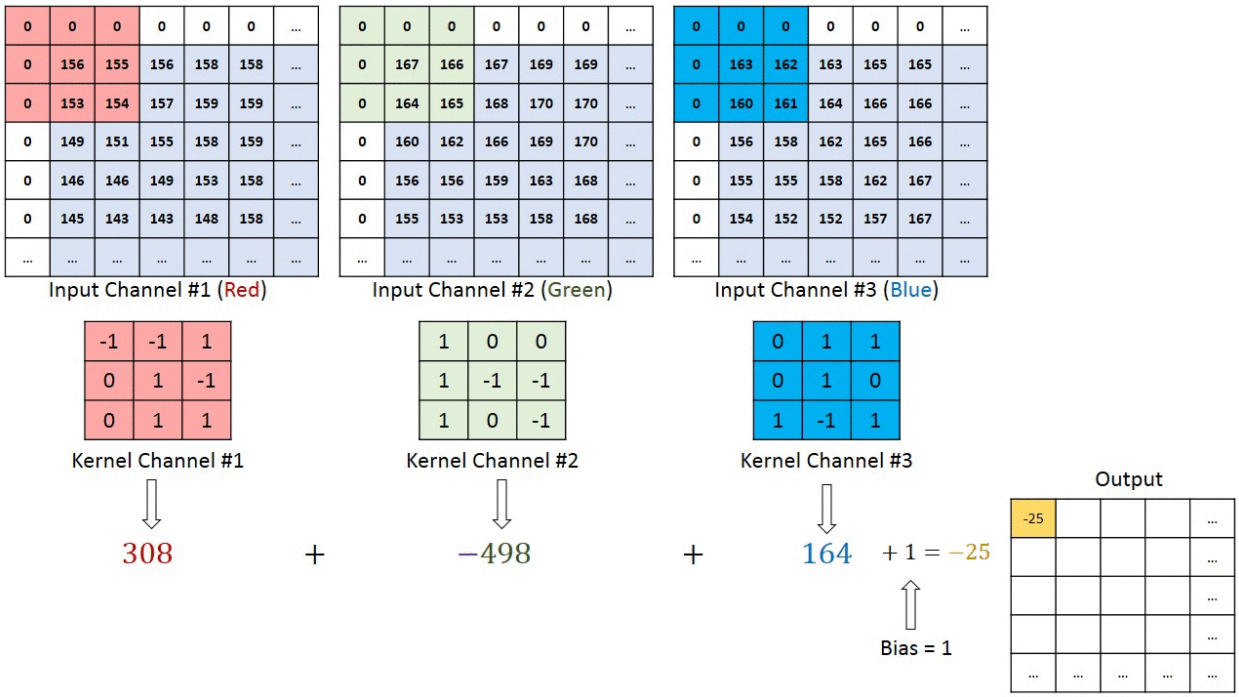
## 2.2 Face Recognition

There are a couple of ways to identify a face using deep learning; facial feature distance identification (using face embeddings) and image-based identification (using pixel values).

Facial-feature distance identification uses a neural network to determine the location of facial features, such as eyes, nose, and mouth, on a given face and compares the location of those features to location of those features on known faces. Ultimately, the network calculates a distance score that is used to compare faces for all desired classes and select the closest-distance (e.g. using euclidean distance) face as the model's prediction.

Image-based identification treats a person's face like any other image recognition task (i.e. types of vehicles or household objects) using a Convolutional Neural Network.

Convolutional Neural Networks are successful in image classification tasks because of the Convolution Operations they perform using kernels (NxN sub selections that stride along the full image) to extract complex features from a flat (black & white) or w*h*3 (RGB color) sized image input of pixel values. These operations are able to extract features like image edges (using max pooling), smooth the image (using average pooling), and extract other complex non-linear features from a relatively simple pixel valued input. The key difference in face recognition tasks are the training data and the transformations applied on the training data to replicate "faces in the wild". We will discuss these in more depth later. The image below offers a nice demonstration of these convolutional operations:




Source

## 2.3 Emotion Recognition

For Emotion detection, we have used two sets of models - one based on the design of ResNet9 and another a simple design based on CNN, both implemented using PyTorch.

**Data Preparation**

The data preparation was done using the following transformation functions on the training data:

```
# Data transforms (Gray Scaling & data augmentation)
train_tfms = tt.Compose([tt.Grayscale(num_output_channels=1),
                         tt.RandomHorizontalFlip(),
                         tt.RandomRotation(30),
                         tt.ToTensor()])

valid_tfms = tt.Compose([tt.Grayscale(num_output_channels=1), tt.ToTensor()])
```

For training, we used GrayScale to make sure all images are in grayscale. Then, we used RandomHorizontalFlip which flips the image horizontally with 50% probability. The image is then rotated at about 30 degrees in random directions(left or right). Finally, we converted the PIL images to tensor since Pytorch layers only work with tensors. This allowed the training data to be fairly random to avoid overfitting.

For validation, we just converted the images to grayscale and then to tensors to avoid model validation inaccuracies by introducing other transformations.

For training, we loaded the data and model into the GPU memory instead of CPU memory (default) using custom functions to get the default device and then load the data and model to the cuda device.

The model creation and training/validation steps are outlined in the Implementation section.

# 3. Datasets

Three different datasets were used to train and validate our models:

## 3.1 Face Recognition Dataset

This dataset was generated using two different methods. The first method uses FFmpeg and MTCNN utilities. FFmpeg is an open-source software project used for handling video, audio, and other multimedia files and streams. And the MTCNN library is used to detect faces. The goal was to feed a video file (.mp4) and extract the faces (500 images with a 0.05 rate).

The second method consisted of manually capturing images from prior class video recordings to capture our classmate's faces. Over 160 images were captured and were split for training and validation purposes.

The images were used to train and validate the face recognition model.
The code for the first method can be found here: /face/m2/Extract Faces.ipynb

## 3.2 Kaggle Dataset

The FER2013 dataset was obtained from Kaggle. The data consists of 48x48 pixel grayscale images of faces. The faces are more or less centred and occupy about the same amount of space in each image. The images are categorized based on the facial expression into seven categories (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). The training set consists of 35887 images, 7178 for testing and 28709 for training. However, not all categories were used in our model.

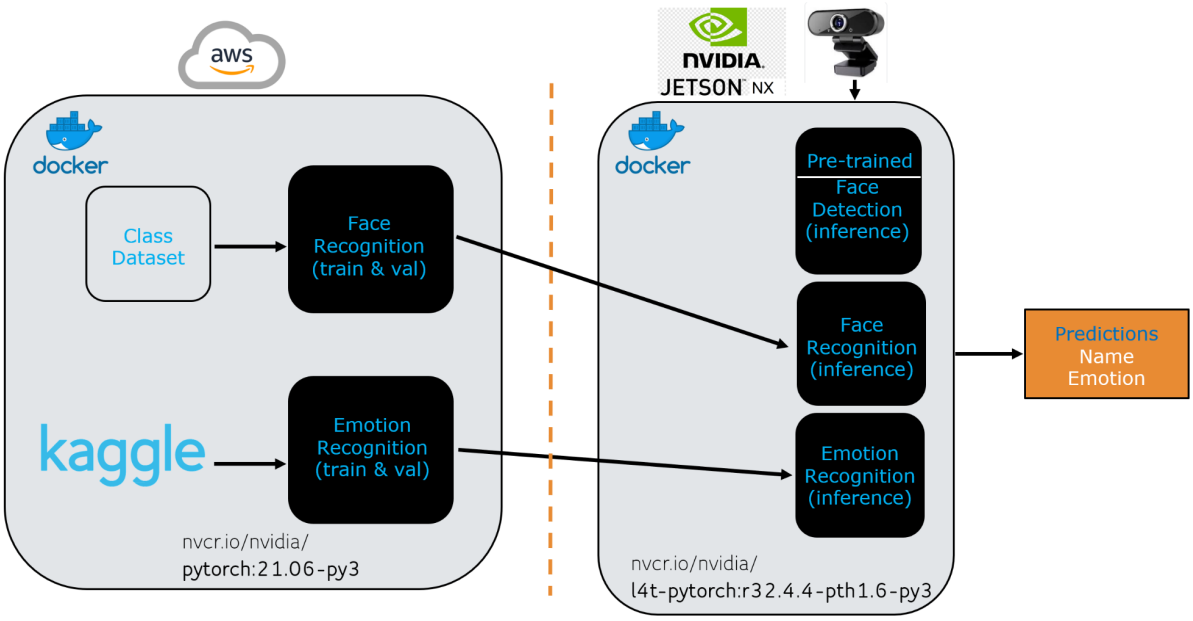
For our model training, we used a total of 24176 images (Happy: 7215, Neutral: 4965, Angry: 3995, Surprise: 3171, Sad: 4830). For our training, we used 3006 images (Happy: 879, Neutral: 626, Angry: 491, Surprise: 416, and Sad: 594)

The images were used to train and validate the emotion model. Data files can be found here: /data/

The rights for the Kaggle dataset belongs to the original authors listed in the file fer2013.bib

# 4. System Design & Implementation

## 4.1 System Design




The models are trained on a container on AWS and then used in the Jetson for inference.

## 4.2 Models & Notebooks

### 4.2.1 Face Detection

**Notebook**: [facenet-pytorch/mtcnn.py at master · timesler/facenet-pytorch]

The MTCNN architecture is complex to implement. However, there are open source implementations of the architecture, as well as pre-trained models that can be used directly for face detection, and that is what we chose to use.

The implementation details around MTCNN can be found in the Arxiv paper titled “Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks.” At the core, MTCNN uses three separate convolutional neural networks and these three networks feed into each other, dramatically increasing the accuracy of face detection.

Pretrained MTCNN that loads the model:

```
model_face_detect = MTCNN(keep_all=True, device=cuda_device)
```

### 4.2.2 Face Recognition

**Notebook:** [mids-w251-project/blob/main/face_emotion/facerec_transfer_training.ipynb]

This model starts with an InceptionResnetV1 model pre-trained on the [VGGFace2 dataset]. This dataset contains 3.31 million images of 9,131 subjects. The images come from Google Image Search and images “have large variations in pose, age, illumination, ethnicity, and profession (e.g. actors, athletes, politicians).”

```
model_ft = InceptionResnetV1(pretrained='vggface2', classify=False, num_classes = len(class_names))
```

We then “freeze” the layers and weights up to the last convolutional block, so that the weights and transformations that occur in the network up until the final inferential block are maintained.

```
#Put all beginning layers in an nn.Sequential .
#model_ft is now a torch model but without the final linear, pooling, batch
norm, and sigmoid layers.
model_ft = nn.Sequential(*list(model_ft.children())[:-5])
```

```
#Set requires_grad false for all prior layers (to freeze them).
for param in model_ft.parameters():
    param.requires_grad = False
```

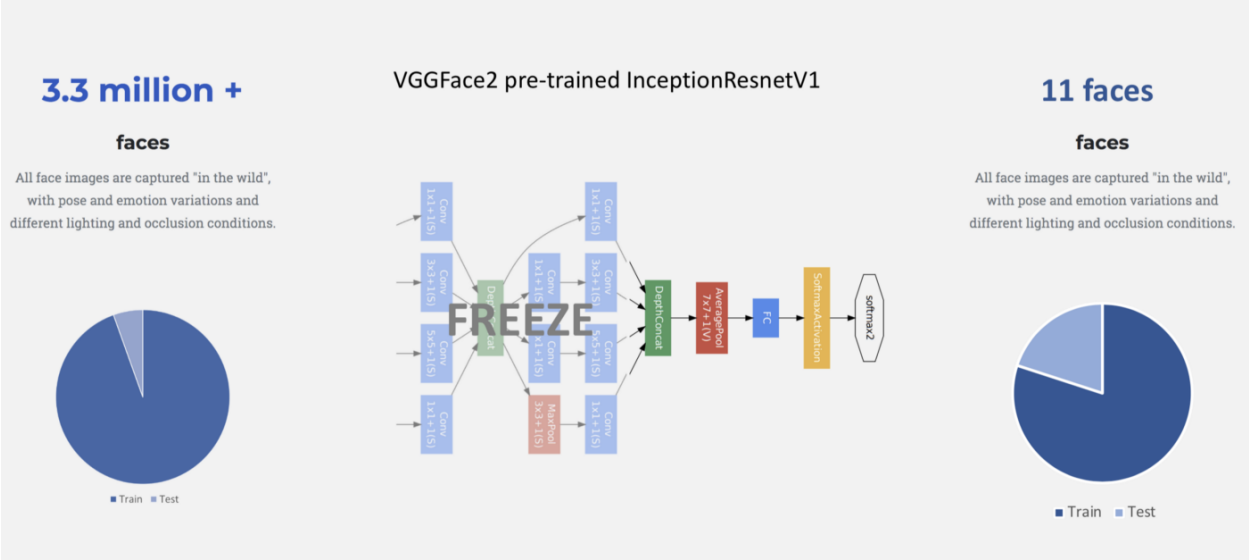
We then assign new labels based on our desired training set, and use this training set to train the weights in the 5 layers of the final convolutional block by setting requires_grad to True. We did make one important change in the dropout layer of the final convolutional block, by adjusting the probability an element of the input tensor is zeroed out from p = 0.6 to p = 0.2. We observed an improvement in training from this adjustment.

Here are the original model’s final 5 layers that we will update:

```
#Remove the last layers after conv block and place in layer_list .
layer_list = list(model_ft.children())[-5:] # all final layers
layer_list
```

```
[AdaptiveAvgPool2d(output_size=1),
 Dropout(p=0.6, inplace=False),
 Linear(in_features=1792, out_features=512, bias=False),
 BatchNorm1d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats
=True),
 Linear(in_features=512, out_features=8631, bias=True)]
```

The entire model fine-tuning process can be summarized as follows:



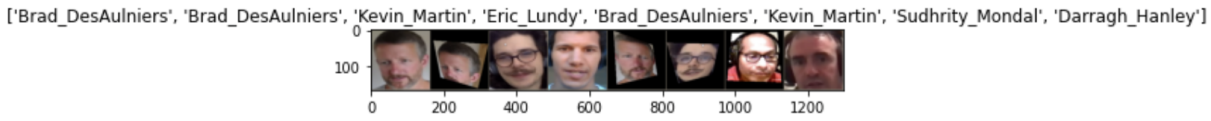

Source

As discussed our training/val dataset (mids-w251-project/data/face/images.m2/) consists of 11 classes (people), with 10 images for each class in the training dataset, and 5 images for each class in the validation dataset. Images were manually screenshotted from Zoom recordings of our synchronous class sessions. Since these images were from often consecutive frames without much variation in pose, angle, and zoom (how close the subject was to their webcam), we apply the following transformations to prevent overfitting on the training set:

```
#Perform transforms and normalization to prepare images for training on ResNet
data_transforms = {
    'train': transforms.Compose([
        #Note: Our images must be 160x160 for model (Training & Val).
        transforms.RandomResizedCrop(size = (160,160), scale = (0.7,1.0)),
        #Add some random transforms in training to help
        #generalize to webcam noise, zoom, and angles.
        #transforms.GaussianBlur(kernel_size = (5,9)),
        #transforms.ColorJitter(),
        transforms.RandomPerspective(),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(size = (160,160)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

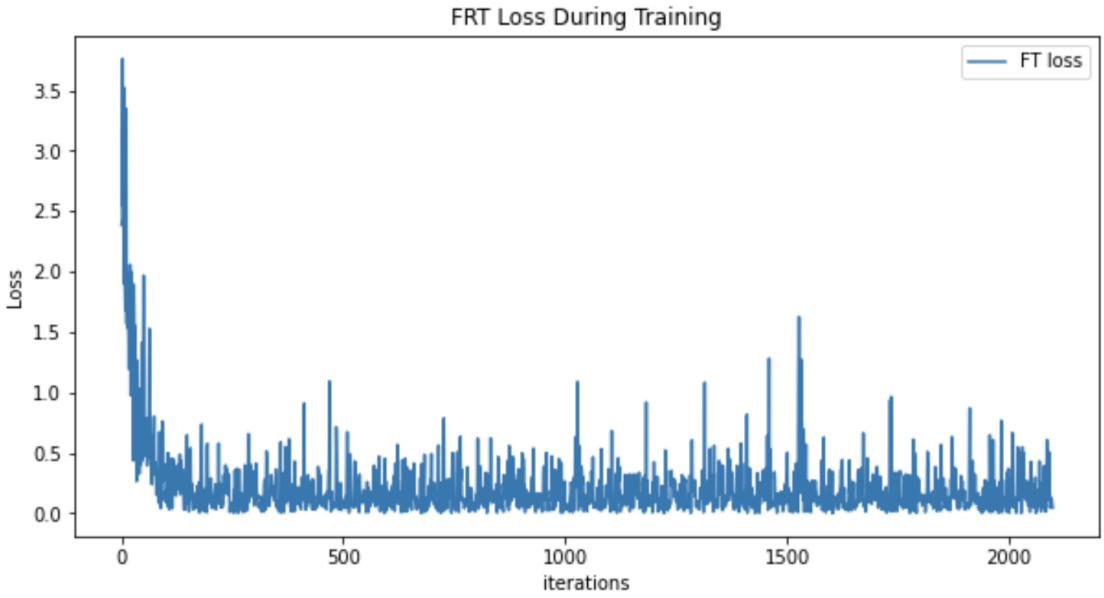
The RandomResizedCrop accounts for subjects moving in and away from their webcam during a live session and the RandomPerspective and RandomHorizontalFlip account for subjects changing angles (side to side). Since the original InceptionResnetV1 is trained on 160x160 images we resize our images and finally normalize to the model specs. Here is an example of the transformed training images with corresponding labels:

```
#Show some images
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated

# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))
# Make a grid from batch
out = utils.make_grid(inputs)
imshow(out, title=[class_names[x] for x in classes])
```




We train the model for 100 epochs (on a batch size of 8 randomly selected images) using a CrossEntropyLoss() criterion. This is a standard criterion for image classification tasks. We use an Adam optimizer, with an initialized learning rate of 0.1 and StepLR learning rate scheduler

that decreases the learning rate by a factor of 0.5 every 20 iterations. The training loss was visualized as follows:




We see modest improvement in training loss through the first 500 iterations, before the model hits a minima and fails to improve further. We tried a number of variations in our hyperparameters but were not able to gain any significant performance improvements beyond those outlined here. This is likely because of our small training/val dataset with limited variation in a person's appearance, position, and video quality. In fact, after 100 epochs we observed the following results for train/val loss and accuracy:

```
----------
train Loss: 0.1825 Acc: 0.9636
val Loss: 0.1166 Acc: 1.0000
Epoch 98/99
----------
train Loss: 0.1556 Acc: 0.9909
val Loss: 0.0806 Acc: 1.0000
Epoch 99/99
----------
train Loss: 0.2251 Acc: 0.9727
val Loss: 0.0880 Acc: 1.0000
Training complete in 7m 31s
Best val Acc: 1.000000
```

As we will discuss later, although our model performed impressively on static images with 100% validation accuracy, the livestream results were varied (closer to ~50% accuracy, although we do not have a validation process for livestream results). This presents a promising area for future work on this project.

Finally we save the updated model to load into our final notebook for inference on live webcam data.

```
torch.save(model_ft, '../models/face_recognition_transf.pt')
```

### 4.2.3 Emotion Recognition

We tried three different approaches to the models used to recognize emotion.

#### Model 1 - ResNet9-based Model

**Notebook**: emotion/emotion-m1tr-0723_64e_.63.ipynb

This model uses the FER2013 dataset from Kaggle previously mentioned. Five training and validation classes were used: 'Neutral', 'Happy', 'Surprise', 'Sad', and 'Angry'. Once the data was loaded, transformations were applied to the training and validation datasets such as Grayscale, Random Horizontal Flip, and Random Rotation.

We used the ImageFolder library to apply transformations to each image. We then used the DataLoader library to load the data and shuffle for training only. We used a batch size of 200 since that fit well with the g4dn.xlarge GPU memory.

```
# Emotion Detection datasets
train_ds = ImageFolder(data_dir + '/train', train_tfms)
valid_ds = ImageFolder(data_dir + '/val', valid_tfms)
```

```
batch_size = 200
```

```
# PyTorch data loaders
train_dl = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4, pin_memory=True)
valid_dl = DataLoader(valid_ds, batch_size*2, num_workers=4, pin_memory=True)
```

Below, is a sample of the data in the train dataset.

```
show_batch(train_dl)
```

```
torch.Size([1, 48, 48])
```

The model was trained and validated in AWS in an nvidia/pytorch:21.06-py3 container using a g4dn.xlarge instance. Dockerfile and scripts to build and run the container can be found in the aws folder /aws/.

The CNN model is based on the design of a ResNet9 implementation. Three convolution blocks were created. Each convolution block takes the number of channels for the input and output tensor, and max-pooling (true or false). Conv2d was used for all the layers, with a kernel size of 3, padding of 1, and input channels: 1, 128, and 256, and output channels 128, 256, and 512. Due to the low resolution of the images, increasing the number of blocks would cause the model to fail, limiting our model to those 3 blocks.

In addition, BatchNorm2d was used for batch normalization, ELU for the activation function, MaxPool2d, and Dropout of 0.5.

```
ResNet(
  (conv1): Sequential(
    (0): Conv2d(1, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
  )
  (conv2): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res1): Sequential(
    (0): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ELU(alpha=1.0, inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ELU(alpha=1.0, inplace=True)
    )
  )
  (drop1): Dropout(p=0.5, inplace=False)
  (conv3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
  )

(drop1): Dropout(p=0.5, inplace=False)
(conv3): Sequential(
  (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ELU(alpha=1.0, inplace=True)
)
(conv4): Sequential(
  (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ELU(alpha=1.0, inplace=True)
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(res2): Sequential(
  (0): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
  )
  (1): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
  )
)
(drop2): Dropout(p=0.5, inplace=False)
(conv5): Sequential(
  (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ELU(alpha=1.0, inplace=True)
)
```

```
  (conv6): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0, inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res3): Sequential(
    (0): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ELU(alpha=1.0, inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ELU(alpha=1.0, inplace=True)
    )
  )
  (drop3): Dropout(p=0.5, inplace=False)
  (classifier): Sequential(
    (0): MaxPool2d(kernel_size=6, stride=6, padding=0, dilation=1, ceil_mode=False)
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): Linear(in_features=512, out_features=5, bias=True)
  )
)
```

The model was run using the following command that takes the number of channels, number of classes, and device (CUDA if available).

Command to run the model

```
model = to_device(ResNet(1, len(classes_train)), device)
model
```

The following  parameters were used to train the model.

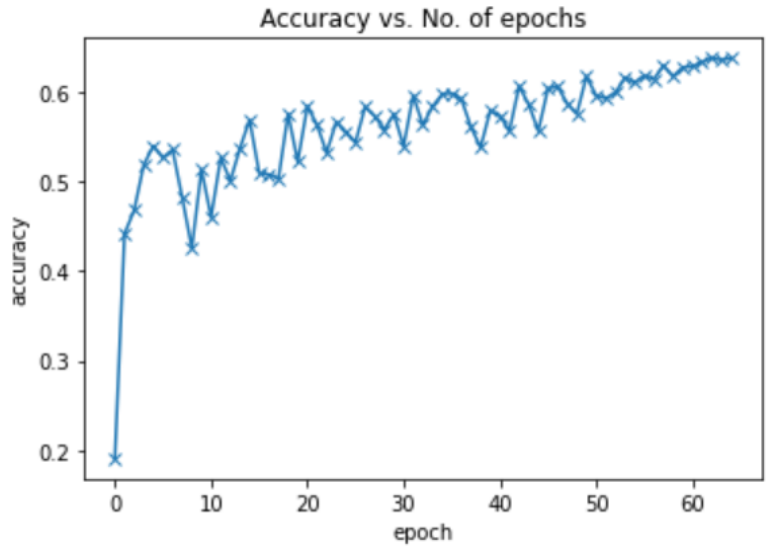
- epochs = 64
- max_lr = 0.008
- grad_clip = 0.1
- weight_decay = 1e-4
- opt_func = torch.optim.Adam.

The model was trained for 1 hour and 17 min, reaching an accuracy of 0.6370 on epoch [63] and a validation loss of 1.1078 (see below).

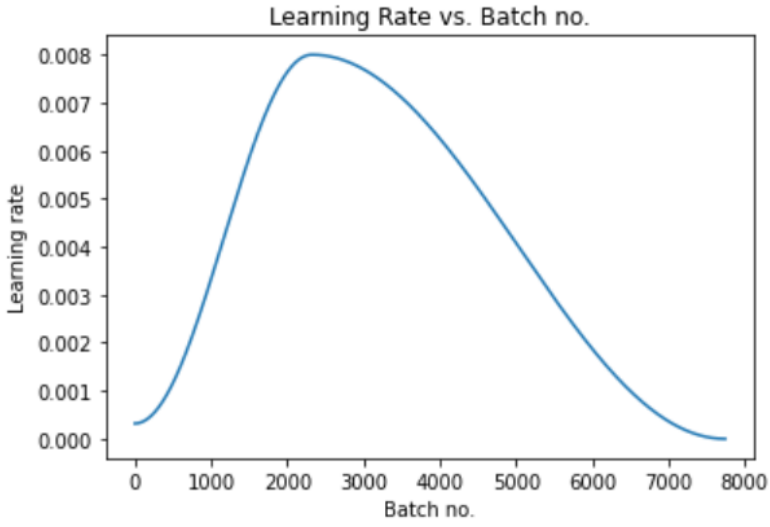
```
Epoch [56], last_lr: 0.00047, train_loss: 0.6974, val_loss: 1.0893, val_acc: 0.6290
Epoch [57], last_lr: 0.00035, train_loss: 0.6836, val_loss: 1.1089, val_acc: 0.6168
Epoch [58], last_lr: 0.00024, train_loss: 0.6722, val_loss: 1.1120, val_acc: 0.6270
Epoch [59], last_lr: 0.00016, train_loss: 0.6621, val_loss: 1.1034, val_acc: 0.6277
Epoch [60], last_lr: 0.00009, train_loss: 0.6602, val_loss: 1.1071, val_acc: 0.6326
Epoch [61], last_lr: 0.00004, train_loss: 0.6527, val_loss: 1.1055, val_acc: 0.6373
Epoch [62], last_lr: 0.00001, train_loss: 0.6500, val_loss: 1.1089, val_acc: 0.6351
Epoch [63], last_lr: 0.00000, train_loss: 0.6429, val_loss: 1.1078, val_acc: 0.6370
CPU times: user 45min 57s, sys: 31min 14s, total: 1h 17min 11s
Wall time: 1h 17min 50s
```

A series of graphs were generated to better understand the model performance.

```
plot_accuracies(history)
```




*The Accuracy vs. No. of epochs* graph shows how the accuracy increased as the number of epochs increased. In addition, we can see fluctuations in the accuracy related to the learning rate scheduler adjusting the learning rate values for each epoch.




*The Learning Rate vs. Batch number* graph shows how the learning rate was increased from a low value at 0 epochs up to approximately the max value configured (0.008) at 2200 epochs and then the learning rate decreased as the batch size increased to approximately 0. No fixed learning rate was used for the model, as previously mentioned the learning rate scheduler modified the learning rate after each batch of training.

After training the model, it was saved to the /models/ folder using the following command:

```
torch.save(model.state_dict(), '../models/emotion-m1tr-0723_64e_.63.pth')
```

We are using this model for part of our demonstration.

### Model 2 - Custom CNN

**Notebook**: emotion/emotion_m2tr-0723_90e_.35.ipynb

This model uses the FER2013 dataset from Kaggle previously mentioned. Five training and validation classes were used: 'Neutral', 'Happy', 'Surprise', 'Sad', and 'Angry'. Once the data was loaded, transformations were applied to the training and validation datasets such as Grayscale, Random Horizontal Flip, and Random Rotation.

A set of the training images is displayed below.

Just like the prior model, this model was trained and validated in AWS in an nvidia/pytorch:21.06-py3 container using a g4dn.xlarge instance. Dockerfile and scripts to build and run the container can be found in the aws folder /aws/

The CNN model used 7 cnn with input channels: 1, 8, 16, 32, 34, 128, and 256, and output channels 8, 16, 32, 34, 128, and 256. Relu was used as the activation function,

In addition, BatchNorm2d was used for batch normalization, ELU for the activation function (ReLU will take the maximal value between 0 and x), MaxPool2d, BatchNorm2d, and Dropout of 0.3

The model was run using the following command that takes the number of channels, number of classes, and device (CUDA if available).

Command to run the model:

```
model = to_device(ResNet(), device)
model
```

The model was run using CUDA.

```
ResNet(
  (cnn1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1))
  (cnn2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (cnn3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (cnn4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (cnn5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (cnn6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (cnn7): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
  (relu): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (cnn1_bn): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn2_bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn3_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn4_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn5_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn6_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn7_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=1024, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=7, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (log_softmax): LogSoftmax(dim=1)
)
```

The following  parameters were used to train the model.

- epochs = 90
- max_lr = 0.008
- grad_clip = 0.1

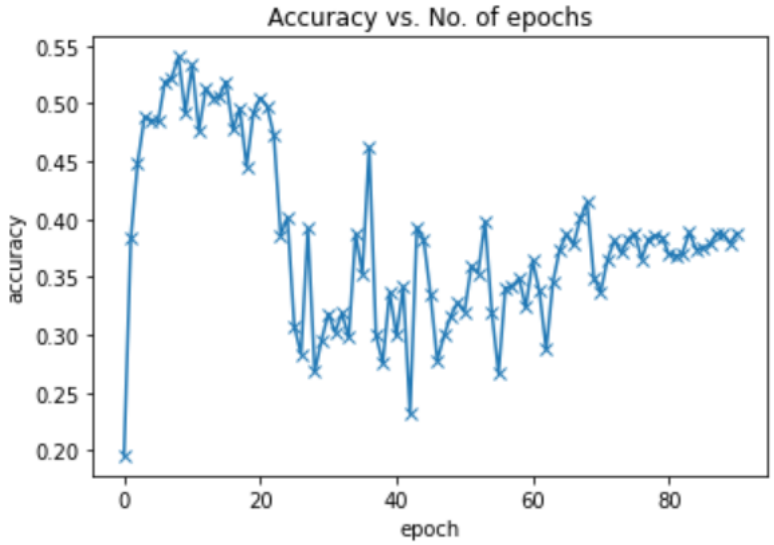
- weight_decay = 1e-4
- opt_func = torch.optim.Adam.

The model was trained for 47 min, reaching an accuracy of 0.39 on epoch [89] and a validation loss of 1.1.5003 (see below).

```
Epoch [82], last_lr: 0.000241, train_loss: 0.7071, val_loss: 1.4889, val_acc: 0.3885
Epoch [83], last_lr: 0.000178, train_loss: 0.7017, val_loss: 1.5072, val_acc: 0.3741
Epoch [84], last_lr: 0.000124, train_loss: 0.6880, val_loss: 1.5031, val_acc: 0.3750
Epoch [85], last_lr: 0.000079, train_loss: 0.6951, val_loss: 1.5046, val_acc: 0.3778
Epoch [86], last_lr: 0.000045, train_loss: 0.6926, val_loss: 1.5028, val_acc: 0.3876
Epoch [87], last_lr: 0.000020, train_loss: 0.6874, val_loss: 1.4977, val_acc: 0.3866
Epoch [88], last_lr: 0.000005, train_loss: 0.6884, val_loss: 1.5146, val_acc: 0.3789
Epoch [89], last_lr: 0.000000, train_loss: 0.6905, val_loss: 1.5003, val_acc: 0.3882
CPU times: user 28min 16s, sys: 18min 49s, total: 47min 6s
Wall time: 47min 41s
```

A series of graphs were generated to better understand the model performance.

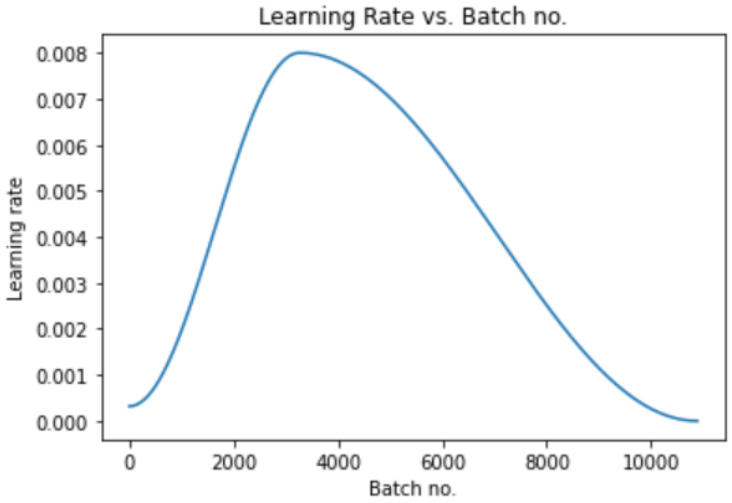
```
plot_accuracies(history)
```




*The Accuracy vs. No. of epochs* graph shows how the accuracy fluctuated as the number of epochs increased. In addition, we can see fluctuations in the accuracy related to the learning rate scheduler adjusting the learning rate values for each epoch.

Comparing Emotion Recognition Model 1 and 2, we were able to achieve better accuracy in Model 1 (63%) compared to the Model 2 accuracy (39%)

```
plot_lrs(history)
```




*The Learning Rate vs. Batch number* graph shows how the learning rate was increased from a low value at 0 epochs up to approximately the max value configured (0.008) at 3200 epochs and then the learning rate decreased as the batch size increased to approximately 0. No fixed learning rate was used for the model, as previously mentioned the learning rate scheduler modified the learning rate after each batch of training.

After training the model, it was saved to the /models/ folder using the following command:

```
torch.save(model.state_dict(), '../models/emotion_m2tr-0723_90e_.35.pth')
```

### Model 3 - Pre-trained using fer2013.csv

**Notebook**: Since this is a pre-trained model, no notebooks were used for validation or training. The implementation of this Model 2 pre-trained can be found in notebooks face_emotion/Face_detection_recognition_and_emotion_recognition.ipynb and other notebooks in the face_emotion folder.

We also used a pre-trained model *emotion_m2pt.pt* to capture emotion for static images and from live video streams based on the article [shangeth/Facial-Emotion-Recognition-PyTorch-ONNX](shangeth/Facial-Emotion-Recognition-PyTorch-ONNX) .

The model is same as hat used in the earlier section and shown below:

```
ResNet(
  (cnn1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1))
  (cnn2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
  (cnn3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (cnn4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (cnn5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (cnn6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (cnn7): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
  (relu): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (cnn1_bn): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn2_bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn3_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn4_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn5_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn6_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (cnn7_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=1024, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=7, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (log_softmax): LogSoftmax(dim=1)
)
```

This pre-trained model was trained using dataset fer2013.csv with 7 classes (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral). The training data consists of 28,709 samples. The validation set used 3,589 samples.

We were not able to locate the exact fer2013.csv dataset used for training and validation of the model which is why we decided to use the pre-trained model instead.

This model achieved an accuracy of between 0.78 - 0.85 for the fer2013.csv dataset.

We are using this model for part of our demonstration.

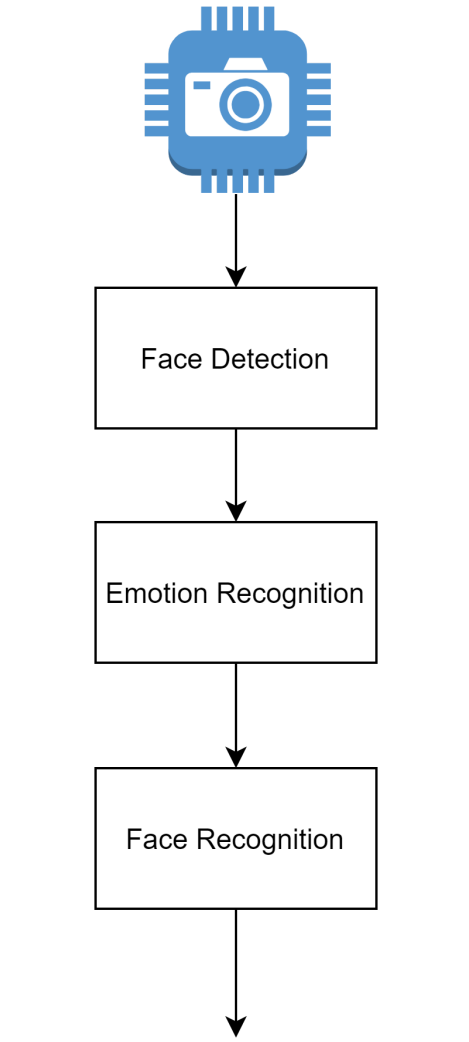
Command to load the pretrained model:

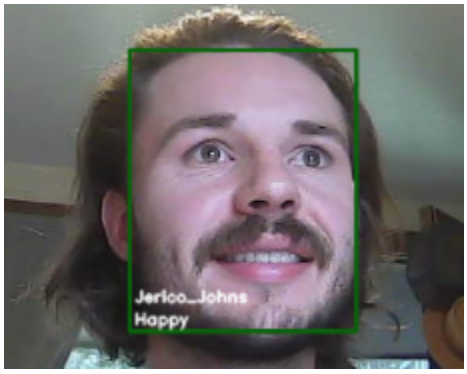
```
model = load_trained_model('../models/emotion_m2pt.pt')
```

## 4.4 Integration

The face detection, face recognition, and emotion recognition models are integrated in notebooks located under the face_emotion/ folder in the repository.

The models are executed sequentially and the inferences are shown in the final image.



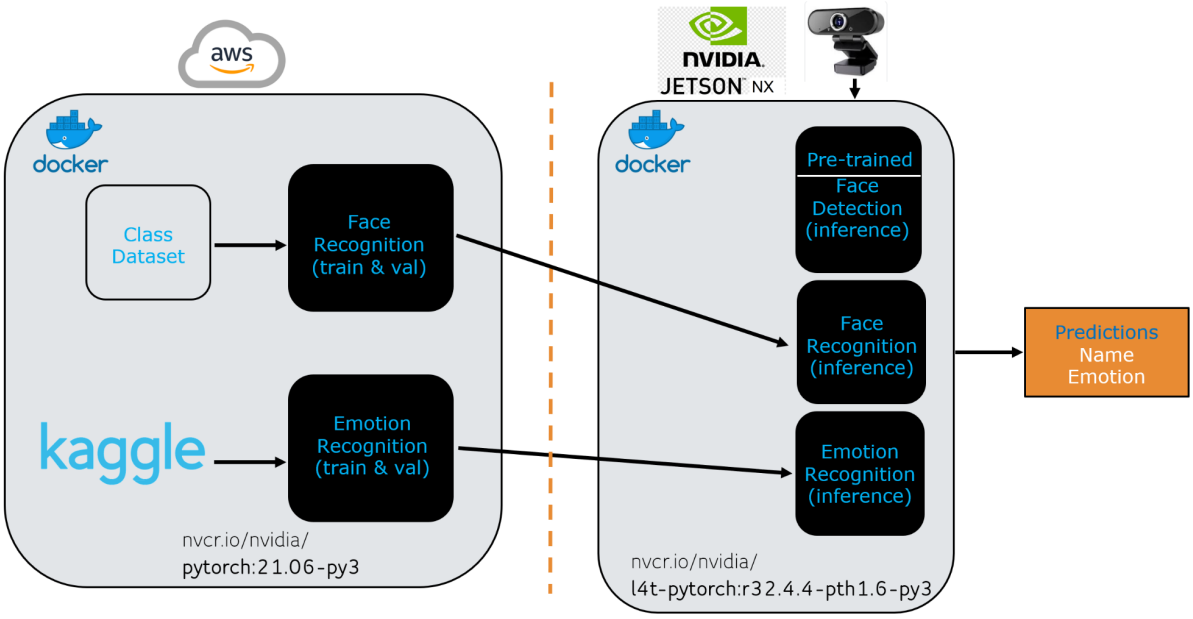




The code-segment that does this integration (*Face_detection_recognition_and_emotion_recognition.ipynb*) is shown below:

```
try:
    while(True):
        frame_count += 1

        clear_output(wait=True)
        _, frame = video_capture.read()
        faces, _ = model_face_detect.detect(frame)

        if faces is not None:

            for top, left, bottom, right  in faces:

                gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                resize_frame = cv2.resize(gray[int(top):int(bottom), int(left):int(right)], (48, 48))

                X = resize_frame/256
                X = PILImage.fromarray((X))
                X = val_transform(X).unsqueeze(0)
                with torch.no_grad():
                    model_emotion_m1.eval()
                    log_ps = model_emotion_m1.cpu()(X)
                    ps = torch.exp(log_ps)
                    top_p, top_class = ps.topk(1, dim=1)
                    pred = emotion_dict[int(top_class.numpy())]

                #face_img = frame[0:300, 0:200]
                face_img = frame[int(top):int(bottom), int(left):int(right)]
                cv2.rectangle(frame, (top, left), (bottom, right), (0, 100, 0), 2)

                #Face name recognition
                rgb_face = face_img[:, :, ::-1]
                face_name = get_face_name(rgb_face)

                font = cv2.FONT_HERSHEY_DUPLEX
                x, y, w, h = int(top+2), int(right-50), int(left-2), int(bottom-26)

                scale = 0.045 # this value can be from 0 to 1 (0,1] to change the size of the text relative to the i
                fontScale = min(w,h)/(25/scale)

                cv2.putText(frame, face_name, ((int(top)+3), (int(right)-18)), font, fontScale, (255, 255, 255), 1,
                cv2.putText(frame, pred, ((int(top)+3), (int(right)-4)), font, fontScale, (255, 255, 255), 1, cv2.LI
```

## 4.3 Implementation

The implementation of the project is done on AWS and the Nvidia Jetson Xavier NX edge device. The system implementation diagram is reproduced below for reference.




We used the **nvcr.io/nvidia/pytorch:21.06-py3** Docker container from Nvidia as the base container to run on g4dn.xlarge x86 VM on AWS for training and validation of both the face recognition and the emotion recognition models. The docker container was updated to add the libraries needed for face recognition.

We used the **nvcr.io/nvidia/l4t-pytorch:r32.4.4-pth1.6-py3** Docker container from Nvidia as the base container to run on Nvidia Jetson Xavier NX Jetpack 4.4 edge device inference of face detection, face recognition and emotion recognition. The docker container was updated to add the libraries needed for face detection, recognition and emotion recognition.

The project repository is available on Github at:

https://github.com/sudhrity/mids-w251-project

The folder structure for the implementation and some description of the folders are shown below.

```
📦mids-w251-project
 ├ 📂.git
 ├ 📂aws        Contains the dockerfile, the docker build and run scripts for container on AWS
 |  ├ 📜build.sh
 |  ├ 📜Dockerfile
 |  └ 📜run.sh
 ├ 📂data       Contains the datasets used for face recognition, emotion recognition - train and validation
 |  ├ 📂emotion
 |  |  ├ 📂train
 |  |  |  ├ 📂Angry
```

```
|   |   |   ├ 📂Happy
|   |   |   ├ 📂Neutral
|   |   |   ├ 📂Sad
|   |   |   ├ 📂Surprised
|   |   ├ 📂val
|   |   |   ├ 📂Angry
|   |   |   ├ 📂Happy
|   |   |   ├ 📂Neutral
|   |   |   ├ 📂Sad
|   |   |   ├ 📂Surprised
|   ├ 📂face
|   |   ├ 📂images.m1
|   |   ├ 📂images.m2
|   ├ 📂test_images
├ 📂emotion Contains the notebooks for training/val of emotion recognition models 1 & 2 using fer2013 datasets
|   ├ 📜emotion-m1tr-0723_64e_.63.ipynb Uses emotion model 1 trained on FER 2013 image dataset
|   ├ 📜emotion_m2tr-0723_90e_.35.ipynb Uses emotion model 2 trained on FER 2013 image dataset
├ 📂face  Contains the notebook and Python script for training/val/inference of face recognition model
|   ├ 📂m1  Contains the Python script to run face_recognition pre-trained model for face_recognition
|   |   ├ 📜barack_obama.jpg
|   |   ├ 📜facerec_from_webcam_faster.py
|   |   ├ 📜jerico_johns.jpg
|   |   ├ 📜joe_biden.jpg
|   ├ 📂m2  Contains Notebooks to extract faces from video and transfer learning models for face recognition
|   |   ├ 📜Extract Faces.ipynb
|   |   ├ 📜face_emotion_transfer.ipynb
|   |   ├ 📜face_tracking.ipynb
|   |   ├ 📜facerec_transfer_training.ipynb
├ 📂face_emotion  Contains Notebooks that integrates Face detection, recognition and emotion recognition
|   ├ 📂utils
|   ├ 📜Face_detection_recognition_and_emotion_recognition.ipynb Final Demonstration notebook
|   ├ 📜emotion_models.py
|   ├ 📜face_encodings.py
|   ├ 📜face_m1_emotion_m1tr-07-24.ipynb Uses face model 1 and emotion model 1 trained
|   ├ 📜face_m1_emotion_m2pt-07-24.ipynb Uses face model 1 and emotion model 2 pretrained
|   ├ 📜face_m1_emotion_m2tr-07-24.ipynb Uses face model 1 and emotion model 2 trained
|   ├ 📜face_m2_emotion_m2pt-07-30.ipynb Uses face model 2 and emotion model 2 pretrained
|   ├ 📜facerec_transfer_training.ipynb Implements transfer learning for face recognition
|   ├ 📜high-level-arch.png
|   ├ 📜inception_resnet_v1.py
|   ├ 📜system-design.png
├ 📂jetson  Contains the dockerfile, the docker build and run scripts for container on Jetson
|   ├ 📜Dockerfile
|   ├ 📜build.sh
```

```
|  └ 📜run.sh
├ 📂models  Contains the trained and pre-trained models for face recognition and emotion recognition
|  ├ 📜emotion_m1tr-0723_64e_.63.pth emotion model 1 - trained using fer2013 image dataset
|  ├ 📜emotion_m2pt.pt emotion model 2 - pre-trained using fer2013.csv dataset
|  └ 📜emotion_m2tr-0723_90e_.35.pth emotion model 2 - trained using fer2013 image dataset
|  ├ 📜face_recognition_transf.pt  face model - pre-trained and transfer learning model
|  └ 📜haarcascade_frontalface_default.xml
├ 📜.dockerignore
├ 📜.gitignore
└ 📜README.md
```

## 4.4 Instructions to run

A Docker image was generated for this project. It contains application code, libraries, tools, dependencies and other files needed for the application to run. It can be used on Jetpack 4.4 and Jetpack 4.5.

**Instructions for installing the image and running the container from the Jetson Terminal:**

1. cd ~
2. git clone https://github.com/sudhrity/mids-w251-project.git
3. cd mids-w251-project
4. cd jetson
5. chmod +x *.sh
6. ./build.sh [if installing for Jetpack 4.5 change version r32.5.0 inside Dockerfile]
7. ./run.sh
8. jupyter notebook --allow-root
9. Open the Final Demonstration notebook -
   **Face_detection_recognition_and_emotion_recognition.ipynb**
10. From Jupyter Notebook run : Kernel -> Restart and Run all

# 5. Challenges

The biggest challenge was acquiring high-quality labelled data. Our method for image collection and labelling was a slow process that required reviewing several previously recorded videos to capture faces.

There was a dramatic loss in performance between training on static images and evaluating live video. We suspect this is due to the The Illumination Problem and the Pose Problem (Source) .

The Fer2013 offered low resolution images compared to our custom labelled dataset.

Measuring model accuracy on live video was difficult due to

# 6. Results

We integrated Face detection pre-trained, Face recognition transfer and Emotion recognition models to detect faces in a video stream, recognize faces and recognize facial image emotion with fair accuracy. This will be demonstrated during the project presentation.

# 7. Conclusions and Next Steps

Using facial features and embeddings may offer more efficient and accurate inference on live facial recognition.

We would like to have more time to experiment with various transformations on the training data to overcome accuracy limitations.

We believe that if our emotion recognition dataset was more customized for the specific people in our dataset, (e.g. pictures of classmates happy or sad) the results would be more accurate.

# 8. Credits/References

timesler/facenet-pytorch: Pretrained Pytorch face detection (MTCNN) and facial recognition (InceptionResnet) models

## Face Detection

[1604.02878] Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks

## Face Recognition

[1710.08092] VGGFace2: A dataset for recognising faces across pose and age

## Emotion Recognition

Emotion Detection Dataset

Emotion Detection Using Pytorch. One day while scrolling through youtube… | by Jayesh Rohan Singh | The Startup

shangeth/Facial-Emotion-Recognition-PyTorch-ONNX

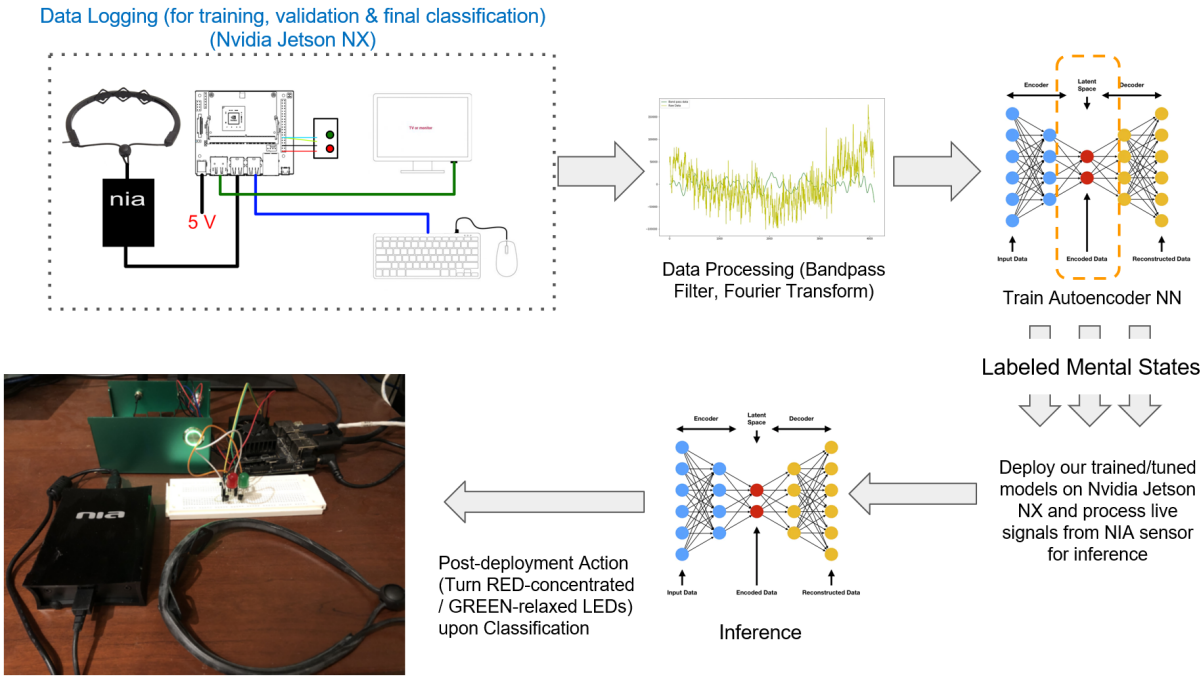
FER-2013 pytorch implementation

# Appendix - Additional project research work done earlier

Our initial project research work involved capturing brain waves from a human using a basic sensor (NIA from OCZ technology) and predict if signals are from concentration or relaxation brain activity.

The project implementation is based on information and code provided in the following article: https://www.hackster.io/dnhkng/aiot-artificial-intelligence-on-thoughts-f62249#overview

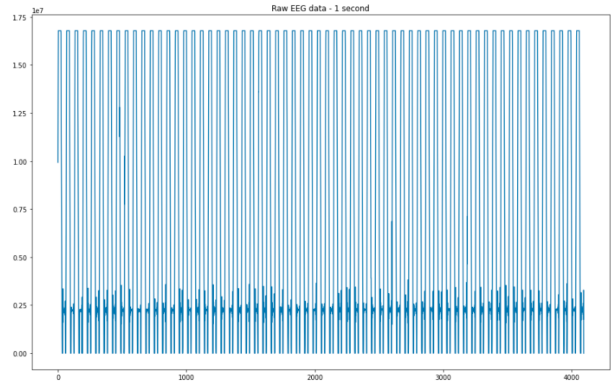
The architecture and data flow used for the project is shown below:



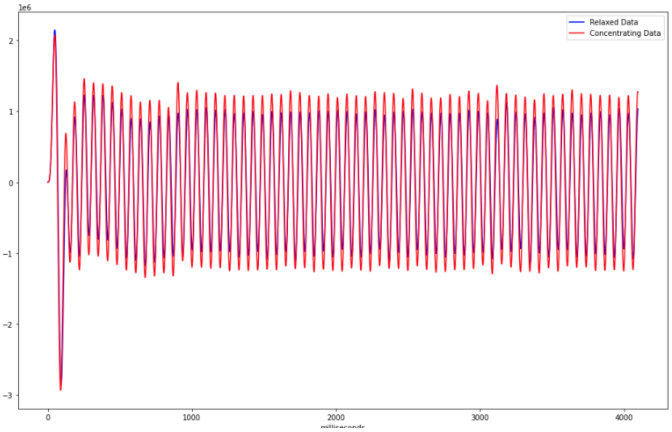

The prediction of mental states were random and as a result, the LEDs (Green and Red) were lighting up randomly. Due to the low precision sensor that we used for the brain waves capture project and the unpredictable nature of the prediction, we decided to stop the project and proceed with the face and emotion recognition project instead.

Below are some results from the project using the NIA sensor on a team member (Sudhrity).

### Raw and Bandpass data signals (training)




### Relaxed and concentrated data signals (training)




### Autoencoder model and final Classification (YES, NO)

```python
# Finally our Autoencoder!
class AE(nn.Module):
    def __init__(self, ae_width):
        super(AE, self).__init__()

        self.fc1 = nn.Linear(1024, 256)
        self.fc2 = nn.Linear(256, 64)
        self.fc3 = nn.Linear(64, ae_width)
        self.fc4 = nn.Linear(ae_width, 64)
        self.fc5 = nn.Linear(64, 256)
        self.fc6 = nn.Linear(256, 1024)

    def encode(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

    def decode(self, x):
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        return self.fc6(x)

    def forward(self, x):
        x = self.encode(x.view(-1, 1024))
        return self.decode(x)
```

```python
# Finally, run the Classifier, and control those LEDs!!!
while True:
    autoencoder_output = process_eeg_data().cpu().numpy()
    print(autoencoder_output)
    state = kmeans.predict(autoencoder_output.reshape(1, -1).astype('double'))
    print(state)
    if state == 0:
        GPIO.output(11, GPIO.HIGH)
        GPIO.output(12, GPIO.LOW)
        print("NO")
    else:
        GPIO.output(12, GPIO.HIGH)
        GPIO.output(11, GPIO.LOW)
        print("YES")
```

```
[-0.10988332  0.07077788  0.85087675 -0.5022227   0.17916398  0.18378733
  0.34307602 -0.01912618  0.26018128  0.36986232 -0.05480983 -0.15280901
 -0.4921522  -0.55692255  0.18741968  0.27328473 -0.40229365  0.5032748
 -0.247329   -0.23850417]
[0]
NO
[-1.7743190e-01  1.4926468e-01  2.9026654e-01 -5.3641051e-01
  5.0924726e-02 -3.7765542e-01  4.4150969e-01 -1.5379593e-01
 -1.0475672e-01  7.8771627e-01 -7.6003373e-05 -5.6774938e-01
 -2.1103024e-01 -4.4489580e-01  2.4640451e-01  3.3439419e-01
 -1.3731012e-01  6.1042911e-01  7.7573180e-02 -4.2160511e-01]
[1]
YES
[-0.04559164  0.03208309  0.5299741  -0.47120973  0.09344241  0.12973432
  0.18877426 -0.04388416  0.11133362  0.23438564 -0.06721763 -0.11658448
 -0.3662679  -0.39997804  0.11876173  0.31659228 -0.1891008   0.42692244
 -0.12527668 -0.073861  ]
[0]
```

Jetson NX GPIO 11 and 12 pins were connected to the Red & Green LEDs for prediction (YES (Concentrating) - RED, NO (Relaxing) - GREEN). Further investigation can be done with improved brain wave (EEG) sensors for improved and predictable results.