```
from google.colab import drive

drive.mount('//content/gdrive')
```

```
Mounted at //content/gdrive
```

```
cd '/content/gdrive/My Drive/'
```

```
/content/gdrive/My Drive
```

```
import pandas as pd

data=pd.read_csv('dataset.csv')
```

Loading data

```
data.tail()
```

	file_name_list	speakers	visual_features	
1331	Ses05M_script03_2_M029	M05	/features/visual_features/Session5/Ses05M_scri...	/featu
1332	Ses05M_script03_2_M039	M05	/features/visual_features/Session5/Ses05M_scri...	/featu
1333	Ses05M_script03_2_M041	M05	/features/visual_features/Session5/Ses05M_scri...	/featu
1334	Ses05M_script03_2_M042	M05	/features/visual_features/Session5/Ses05M_scri...	/featu
1335	Ses05M_script03_2_M043	M05	/features/visual_features/Session5/Ses05M_scri...	/featu

Class Imabalance

```
from collections import Counter

Counter(data['emotion_labels'])
```

```
Counter({0: 328, 1: 308, 2: 180, 3: 520})
```

```
import numpy as np
```

Pytorch Data loaders for all features reading the .npy files

```
import torch
```

```
class Dataset(torch.utils.data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, list_IDs, labels,feature):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs
        self.feature_type=feature

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

  def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]
        number=ID[4]
        val=ID.split("_")
        direc=""
        for i in range(len(val)-1):
          direc+=val[i]
          direc+="_"
        direc=direc[:-1]
        # Load data and get label-features
        if self.feature_type =="lexical_features":
          feature=np.load('features/'+self.feature_type+'/Session' + number+"/"+direc+"/"+ID
        else:
          feature=np.load('features/'+self.feature_type+'/Session' + number+"/"+direc+"/"+ID
          feature=feature.mean(axis=0)

        X =torch.tensor(feature)
        y = self.labels[ID]

        return X, y
```

Creating separate training and validation sets for 10 fold cross validation

```
def dataset_preparation(train,test):
  params = {'batch_size': 32,
          'shuffle': True,
          'num_workers': 1}
  params_test = {'batch_size': 32,
          'num_workers': 1}
  labels={}
  for i in train.iterrows():

    labels[i[1][0]]=i[1][5]

  training_set_f1 = Dataset(train['file_name_list'].values, labels,"visual_features")
```

```
  training_generator_f1 = torch.utils.data.DataLoader(training_set_f1, **params)
  training_set_f2 = Dataset(train['file_name_list'].values, labels,"acoustic_features")
  training_generator_f2 = torch.utils.data.DataLoader(training_set_f2, **params)
  training_set_f3 = Dataset(train['file_name_list'].values, labels,"lexical_features")
  training_generator_f3 = torch.utils.data.DataLoader(training_set_f3, **params)
  labels={}
  for i in test.iterrows():
    labels[i[1][0]]=i[1][5]
  testing_set_f1 = Dataset(test['file_name_list'].values, labels,"visual_features")
  testing_generator_f1 = torch.utils.data.DataLoader(testing_set_f1, **params_test)
  testing_set_f2 = Dataset(test['file_name_list'].values, labels,"acoustic_features")
  testing_generator_f2 = torch.utils.data.DataLoader(testing_set_f2, **params_test)
  testing_set_f3= Dataset(test['file_name_list'].values, labels,"lexical_features")
  testing_generator_f3 = torch.utils.data.DataLoader(testing_set_f3, **params_test)
  return training_generator_f1,training_generator_f2,training_generator_f3,testing_generator_
```

Taking into account the label distribution in each training set

```
counter_speaker_wise={}


def extract(data,speaker):
  val="speaker_"+speaker
  train=data[data["speakers"]!=speaker]
  test=data[data["speakers"] ==speaker]
  samples=Counter(train['emotion_labels'])
  counter_speaker_wise[speaker]=list(samples.values())
  training_generator_f1,training_generator_f2,training_generator_f3,testing_generator_f1,test
  return {"train_visual":training_generator_f1,"train_acoustic":training_generator_f2,"train_


cv10_fold_data={}
for i in data.speakers.unique():
  cv10_fold_data[i]=extract(data,i)
```

A sample data loader

```
for i in cv10_fold_data:
  print(i)
  print(cv10_fold_data[i]['test_lexical'])
  for j,y in cv10_fold_data[i]['test_lexical']:
    print(j,y)
    break
  break
```

```
F01
<torch.utils.data.dataloader.DataLoader object at 0x7f626d06abd0>
tensor([[-1.1109, -0.0768, -0.4929,  ...,  0.8613,  0.8013, -0.6062],
        [ 0.4657, -0.1908,  1.0678,  ..., -0.3106, -1.2877, -0.3745],
```

```
        [ 0.5843,  1.0325, -1.3516,  ...,  0.8121, -0.5970,  1.4111],
        ...,
        [ 0.8674, -0.5370,  0.7117,  ...,  0.7471, -0.9581, -0.7356],
        [ 1.4550,  1.1453,  0.1690,  ...,  0.2822,  0.4188,  1.3527],
        [ 1.2102, -0.2770,  1.1886,  ..., -0.9448,  0.2593,  0.9709]]) tensor([3, 1, 3,
        0, 0, 3, 1, 1, 3, 3, 1])
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

Visual Features

```
import torch.nn.functional as F
import torch
from functools import partial
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
```

Model

```
class Net(nn.Module):
    def __init__(self,l3=32,l2=256):
      super(Net, self).__init__()

      self.fc1 = nn.Linear(2048, 1024)
      self.fc2 = nn.Linear(1024, l2)
      self.fc3 = nn.Linear(l2, l3)
      self.fc4 = nn.Linear(l3, 4)

    # x represents our data
    def forward(self, x):
      # Pass data through fc1


      x = self.fc1(x)

      x = self.fc2(x)
      x = self.fc3(x)
      x = self.fc4(x)

      output = F.softmax(x, dim=1)
      return output
```

```
import torch.optim as optim
```

```
from sklearn.metrics import f1_score
import numpy as np
```

Module for validation and finding F1 score

```
def find_f1(loader,model):
  model.eval()  # eval mode (batchnorm uses moving mean/variance instead of mini-batch mean/v
  with torch.no_grad():
    correct = 0
    y_true=[]
    y_pred=[]
    for i, samples in enumerate(loader, 0):

        inputs, labels = samples
        inputs,labels = inputs.to(device),labels.cpu().detach().numpy()
        outputs = model(inputs.float())
        outputs=outputs.cpu().detach().numpy()


        y_pred.extend(np.argmax(outputs,axis=1))
        y_true.extend(labels)
    return f1_score(y_true, y_pred, average='micro')
```

Training visual model

```
def train(l1,l2,learning_rate,speaker,loader,weights):
  my_nn = Net(l1, l2)
  my_nn = my_nn.to(device)
  criterion = nn.CrossEntropyLoss(weight=weights)
  optimizer = optim.Adam(my_nn.parameters(), lr=learning_rate)
  running_loss = []

  for epoch in range(50):
      run_loss=0.0


      for i, samples in enumerate(loader, 0):

        inputs, labels = samples
        inputs,labels = inputs.to(device),labels.to(device)
        optimizer.zero_grad()
```

```
        outputs = my_nn(inputs.float())

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        run_loss+=loss.item()
      #print(loss.item())


  f1_scores = find_f1(cv10_fold_data[speaker]['test_visual'],my_nn)
  return f1_scores,my_nn


  print('Finished Training')
```

Hyper-parameter Tuning

```
best_model={}
best_f1={}
hyper_parameter_results={}
for l1 in [1024,512,256]:
  for l2 in [256,64,32]:
    if l1==512 and l2==256:
      continue
    if l1==256 and l2 ==256:
      continue
    for lr in [0.00001,0.0001]:
      hyper_parameter_results[tuple([l1,l2,lr])]=[]
      for k in cv10_fold_data:

        samples=counter_speaker_wise[k]
        max_value=max(samples)
        weights=[]
        for i in samples:
          weights.append(max_value/i)
        weights=torch.tensor(weights).to(device)
        loader=cv10_fold_data[k]['train_visual']
        value,model = train(l1,l2,lr,k,loader,weights)
        hyper_parameter_results[tuple([l1,l2,lr])].append([k,value])
print(hyper_parameter_results)
```

```
'M04', 0.3333333333333333], ['F05', 0.35185185185185186], ['M05', 0.33884297520661155]]}
```

Finding best results

```
final_result={}
best_f1=-float("inf")
best_parameter=None
for key in hyper_parameter_results:
  f1=0
  for speaker in hyper_parameter_results[key]:
    f1+=speaker[1]
  final_result[key]=f1/10
  if f1>best_f1:
    best_f1=f1
    best_parameter=key

print(final_result)
print(best_f1/10)
print(best_parameter)
```

```
{(1024, 256, 1e-05): 0.36033845779235857, (1024, 256, 0.0001): 0.33379745451357923, (102
0.3655608733757713
(512, 64, 1e-05)
```

Saving models

```
f1=0
for k in cv10_fold_data:
  loader=cv10_fold_data[k]['train_visual']
  value = train(512, 64, 1e-05,k,loader)
  f1+=value[0]
  name="visual_"+str(k)+".pth"
  torch.save(value[1].state_dict(), name)
```

Printing Confusion matrix

```
from sklearn.metrics import confusion_matrix
```

```
final_confusion_matrix_visual=[[0]*4 for _ in range(4)]
confusion_matrix_visual=[]
```

```
model = Net(512,64)
for k in cv10_fold_data:
  name="visual_"+str(k)+".pth"
  model.load_state_dict(torch.load(name))
  model.eval()
```

```
    with torch.no_grad():
      y_true=[]
      y_pred=[]
      loader=cv10_fold_data[k]['test_visual']
      for i, samples in enumerate(loader, 0):
        inputs, labels = samples
        inputs,labels = inputs,labels.cpu().detach().numpy()
        outputs = model(inputs.float())
        outputs=outputs.cpu().detach().numpy()
        y_pred.extend(np.argmax(outputs,axis=1))
        y_true.extend(labels)
      cm = confusion_matrix(y_true, y_pred)
      print(cm)
      confusion_matrix_visual.append(cm)
      final_confusion_matrix_visual+=cm

print(final_confusion_matrix_visual)
```

```
print(final_confusion_matrix_visual)
```

```
[[318 188  30 279]
 [155 300  42 269]
 [131 109  57 166]
 [366 322  91 603]]
```

```
import pickle

f = open("file.pkl","wb")
pickle.dump(hyper_parameter_results,f)
f.close()
```

Textual Features

Model

```
class EmoGRU(nn.Module):
    def __init__(self, embedding_dim, hidden_units, batch_sz, output_size):
        super(EmoGRU, self).__init__()
        self.batch_sz = batch_sz
        self.hidden_units = hidden_units
        self.embedding_dim = embedding_dim
        self.output_size = output_size

        # layers
        #self.embedding = nn.Embedding(self.vocab_size, self.embedding_dim)
        self.dropout = nn.Dropout(p=0.5)
        self.gru = nn.GRU(self.embedding_dim, self.hidden_units)
        self.fc = nn.Linear(self.hidden_units, self.output_size)
```

```
    def initialize_hidden_state(self,batch_sz):
        return torch.zeros((1, batch_sz, self.hidden_units))

    def forward(self, x):
        #x = self.embedding(x)
        x=x.view(1,-1,768)
        self.hidden = self.initialize_hidden_state(x.shape[1]).to(device)
        output, self.hidden = self.gru(x, self.hidden) # max_len X batch_size X hidden_units
        out = output[-1, :, :]
        out = self.dropout(out)
        out = self.fc(out)
        out = F.softmax(out, dim=1)
        return out
```

Training textual model

```
def train_lexical(units,learning_rate,speaker,loader,weights):
  embedding_dim=768

  BATCH_SIZE=32
  target_size=4

  model = EmoGRU( embedding_dim, units, BATCH_SIZE, target_size)
  criterion = nn.CrossEntropyLoss(weight=weights)
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)

  model = model.to(device)

  running_loss = []

  for epoch in range(50):
      run_loss=0.0


      for i, samples in enumerate(loader, 0):

        inputs, labels = samples
        inputs,labels = inputs.to(device),labels.to(device)
        optimizer.zero_grad()


        outputs = model(inputs.float())

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        run_loss+=loss.item()
```

```
  f1_scores = find_f1(cv10_fold_data[speaker]['test_lexical'],model)
  return f1_scores,model


  print('Finished Training')
```

Hyper-parameter tuning

```
hyper_parameter_results_textual={}
for l2 in [300, 100,64,32]:
  for lr in [0.000001,0.00001,0.0001,0.001]:
    hyper_parameter_results_textual[tuple([l2,lr])]=[]
    for k in cv10_fold_data:
       samples=counter_speaker_wise[k]
        max_value=max(samples)
        weights=[]
        for i in samples:
          weights.append(max_value/i)
       weights=torch.tensor(weights).to(device)
       loader=cv10_fold_data[k]['train_lexical']
       value = train_lexical(l2,lr,k,loader,weights)
       hyper_parameter_results_textual[tuple([l2,lr])].append([k,value[0]])

print(hyper_parameter_results_textual)
```

```
{(100, 1e-06): [['F01', 0.20134228187919462], ['M01', 0.3375], ['F02', 0.22881355932203
```

Finding best results

```
final_result={}
best_f1=-float("inf")
best_parameter=None
for key in hyper_parameter_results_textual:
  f1=0
  for speaker in hyper_parameter_results_textual[key]:
    f1+=speaker[1]
  final_result[key]=f1/10
  if f1>best_f1:
    best_f1=f1
    best_parameter=key

print(final_result)
print(best_f1/10)
print(best_parameter)
```

```
{(300, 1e-06): 0.40926304826095194, (300, 1e-05): 0.5760339107327832, (300, 0.0001): 0.6
0.6350955534644396
(300, 0.0001)
```

```
import pickle

f = open("file_textual.pkl","wb")
pickle.dump(hyper_parameter_results_textual,f)
f.close()
```

Saving the models

```
f1=0
for k in cv10_fold_data:
  loader=cv10_fold_data[k]['train_lexical']
  samples=counter_speaker_wise[k]
  max_value=max(samples)
  weights=[]
  for i in samples:
    weights.append(max_value/i)
  weights=torch.tensor(weights).to(device)
  value = train_lexical(300,0.0001,k,loader,weights)
  f1+=value[0]
  name="lexical_"+str(k)+".pth"
  torch.save(value[1].state_dict(), name)
```

Creating Confusion matrix

```
final_confusion_matrix_lexical=[[0]*4 for _ in range(4)]
confusion_matrix_lexical=[]
```

```
model = EmoGRU( 768, 300, 32, 4)
for k in cv10_fold_data:
  name="lexical_"+str(k)+".pth"
  model.load_state_dict(torch.load(name))
  model.eval()
  with torch.no_grad():
    y_true=[]
    y_pred=[]
    loader=cv10_fold_data[k]['test_lexical']
    for i, samples in enumerate(loader, 0):
      inputs, labels = samples
      inputs,labels = inputs,labels.cpu().detach().numpy()
      outputs = model(inputs.float())
```

```
            outputs=outputs.cpu().detach().numpy()
            y_pred.extend(np.argmax(outputs,axis=1))
            y_true.extend(labels)
        cm = confusion_matrix(y_true, y_pred)
        confusion_matrix_lexical.append(cm)
        final_confusion_matrix_lexical+=cm

print(final_confusion_matrix_lexical)
```

```
print(final_confusion_matrix_lexical)
```

```
     [[229  17   9  73]
      [ 24 192  23  69]
      [ 21  27  64  68]
      [ 65  72  33 350]]
```

Audio Features

Model

```
class AudioGRU(nn.Module):
    def __init__(self, embedding_dim, hidden_units, batch_sz, output_size):
        super(AudioGRU, self).__init__()
        self.batch_sz = batch_sz
        self.hidden_units = hidden_units
        self.embedding_dim = embedding_dim
        self.output_size = output_size

        # layers
        #self.embedding = nn.Embedding(self.vocab_size, self.embedding_dim)
        self.dropout = nn.Dropout(p=0.5)
        self.gru = nn.GRU(self.embedding_dim, self.hidden_units)
        self.fc = nn.Linear(self.hidden_units, self.output_size)

    def initialize_hidden_state(self,batch_sz):
        return torch.zeros((1, batch_sz, self.hidden_units))

    def forward(self, x):
        #x = self.embedding(x)
        x=x.view(1,-1,128)
        self.hidden = self.initialize_hidden_state(x.shape[1]).to(device)
        output, self.hidden = self.gru(x, self.hidden) # max_len X batch_size X hidden_units
        out = output[-1, :, :]
        out = self.dropout(out)
        out = self.fc(out)
        out = F.softmax(out, dim=1)
        return out
```

Training acoustic model

```
def train_audio(units,learning_rate,speaker,loader,weights):
  embedding_dim=128

  BATCH_SIZE=32
  target_size=4

  model = AudioGRU(embedding_dim, units, BATCH_SIZE, target_size)
  criterion = nn.CrossEntropyLoss(weight=weights)
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)

  model = model.to(device)

  running_loss = []

  for epoch in range(50):
      run_loss=0.0


      for i, samples in enumerate(loader, 0):

        inputs, labels = samples
        inputs,labels = inputs.to(device),labels.to(device)
        optimizer.zero_grad()


        outputs = model(inputs.float())

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        run_loss+=loss.item()




  f1_scores = find_f1(cv10_fold_data[speaker]['test_acoustic'],model)
  return f1_scores,model


  print('Finished Training')
```

Hyper-parameter tuning

```
hyper_parameter_results_audio={}
for l2 in [100,64,32]:
  for lr in [0.000001,0.00001,0.0001,0.001]:
```

```
    hyper_parameter_results_audio[tuple([l2,lr])]=[]
    for k in cv10_fold_data:
       samples=counter_speaker_wise[k]
        max_value=max(samples)
        weights=[]
        for i in samples:
          weights.append(max_value/i)
       loader=cv10_fold_data[k]['train_acoustic']
       value = train_audio(l2,lr,k,loader,weights)
       hyper_parameter_results_audio[tuple([l2,lr])].append([k,value[0]])
print(hyper_parameter_results_audio)
```

```
{(100, 1e-06): [['F01', 0.20134228187919462], ['M01', 0.16875], ['F02', 0.2372881355932
```

Finding best results

```
final_result={}
best_f1=-float("inf")
best_parameter=None
for key in hyper_parameter_results_audio:
  f1=0
  for speaker in hyper_parameter_results_audio[key]:
    f1+=speaker[1]
  final_result[key]=f1/10
  if f1>best_f1:
    best_f1=f1
    best_parameter=key

print(final_result)
print(best_f1/10)
print(best_parameter)
```

```
{(100, 1e-05): 0.4796765013736997, (100, 0.0001): 0.5049969358760285, (64, 1e-05): 0.43
0.5049969358760285
(100, 0.0001)
```

Saving models

```
f1=0
for k in cv10_fold_data:
  loader=cv10_fold_data[k]['train_acoustic']
  samples=counter_speaker_wise[k]
  max_value=max(samples)
  weights=[]
  for i in samples:
    weights.append(max_value/i)
```

```
  weights=torch.tensor(weights).to(device)
  value = train_audio(100,0.0001,k,loader,weights)
  f1+=value[0]
  name="acoustic_new_"+str(k)+".pth"
  torch.save(value[1].state_dict(), name)
```

Creating confusion matrix

```
final_confusion_matrix_acoustic=[[0]*4 for _ in range(4)]
confusion_matrix_acoustic=[]
```

```
model = AudioGRU( 128, 100, 32, 4)
for k in cv10_fold_data:
  name="acoustic_new_"+str(k)+".pth"
  model.load_state_dict(torch.load(name))
  model.eval()
  with torch.no_grad():
    y_true=[]
    y_pred=[]
    loader=cv10_fold_data[k]['test_acoustic']
    for i, samples in enumerate(loader, 0):
      inputs, labels = samples
      inputs,labels = inputs,labels.cpu().detach().numpy()
      outputs = model(inputs.float())
      outputs=outputs.cpu().detach().numpy()
      y_pred.extend(np.argmax(outputs,axis=1))
      y_true.extend(labels)
    cm = confusion_matrix(y_true, y_pred)
    confusion_matrix_acoustic.append(cm)
    final_confusion_matrix_acoustic+=cm
```

```
print(final_confusion_matrix_acoustic)
```

```
[[138  10   2 178]
 [  7 112  10 177]
 [ 21  26  43  92]
 [ 30  64   2 424]]
```

Early Fusion

Creating new data loader to hold features in a concatenated form

```
import torch
import numpy as np
class Dataset_Concat(torch.utils.data.Dataset):
```

```
    'Characterizes a dataset for PyTorch'
    def __init__(self, list_IDs, labels):
          'Initialization'
          self.labels = labels
          self.list_IDs = list_IDs


    def __len__(self):
          'Denotes the total number of samples'
          return len(self.list_IDs)

    def __getitem__(self, index):
          'Generates one sample of data'
          # Select sample
          ID = self.list_IDs[index]
          number=ID[4]
          val=ID.split("_")
          direc=""
          for i in range(len(val)-1):
            direc+=val[i]
            direc+="_"
          direc=direc[:-1]

          feature_lexical=np.load('features/lexical_features/Session' + number+"/"+direc+"/"+ID
          feature_acoustic=np.load('features/acoustic_features/Session' + number+"/"+direc+"/"+
          feature_acoustic=feature_acoustic.mean(axis=0)
          feature_visual=np.load('features/visual_features/Session' + number+"/"+direc+"/"+ID +
          feature_visual=feature_visual.mean(axis=0)
          feature=np.concatenate((feature_lexical, feature_acoustic), axis=0)
          feature=np.concatenate((feature, feature_visual), axis=0)
          X =torch.tensor(feature)
          y = self.labels[ID]

          return X, y


  def dataset_preparation_concat(train,test):
    params = {'batch_size': 128,
            'shuffle': True,
            'num_workers': 1}

    labels={}
    for i in train.iterrows():

      labels[i[1][0]]=i[1][5]

    training_set_f1 = Dataset_Concat(train['file_name_list'].values, labels)
    training_generator_f1 = torch.utils.data.DataLoader(training_set_f1, **params)

    labels={}
    for i in test.iterrows():
      labels[i[1][0]]=i[1][5]
```

```
  testing_set_f1 = Dataset_Concat(test['file_name_list'].values, labels)
  testing_generator_f1 = torch.utils.data.DataLoader(testing_set_f1, **params)

  return training_generator_f1, testing_generator_f1


def extract_concat(data,speaker):
  val="speaker_"+speaker
  train=data[data["speakers"] !=speaker]
  test=data[data["speakers"] ==speaker]
  training_generator, testing_generator = dataset_preparation_concat(train,test)
  return [training_generator,testing_generator]


cv10_fold_concat_data={}
for i in data.speakers.unique():
  cv10_fold_concat_data[i]=extract_concat(data,i)
```

Sample data loader

```
for i in cv10_fold_concat_data:

  for j,y in cv10_fold_concat_data[i][1]:
    print(j.shape,y)
    break
  break
```

```
torch.Size([128, 2944]) tensor([1, 3, 0, 3, 3, 1, 2, 0, 1, 1, 3, 2, 1, 0, 0, 0, 2, 0, 0,
        0, 3, 0, 2, 3, 2, 1, 2, 2, 3, 0, 0, 1, 2, 3, 0, 2, 0, 1, 0, 0, 0, 1, 0,
        3, 3, 3, 3, 3, 1, 3, 2, 3, 3, 3, 2, 3, 0, 3, 3, 2, 0, 0, 0, 3, 0, 1, 3,
        0, 2, 0, 3, 0, 0, 0, 0, 0, 3, 3, 0, 0, 2, 1, 0, 0, 3, 3, 3, 0, 0, 3, 2,
        3, 3, 1, 1, 0, 0, 1, 0, 1, 0, 1, 2, 2, 0, 3, 3, 0, 3, 3, 0, 3, 3, 0, 3,
        0, 2, 1, 1, 0, 2, 3, 2])
```

Model for Early Fusion

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc0 = nn.Linear(2944, 1024)
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 120)
        self.fc2 = nn.Linear(120, 84)
```

```
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 4)

    def forward(self, x):

        x = self.fc0(x)

        x=x.view(-1,1,32,32)

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

        x = self.pool(F.relu(self.conv2(x)))

        x = x.view(-1, 16 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Model Training

```
def train_multimodal(learning_rate,speaker,loader,weights):
  my_nn = Net()
  my_nn = my_nn.to(device)
  criterion = nn.CrossEntropyLoss(weigth=weights)
  optimizer = optim.Adam(my_nn.parameters(), lr=learning_rate)
  running_loss = []

  for epoch in range(50):
      run_loss=0.0


      for i, samples in enumerate(loader, 0):

        inputs, labels = samples
        inputs,labels = inputs.to(device),labels.to(device)
        optimizer.zero_grad()


        outputs = my_nn(inputs.float())

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        run_loss+=loss.item()
```

```
  f1_scores = find_f1(cv10_fold_concat_data[speaker][1],my_nn)
  return f1_scores,my_nn


  print('Finished Training')
```

Hyper-parameter tuning

```
hyper_parameter_results_multimodal={}
for lr in [1e-06, 1e-05,0.0001,0.001]:
  hyper_parameter_results_multimodal[lr]=[]
  for k in cv10_fold_concat_data:
    samples=counter_speaker_wise[k]
    max_value=max(samples)
    weights=[]
    for i in samples:
      weights.append(max_value/i)
    weights=torch.tensor(weights).to(device)
    loader=cv10_fold_concat_data[k][0]
    value = train_multimodal(lr,k,loader,weights)
    hyper_parameter_results_multimodal[lr].append([k,value[0]])
print(hyper_parameter_results_multimodal)
```

```
{1e-06: [['F01', 0.2953020134228188], ['M01', 0.1875], ['F02', 0.1440677966101695], ['M0
```

Finding the best

```
final_result={}
best_f1=-float("inf")
best_parameter=None
for key in hyper_parameter_results_multimodal:
  f1=0
  for speaker in hyper_parameter_results_multimodal[key]:
    f1+=speaker[1]
  final_result[key]=f1/10
  if f1>best_f1:
    best_f1=f1
    best_parameter=key

print(final_result)
print(best_f1/10)
print(best_parameter)
```

```
{1e-06: 0.3241467729898147, 1e-05: 0.43003193841352677, 0.0001: 0.6306670630712599, 0.00
0.6541677627481584
0.001
```

Saving Models

```
f1=0
for k in cv10_fold_concat_data:
  loader=cv10_fold_concat_data[k][0]
  samples=counter_speaker_wise[k]
  max_value=max(samples)
  weights=[]
  for i in samples:
    weights.append(max_value/i)
  weights=torch.tensor(weights).to(device)
  value = train_multimodal(0.001,k,loader,weights)
  f1+=value[0]
  name="multi_modal_concat"+str(k)+".pth"
  torch.save(value[1].state_dict(), name)
```

Creating confusion matrix

```
final_confusion_matrix_early_fusion=[[0]*4 for _ in range(4)]
confusion_matrix_early_fusion=[]
```

```
model = Net()
for k in cv10_fold_concat_data:
  name="multi_modal_concat"+str(k)+".pth"
  model.load_state_dict(torch.load(name))
  model.eval()
  with torch.no_grad():
    y_true=[]
    y_pred=[]
    loader=cv10_fold_concat_data[k][1]
    for i, samples in enumerate(loader, 0):
      inputs, labels = samples
      inputs,labels = inputs,labels.cpu().detach().numpy()
      outputs = model(inputs.float())
      outputs=outputs.cpu().detach().numpy()
      y_pred.extend(np.argmax(outputs,axis=1))
      y_true.extend(labels)
    cm = confusion_matrix(y_true, y_pred)
    print(cm)
    confusion_matrix_early_fusion.append(cm)
    final_confusion_matrix_early_fusion+=cm

print(final_confusion_matrix_early_fusion)
```

```
print(final_confusion_matrix_early_fusion)
```

```
[[228  18  14  68]
 [ 20 213  21  54]
 [ 22  22  71  65]
 [ 60  67  42 351]]
```

Late Fusion

```
final_confusion_matrix_late_fusion=[[0]*4 for _ in range(4)]
confusion_matrix_late_fusion=[]
f1_scores_late_fusion=[]
```

Loading all the best models and validating the output and creating confusion matrix

Majority Vote: by adding output probbalities and them using these to find the label

```
model_a=EmoGRU( 768, 300, 32, 4)
model_b=AudioGRU( 128, 100, 32, 4)
model_c=Net(512,64)
for k in cv10_fold_data:
  a,b,c =cv10_fold_data[k]['test_lexical'],cv10_fold_data[k]['test_acoustic'],cv10_fold_data[
  name="lexical_"+str(k)+".pth"
  model_a.load_state_dict(torch.load(name))
  model_a.eval()
  name="acoustic_"+str(k)+".pth"
  model_b.load_state_dict(torch.load(name))
  model_b.eval()
  name="visual_"+str(k)+".pth"
  model_c.load_state_dict(torch.load(name))
  model_c.eval()
  with torch.no_grad():
    y_true=[]
    y_pred=[]
    for i,j,k in zip( enumerate(a, 0), enumerate(b, 0), enumerate(c, 0)):

        inputs, labels = i[1][0],i[1][1]
        inputs,labels = inputs,labels.cpu().detach().numpy()
        outputs_a = model_a(inputs.float())
        outputs_a=outputs_a.cpu().detach().numpy()

        inputs, labels = j[1][0],j[1][1]
        inputs,labels = inputs,labels.cpu().detach().numpy()
        outputs_b = model_b(inputs.float())
        outputs_b=outputs_b.cpu().detach().numpy()

        inputs, labels = k[1][0],k[1][1]
        inputs,labels = inputs,labels.cpu().detach().numpy()
        outputs_c = model_c(inputs.float())
```

```
        outputs_c=outputs_c.cpu().detach().numpy()

        outputs=outputs_a+outputs_b+outputs_c


        y_pred.extend(np.argmax(outputs,axis=1))
        y_true.extend(labels)

  f1 = f1_score(y_true, y_pred, average='micro')
  f1_scores_late_fusion.append(f1)
  cm = confusion_matrix(y_true, y_pred)
  confusion_matrix_late_fusion.append(cm)
  final_confusion_matrix_late_fusion+=cm
```

```
print(final_confusion_matrix_late_fusion)
```

```
[[240  10   0  78]
 [ 15 205   3  85]
 [ 30  24  29  97]
 [ 61  76   3 380]]
```

```
print(sum(f1_scores_late_fusion)/10)
```

```
0.6395977983715335
```

0s completed at 10:04 PM