Speech Emotion Recognition

Hugo Alves

Juan Nunes

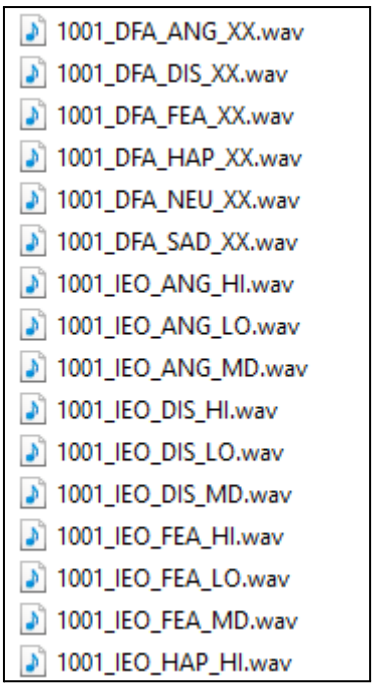
1. Contextualização

Esse projeto foi desenvolvido a partir do artigo de Konstantinos Mountzouris, et. al., intitulado, em inglês, Speech Emotion Recognition Using Convolutional Neural Networks with Attention Mechanism (DOI: https://doi.org/10.3390/electronics12204376). Esse artigo aborda 6 modelos de Deep Learning aplicados às bases Surrey Audio-Visual Expressed Emotion (SAVEE) e Rayerson Audio-Visual Database (RAVDESS), com o objetivo de solucionar problemas de reconhecimento de emoções através da fala e também de comparar a eficácia das redes neurais com o mecanismo de atenção. Esses modelos são: Deep Belief Network (DBN); um Deep Neural Network simples (SDNN); uma rede LSTM; LSTM com mecanismo de Attention; um Convolutional Neural Network (CNN); e um CNN com mecanismo de Attention.

Como o autor do artigo não disponibiliza códigos fontes, para o nosso projeto, decidimos implementar 5 dos 6 modelos anteriormente citados, sendo o Deep Belief Network (DBN) o modelo não abordado. Também como base de dados, decidimos utilizar uma base diferente. Ao invés de utilizar a SAVEE e a RAVDESS, utilizamos a CREMA-D, uma base de dados pública contendo arquivos .wav anotados, para treinar e testar nossos modelos.

2. Abordagem

O primeiro passo da nossa abordagem foi entender e processar a base de dados para que pudéssemos aplicar os modelos de rede neural. A base CREMA-D possui anotação dos dados de áudio no seu título, descrevendo o sentimento transmitido pela gravação de atores. Essa base possui cerca 7442 áudios gravados por 91 atores, em inglês, sendo 48 homens e 43 mulheres de diferentes etnias, para que seus sotaques possam ser levados em conta na hora de treinar modelos. Abaixo segue figura do formato dos dados:



Para que pudéssemos seguir com a proposta, precisamos tratar esses arquivos de áudio para conseguir extrair suas labels. Ao entender as especificações da base de dados, desenvolvemos o seguinte código para resolver esse passo:

```
paths=[]
labels=[]
for filename in os.listdir('./AudioWAV'):

    paths.append('./AudioWAV/' + filename)
    file = filename.split('.')[0]

    label = file.split('_')[2]
    if label == 'ANG':
        labels.append('angry.wav')
    elif label == 'DIS':
        labels.append('disgust.wav')
    elif label == 'FEA':
        labels.append('fear.wav')
    elif label == 'HAP':
        labels.append('happy.wav')
    elif label == 'NEU':
        labels.append('neutral.wav')
    elif label == 'SAD':
        labels.append('sad.wav')
```

```
df_cremad = pd.DataFrame({'speech':paths,'label':labels})
```

Dessa forma, alteramos as especificações no título dos arquivos para somente o sentimento que ele transmite e, na sequência, criando um DataFrame para utilizarmos no restante do projeto.

Para extrair as características da voz do ator através do arquivo de áudio, utilizamos a Mel Frequency Cepstral Coefficients (MFCCs) com a ajuda da biblioteca Librosa, assim como sugerido pelo autor do artigo. Isso irá gerar um vetor de características em coeficientes para que possamos treinar e testar as redes posteriormente.

```
def MFCC(filename):
    y, sr = librosa.load(filename,duration=3,offset=0.5)
    return np.mean(librosa.feature.mfcc(y=y,sr=sr,n_mfcc=40).T,axis=0)

mfcc_cremad = df_cremad['speech'].apply(lambda x:MFCC(x))
```

```
0       [-324.20966, 128.3947, -19.360773, 45.353886, ...
1       [-364.7359, 135.75037, -26.445019, 55.168713, ...
2       [-314.81885, 115.73832, -14.47054, 34.524033, ...
3       [-335.19382, 123.35851, -19.410343, 43.848305,...
4       [-364.19574, 126.68473, -6.776453, 40.001396, ...
                              ...
7437    [-445.5911, 134.99956, 9.402614, 52.412285, -1...
7438    [-449.06717, 126.30141, 9.401833, 60.209625, -...
7439    [-373.89056, 125.2806, -2.8478541, 54.36043, -...
7440    [-404.8829, 124.196915, 13.040497, 43.833916, ...
7441    [-454.50098, 130.466, 9.12131, 55.879097, -15....
Name: speech, Length: 7442, dtype: object
```

Após carregada nossa base de dados, realizamos um One Hot Encoding das labels e separamos os dados para, por fim, partir para as redes neurais.

```
ohe=OneHotEncoder()
y = ohe.fit_transform(df_cremad[['label']] )
y = y.toarray()
```

```
X =[x for x in mfcc_cremad]
X =np.array(X)
X.shape
```

```
X =np.expand_dims(X,-1)
```

Para os modelos de redes neurais, utilizamos implementações bases que estão presentes em suas documentações, realizando alterações de acordo com nossos dados em alguns parâmetros de entrada. Em alguns casos, como LSTM, CNN e o mecanismo de Attention, precisamos realizar algumas alterações em alguns passos da rede neural, já que a forma como a biblioteca Librosa devolve os dados podia não ser compatível com a arquitetura do modelo. Técnicas de Dropout e Batch Normalization também foram utilizadas para evitar Overfitting e aumentar sua velocidade de treinamento. A seguir estão os códigos das 5 redes neurais utilizadas.

```
class SimpleDNN(nn.Module):
    def __init__(self, input_size, num_classes):
        super(SimpleDNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, 128)
        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 64)
        self.dropout2 = nn.Dropout(0.5)
        self.fc3 = nn.Linear(64, num_classes)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

```
class LSTMModel(nn.Module):
          def   __init__(self,   input_size,   hidden_size,   num_layers,
num_classes):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
                    h0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)
                    c0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)

        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out
```

```
class LSTMAttention(nn.Module):
          def __init__(self, input_size, hidden_size, num_layers,
num_classes):
        super(LSTMAttention, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
            self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.attention = nn.Linear(hidden_size, 1)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
                    h0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)
                    c0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)

        out, _ = self.lstm(x, (h0, c0))
        attn_weights = torch.softmax(self.attention(out), dim=1)
        out = torch.sum(attn_weights * out, dim=1)
        out = self.fc(out)
        return out
```

```
class CNNModel(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(CNNModel, self).__init__()
            self.conv1 = nn.Conv1d(input_channels, 64, kernel_size=3,
padding=1)
```

```
        self.conv2 = nn.Conv1d(64, 32, kernel_size=3, padding=1)
        self.fc_input_size = 32 * 1 * 1
        self.fc1 = nn.Linear(self.fc_input_size, 128)
        self.fc2 = nn.Linear(128, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.dropout(x)
        x = x.view(-1, self.fc_input_size)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
class CNNAttention(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(CNNAttention, self).__init__()
            self.conv1 = nn.Conv1d(input_channels, 64, kernel_size=3,
padding=1)
        self.conv2 = nn.Conv1d(64, 32, kernel_size=3, padding=1)
        self.attention = nn.Linear(32, 1)
        self.fc = nn.Linear(32, num_classes)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
            attn_weights = self.softmax(self.attention(x.permute(0, 2,
1))).squeeze(-1)
        attn_weights = attn_weights.unsqueeze(-1)
        x = torch.sum(attn_weights * x, dim=2)
        x = self.fc(x)
        return x
```

3. Resultados

Após criar nossas redes neurais, realizamos o seu treinamento e teste. Utilizamos a Cross Entropy Loss como critério nas redes. Realizamos diferentes treinamentos e testes,

alterando o learning rate, a quantidade de épocas e o optimizer, utilizando o Adam e o AdaGrad. os códigos estão disponíveis em 8 notebooks no GitHub: https://github.com/Hugo-rAlves/SER-RedesNeurais.

Teste 1:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,001	Adam	100	0.3966	0.3962
LSTM	0,001	Adam	100	0.5238	0.4318
LSTM+ATN	0,001	Adam	100	0.5218	0.4278
CNN	0,001	Adam	100	0.4749	0.3801
CNN+ATN	0,001	Adam	100	0.5799	0.4439

Teste 2:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,001	Adam	200	0.4006	0.4291
LSTM	0,001	Adam	200	0.5908	0.4305
LSTM+ATN	0,001	Adam	200	0.5933	0.4338
CNN	0,001	Adam	200	0.4997	0.3875
CNN+ATN	0,001	Adam	200	0.6321	0.4090

Teste 3:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,001	Adam	500	0.4366	0.4332
LSTM	0,001	Adam	500	0.6751	0.4043
LSTM+ATN	0,001	Adam	500	0.6887	0.4063
CNN	0,001	Adam	500	0.5216	0.4117
CNN+ATN	0,001	Adam	500	0.6518	0.4144

Teste 4:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,001	AdaGrad	500	0.2115	0.2639
LSTM	0,001	AdaGrad	500	0.4386	0.3835
LSTM+ATN	0,001	AdaGrad	500	0.4413	0.3915
CNN	0,001	AdaGrad	500	0.3806	0.3714
CNN+ATN	0,001	AdaGrad	500	0.4258	0.3969

Teste 5:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,0001	Adam	500	0.4596	0.4486
LSTM	0,0001	Adam	500	0.6785	0.4312
LSTM+ATN	0,0001	Adam	500	0.6558	0.4439
CNN	0,0001	Adam	500	0.4977	0.3976
CNN+ATN	0,0001	Adam	500	0.5512	0.4520

Teste 6:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,00001	Adam	500	0.3514	0.3526
LSTM	0,00001	Adam	500	0.4509	0.4184
LSTM+ATN	0,00001	Adam	500	0.4364	0.4150
CNN	0,00001	Adam	500	0.3880	0.4150
CNN+ATN	0,00001	Adam	500	0.4295	0.4150

Teste 7:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,0001	Adam	1000	0.4769	0.4533
LSTM	0,0001	Adam	1000	0.7715	0.4063
LSTM+ATN	0,0001	Adam	1000	0.7564	0.4204

CNN	0,0001	Adam	1000	0.5527	0.3606
CNN+ATN	0,0001	Adam	1000	0.6014	0.4379

Teste 8:

Network	LR	Optimizer	Num. Epochs	Acc. Train	Acc. Test
SDDN	0,00001	Adam	1000	0.3758	0.3902
LSTM	0,00001	Adam	1000	0.5018	0.4433
LSTM+ATN	0,00001	Adam	1000	0.4959	0.4473
CNN	0,00001	Adam	1000	0.4188	0.4372
CNN+ATN	0,00001	Adam	1000	0.4636	0.4412