University of Tehran
College of Engineering
Department of Mechanical Engineering

Title:

4th Homework (Recurrent Networks)

Course:

Artificial Intelligence

Author:

Hatef Mohammadi Alashti

810601123

Course Instructor:

Dr. Shariat Panahi

June 2024

University of Tehran
College of Engineering
Department of Mechanical Engineering

Introduction

In this project, we are asked to train two models (CNN, and CNN-LSTM) to detect the emotion of audio files.

A. Data gathering

In this section, first, we downloaded a zip file that contains 535 audio files spanning from 2 to 3 seconds. To achieve this, we used the code below, as the question suggested.

```python
import os
import urllib.request
import zipfile

# Defining the data folder
data_folder = os.path.join(os.path.expanduser("~"), "Emo-DB")

# Downloading
if not os.path.exists(data_folder):
    url = "http://emodb.bilderbar.info/download/download.zip"
    zip_path = os.path.join(data_folder, "download.zip")
    os.makedirs(data_folder, exist_ok=True)
    print("Downloading Emo-DB (40.5 MB) ...")
    urllib.request.urlretrieve(url, zip_path)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(data_folder)
```

Since we used the Google Collaboratory environment, we mounted Google Drive to, and then defined folder paths to save data.

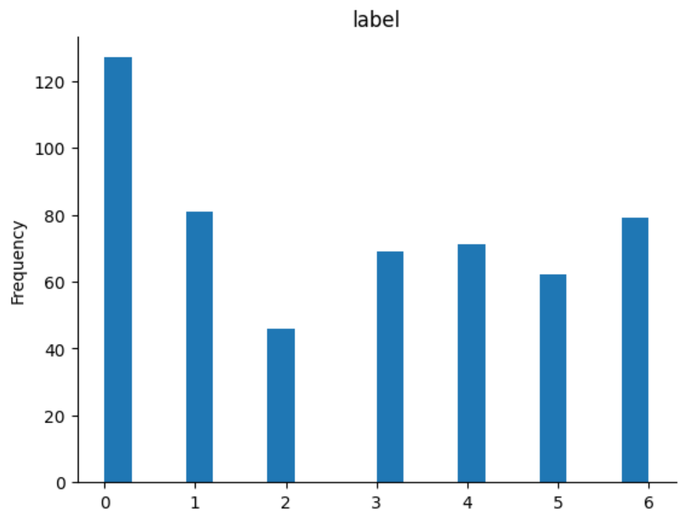
```python
from google.colab import drive
drive.mount('/content/drive')
```

```python
# Defining paths
wav_folder = os.path.join(data_folder, "wav")
save_folder = "/content/drive/My Drive/AI_HW#4"
```

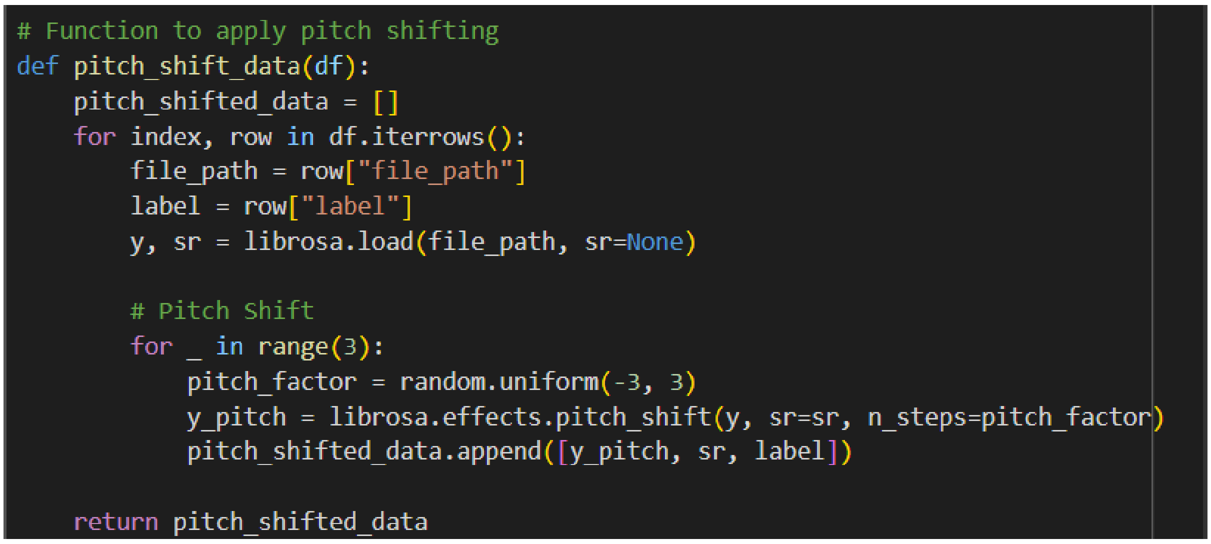
In the next step, we mapped the labels. Each audio file had a name whose second to last letter is the label and was converted to numerical values as the questioned asked.

Letters	Emotion Labels	Numerical Values
W	Anger	0
L	Boredom	1
E	Disgust	2
A	Anxiety/Fear	3
F	Happiness	4
T	Sadness	5
N	Neutral	6

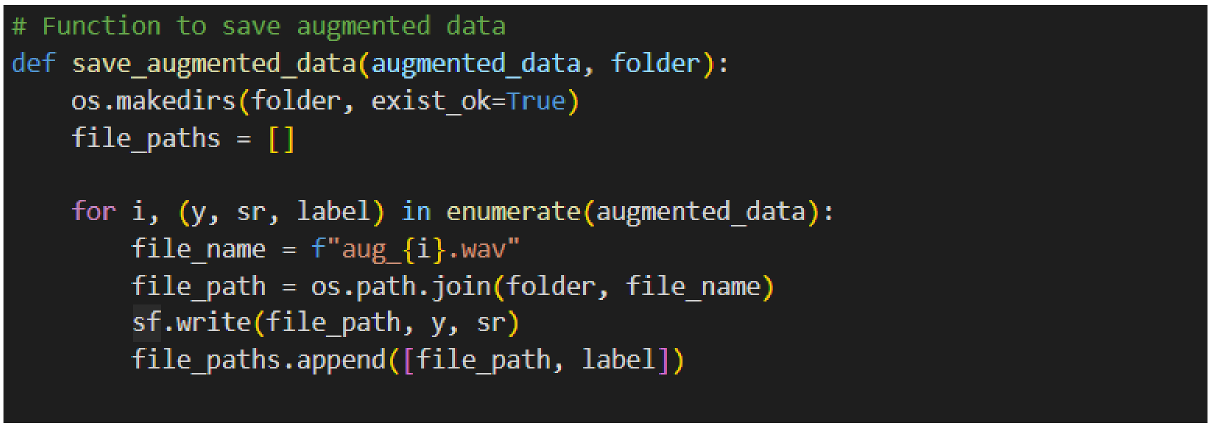
Here, is the preview of the distribution of the audio files we had.



In the next stage, we augmented the data set we had. We used pitch shift and time stretch. We increased the number of audio files by changing each audio file six times (three times with each augmentation method). Hence, at the end of this section, our data frame was 7 times of its previous size (3,745 audio files).

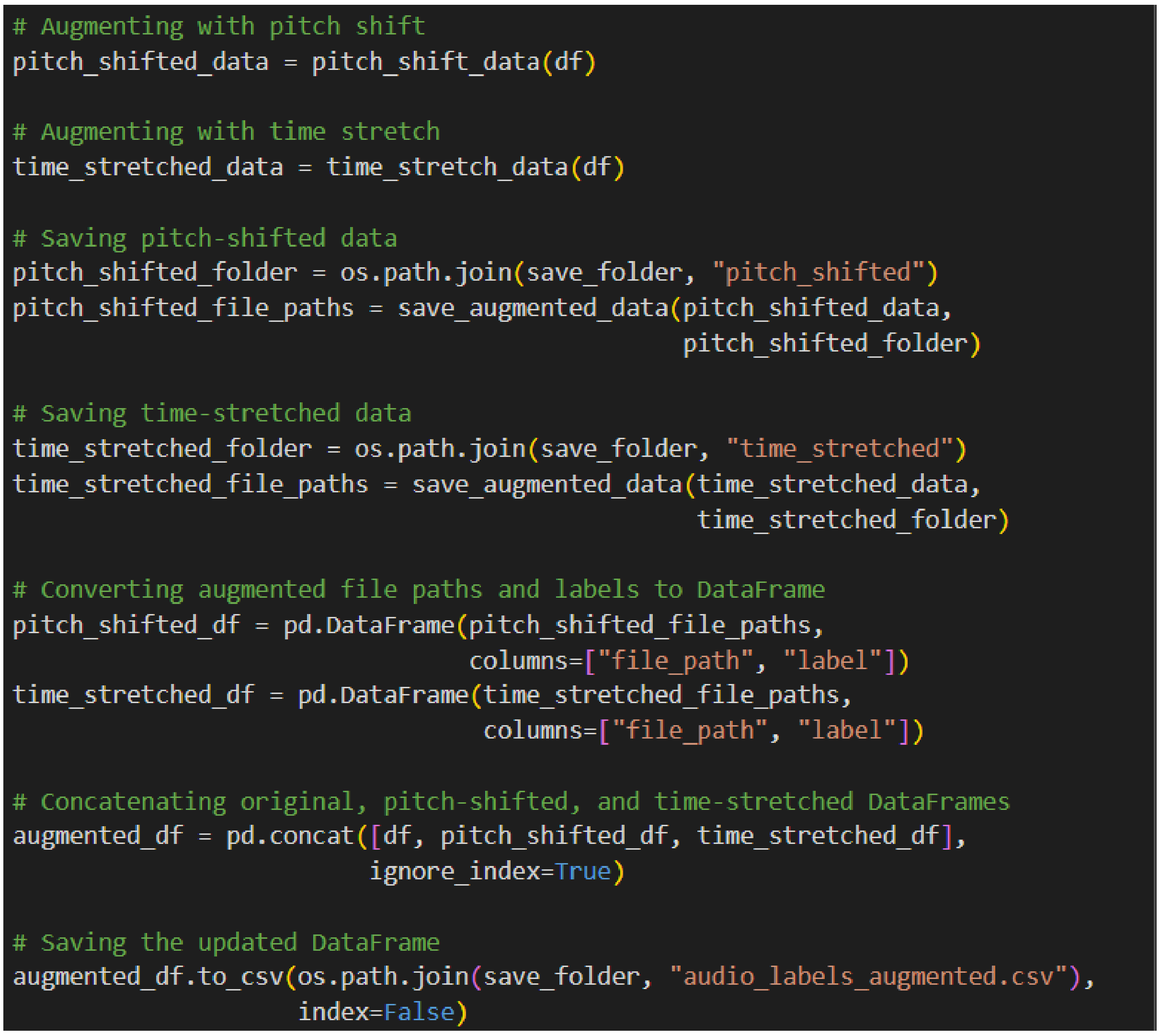


```python
# Function to apply pitch shifting
def pitch_shift_data(df):
    pitch_shifted_data = []
    for index, row in df.iterrows():
        file_path = row["file_path"]
        label = row["label"]
        y, sr = librosa.load(file_path, sr=None)

        # Pitch Shift
        for _ in range(3):
            pitch_factor = random.uniform(-3, 3)
            y_pitch = librosa.effects.pitch_shift(y, sr=sr, n_steps=pitch_factor)
            pitch_shifted_data.append([y_pitch, sr, label])

    return pitch_shifted_data
```



```python
# Function to save augmented data
def save_augmented_data(augmented_data, folder):
    os.makedirs(folder, exist_ok=True)
    file_paths = []

    for i, (y, sr, label) in enumerate(augmented_data):
        file_name = f"aug_{i}.wav"
        file_path = os.path.join(folder, file_name)
        sf.write(file_path, y, sr)
        file_paths.append([file_path, label])
```

Then, we augmented our data frame with the functions defined above. Next, we saved the audio files, and respective data frames.



```python
# Augmenting with pitch shift
pitch_shifted_data = pitch_shift_data(df)

# Augmenting with time stretch
time_stretched_data = time_stretch_data(df)

# Saving pitch-shifted data
pitch_shifted_folder = os.path.join(save_folder, "pitch_shifted")
pitch_shifted_file_paths = save_augmented_data(pitch_shifted_data,
                                               pitch_shifted_folder)

# Saving time-stretched data
time_stretched_folder = os.path.join(save_folder, "time_stretched")
time_stretched_file_paths = save_augmented_data(time_stretched_data,
                                                time_stretched_folder)

# Converting augmented file paths and labels to DataFrame
pitch_shifted_df = pd.DataFrame(pitch_shifted_file_paths,
                                columns=["file_path", "label"])
time_stretched_df = pd.DataFrame(time_stretched_file_paths,
                                 columns=["file_path", "label"])

# Concatenating original, pitch-shifted, and time-stretched DataFrames
augmented_df = pd.concat([df, pitch_shifted_df, time_stretched_df],
                         ignore_index=True)

# Saving the updated DataFrame
augmented_df.to_csv(os.path.join(save_folder, "audio_labels_augmented.csv"),
                    index=False)
```

To complete this section, we split the data frame into 70% train, 15% test, and 15% validation sets.

```
from sklearn.model_selection import train_test_split

# Splitting data sets:
train_df, temp_df = train_test_split(augmented_df, test_size=0.3, random_state=42)
test_df, val_df = train_test_split(temp_df, test_size=0.5, random_state=42)

print(f"Length of Train Set: {len(train_df)}")
print(f"Length of Validation Set: {len(val_df)}")
print(f"Length of Test Set: {len(test_df)}")

# Saving the data sets
train_df.to_csv(os.path.join(save_folder, "train.csv"), index=False)
test_df.to_csv(os.path.join(save_folder, "test.csv"), index=False)
val_df.to_csv(os.path.join(save_folder, "val.csv"), index=False)
```

Below, is the number of rows in each data set.

```
Length of Train Set: 2621
Length of Validation Set: 562
Length of Test Set: 562
```

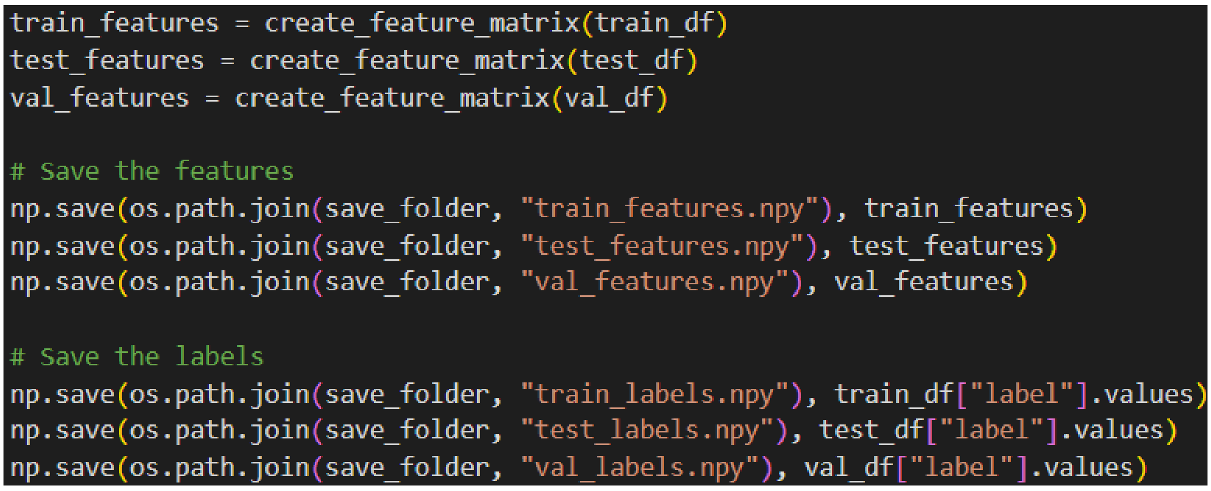
B. Data Preparation

In this section, we are asked to make data frames related to the characteristics of the audio files (in the frequency domain). First, we reduced the noise of each audio file. Then, we used the Discrete Fourier Transform (DTF). Next, we extracted the features we needed (energy, and 13 MFCCs). Lastly, we normalized the features and ensured that the length of all features are 150.

```python
def process_audio(file_path):
    y, sr = librosa.load(file_path, sr=None)

    # Convert audio to digital with 20ms spans
    y_resampled = librosa.resample(y, orig_sr=sr, target_sr=16000)

    # Reduce noise
    y_denoised = nr.reduce_noise(y=y_resampled, sr=16000)

    # DFT
    y_dft = np.fft.fft(y_denoised)

    # Extracting MFCC features
    mfccs = librosa.feature.mfcc(y=y_denoised, sr=16000, n_mfcc=12)
    energy = np.sum(y_denoised**2) / len(y_denoised)

    features = np.vstack([mfccs, np.full((1, mfccs.shape[1]), energy)])

    # Normalizing the features
    features = librosa.util.normalize(features, axis=1)

    # Leveling Lengths to 150
    if features.shape[1] < 150:
        features = np.pad(features, ((0, 0), (0, 150 - features.shape[1])),
                                  mode='constant')
    else:
        features = features[:, :150]

    return features
```

```python
def create_feature_matrix(df):
    feature_list = []
    for index, row in df.iterrows():
        features = process_audio(row["file_path"])
        feature_list.append(features)

    return np.array(feature_list)
```

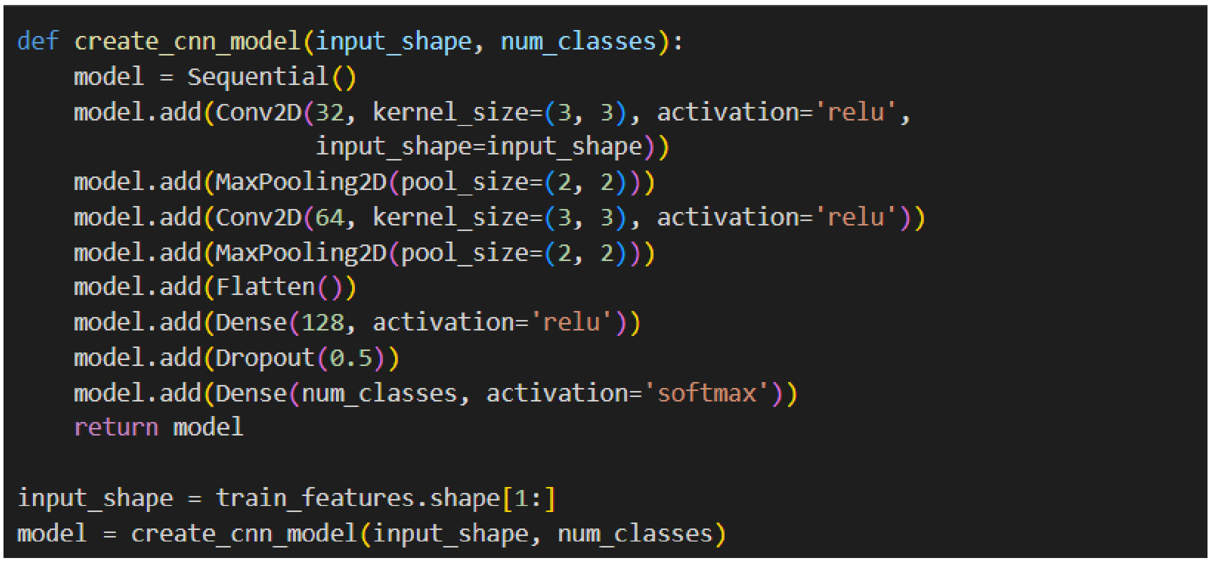
Lastly, we did this processing over all the three data sets. And stored them in Drive.



```
train_features = create_feature_matrix(train_df)
test_features = create_feature_matrix(test_df)
val_features = create_feature_matrix(val_df)

# Save the features
np.save(os.path.join(save_folder, "train_features.npy"), train_features)
np.save(os.path.join(save_folder, "test_features.npy"), test_features)
np.save(os.path.join(save_folder, "val_features.npy"), val_features)

# Save the labels
np.save(os.path.join(save_folder, "train_labels.npy"), train_df["label"].values)
np.save(os.path.join(save_folder, "test_labels.npy"), test_df["label"].values)
np.save(os.path.join(save_folder, "val_labels.npy"), val_df["label"].values)
```

C. CNN Model

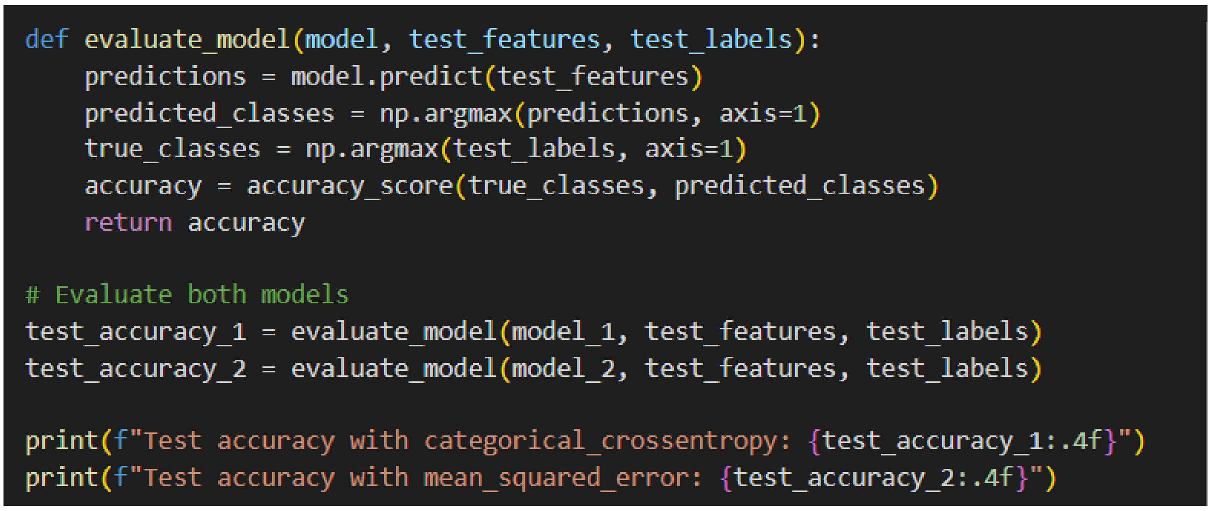
Then, we trained a CNN model. The parameters and the structure of the model is given below. As been asked, except for the output layer whose activation function is softmax, other layers are relu.



```python
def create_cnn_model(input_shape, num_classes):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    return model

input_shape = train_features.shape[1:]
model = create_cnn_model(input_shape, num_classes)
```

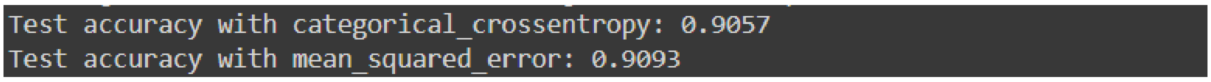
Then, we compiled the model with two different cost functions: Mean Squared Error, and Categorical CrossEntropy. To avoid over fitting, we used EarlyStopping function with the patience of 5, monitoring val_loss.

After compiling for 50 epochs, which never have been reached, due to early stopping, we evaluated the two models.



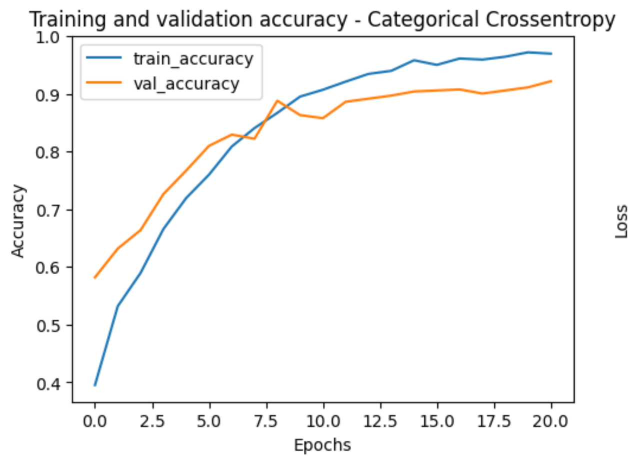
```python
def evaluate_model(model, test_features, test_labels):
    predictions = model.predict(test_features)
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(test_labels, axis=1)
    accuracy = accuracy_score(true_classes, predicted_classes)
    return accuracy

# Evaluate both models
test_accuracy_1 = evaluate_model(model_1, test_features, test_labels)
test_accuracy_2 = evaluate_model(model_2, test_features, test_labels)

print(f"Test accuracy with categorical_crossentropy: {test_accuracy_1:.4f}")
print(f"Test accuracy with mean_squared_error: {test_accuracy_2:.4f}")
```

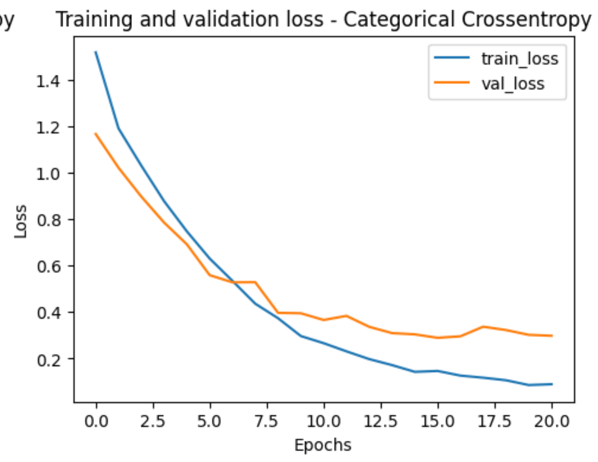
The performance of both on test data were fairly similar, and acceptable.

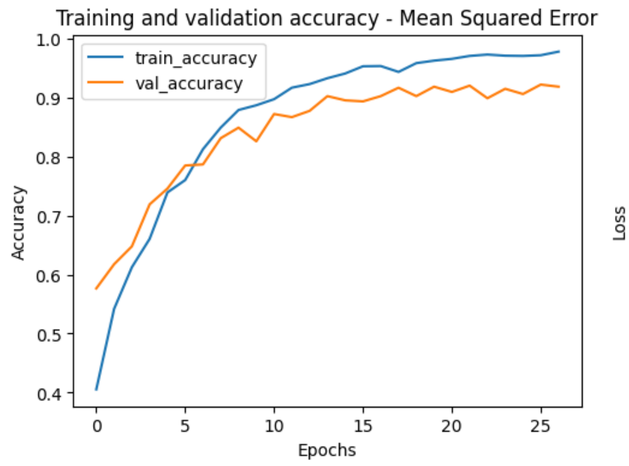


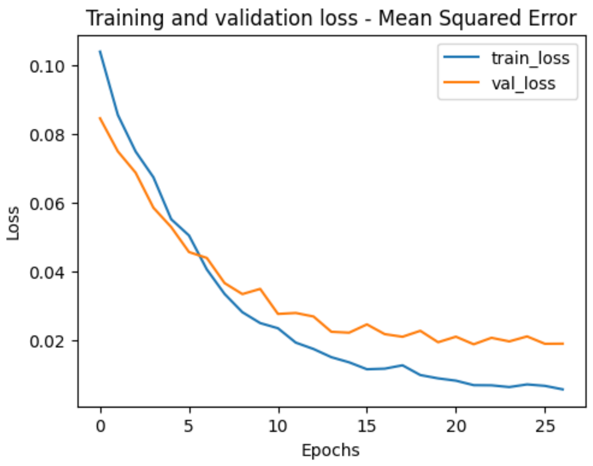
```
Test accuracy with categorical_crossentropy: 0.9057
Test accuracy with mean_squared_error: 0.9093
```

The trend of accuracies and losses are shown below, for both cost functions.

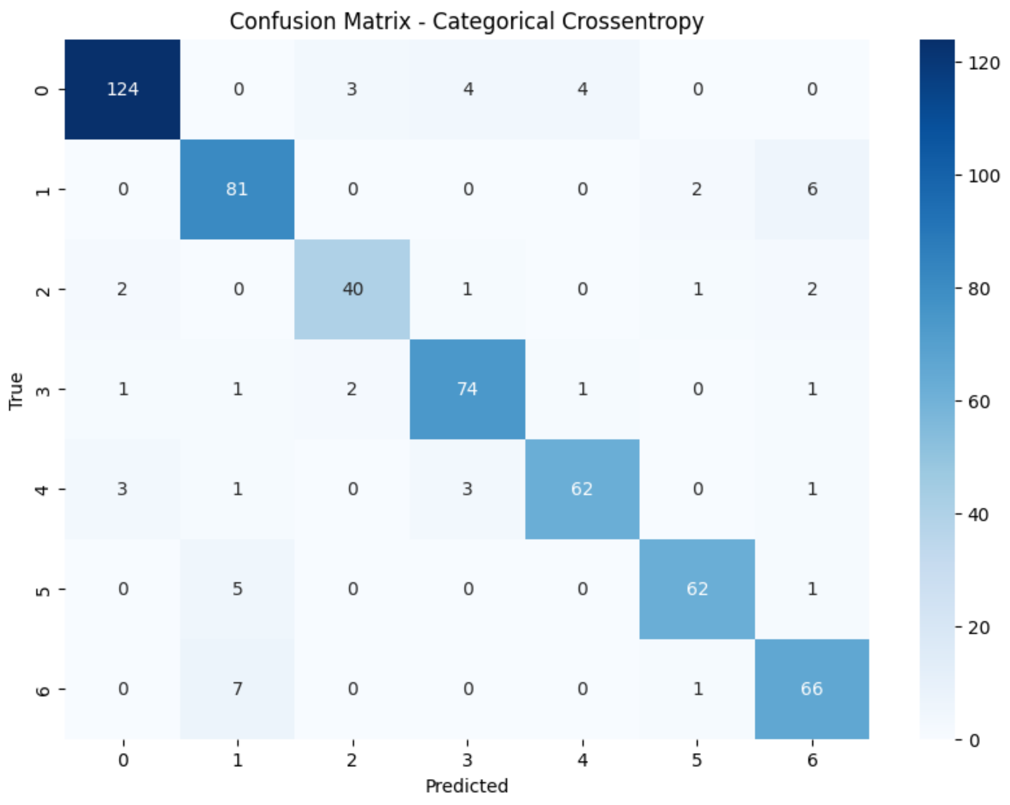


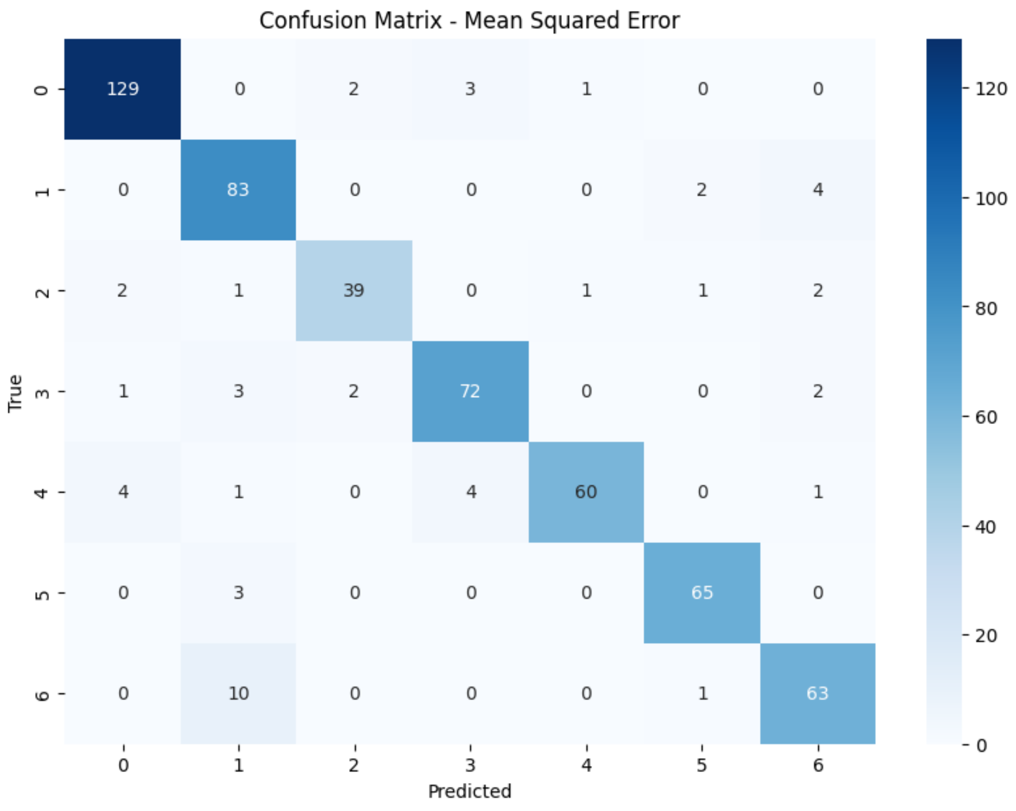






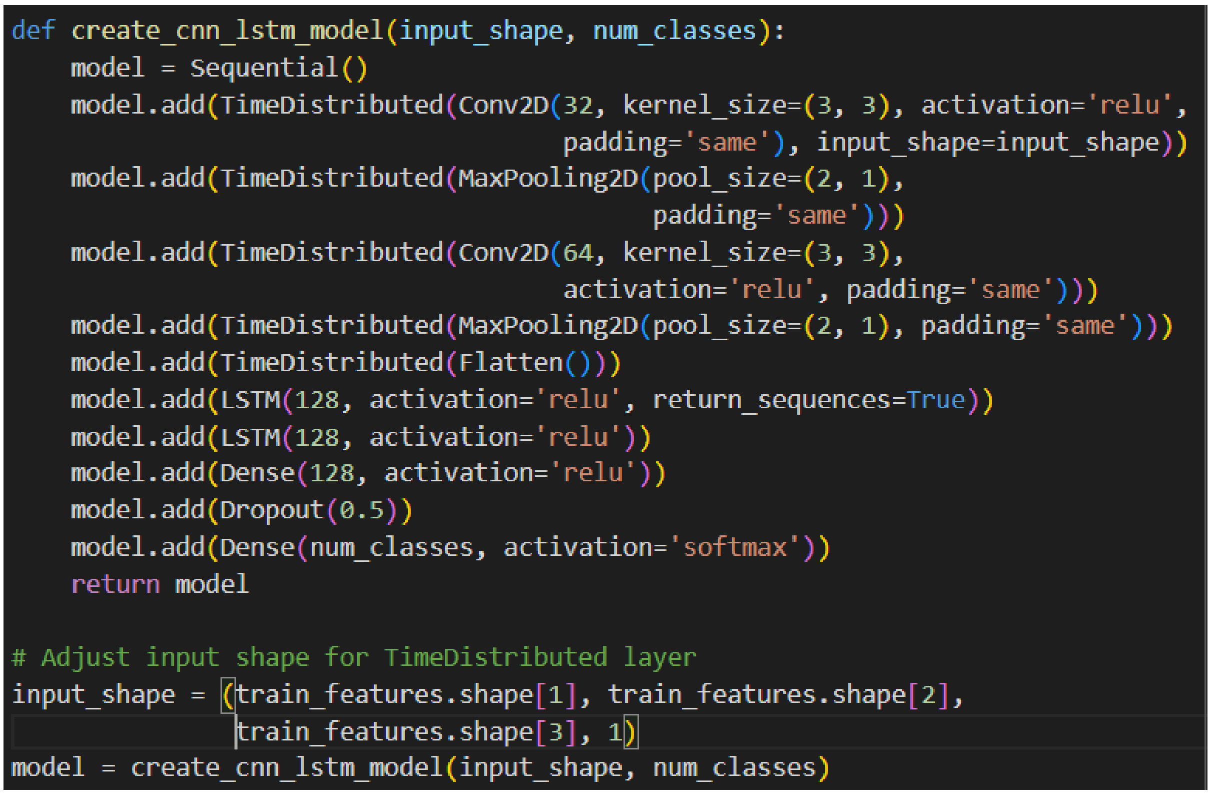
Then, we showed the confusion matrices. By comparing them, in spite of the fact that both models performed identical on test data it can be concluded that Categorical CrossEntropy appears to perform slightly better in terms of correctly identifying the classes (True Positives), and generally has fewer False Negatives. While Mean Squared Error has a tendency to produce fewer False Positives but results in higher False Negatives, indicating a higher misclassification rate for some classes. Overall, Categorical CrossEntropy seems to be more effective for this particular classification task, balancing between True Positives and False Negatives more efficiently.


Confusion Matrix - Categorical Crossentropy
True
Predicted
0: 124 0 3 4 4 0 0
1: 0 81 0 0 0 2 6
2: 2 0 40 1 0 1 2
3: 1 1 2 74 1 0 1
4: 3 1 0 3 62 0 1
5: 0 5 0 0 0 62 1
6: 0 7 0 0 0 1 66
0 1 2 3 4 5 6
120 100 80 60 40 20 0


Confusion Matrix - Mean Squared Error
True
Predicted
0: 129 0 2 3 1 0 0
1: 0 83 0 0 0 2 4
2: 2 1 39 0 1 1 2
3: 1 3 2 72 0 0 2
4: 4 1 0 4 60 0 1
5: 0 3 0 0 0 65 0
6: 0 10 0 0 0 1 63
0 1 2 3 4 5 6
120 100 80 60 40 20 0

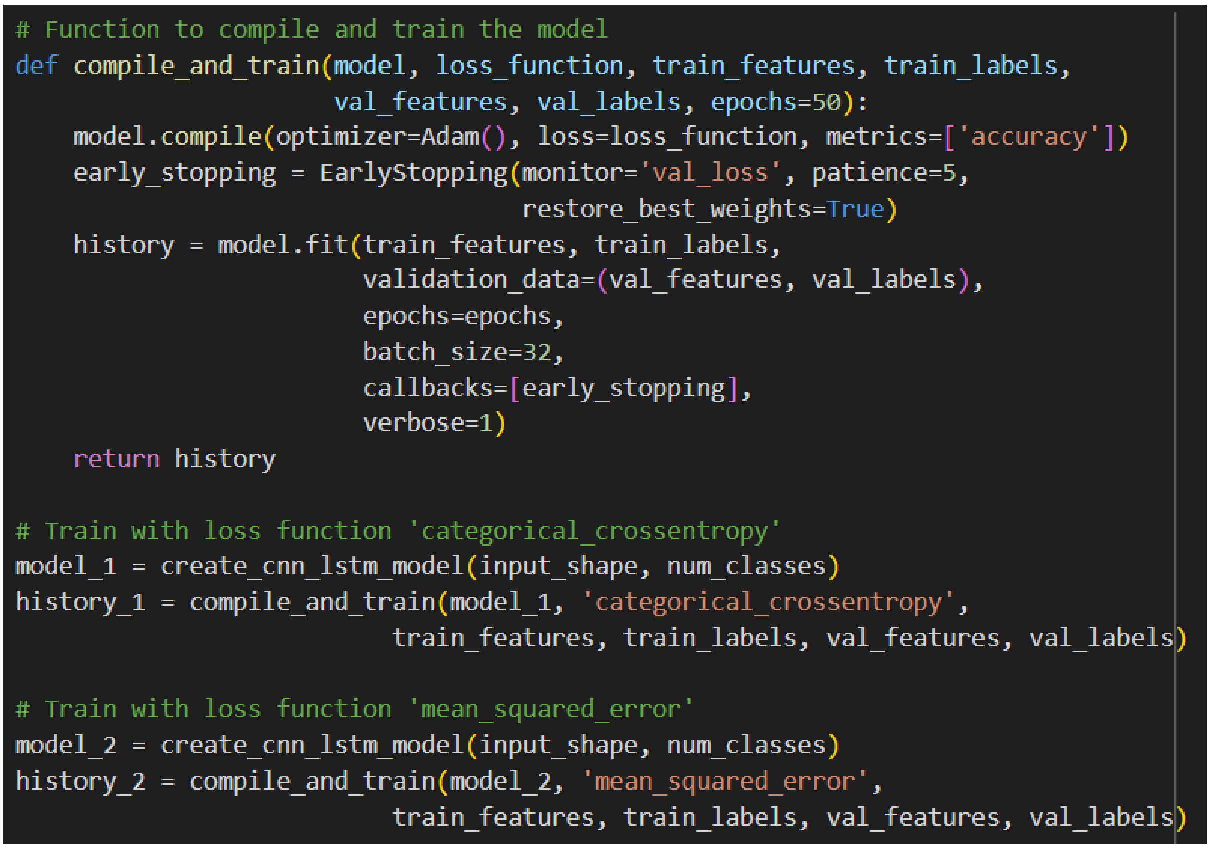
D. CNN-LSTM Model

Then, we designed a CNN_LSTM model as below. We almost had the same approach as the previous model. In addition, we were forced to rearrange input features to fit in the input layer for this model.



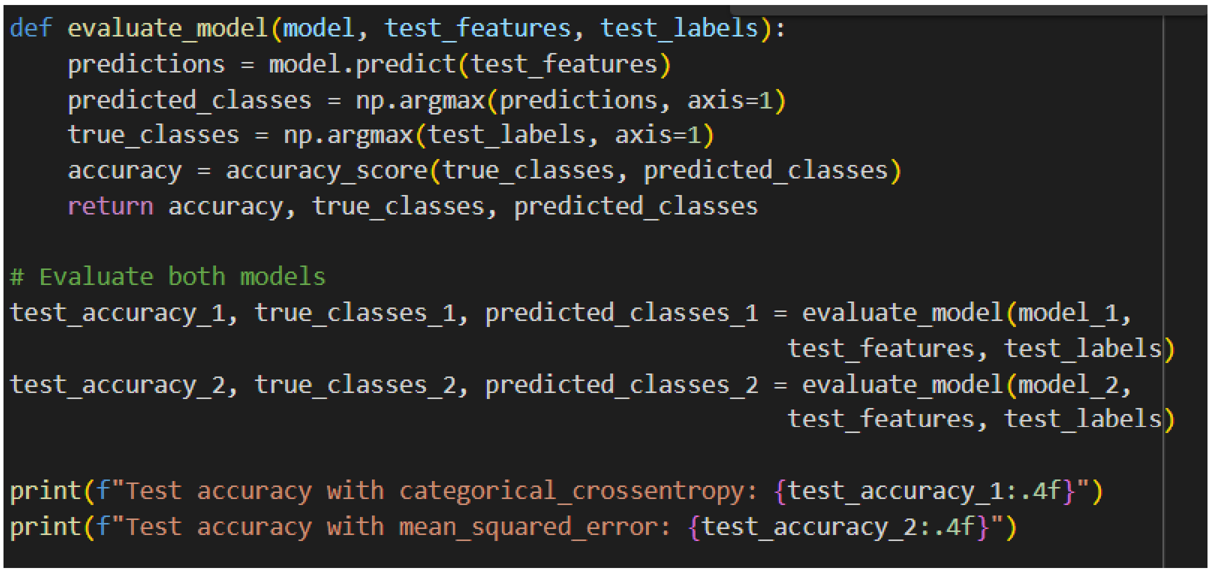
```python
def create_cnn_lstm_model(input_shape, num_classes):
    model = Sequential()
    model.add(TimeDistributed(Conv2D(32, kernel_size=(3, 3), activation='relu',
                                     padding='same'), input_shape=input_shape))
    model.add(TimeDistributed(MaxPooling2D(pool_size=(2, 1),
                                           padding='same')))
    model.add(TimeDistributed(Conv2D(64, kernel_size=(3, 3),
                                     activation='relu', padding='same')))
    model.add(TimeDistributed(MaxPooling2D(pool_size=(2, 1), padding='same')))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(128, activation='relu', return_sequences=True))
    model.add(LSTM(128, activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    return model

# Adjust input shape for TimeDistributed layer
input_shape = (train_features.shape[1], train_features.shape[2],
               train_features.shape[3], 1)
model = create_cnn_lstm_model(input_shape, num_classes)
```

Then, we compiled two models with the aforementioned cost functions.



```python
# Function to compile and train the model
def compile_and_train(model, loss_function, train_features, train_labels,
                      val_features, val_labels, epochs=50):
    model.compile(optimizer=Adam(), loss=loss_function, metrics=['accuracy'])
    early_stopping = EarlyStopping(monitor='val_loss', patience=5,
                                   restore_best_weights=True)
    history = model.fit(train_features, train_labels,
                        validation_data=(val_features, val_labels),
                        epochs=epochs,
                        batch_size=32,
                        callbacks=[early_stopping],
                        verbose=1)
    return history

# Train with loss function 'categorical_crossentropy'
model_1 = create_cnn_lstm_model(input_shape, num_classes)
history_1 = compile_and_train(model_1, 'categorical_crossentropy',
                          train_features, train_labels, val_features, val_labels)

# Train with loss function 'mean_squared_error'
model_2 = create_cnn_lstm_model(input_shape, num_classes)
history_2 = compile_and_train(model_2, 'mean_squared_error',
                          train_features, train_labels, val_features, val_labels)
```

Next, we evaluated the models, reported their accuracy on test data, and plotted their respective confusion matrices.

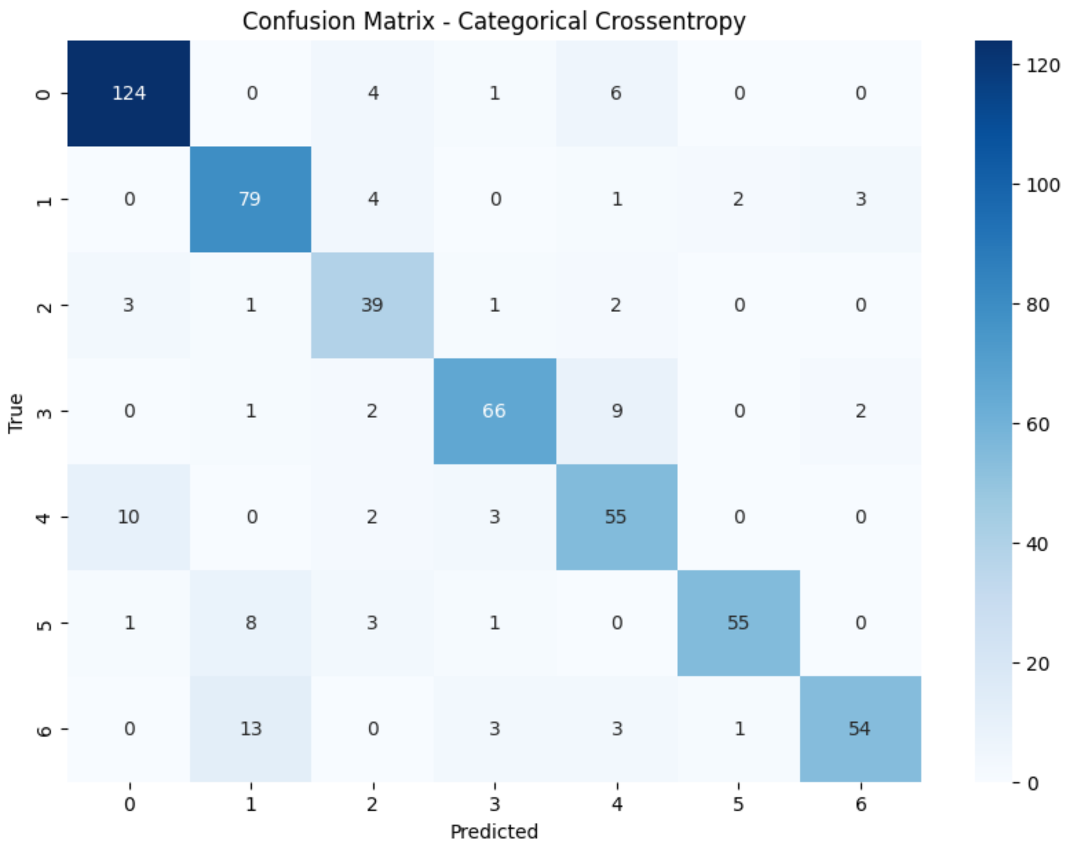


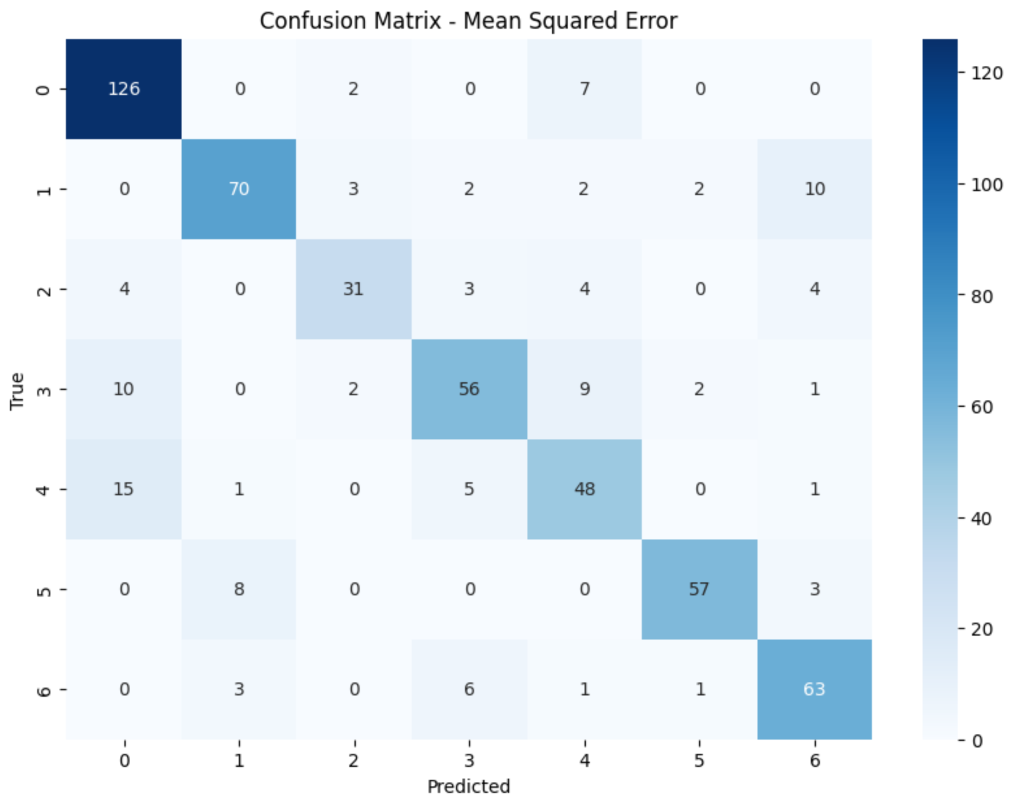
```python
def evaluate_model(model, test_features, test_labels):
    predictions = model.predict(test_features)
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(test_labels, axis=1)
    accuracy = accuracy_score(true_classes, predicted_classes)
    return accuracy, true_classes, predicted_classes

# Evaluate both models
test_accuracy_1, true_classes_1, predicted_classes_1 = evaluate_model(model_1,
                                                     test_features, test_labels)
test_accuracy_2, true_classes_2, predicted_classes_2 = evaluate_model(model_2,
                                                     test_features, test_labels)

print(f"Test accuracy with categorical_crossentropy: {test_accuracy_1:.4f}")
print(f"Test accuracy with mean_squared_error: {test_accuracy_2:.4f}")
```

```python
# Confusion Matrix
def plot_confusion_matrix(true_classes, predicted_classes, title):
    cm = confusion_matrix(true_classes, predicted_classes)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=range(num_classes), yticklabels=range(num_classes))
    plt.title(f'Confusion Matrix - {title}')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.show()

plot_confusion_matrix(true_classes_1, predicted_classes_1,
                      'Categorical Crossentropy')
plot_confusion_matrix(true_classes_2, predicted_classes_2, 'Mean Squared Error')
```

The outputs were as below.

```
18/18 [==============================] - 3s 126ms/step
18/18 [==============================] - 2s 80ms/step
Test accuracy with categorical_crossentropy: 0.8399
Test accuracy with mean_squared_error: 0.8025
```





Categorical CrossEntropy outperforms Mean Squared Error in terms of True Positives and False Negatives, making it more effective for this classification task. Mean Squared Error tends to result in higher False Positives and False Negatives, indicating less precise classification performance. Hence, Categorical CrossEntropy appears to be the more suitable loss function for this particular task, providing better balance and accuracy in classification. Also, it is worth mentioning that Categorical CrossEntropy yielded in a better accuracy on test data.

E. Comparison and Analysis

The CNN model outperforms the CNN-LSTM model in terms of both True Positives and False Positives across all classes. The CNN-LSTM model, while incorporating temporal features via the LSTM layers, seems to struggle more with accurate classification in this specific task. This may suggest that the sequential dependencies captured by the LSTM layers do not significantly enhance performance for this particular dataset or that the LSTM layers may introduce complexity that leads to higher misclassification rates. Thus, for this specific task and dataset, the CNN model is more effective.

Class	Metric	CNN	CNN-LSTM
0 (Anger)	TP	124	124
	FP	11	11
	FN	11	11
1 (Boredom)	TP	81	79
	FP	8	10
	FN	8	10
2 (Disgust)	TP	40	39
	FP	3	6
	FN	3	6
3 (Anxiety/Fear)	TP	74	66
	FP	3	12
	FN	3	12
4 (Happiness)	TP	62	55
	FP	4	15
	FN	4	15
5 (Sadness)	TP	62	55
	FP	5	12
	FN	5	12
6 (Neutral)	TP	66	54
	FP	7	19
	FN	7	19