Image Recognition model writeup justification

As a part of the EDA, it is crucial to know the size of the images, the quantity of images and number of images in each class (disproportional number of entities in each class causes bias). Cell 4 and 11 did exactly this. This allowed us to iteratively go back and declare our input_shape variable for our model and attain a degree of confidence that our dataset does should not have any label ratio bias, i.e large variations in the amounts of images in each class.

I chose to keep the original image size (100, 100, 3) and not shrink or enlarge it, to preserve the images features. Data was loaded by storing the CSV file contents in a data frame and using the "image_paths" and "class_label" to store images in a tf.data.Dataset. To do this the "spectrograms/" segment of "image_path" was removed as it was not in the directory structure.

The tf.data.Dataset data type is more efficient for machine learning models. It allows for loading batches of the data into memory instead of all the data at once. This is less resource intensive, especially for image classification tasks. Should the dataset grow for future iterations this will be more efficient. This also allows less resource intensive machines to be able to train models. It is a more scalable approach which accommodates distributed computing.

It also simplifies pipeline creation as we are only working train_ds instead of X_train and y_train. Batch sizes of 32 was used as smaller batch sizes on average train quicker.

I normalised the data by scaling the images. This was done so by dividing each float32 pixel representative by 255.0. Normalising increases the speed and efficiency of training; it helps the training set converge faster and reduces bias by preventing pixels with more intense values from dominating in the learning process.

Later, Data is reloaded, for the transfer learning segment, using the image_dataset_from_directory function, which loads images from directories, assigning the name of the folder as the class name. This is because images needed to be (224, 224, 3) for the EfficientNetB0 model and I was having issues loading the data in that size using the first method.

Data loaded in both instances was shuffled to prevent bias or overfitting in when training the models.

Since machine learning models only understand integer labels, LabelEncoder was used to create a new column where the classes had been converted to their integer counterparts. AUTOTUNE was used to let tensorflow determine the optimal values for

buffer size, number of parallel calls, and prefetch during runtime for our specific hardware, thus increasing the speed and efficiency of training the model.

The dataset needed to be split up into 3 datasets, training, validation and testing. Training would be used to train the model, and the validation set was used to evaluate the training and finetune the parameters. The test set would then evaluate how well the model performed, testing if it overfit any of the data. A training, validation, test split ratio used was 70:15:15 instead of a typical 80:10:10. This was so that more data could be used to tune the hyperparameters and test.

```
# Creating a CNN network
# THIS IS THE FINAL MODEL TO BE USED, YIELDS BEST RESULTS WITH SMALLEST AMOUNT OF TRAINING TIME
model_1 = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),

        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(256, activation="relu"),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

# BEST MODEL, YET MOST COMPUTATIONALY AND TIME HEAVY
# model_1 = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(256, activation="relu"),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

# GOOD FOR SPEED BUT AS LOWEST ACCURACY READINGS
model_1 = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(16, kernel_size=(3, 3), activation="relu", padding="same"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu", padding="same"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(16, kernel_size=(3, 3), activation="relu", padding="same"),
        layers.MaxPooling2D(pool_size=(2, 2)),

        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(256, activation="relu"),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

Continuing EDA, plotting random images from each class helped understand the types of images in each class. This also led to the decision to not applying image augmentation to our original model. This was because the details and features in the spectrograms on average covered the whole image. I had worries that this would remove too much critical data as accuracy dropped to 44%. I did apply augmentation to the transfer learning as after testing with and without it I found it helped the model generalize better and achieve better accuracies of 59%. When it came to choosing the model several variations of the network were tried and tested.

Although all 3 models performed well with test accuracy of around 50% their speed and performance differed dramatically.

The first model showed results 49-53% and showed the best results for time needed to train the model.

The 2nd model was computationally heavy and only yielded results of around 50-53%. It was computationally excessive.

The second model was the quickest and achieved good test accuracy results of 47-51%, The model works well for slower machines but model 1 was chosen due to minimal compute time increases and better results

Three layers of Conv2D was used to apply kernels to extract all the relevant features for the model to make its predictions. Through trial and error 3 was found to be the optimal number of layers to have a powerful enough model to extract all the features but not overfit. The number of kernels used was 32, 64, and 32 in that order. These were found to minimise overfitting and were again known through trial and error. Max pooling layers were applied after kernel layers to decrease the size of the image and consequently reduce the number of computational operations. Thus, making the model train faster and solve previous iterations issue of overfitting.

A dropout layer was added to increase randomness in the model and reduce the possibility of the model overfitting the data. Padding was purposefully not used because it was found to contribute towards overfitting/ introducing noise in the first 2 models. Padding was only useful in model 3 as it showed an improvement. A deep dense layer (256) was added before the output to help with feature extraction and complexity handling. It helps the model to generalise without requiring excessive computational power.

Transfer learning using EfficientNetB0 was tried to see if it could improve model performance, EfficientNetB0 has a lower parameter size and quicker training speeds. I originally implemented InceptionV3 deep learning model but found that it was overkill for the task at hand and did not work well with the amount of training data we had for our model, achieving accuracy of 21%. EfficientNetB0 works better with smaller amounts of

data and achieved accuracy of 53%, later 59% with augmentation, on the first training iteration.

EfficientNetB0 has its own preprocessing module which we can easily apply. I added a 3 dense end layers with a deep dense layer so that the output of the model would better understand my datasets features. GlobalAveragePooling2D was used as it reduces overfitting as the feature map is replaced with its spacial average, helping reduce the number of parameters while still preserving spacial information.

The same metric was used except this time custom small learning rates were used as lower learning rates are better for fine tuning models. The model's best version was also saved so that it could be easily loaded and trained again/ further. Final results were not much better than the original CNN, different models for transfer learning or different techniques could be explored to help increase the reading.