Project 3: Neural Networks

Gabriel Urbaitis and Rahul Payeli

In this project, we once again performed audio classification in two data modalities. The features extracted from Project 2, and images of spectral frequencies over time. We created a Multi-Layer Perceptron model for the former and utilized a Convolutional Neural Network and Transfer Learning on the latter. Our comparisons within each learning method and between each one follow with regard to performance and accuracy.

I. Introduction

In the project, we are tasked with using Multi-Layer Perceptrons (MLPs), convolutional neural networks (CNNs), and Transfer Learning.

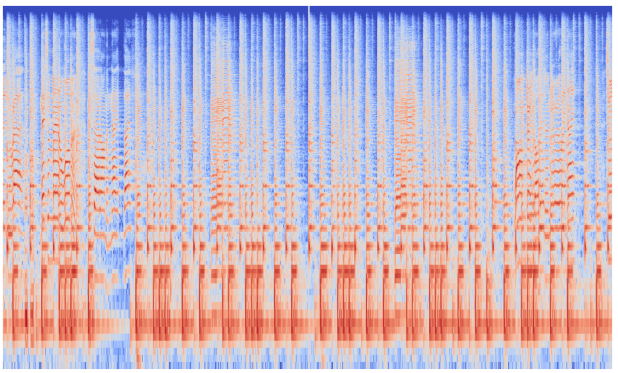
For our extracted Project 2 features, we employ MLPs, for image classification, we employ CNNs, which are particularly adept at recognizing patterns in image data. Additionally, we us transfer learning by adapting the VGG-16, GoogleNet and Resnet models, pre-trained neural networks for image classification tasks.

II. Design and Implementation

A. Data Conversion

We transformed the raw audio files into spectral images in an RGB colormap. These are visual representations of spectral frequencies in a sound over time. Applied normalization, such that the loudest point will reach +1 or -1, and all other points are scaled accordingly, in proportion. This ensures that the output signal utilizes the entire available dynamic range, and helps in retaining the audio quality while enhancing the performance of the following processing stages. To translate the audio signals from the time domain to the frequency domain, we applied the Short-Time Fourier Transform (STFT). In this transformation, we chose the **framesize as 2048** and **hopsize as 512**, which play a very crucial role in maintaining the window size and successive window overlaps that directly affects the image resolution. Then this is converted to the log scale which helps to compress a wide range of sound intensities onto a scale closer to the range of human auditory perception. Each spectrogram is plotted without any axes to make sure we get clean images for feeding into neural networks. These images are saved in RGB format for better pattern recognition with the neural networks.

Log spectrogram of a Rock music file



B. Data Splitting

We used an 80-20 split on the data. 80% is allocated to training, and 20% goes to validation and testing in equal parts. We followed the same strategy in all the neural networks. There is no overlap of data between the splits.

Data Splitting part

```
total_size = len(allData)
train_size = int(0.8 * total_size) # 80% split for train
test_val_size = total_size - train_size # 20% split for validation and test

train_data, test_val_data = random_split(allData, [train_size, test_val_size]) # 80-20 split between train and validation+test

val_size = test_size = int(test_val_size / 2) # splitting 10% each between validation and test
val_data, test_data = random_split(test_val_data, [val_size, test_size]) # 10-10 split between validation and test

train_data.dataset.transform = trainDataTransform #Applying transform to train dataset
val_data.dataset.transform = dataTransform #Applying transform to validation dataset
test_data.dataset.transform = dataTransform #Applying transform to test dataset
```

C. Data Augmentation

Random rotation of images by 15 degrees was applied to the training set data. This increased the diversity of the training set by applying random transformations to the original images. The strategy

prevents the model from overfitting and increases its ability to generalize new data.

Data Augmentation part

```
trainDataTransform = transforms.Compose([
    transforms.Resize((512, 512)),
    transforms.RandomRotation(15),
    #transforms.RandomApply([transforms.RandomRotation(15)], p=0.5),
    transforms.ToTensor()
])

dataTransform = transforms.Compose([
    transforms.Resize((512, 512)),
    transforms.ToTensor()
])
```

D. Convolutional Neural Network (CNN)

The CNN architecture, **ConvolNeuNet** has been designed to classify spectral images of music files into the genre to which it belong. This model has 3 convolutional layers, which increases the number of filters from 32 to 96. Each convolutional layer used a 3*3 sized kernel, stride of 1, and padding of 1. ReLU is applied throughout the convolutional layers for non-linearity. Max pooling is applied post ReLU in each convolutional layer of size 2*2, which reduces the computational load by enhancing the detection of important features.

After the convolution, the layers are flattened i.e., Fully connected layers are formed. This includes a series of layers that reduce in size, from 512 to 10 (number of classes for classification). Also some dropout layers, and ReLU functions are applied between these dense fully connected layers for better and faster classifications.

We have used the default pytorch lightning for training the CNN, which reduced the complexity of training loops and improved the reproducibility. We have used Adam optimizer with a learning rate of 0.001 for optimizing the weights in the network and learning better. This will help us to reach the global maximum.

CNN Architecture

```
class ConvolNeuNet(pl.LightningModule):
  def __init__(self, num_classes):
    super().__init__()
    # convolutional layers
    self.convol1 = nn.Conv2d(3, 32, kernel_size = 3, stride = 1, padding = 1)
    self.convol2 = nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)
    self.convol3 = nn.Conv2d(64, 96, kernel_size = 3, stride = 1, padding = 1)
    self.pool = nn.MaxPool2d(2, 2)

    #linear layers
    self.fc1 = nn.Linear(96 * 64 * 64, 512)
    self.fc2 = nn.Linear(512, 256)
    self.fc3 = nn.Linear(256, 128)
    self.fc4 = nn.Linear(128, 64)
    self.fc5 = nn.Linear(64, num_classes)

    self.relu = nn.ReLU() #non-linear activation function
    self.dropout = nn.Dropout(0.5) #dropout function
    self.cost = nn.CrossEntropyLoss() #cost function

  def forward(self, x):
    #CNN architechture
    x = self.pool(self.relu(self.convol1(x)))
    x = self.pool(self.relu(self.convol2(x)))
    x = self.pool(self.relu(self.convol3(x)))
    x = torch.flatten(x, 1)
    x = self.relu(self.fc1(x))
    x = self.dropout(x)
    x = self.relu(self.fc2(x))
    x = self.relu(self.fc3(x))
    x = self.dropout(x)
    x = self.relu(self.fc4(x))
    x = self.fc5(x)
    return x
```

We have used **'ReduceLROnPlateau'**, a learning rate scheduler, which is useful in adjusting the learning rate by reducing it by a factor of 0.1 if there is no improvement seen in the last 3 epochs. This is quite a useful technique in the coming stages of training for better learning and classification by the network by reducing the loss.

Learning scheduler part

```
def configure_optimizers(self):
  optimizer = optim.Adam(self.parameters(), lr=0.001) #Optimizer for better adjusting the weights
  scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3) #ReduceLROnPlateau scheduler helps in fine tuning the learning parameter
  return {"optimizer": optimizer, "lr_scheduler": {"scheduler": scheduler, "monitor": "val_loss"}}
```

We have implemented **early stopping** of the algorithm to ensure efficient training. This mechanism will stop the training if there is no improvement in the accuracy in the last 5 epochs. This is very useful in saving the memory of computational resources and helping the model from degrading.

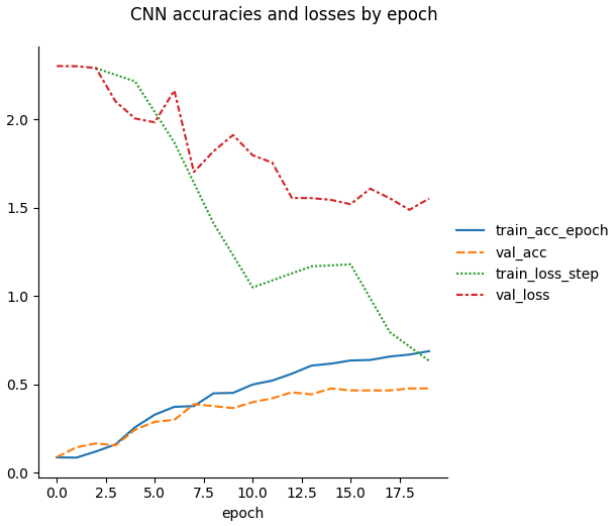
Early stopping part

```
#Helps in stopping the algorithm, if there is no improment in the accuracy in last 5 epochs
early_stopping = EarlyStopping(monitor='val_acc', patience=5, verbose=True, mode='max')
```

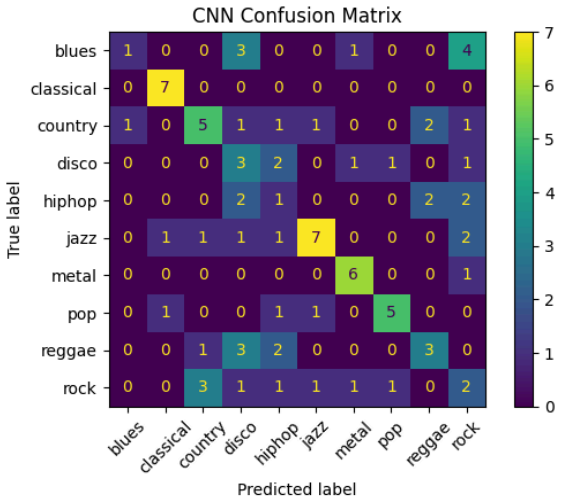
Initially, we tested with 5-fold data preparation, each fold having an 80-20 split but that resulted in ending up with 0.8023 test accuracy which gave a balanced accuracy of 0.4634. This clearly says that it is overfitting.

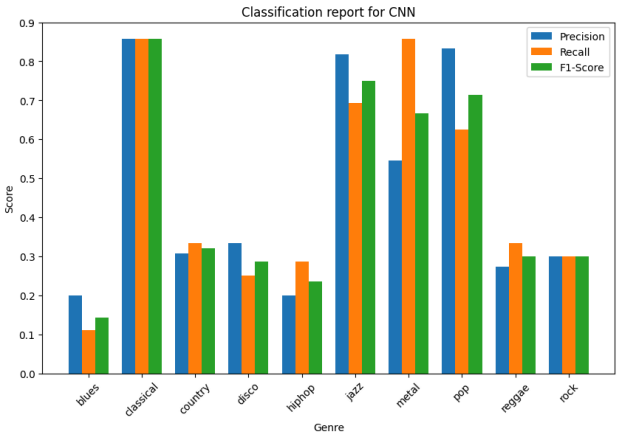
After we did the 80-20 split with 20 epochs, a Dropout of 0.5, and a learning rate of 0.001, gave the below accuracy of 0.4600. The below plot represents the performance of the CNN based on train accuracy, test

accuracy, train loss, and test loss. It is evident that the model is learning and trying to classify the image properly.



The below confusion matrix shows that CNN is able to classify classical, country, jazz, metal, and pop easily but it is struggling to classify the remaining genres correctly. We can also see that it is more struggling to classify the blues and rock genres.





E. Transfer learning

In transfer learning, we chose spectrogram-based classification. They represent audio signals with frequency and time information, which is very important for distinguishing the audio files. These files are very useful for convolutional neural networks, in finding patterns and classifying the images.

VGG-16 is a very well-known deep-learning network for classifying images with high accuracy. The architecture of this network is simple and yet captures the effective patterns for image classification, which is evident in our experiment on this network. VGG is initially pre-trained on an Image-net dataset, which has a very diverse set of images. To use this for our task, we need to freeze the convolutional layers to leverage the learning it had and then fine-tune the classifier portion. The fully connected layers are redefined to align with our 10-class classifier, also added some dropouts to reduce the over-fitting changes, and used the ReLU function to add the non-linearity. The size is reduced from 512 to 256, 256 to 128, and so on until it is reduced to 10 (classes to predict).

When we compare this performance with the CNN, it has been evident that this transfer learning has enhanced accuracy and reduced loss with less overfitting. The CNN has achieved a test accuracy of 0.5223 while the VGG-16 has achieved an accuracy of 0.6993 with data augmentation and 0.76 without data augmentation. Adding the ReLU and dropout at each layer, helped the model to achieve higher accuracy with no overfitting.

VGG16 Architecture for transfer learning

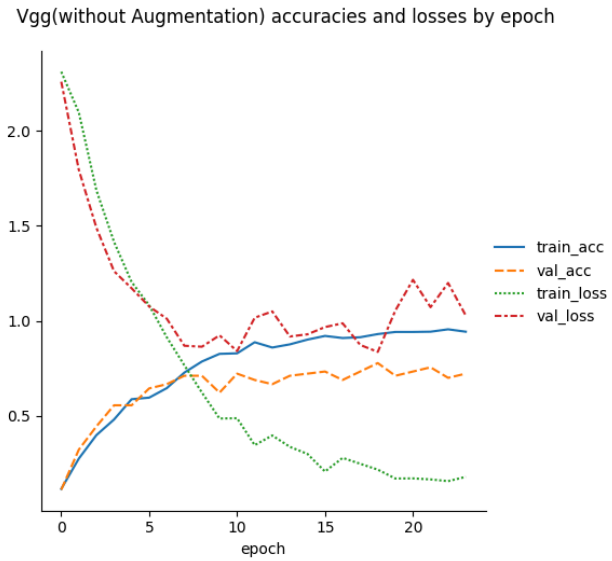
```python
class VGG16Net(pl.LightningModule):
    def __init__(self, num_classes):
        super().__init__()

        # Load a pre-trained VGG16 model
        backbone = models.vgg16(pretrained=True)

        # Freezing all the parameters in the feature extraction
        for param in backbone.features.parameters():
            param.requires_grad = False

        # Modify the classifier part by adding dropouts and ReLU at each layer
        num_features = backbone.classifier[0].in_features
        backbone.classifier = nn.Sequential(
            nn.Linear(num_features, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, num_classes)
        )

        self.model = backbone
        self.loss_func = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.model(x)
```

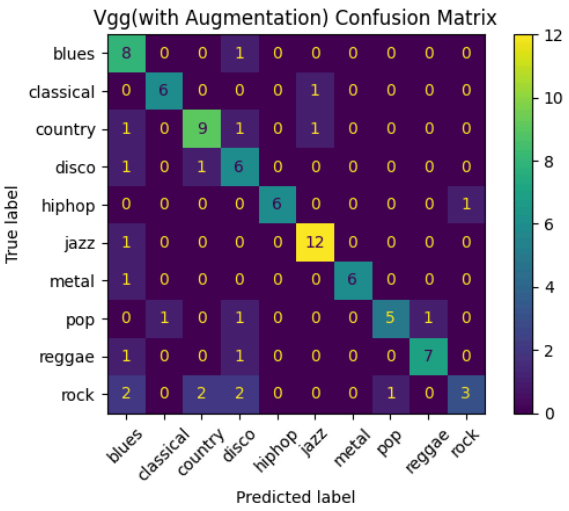
NOTE:

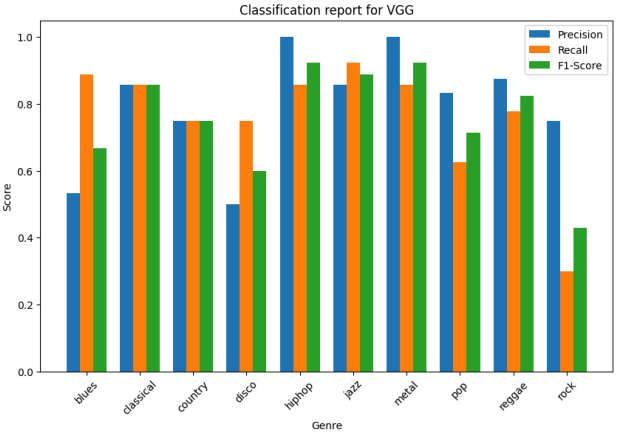
I. With Augmentation: VGG16 implemented with Data Augmentation like image rotations in the training set.

II. Without/No Augmentation: VGG16 was implemented without any Data Augmentation techniques.

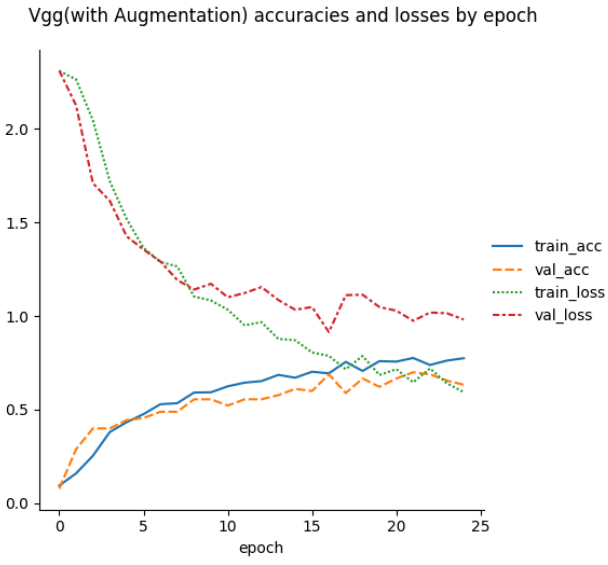
The below three plots show the accuracy and loss by epoch plot, confusion matrix, and the classification report for VGG16 (with augmented data). From the below graphs, we can say that the model had a significant amount of learning and is able to classify the images with a pretty good amount of accuracy.

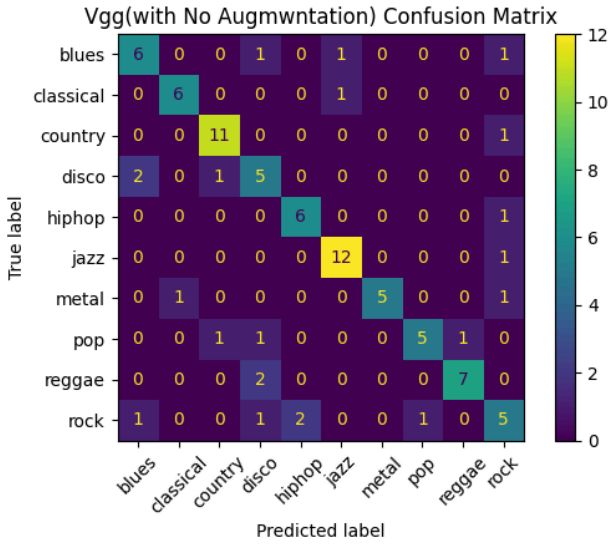


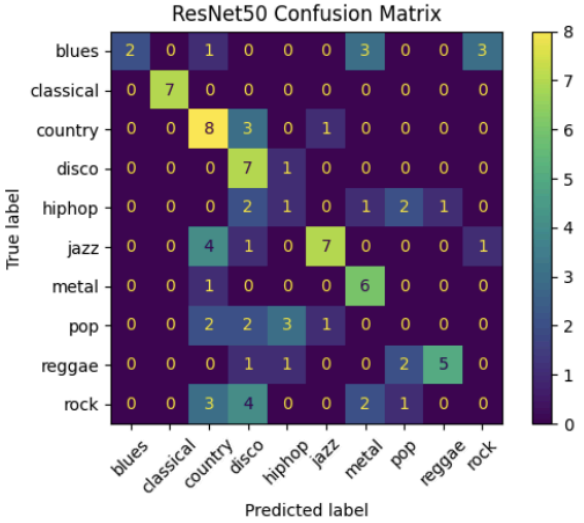


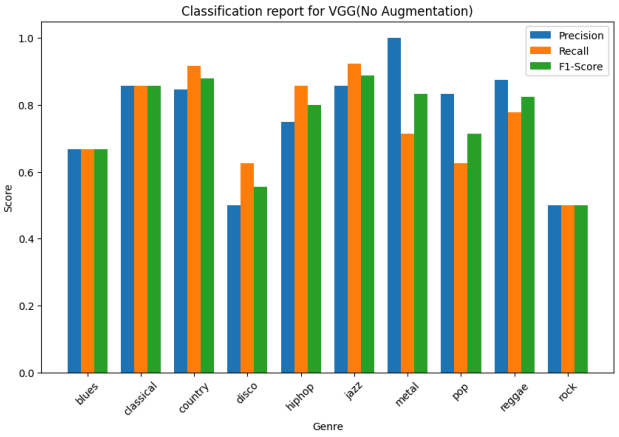


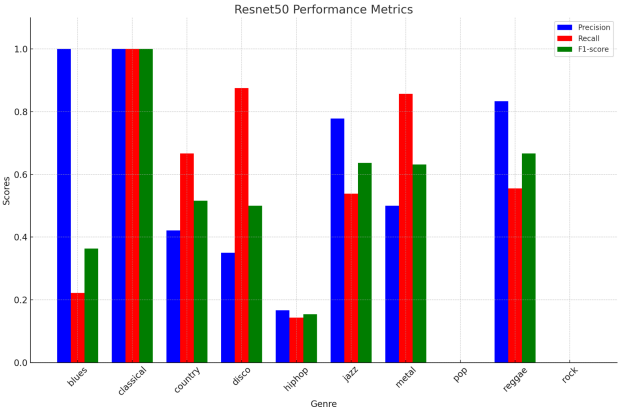
The below three plots show the accuracy and loss by epoch plot, confusion matrix, and classification report for VGG16 (without augmented data). The VGG(with no Augmentation) has learned the country patterns well compared the VGG(with Augmentation)





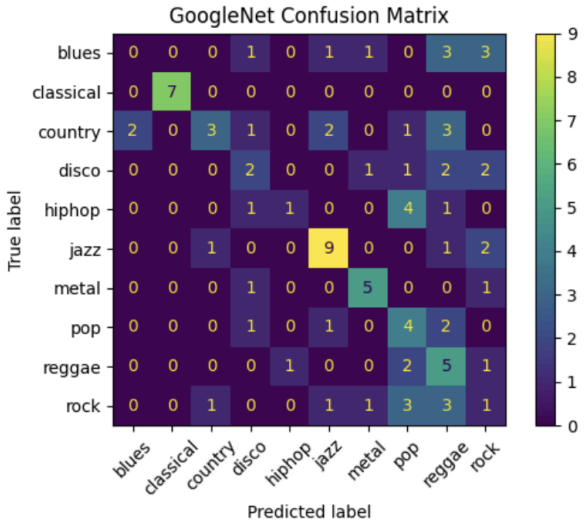


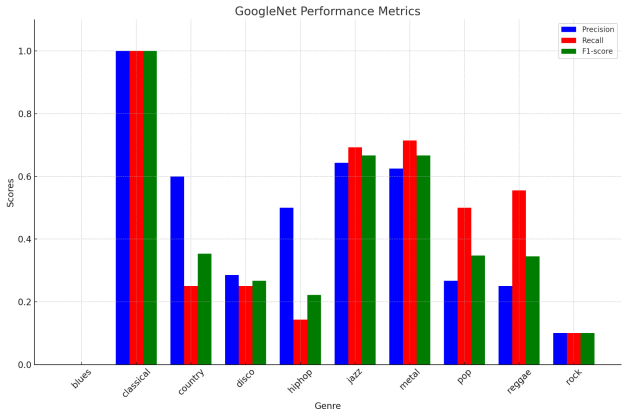




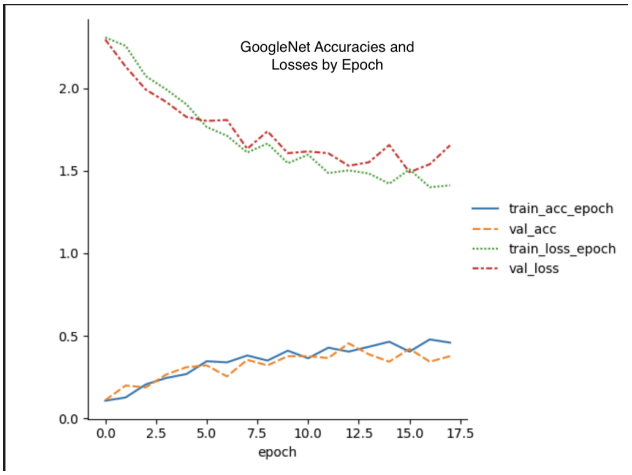
We also compared implementations with GoogleNet, which had 0.42702 balanced accuracy and Resnet50 which had 0.50488 balanced accuracy. All 3 had the same layers added to their base models, though the VGG models later added dropout layers intermittently when they were clearly selected as the best model to focus on.

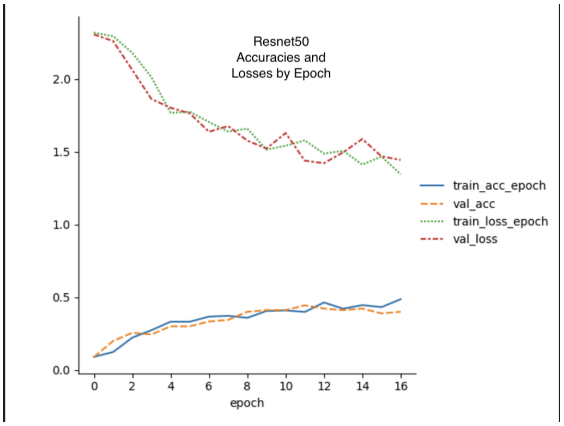
ResNet failed to classify any pop or rock samples correctly, while GoogleNet failed to classify any blues samples correctly. ResNet did fairly well with classical, country, disco and jazz, while GoogleNet only did well with classical and jazz, though it did a little better with jazz than ResNet.





GoogleNet reached stopping criteria in 17 epochs whereas Resnet reached in 16 epochs.
The losses and accuracies are remarkably similar for training and validation, unlike the MLP models looked at below.





F. Multi-Layer Perceptrons (MLP)

The dataset used for MLP analysis was from Project 2, where we extracted Mel-Frequency Cepstral Coefficients (MFCCs) to capture timbral and spectral qualities, Chroma Features for musical pitches, Spectral Features for texture and frequency content, Rhythm Features for tempo and beat onset strength, and Zero-Crossing Rate for the rate of sign changes in the waveform. The mean of large vectors and matrices was taken for feature reduction and Principal Component Analysis was applied, taking only PCAs that explained the top 95% of the data.

The following template was used to train the Project 2 features.

```
class MLP(pl.LightningModule):
    def __init__(self, input_size, hidden_size, output_size, dropout_rate=0.2):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout_rate)
        self.relu = nn.ReLU()
        self.cost = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

    def evaluate(self, batch, stage=None):
        x, y = batch
        y_hat = self.forward(x)
        loss = self.cost(y_hat, y)
        preds = torch.argmax(y_hat, dim=1)
        acc = (preds == y).float().mean()
        if stage:
            self.log(f"{stage}_loss", loss, on_epoch=True, prog_bar=True, logger=True) #
            self.log(f"{stage}_acc", acc,  on_epoch=True, prog_bar=True, logger=True)
        return loss
```

The testing accuracy was not recorded for the first run, however, the training accuracy for a hidden size of 64 and 100 epochs was only 53%. Adding another hidden layer of width 48 with tanh activation increased training accuracy from 53% to 72%. However, at this point, we realized it was more important to look at Testing accuracy, which was 0.52222 for the first modification. From here on out, all accuracies reported in this section will be test accuracy.

One change made from the first modification was altering the tanh layer to width 64. This moderately improved accuracy to .54444. Another change to the 64, 48 width first modification was to make both layers Relu activations. Accuracy took a step back at .48888.

Recognizing that a mix of activation functions and greater widths had resulted in the highest accuracy, a third layer was added. With a first layer of 64 using relu, second 96 using tanh, and third 64 using relu again, accuracy reached .51111.

To test if perhaps the gains had simply come from the use of tanh and not from a mixture, all three

layers were changed to tanh, but once again accuracy took a step back to .46666.

Maintaining the rest of the parameters, the middle layer was reduced to be equal to the outer layers at 64 and accuracy improved to .48888.

Attempting to reduce the middle layer to 48 took a minor step back to .47777. Using this “hourglass” shape on a tanh-relu-tanh layer order increased accuracy to .5.

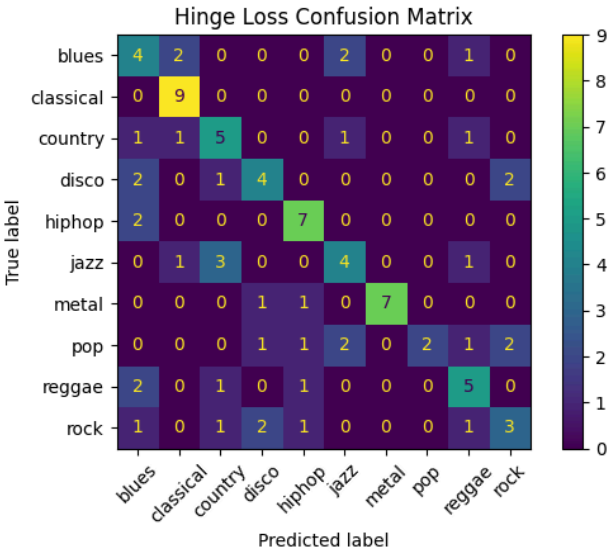
Setting all three layers back to 64 width, accuracy improved to .52222. Lowering all three to 48 drastically reduced accuracy to .45555.

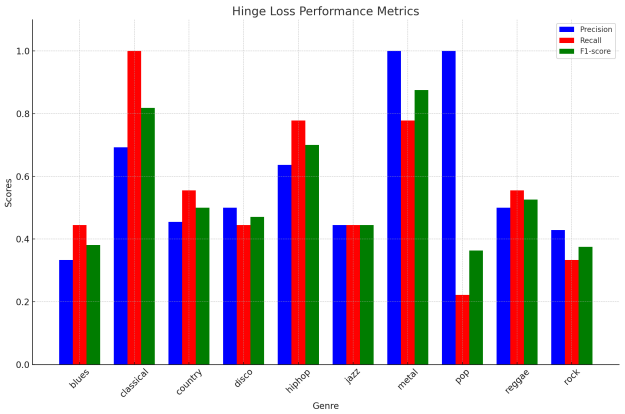
Increasing width of all three layers increased accuracy, until no difference was found between 128 and 256 at .54444 again.

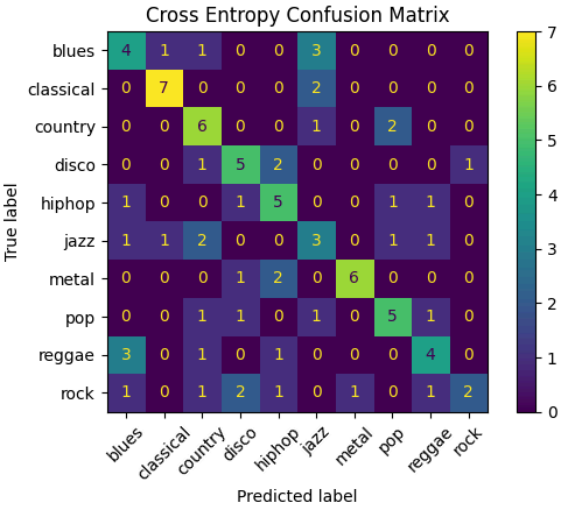
Due to an error, we tested balanced accuracy for the alteration of using hinge loss as an activation function at 512 width for all three layers. Whereas the 256 Cross entropy had balanced accuracy of .48310, the 512 Hinge Loss had balanced accuracy of 0.59896. We only realized the day of the deadline the reason for the bump in accuracy was more due to the extra width than the choice of cost function, as when we adjusted cross entropy to 512 width layers, balanced accuracy rose to 0.54732. Kullback Leibler had a balanced accuracy of .54256.

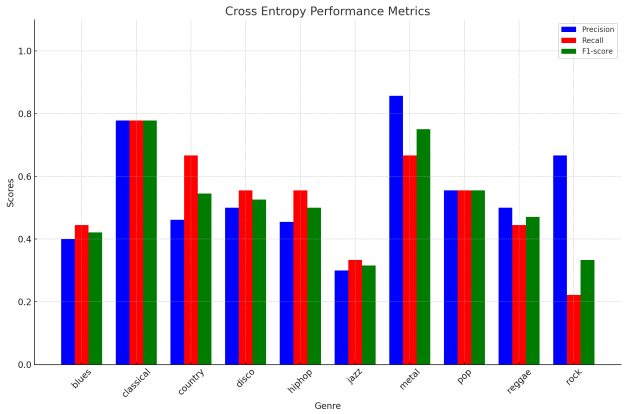
As the discrepancy in accuracies per width didn’t appear with the automatically generated test_acc, we came away with the false conclusion that cost function was the parameter that had the most effect on accuracy. In future studies, we would like to examine why the value it returns is different from balanced accuracy using the same testing_loader, and if it continues to be an unreliable measure, we will use balanced accuracy through the entire process instead.

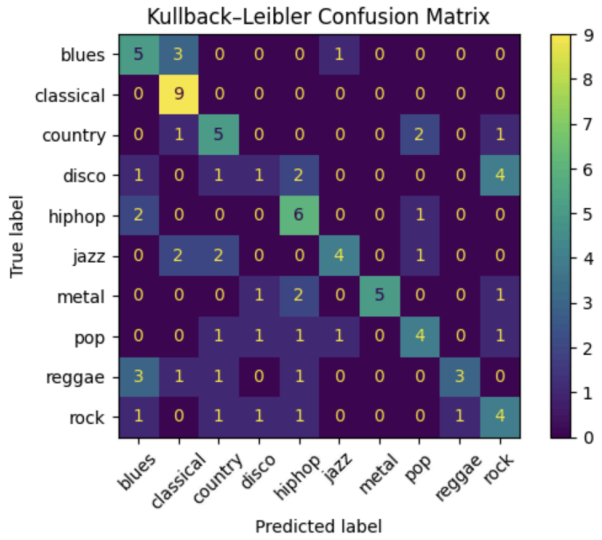
Below are our comparisons for the three cost functions, Cross_Entropy has been corrected to use the 512 width layers, and all three share all other parameters, including tanh-relu-tanh order of activation functions.

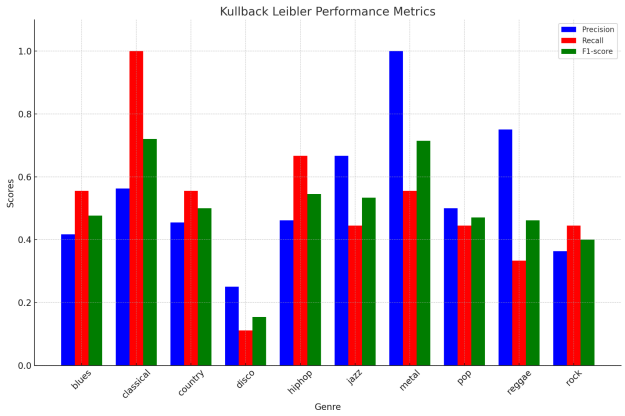




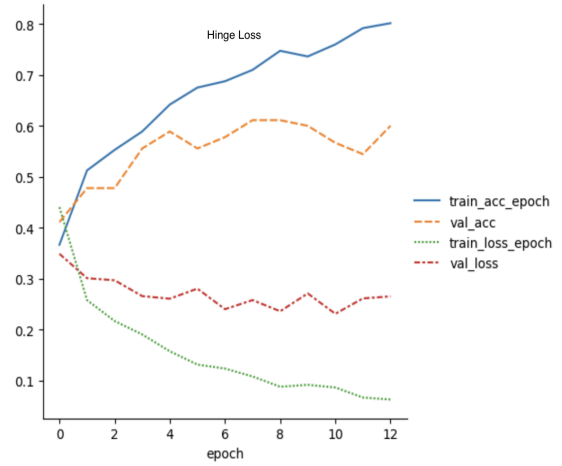


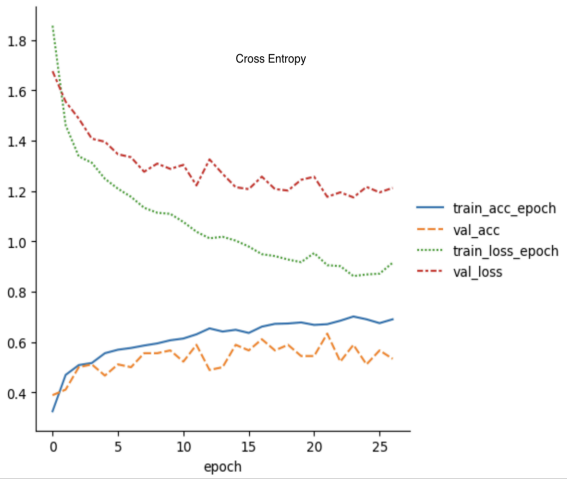


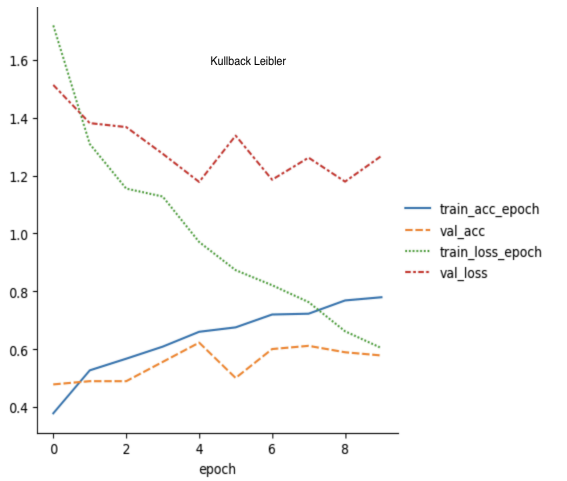




Hinge Loss and Kullback Leibler both perfectly classified all 9 classical samples. Kullback Leibler and Hinge Loss also didn't misclassify any samples of metal, though Hinge loss gained much of its separation in accuracy from the other two by not misclassifying any samples as pop and by classifying 7/9 hip hop samples. Cross Entropy made up its lost ground on classical misclassification and over-classifying samples as metal by performing much better than Kullback Leibler on disco, which was the worst genre classification out of all 3 models. Overall Kullback Leibler had the greatest variance in classifying-ability per genre, Cross entropy and Hinge Loss were pretty well rounded, but Hinge Loss had the highest highs.







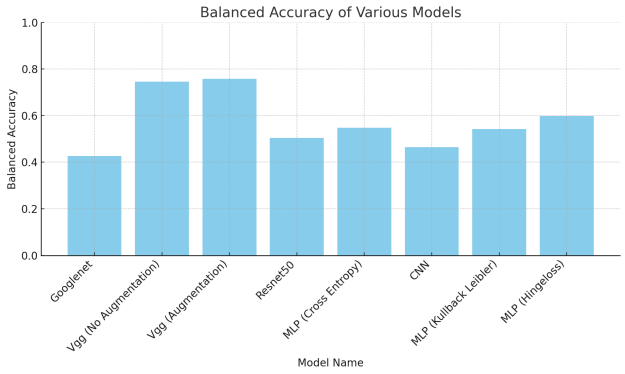
Above are the three models loss and accuracy curves for training and validation. Early stopping criteria was reached when validation accuracy failed to improve after 5 epochs. This took 12 epochs for Hinge Loss, 25

(maximum) for Cross Entropy, and just 9 for Kullback Leibler. Kullback Leibler had the steepest decline in loss for training data, and Hinge Loss had the steepest increase in training accuracy. Validation accuracies and losses all pretty much followed training curves, albeit more jaggedly and overall closer to a 0 slope.

III. Experiments

A. *Balanced Accuracy*

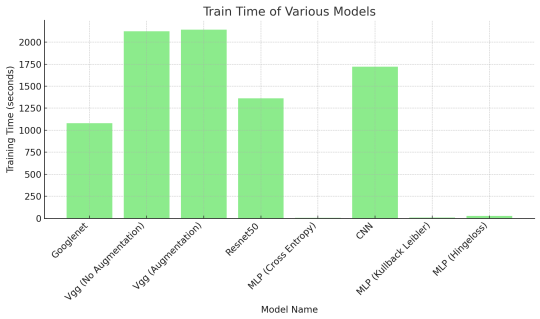
The below graph shows the balanced accuracy across different neural net models. Vgg (with Augmentation) has the highest balanced accuracy while Googlenet has the lowest. Vgg16 has done great work in generalizing the data.



B. *Time to Converge*

This talks about the take taken by the algorithms to converge.

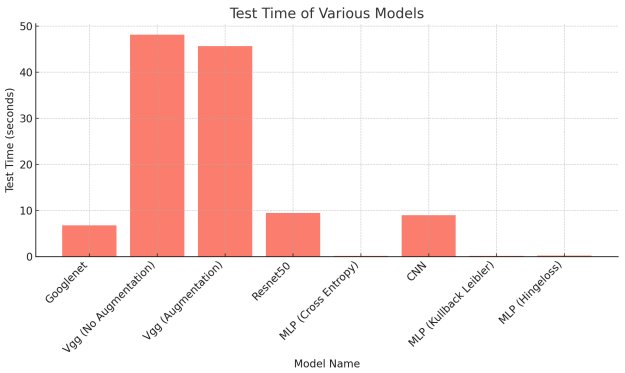
The below graph shows that the Vgg (with Augmentation) has taken more time to converge while MLP(with three different losses) has taken very less time to converge. MLPs are very fast in training the model.



C. *Prediction time*

This talks about the time taken by the model to predict the inputs.

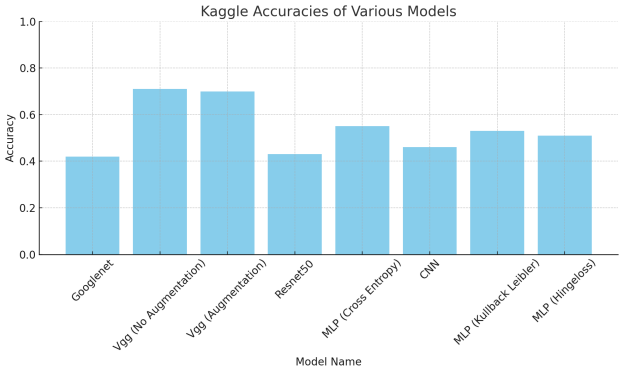
The below graph shows that, again the MLPs are leading the way in predicting the outputs quickly. On the other had, the VGG16 is taking lot of time compared to other models in the graph.



E. *Kaggle submission*

The below graph shows the score achieved on Kaggle with each model.

Vgg has the highest Kaggle score, while the ResNet is the lowest in the graph.



IV. *Conclusion*

In conclusion, this project was able to show that the CNNs and VGG-16, are indeed very powerful solutions for spectrogram audio classification. The VGG-16 is especially held slightly higher ground with more learning and accuracy. This study shows the importance of effective data preprocessing.

Our top Kaggle submission was Vgg (No Augmentation) with 71%. This represented a significant jump over our Logistic Regression final

submission with 57%. Incidentally, our top MLP model, with Cross Entropy scored 55%, suggesting that the methods used in the previous project may have a maximum correlation in the high 50- low 60 percentages. If we were to pursue this project, it seems a Transfer learning model is the best approach, given both its accuracy and advantage of extensive training.