Audio Analysis Using Gramian Angular Field (GAF) and FAST

Andrew Fausak (andrewfausak@my.unt.edu), Andy Fausak (andyfausak@my.unt.edu), Andre Sharp (asharpasharp99@gmail.com), Mica Haney (micahaney@my.unt.edu)

GitHub Repository: https://github.com/dallasai/csce5222-project

1 Introduction

1.1 Motivation

Motivation to create this work is based on review of a pollution analysis method that utilizes Gramian Angular Summation Fields (GASFs) to generate images representing normalized discrete readings of air pollution time series in summary, thereby enabling tools to perform similarity tests to qualify air quality conditions needing to be monitored. Having a solution whereby similarity of works can be compared is useful and readily consumed by AI tools needing features to isolate desired inferences. Experimentation is required to determine resolution needed to uniquely identify or classify a work using Machine Learning methods.

Converting data from one domain of information representation can provide useful information that would not otherwise be noticeable, such as in spectrograms. In this case, audio within a cartesian time series domain is converted to a polar coordinate domain before creating a graphical image in the form of a GASF & Gramian Angular Difference Field (GADF). Prior works have hidden "text" and images in sound files that can be identified only by using specific graphical tools. Using graphical processing techniques to find hidden information or represent features in audio is an interesting topic with the potential to improve the current methods of working with audio by representing an image for feature extraction and representation for the purpose of AI related processing and interpretation.

1.2 Significance

Having a means to identify works quickly in terms of classification and/or origin is very useful. Moreover, this may provide alternate means of tagging documents for referencing. With this technique more information may be extracted from the audio, which will allow for better handling of audio tasks. If the proposed method does extract information from sound that is not usually or easily picked up by current audio processing techniques, then adding such features and procedures to downstream tasks could notably improve results involving audio, particularly in machine learning models.

1.3 Objectives

The overall objectives of this project are to A) extract features from audio files that allows them to be converted to an image format, B) generate multiple image formats to view the effects of different extraction/generation methods, and C) run models with the converted images.

Ideally we would like to apply these above objectives in two fashions. First we would like to compare results of the converted features to results obtained by the same or a very similar architecture that can handle the original audio data. Second, we would like to attempt to re-convert the obtained feature images back to their original form to evaluate the amount of information loss that occurs during the conversion process.

1.4 Features

The completed project will work with three main features, GADF, GASF, and spectrograms.

Spectrograms are visual representations of the frequencies of an audio signal. Currently our implementation of spectrogram generation uses the pre-built function in the matplotlib.pyplot library.

GADF and GASF are types of Gramian Angular Fields (GAFs). GAFs are created from a time series, and each point in the image represents the correlation between two points within the time series where the time series sits on both the x and y axis. GADFs use the sine function to compute values, while GASFs use the cosine function.

2 Background

GAFs are currently used as means to relate real-time events to data. The graph represents changes over time for the data being sampled. Applications vary and have been shown to have significant value within instrumentation [1] and for use in AI applications [2]. In theory the time/data relationship has value for applications in voice recognition where the GAFs represent labels for immediate referencing and inference.

3 Your Model

Figure 1 shows the architecture of all models in this project. After the input, there are three repeating blocks of a convolution layer followed by a batch norm layer, then a max pooling layer, then another batch norm layer. The hidden unit counts of the convolution layers in order are 32, 64, 128. After these three blocks there is a flatten layer, followed by a dense layer with 256 hidden units, then a batch norm, a 50% dropout layer, then the final output layer which is a dense layer with 10 hidden units. All activations are ReLU except for the output layer's, which is a softmax.

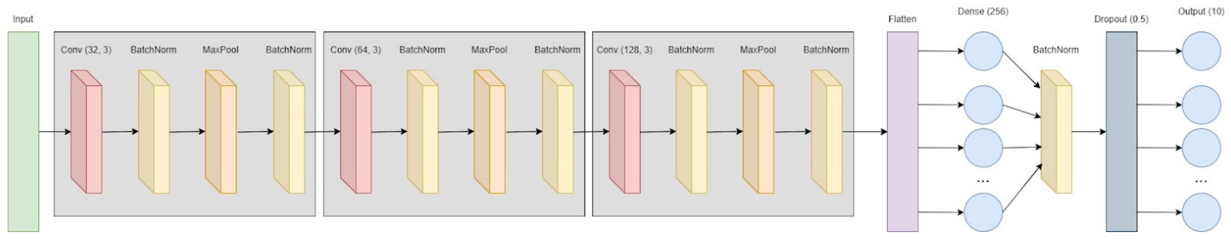


Figure 1: CNN architecture.

Figure 2 shows the workflow of the project. The first four blocks of the first row represent the construction of the initial CNN, done on the Kaggle version of the dataset. The first three blocks of the second row represent the generation of the GADF and GASF files. The right section of blocks (after the diamond-shaped blocks) are the training, testing, comparison and evaluation of full models trained on the raw audio, GADF, and GASF files.

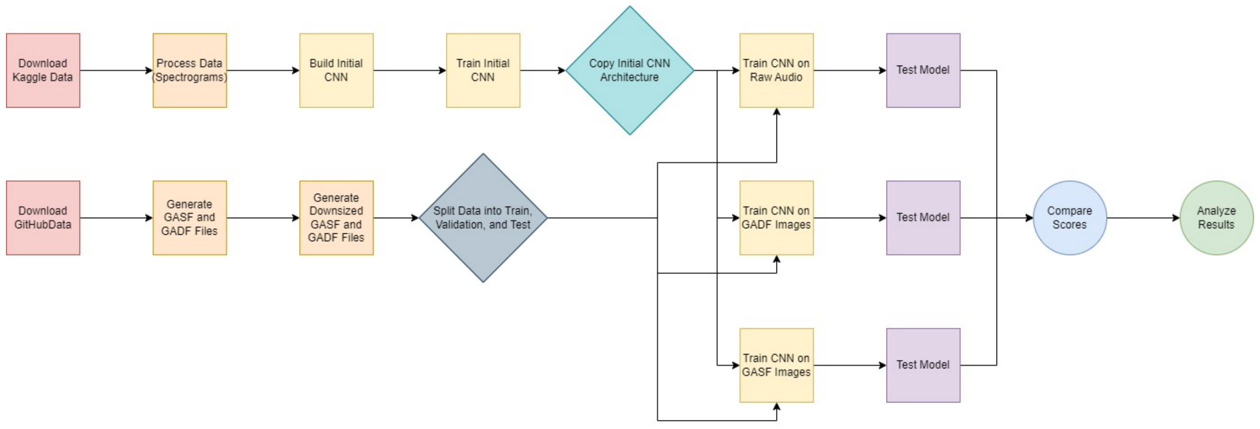


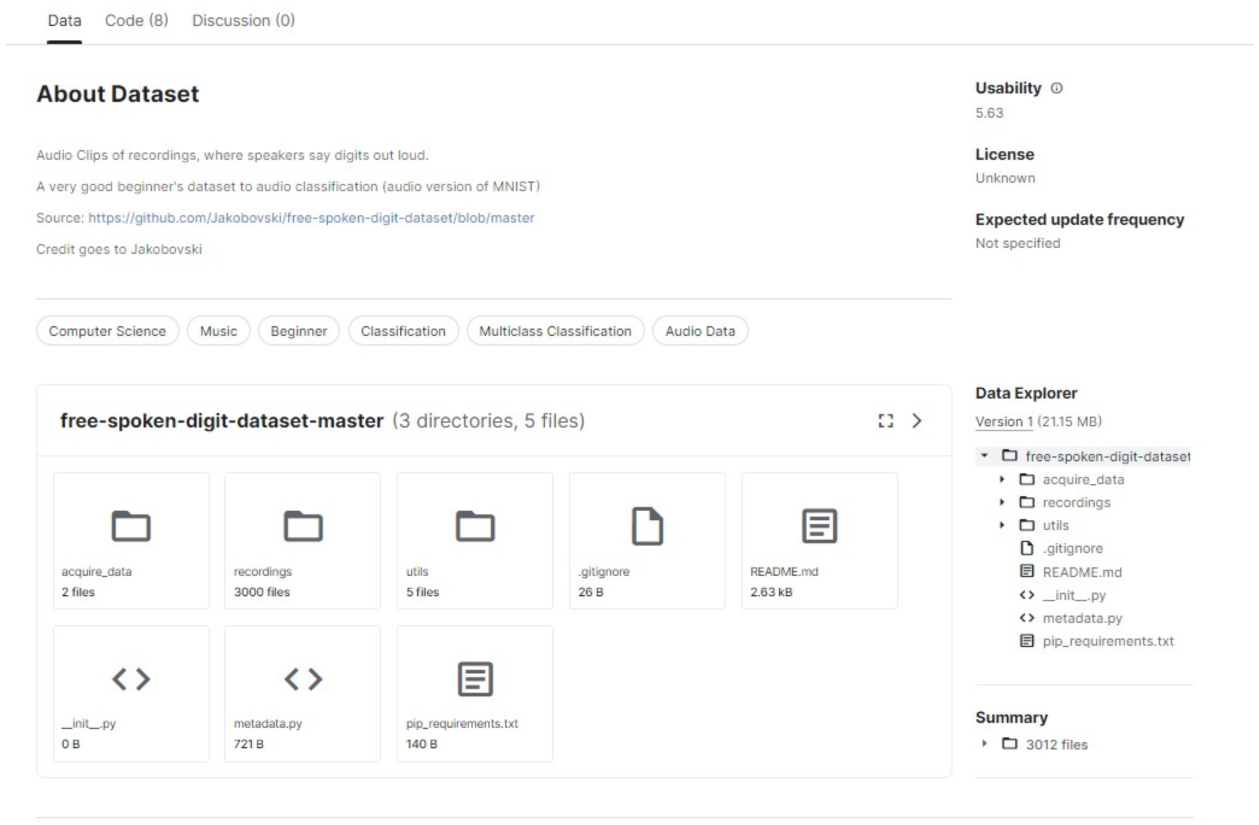
Figure 2: Project workflow.

4 Dataset

The Audio MNIST dataset consists of audio recordings of individuals saying numbers in the zero to nine range in English. There are two versions of this dataset that we use, one on Kaggle and one on GitHub.

4.1 Kaggle

There are six individuals speaking, with 50 recordings of each digit being spoken for each individual for a total of 3,000 recordings. These recordings are .wav files. This dataset was used to build the initial CNN classifier due to its smaller size, and is not the final dataset.


Data Code (8) Discussion (0)

About Dataset

Audio Clips of recordings, where speakers say digits out loud.

A very good beginner's dataset to audio classification (audio version of MNIST)

Source: https://github.com/Jakobovski/free-spoken-digit-dataset/blob/master

Credit goes to Jakobovski

Computer Science | Music | Beginner | Classification | Multiclass Classification | Audio Data

free-spoken-digit-dataset-master (3 directories, 5 files)

acquire_data 2 files	recordings 3000 files	utils 5 files	.gitignore 26 B	README.md 2.63 kB
__init__.py 0 B	metadata.py 721 B	pip_requirements.txt 140 B		

Usability 5.63

License Unknown

Expected update frequency Not specified

Data Explorer

Version 1 (21.15 MB)

```
free-spoken-digit-dataset
  acquire_data
  recordings
  utils
  .gitignore
  README.md
  __init__.py
  metadata.py
  pip_requirements.txt
```

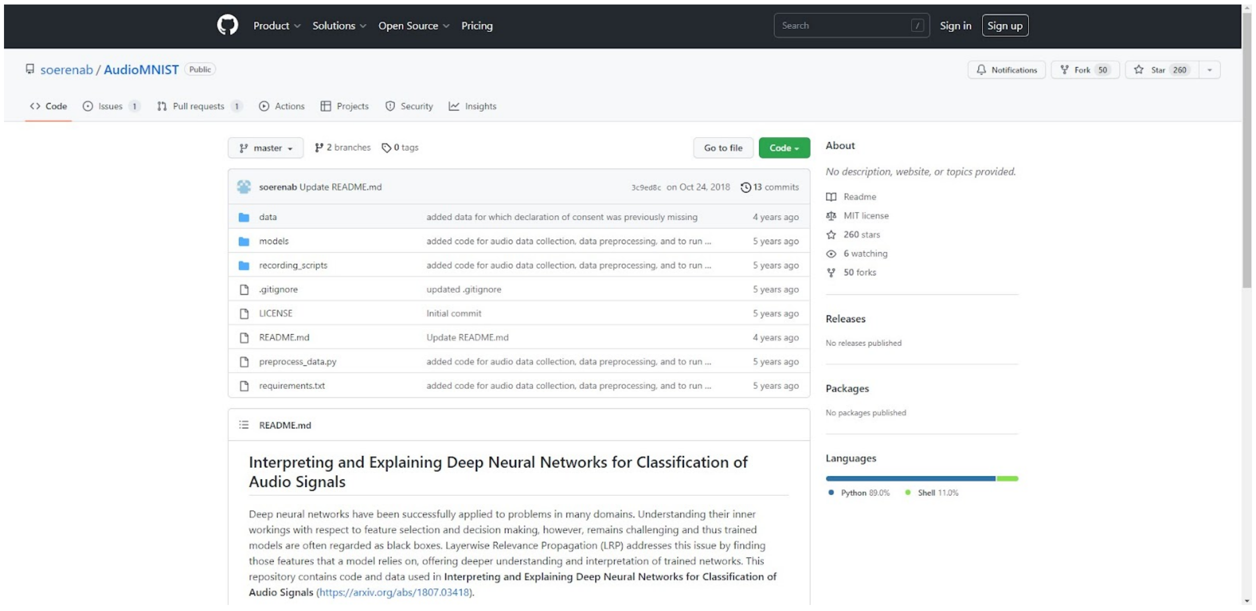
Summary

3012 files

Figure 3: Kaggle dataset webpage.

4.2 GitHub

There are sixty individuals speaking, with 50 recordings of each digit being spoken for each individual for a total of 30,000 recordings. These recordings are .wav files. This is the dataset that the GAF images are generated from and final training of the model(s) is done on.



soerenab / AudioMNIST Public

2 branches 0 tags

soerenab Update README.md 3c9ed8c on Oct 24, 2018 13 commits

data	added data for which declaration of consent was previously missing	4 years ago
models	added code for audio data collection, data preprocessing, and to run ...	5 years ago
recording_scripts	added code for audio data collection, data preprocessing, and to run ...	5 years ago
.gitignore	updated .gitignore	5 years ago
LICENSE	Initial commit	5 years ago
README.md	Update README.md	4 years ago
preprocess_data.py	added code for audio data collection, data preprocessing, and to run ...	5 years ago
requirements.txt	added code for audio data collection, data preprocessing, and to run ...	5 years ago

README.md

Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals

Deep neural networks have been successfully applied to problems in many domains. Understanding their inner workings with respect to feature selection and decision making, however, remains challenging and thus trained models are often regarded as black boxes. Layerwise Relevance Propagation (LRP) addresses this issue by finding those features that a model relies on, offering deeper understanding and interpretation of trained networks. This repository contains code and data used in **Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals** (https://arxiv.org/abs/1807.03418).

About

No description, website, or topics provided.

Readme
MIT license
260 stars
6 watching
50 forks

Releases

No releases published

Packages

No packages published

Languages

● Python 89.0% ● Shell 11.0%

Figure 4: GitHub dataset webpage.

5 Analysis of data

5.1 Data Preprocessing

5.1.1 Kaggle Data

Each spectrogram was normalized to the 0 to 1 range. Some of the spectrograms were randomly flipped along an axis, and some were randomly rotated. The spectrograms that were rotated or flipped were not replaced with the original scaled spectrogram.

The data was split into an 80/20 ratio of training and validation data, respectively. No testing data was set aside.

5.1.2 GitHub Data

For each sound file in the GitHub version of the dataset, both GADF and GASF images are generated from each file. Initially image files are saved at a high resolution (about 4,000 x 4,000 pixels). However, the size required too many parameters for a ML model and caused out-of-memory crashes, so for training purposes downsized resolution (192 x 192 pixels) versions of these images were used for training. The GADF and GASF downsized files are not of uniform sizes. To compensate, all images with side lengths over 192 were discarded and all images with side lengths under 192 were padded with 0, with the image values being centered in the padding.

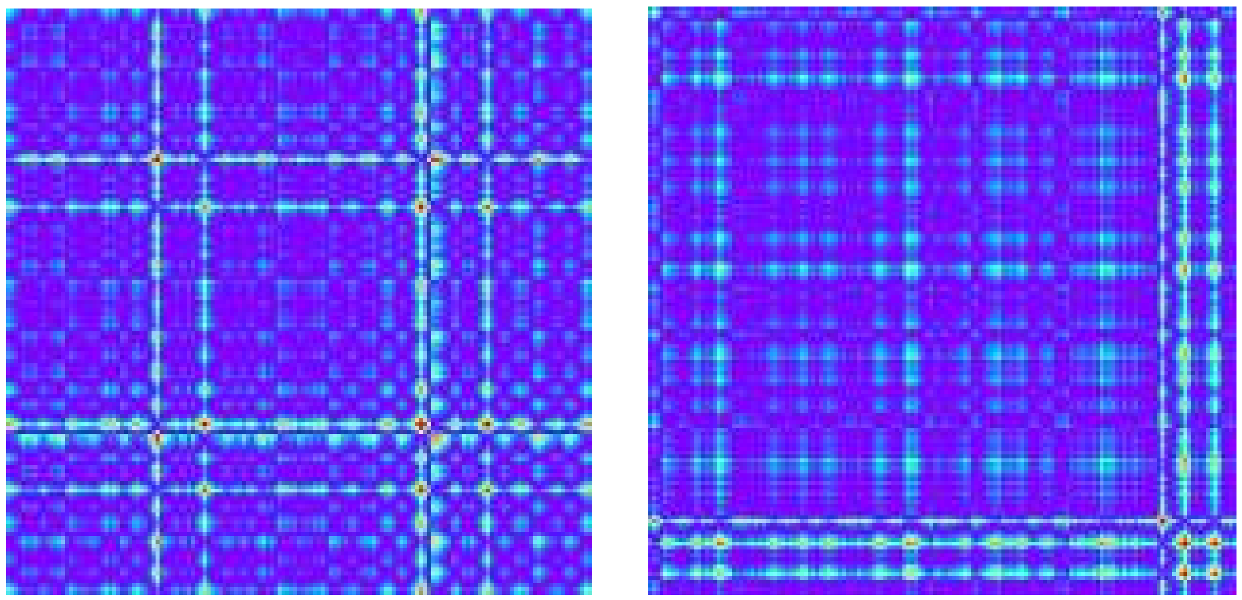


Figure 5: GAF images of a speaker saying “zero”.

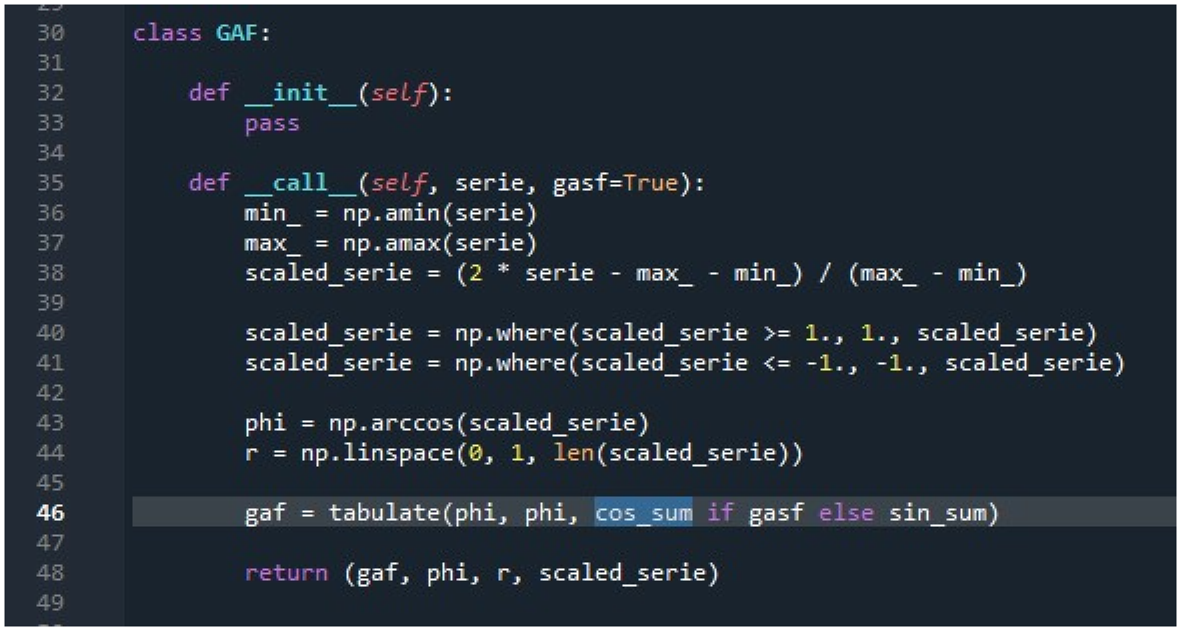
The raw audio values were not of uniform size and were in a 1D array. To compensate, the arrays were padded with 0 to reach a total length of 48,400, again with the array value being centered. The arrays were then reshaped into 220 x 220 matrices. Finally the matrices were normalized so that all values were between 0 and 1, as there were negative values present in the data.

The dataset split was done in an 80/10/10 for training, validation, and testing sets, respectively. Splits were done by file name so as to maintain identical distributions of file location, i.e. generated files going into the same set as the parent file.

6 Implementation

6.1 GAF Generation

The GAF file generation code uses custom code to generate the GAF values, with matplotlib, scipy, pandas, and numpy as supporting libraries for loading the audio signal and plotting the GAF values. Both GASF and GADF values are computed.



```
class GAF:

    def __init__(self):
        pass

    def __call__(self, serie, gasf=True):
        min_ = np.amin(serie)
        max_ = np.amax(serie)
        scaled_serie = (2 * serie - max_ - min_) / (max_ - min_)

        scaled_serie = np.where(scaled_serie >= 1., 1., scaled_serie)
        scaled_serie = np.where(scaled_serie <= -1., -1., scaled_serie)

        phi = np.arccos(scaled_serie)
        r = np.linspace(0, 1, len(scaled_serie))

        gaf = tabulate(phi, phi, cos_sum if gasf else sin_sum)

        return (gaf, phi, r, scaled_serie)
```

Figure 6: GAF generation class.

6.2 Initial CNN

For the initial CNN classifier, each audio file had a spectrogram created from it via the raw audio values and framerate using pylab. Each spectrogram was saved as a .png file of the dimensions 256 by 256. Each spectrogram was normalized to the 0 to 1 range. Some of the spectrograms were randomly flipped along an axis, and some were randomly rotated. The spectrograms that were rotated or flipped were not replaced with the original scaled spectrogram.

This model was neither created nor run during this increment. However, significant portions were copied to create the full CNN, so we have included a section for it here.

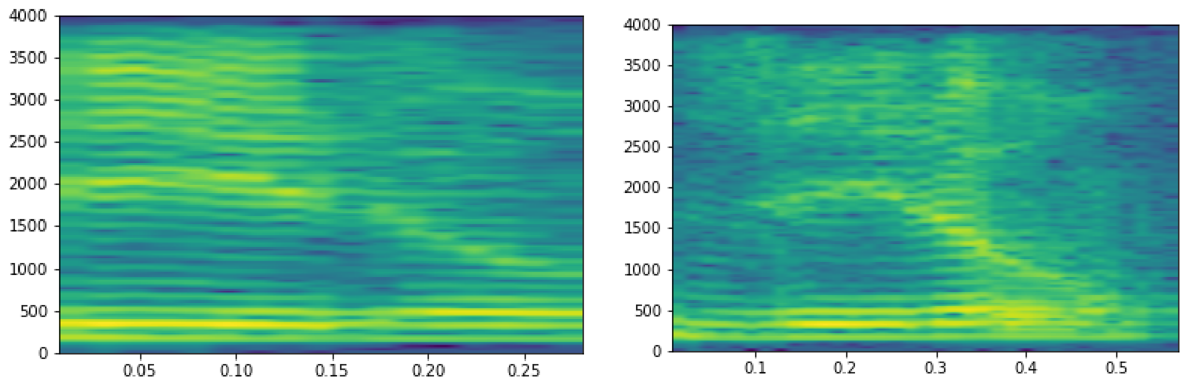
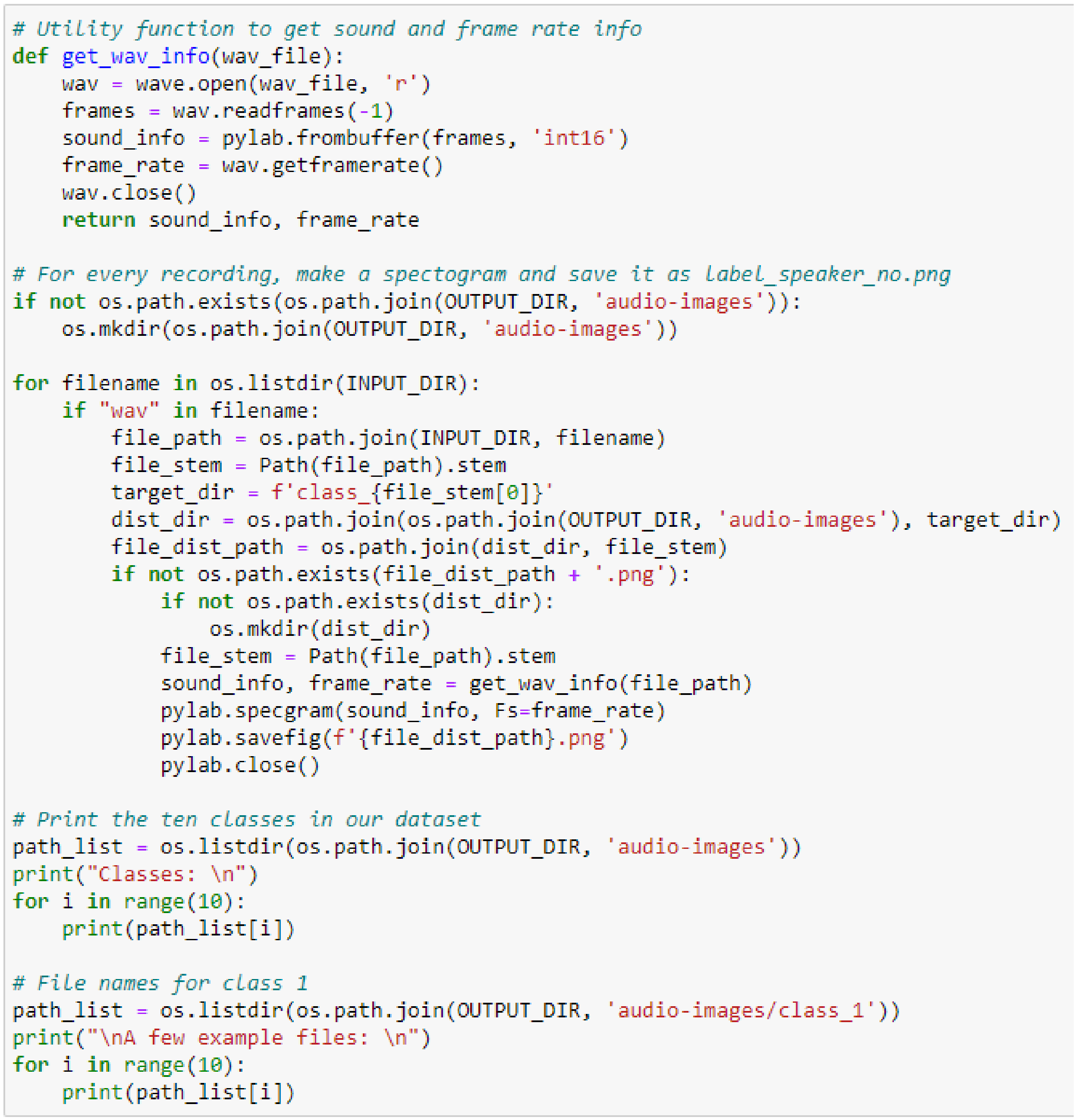


Figure 7: Spectrograms of a speaker saying “zero”.



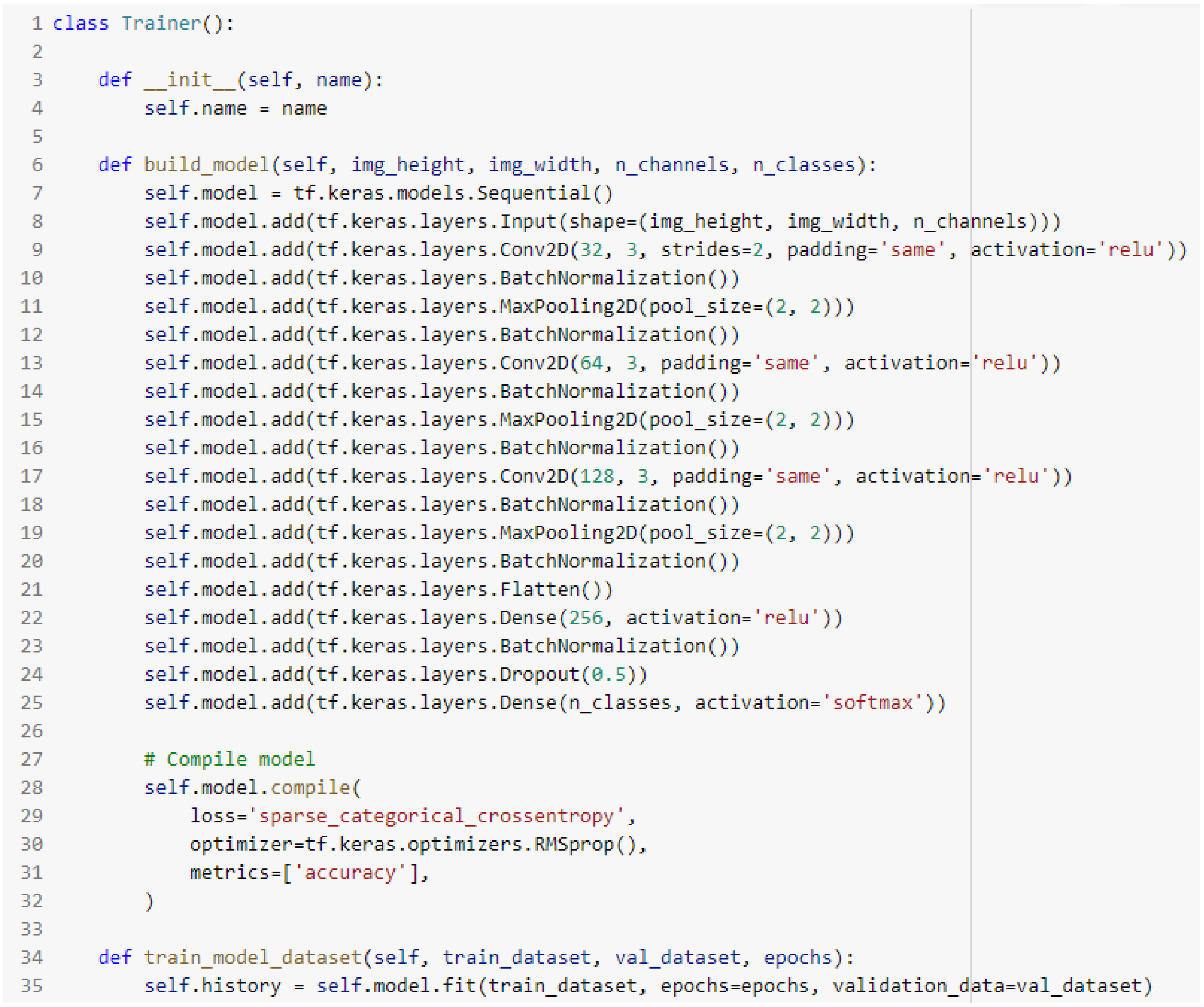
```python
# Utility function to get sound and frame rate info
def get_wav_info(wav_file):
    wav = wave.open(wav_file, 'r')
    frames = wav.readframes(-1)
    sound_info = pylab.frombuffer(frames, 'int16')
    frame_rate = wav.getframerate()
    wav.close()
    return sound_info, frame_rate

# For every recording, make a spectogram and save it as label_speaker_no.png
if not os.path.exists(os.path.join(OUTPUT_DIR, 'audio-images')):
    os.mkdir(os.path.join(OUTPUT_DIR, 'audio-images'))

for filename in os.listdir(INPUT_DIR):
    if "wav" in filename:
        file_path = os.path.join(INPUT_DIR, filename)
        file_stem = Path(file_path).stem
        target_dir = f'class_{file_stem[0]}'
        dist_dir = os.path.join(os.path.join(OUTPUT_DIR, 'audio-images'), target_dir)
        file_dist_path = os.path.join(dist_dir, file_stem)
        if not os.path.exists(file_dist_path + '.png'):
            if not os.path.exists(dist_dir):
                os.mkdir(dist_dir)
            file_stem = Path(file_path).stem
            sound_info, frame_rate = get_wav_info(file_path)
            pylab.specgram(sound_info, Fs=frame_rate)
            pylab.savefig(f'{file_dist_path}.png')
            pylab.close()

# Print the ten classes in our dataset
path_list = os.listdir(os.path.join(OUTPUT_DIR, 'audio-images'))
print("Classes: \n")
for i in range(10):
    print(path_list[i])

# File names for class 1
path_list = os.listdir(os.path.join(OUTPUT_DIR, 'audio-images/class_1'))
print("\nA few example files: \n")
for i in range(10):
    print(path_list[i])
```

Figure 8: Spectrogram generation code.

6.3 Full CNNs

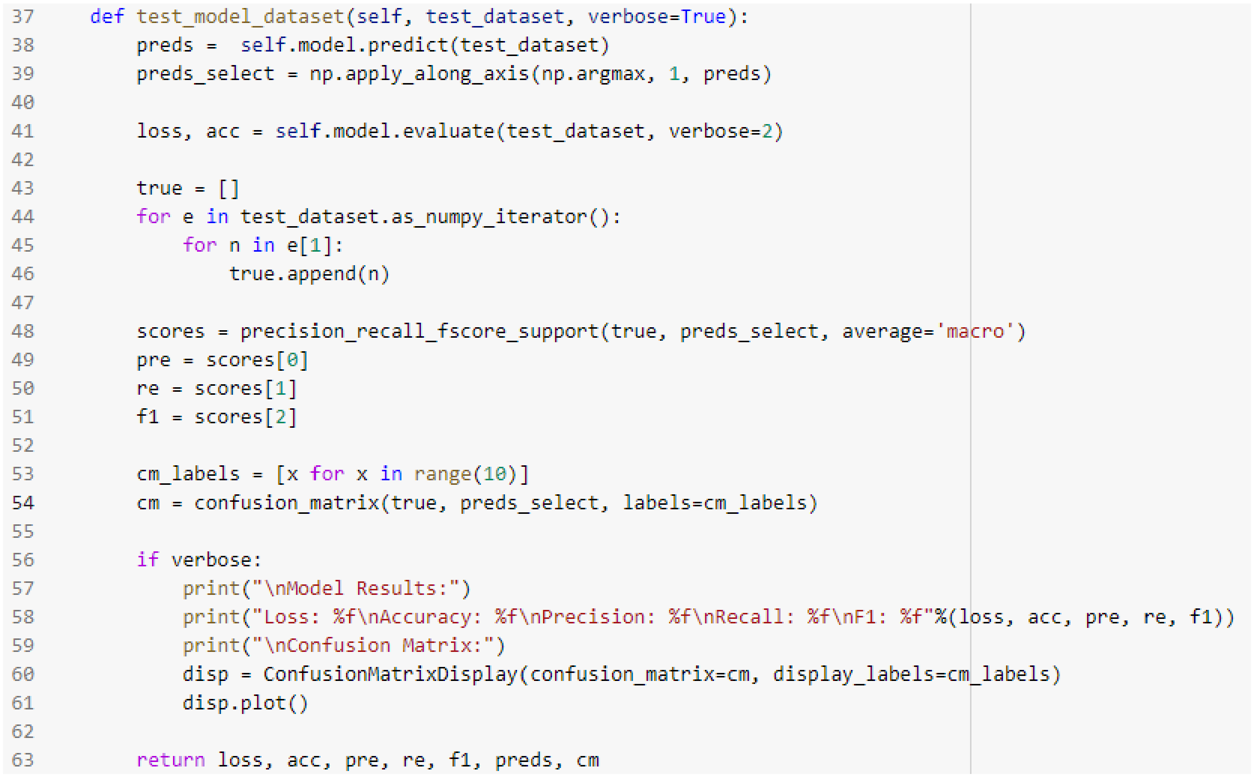
6.3.1 CNN Architecture

The full CNN maintained the same architecture and hidden unit counts as the initial CNN, only changing the input shapes to account for the different shaped inputs (Figure 9). Due to memory constraints, all data is loaded into Tensorflow Datasets via generators that only open files 10 at a time. All models are trained for 10 epochs. The model is constructed and trained via Tensorflow, with precision, recall, and F1 scores being computed using scikit-learn’s evaluation functions (Figure 10).



```
class Trainer():

    def __init__(self, name):
        self.name = name

    def build_model(self, img_height, img_width, n_channels, n_classes):
        self.model = tf.keras.models.Sequential()
        self.model.add(tf.keras.layers.Input(shape=(img_height, img_width, n_channels)))
        self.model.add(tf.keras.layers.Conv2D(32, 3, strides=2, padding='same', activation='relu'))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu'))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.Conv2D(128, 3, padding='same', activation='relu'))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.Flatten())
        self.model.add(tf.keras.layers.Dense(256, activation='relu'))
        self.model.add(tf.keras.layers.BatchNormalization())
        self.model.add(tf.keras.layers.Dropout(0.5))
        self.model.add(tf.keras.layers.Dense(n_classes, activation='softmax'))

        # Compile model
        self.model.compile(
            loss='sparse_categorical_crossentropy',
            optimizer=tf.keras.optimizers.RMSprop(),
            metrics=['accuracy'],
        )

    def train_model_dataset(self, train_dataset, val_dataset, epochs):
        self.history = self.model.fit(train_dataset, epochs=epochs, validation_data=val_dataset)
```

Figure 9: Full CNN class. Architecture construction and training only.



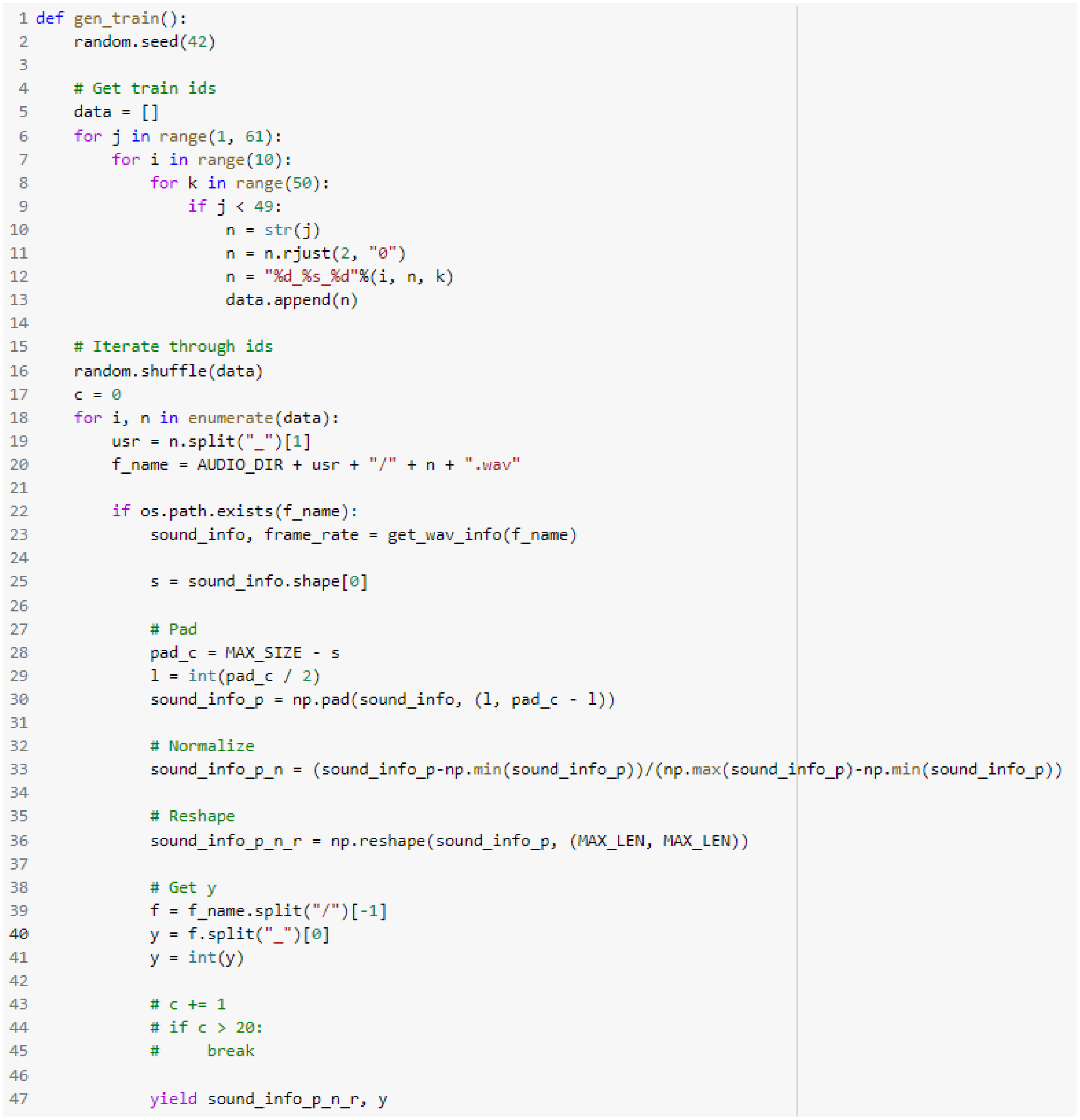
```python
    def test_model_dataset(self, test_dataset, verbose=True):
        preds =  self.model.predict(test_dataset)
        preds_select = np.apply_along_axis(np.argmax, 1, preds)

        loss, acc = self.model.evaluate(test_dataset, verbose=2)

        true = []
        for e in test_dataset.as_numpy_iterator():
            for n in e[1]:
                true.append(n)

        scores = precision_recall_fscore_support(true, preds_select, average='macro')
        pre = scores[0]
        re = scores[1]
        f1 = scores[2]

        cm_labels = [x for x in range(10)]
        cm = confusion_matrix(true, preds_select, labels=cm_labels)

        if verbose:
            print("\nModel Results:")
            print("Loss: %f\nAccuracy: %f\nPrecision: %f\nRecall: %f\nF1: %f"%(loss, acc, pre, re, f1))
            print("\nConfusion Matrix:")
            disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=cm_labels)
            disp.plot()

        return loss, acc, pre, re, f1, preds, cm
```

Figure 10: Full CNN testing code.

6.3.2 Generators

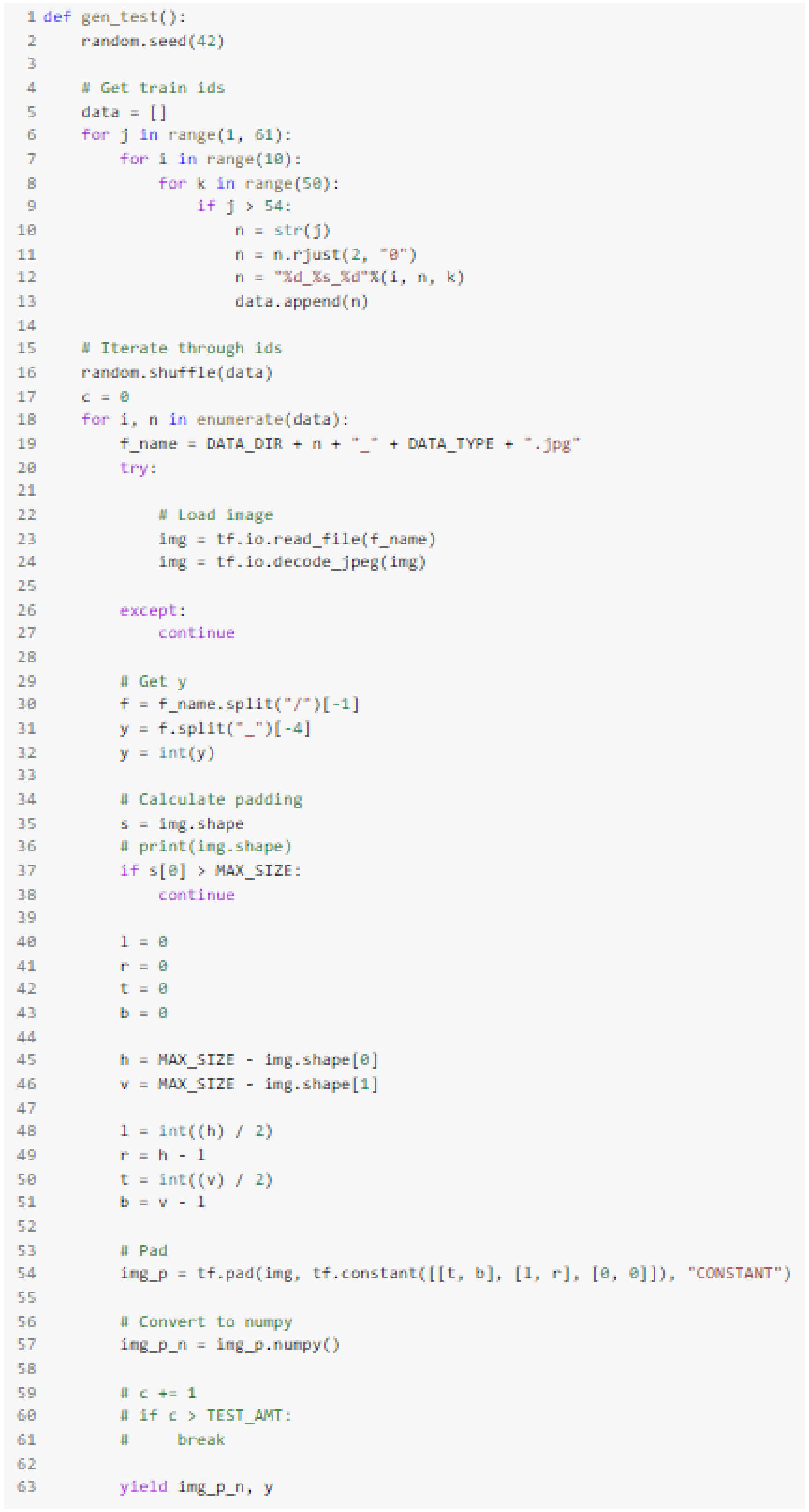
The raw audio, and GAF files all have different generators to handle preprocessing needs (padding, normalization, specific file requirements, etc.). Both the GASF and GADF files can use the same generator due to the method of file generation and file similarity.

For each file type three generators were constructed, a train, a validation, and a test generator. The only differences between each generator within a file type was which files are loaded. Different files are loaded into different sets in order to perform subject-wise separation between the sets. Both the validation and test set have 6 speakers in them that do not appear in any of the other sets. The train set has all of the other speakers.



```python
def gen_train():
    random.seed(42)

    # Get train ids
    data = []
    for j in range(1, 61):
        for i in range(10):
            for k in range(50):
                if j < 49:
                    n = str(j)
                    n = n.rjust(2, "0")
                    n = "%d_%s_%d"%(i, n, k)
                    data.append(n)

    # Iterate through ids
    random.shuffle(data)
    c = 0
    for i, n in enumerate(data):
        usr = n.split("_")[1]
        f_name = AUDIO_DIR + usr + "/" + n + ".wav"

        if os.path.exists(f_name):
            sound_info, frame_rate = get_wav_info(f_name)

            s = sound_info.shape[0]

            # Pad
            pad_c = MAX_SIZE - s
            l = int(pad_c / 2)
            sound_info_p = np.pad(sound_info, (l, pad_c - l))

            # Normalize
            sound_info_p_n = (sound_info_p-np.min(sound_info_p))/(np.max(sound_info_p)-np.min(sound_info_p))

            # Reshape
            sound_info_p_n_r = np.reshape(sound_info_p, (MAX_LEN, MAX_LEN))

            # Get y
            f = f_name.split("/")[-1]
            y = f.split("_")[0]
            y = int(y)

            # c += 1
            # if c > 20:
            #     break

            yield sound_info_p_n_r, y
```

Figure 11: Training generator for raw audio data.



```python
def gen_test():
    random.seed(42)

    # Get train ids
    data = []
    for j in range(1, 61):
        for i in range(10):
            for k in range(50):
                if j > 54:
                    n = str(j)
                    n = n.rjust(2, "0")
                    n = "%d_%s_%d"%(i, n, k)
                    data.append(n)

    # Iterate through ids
    random.shuffle(data)
    c = 0
    for i, n in enumerate(data):
        f_name = DATA_DIR + n + "_" + DATA_TYPE + ".jpg"
        try:

            # Load image
            img = tf.io.read_file(f_name)
            img = tf.io.decode_jpeg(img)

        except:
            continue

        # Get y
        f = f_name.split("/")[-1]
        y = f.split("_")[-4]
        y = int(y)

        # Calculate padding
        s = img.shape
        # print(img.shape)
        if s[0] > MAX_SIZE:
            continue

        l = 0
        r = 0
        t = 0
        b = 0

        h = MAX_SIZE - img.shape[0]
        v = MAX_SIZE - img.shape[1]

        l = int((h) / 2)
        r = h - l
        t = int((v) / 2)
        b = v - l

        # Pad
        img_p = tf.pad(img, tf.constant([[t, b], [l, r], [0, 0]]), "CONSTANT")

        # Convert to numpy
        img_p_n = img_p.numpy()

        # c += 1
        # if c > TEST_AMT:
        #     break

        yield img_p_n, y
```

Figure 12: GAF test generator.

7 Results

7.1 GAF Generation

GAF images are generated as 6000 x 6000 pixel images from original waveform. The generation and training of these large images is costly (required excessive CPU time to simply generate these images). Moreover, the overall ability to use these for training tended to be slow (due to the bulky size). GAF's were then modified to be greatly reduced (200 x 200 pixel images). Although these were able to be used for training, the results were unexpected and disappointing.

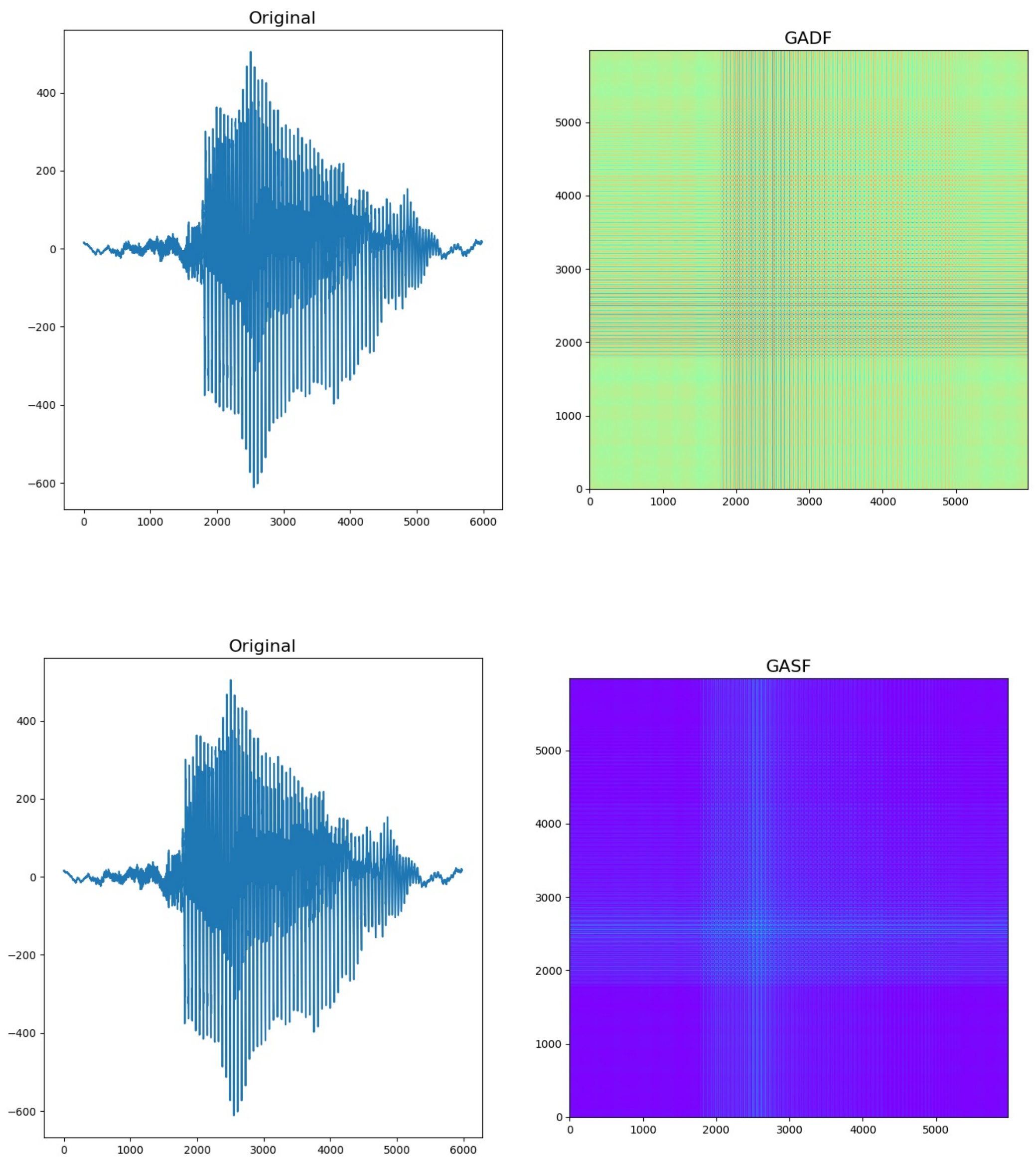


Figure 13: GADF and GASF images with the original waveform.

7.2 GAF Reverse Conversion

Inverse GAF functions operate on the diagonal of the GAF images. GAFs are square matrices (M=N). The length of the diagonal is the same as the number of sample periods. The original waveform can be reproduced, but at a cost that the diagonal contains sufficient length for the regeneration.

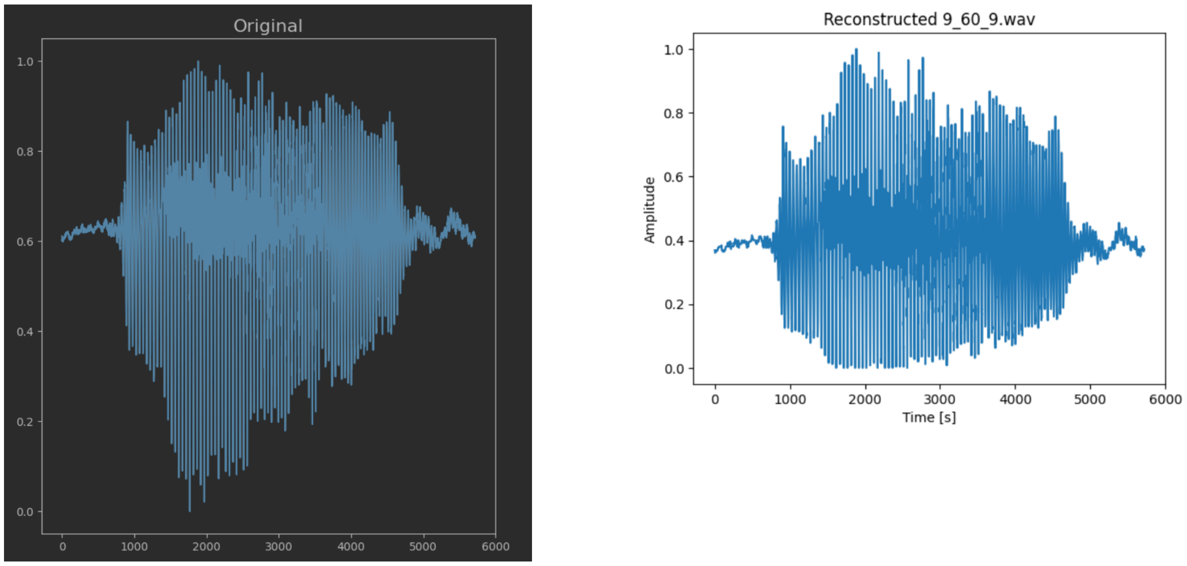


Figure 14: An original waveform and the waveform reconstructed from a GAF image.

7.3 Initial CNN Classifier

The initial classifier trained for 10 epochs with training and validation. The final and best scores are listed in Table 1.

	Loss	**Accuracy**	**Val. Loss**	**Val. Accuracy**
Last Epoch	0.0271	0.9946	0.2764	0.9033
Best Epoch (epoch found)	0.0190 (9)	0.9971 (9)	0.1410 (8)	0.9617 (8)

Table 1: Results of initial classifier for training.

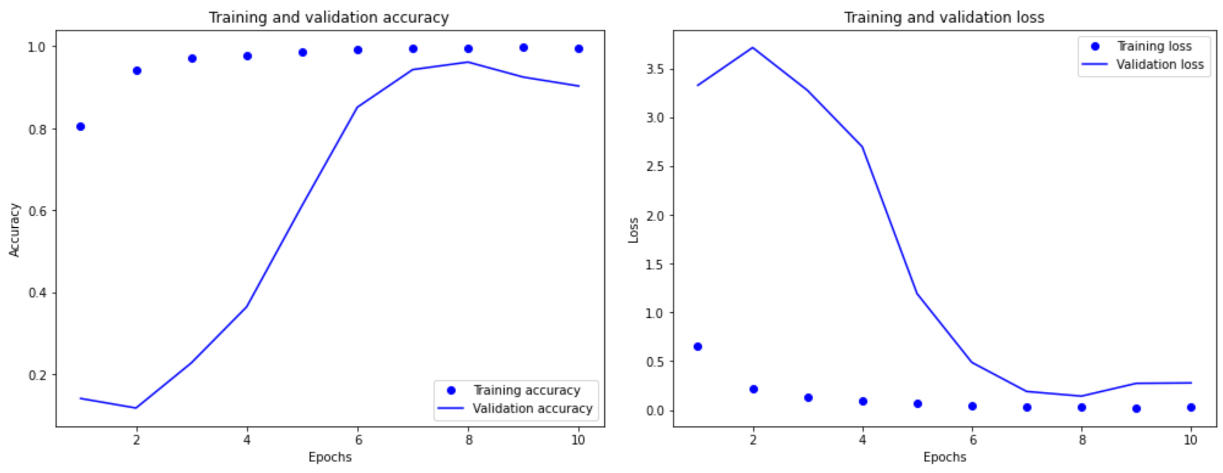


Figure 15: Initial classifier accuracy and loss.

7.4 Full CNN Classifier

All results are achieved via training for 10 epochs with identical hyperparameters aside from input shape.

7.4.1 Results on Raw Audio

The scores for the classifier trained on the raw audio data achieve final accuracies of 0.9617 and 0.8097 for training and validation scores, respectively. These are also the best of the training scores. The final test scores are an accuracy of 0.834333, precision of 0.844516, recall of 0.834333, and F1 of 0.833827.

Looking at Figure 17 we can see minor misclassification rates in most cases, with only a few labels having notable misclassification counts.

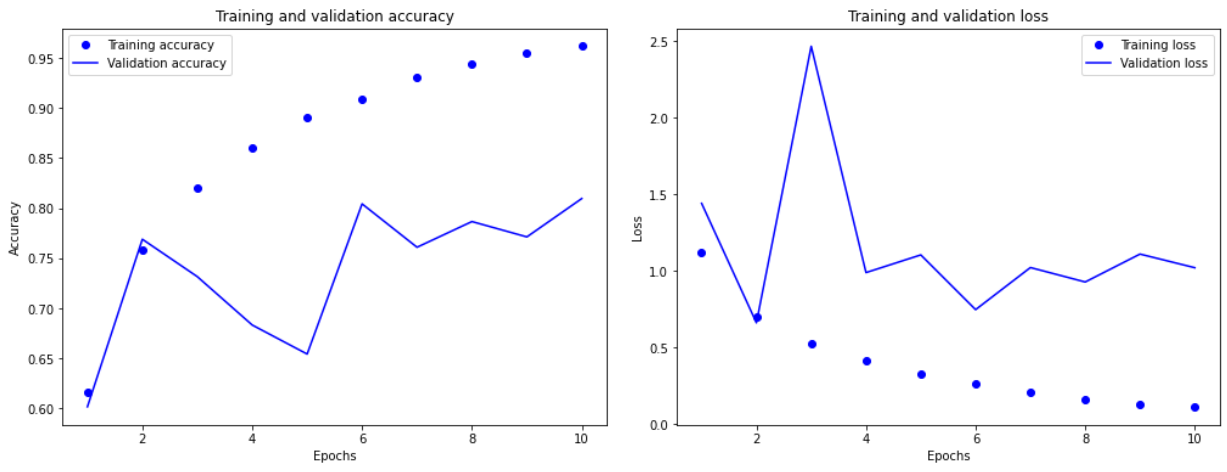


Figure 16: Full classifier accuracy (left) and loss (right) on raw audio input.

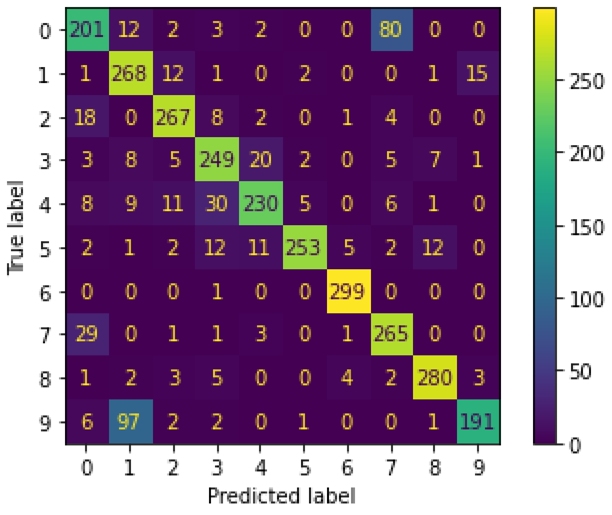


Figure 17: Full classifier confusion matrix on raw audio input.

7.4.2 Results on GADF Images

The final accuracy from training was 0.3930, with a validation accuracy of 0.2859. At epoch 6 there is a higher validation accuracy of 0.2956, but overall scores were both relatively consistent as well as low. The final testing scores were accuracy: 0.202128, precision: 0.221538, recall: 0.189075, and F1: 0.158970.

The Figure 19 shows strong misclassification tendencies, particularly for higher value digits. Overall this model performed very poorly in comparison to the raw audio classifier.

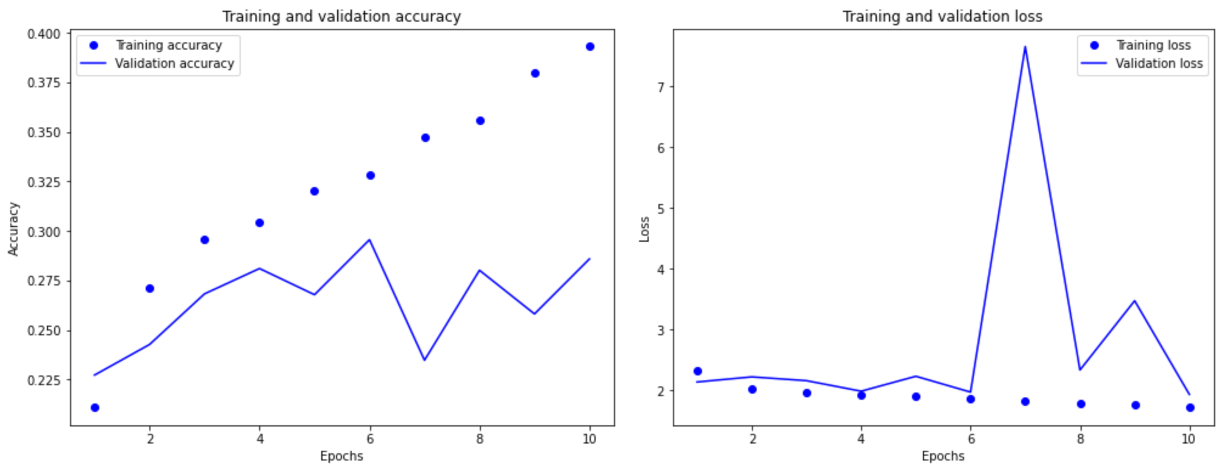
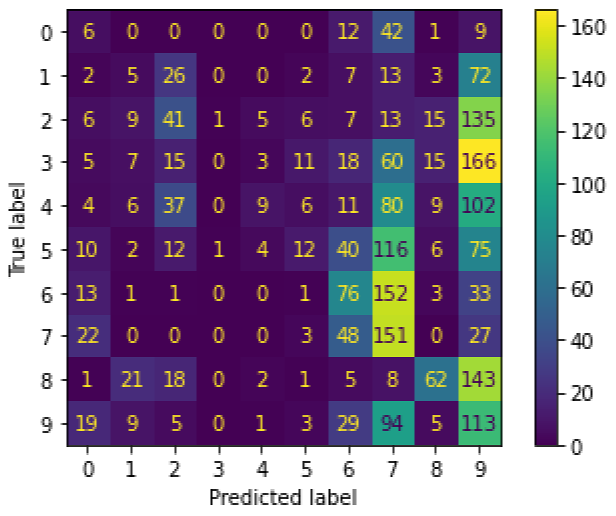


Figure 18: Full classifier accuracy (left) and loss (right) on GADF image input.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	6	0	0	0	0	0	12	42	1	9
1	2	5	26	0	0	2	7	13	3	72
2	6	9	41	1	5	6	7	13	15	135
3	5	7	15	0	3	11	18	60	15	166
4	4	6	37	0	9	6	11	80	9	102
5	10	2	12	1	4	12	40	116	6	75
6	13	1	1	0	0	1	76	152	3	33
7	22	0	0	0	0	3	48	151	0	27
8	1	21	18	0	2	1	5	8	62	143
9	19	9	5	0	1	3	29	94	5	113

Figure 19: Full classifier confusion matrix on GADF image input.

7.4.3 Results on GASF Images

The final training and validation accuracies were 0.3819 and 0.2753, respectively. The best validation accuracy actually occurred on epoch 5, with a value of 0.2991. The final testing scores ended up being accuracy: 0.222881, precision: 0.218572, recall: 0.206050, F1: 0.190465. These scores, while being slightly better than the GADF scores, are not notably better and still show horrible performance when compared to the raw audio scores.

Figure 21 again shows a strong pattern of misclassification for the model, favoring higher-value digits.

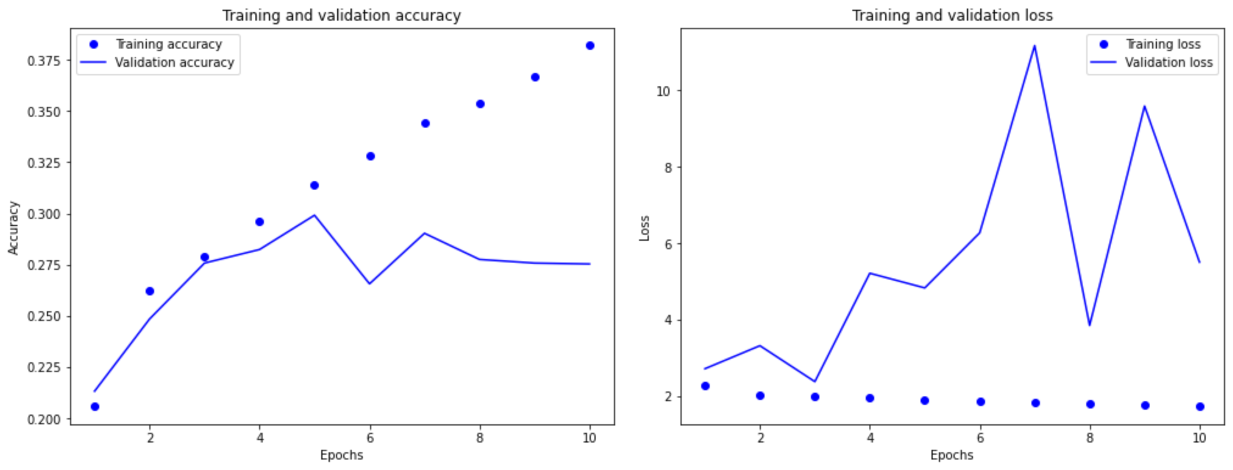


Figure 20: Full classifier accuracy (left) and loss (right) on GASF image input.

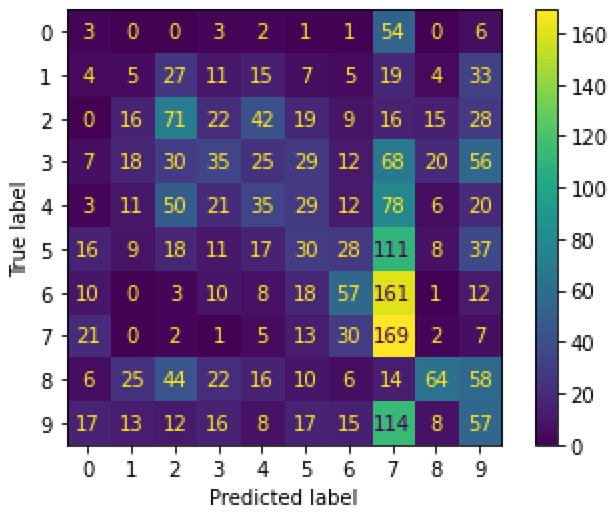


Figure 21: Full classifier confusion matrix on GASF image input.

7.4.4 Comparison of Results

Table 2 displays the testing scores obtained after ten epochs of training. The scores of the classifiers trained on the GAF images are horrible. Given the decent scores achieved by the classifier trained on the raw audio, this wild disparity in score values indicate that the classifiers are not able to identify relevant information contained within the GAF images.

It's possible that scores would improve with further training. The trends of the learning curves are vaguely encouraging. However, it would likely take significantly more training. Given the time and storage constraints, this is impractical and could not be done for this increment.

The massive underperformance of the GAF image classifiers was a startling result. While it would not have been surprising for the GAF classifiers to perform somewhat worse, the massive gap was both startling and intriguing. If more time was available, we would have liked to investigate this.

	Accuracy	**Precision**	**Recall**	**F1**
Raw Audio	**0.8343**	**0.8445**	**0.8343**	**0.8338**
GADF	0.2021	0.2215	0.1891	0.1590
GASF	0.2229	0.2186	0.2061	0.1905

Table 2: Comparison of testing scores.

8 Project Management

8.1 Implementation status report

8.1.1 Work completed

8.1.1.1 Description

Two datasets, a construction and a full dataset, were acquired.

The GADF and GASF files have been generated at full size.

The GADF and GASF files have a reduced-size version of each.

The GADF and GASF files have been converted back to the original data format.

Further necessary data preprocessing was done.

An initial CNN classifier was constructed and trained.

A full CNN classifier was constructed and trained on: raw audio data, GADF images, GASF images.

8.1.1.2 Responsibility

Andrew Fausak:

- Generated GAF files.
- Generated reduced-size GAF files.
- Converted GAF files back to audio data.
- Worked on the project proposal.
- Worked on the project update.
- Worked on the final report.
- Made a demo video.
- Recorded final presentation.

Andy Fausak:

- Generated GAF files.
- Generated reduced-size GAF files.
- Converted GAF files back to audio data.
- Worked on the project proposal.
- Worked on the project update.
- Worked on the final report.
- Made a demo video.
- Recorded final presentation.

Andre Sharp:

- Generated spectrograms.

- Made initial classifier.
- Worked on the final classifier.
- Worked on the project proposal.
- Worked on the project update.
- Worked on the final report.
- Made a demo video.
- Recorded final presentation.

Mica Haney:

- Preprocessed data files.
- Made generators.
- Worked on the final classifier.
- Worked on the project proposal.
- Worked on the project update.
- Worked on the final report.
- Made a demo video.
- Recorded final presentation.

8.1.1.3 Contributions

Andrew Fausak: 25%

Andy Fausak: 25%

Andre Sharp: 25%

Mica Haney: 25%

8.1.1.4 Issues/Concerns

We were not able to run as many experiments as we wanted to due to memory issues. The large file sizes often crashed whatever runtime was being used, trying to compile a model that can handle that size input crashed runtimes, and our cloud storage (Google Drive) often would time out and prevent models from accessing the data with no explained reason. The number of experiments we could run and use to compare the features and approaches was disappointingly small.

There were no comments on our previous increment to address.

9 References

1. Shah, S. K., Tariq, Z., Lee, J., & Lee, Y. (2021). Event-Driven Deep Learning for Edge Intelligence (EDL-EI). *Sensors, 21*(18), 6023.

2. Wang, Z., & Oates, T. (2015, June). Imaging time-series to improve classification and imputation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.