Exposing AI Audio: How Spectrograms and LSTM Can Detect Deepfakes

Matthew Perez
College of Information Science & Technology
University of Nebraska at Omaha
Omaha, United States of America
0009-0003-3760-5011

***Abstract*—It is the belief of this paper that audio generated mostly, or even entirely, using new Generative AI models poses a major threat to the security of organizations internationally. The opportunities for malicious phishing campaigns seem especially dangerous. Therefore, it would be worthwhile for those within the cybersecurity space to treat this threat seriously, working to try and find any way possible to more easily detect this type of content. This paper introduces a Long-Short Term Memory (LSTM) model implemented in PyTorch to help solve this issue. It has been trained on hours of audio data, including human-made audio, as well as audio put through Neural Network vocoders. The model was able to achieve an overall testing accuracy of 0.941, with an EER of 0.0667, and an AUC of 0.9678.**

***Index Terms*—Artificial Intelligence, Audio Deepfakes, Cybersecurity, Deep Learning, LSTM, MFCC**

I. Introduction & Motivation

In the last decade, it has become increasingly popular for people to create and distribute content made entirely with AI tools. This is, in large part, because modern neural network generators, such as Large Language Models (LLMs) and Generative AI, have improved greatly to the point where they come close to sounding human [1], [2], [3], [4], [5]. However, with this improvement, many threats have come to the cybersecurity space, often in the form of various types of phishing and misinformation. For example, in [3], it was mentioned that hundreds of thousands of Euros were transferred illegally when an executive's voice was copied with AI audio technology. Another source, [5], spoke of an AI-generated video of former President Joe Biden, the content of which involved him telling people in certain counties not to vote. CEOs have been used as bait for malicious cybercriminals to try and extract funds illegally from corporations. Even major cybersecurity firms such as Kaspersky [6] have noted the danger of AI-Generated Audio. Audio and video like this exist within a group of content commonly known as "deepfakes," and show no signs of stopping or slowing in the near future [1], [2], [3], [4], [5], [6], [7].

With these "deepfakes" proliferating so easily now [1], [2], [3], [4], [5], [6], it seems clear that the Cybersecurity field should take steps to try and mitigate the threats this technology poses. Specifically, detection using spectral data, such as Mel-Spectrograms, seems to be a promising method [1], [2], [4], [5]. Additionally, many papers have focused on pre-trained Deep Learning (DL) models, like RawNet2 or variants of ResNet [2], [4], [7], as their backbone. Other researchers have used Machine Learning (ML) algorithms, with Support Vector Machines (SVM) being quite popular [1], [2], [4], [5]. Finally, some of the most common from-scratch DL-type classifiers involve using Convolutional Neural Networks (CNN) [1], [2], [4], [5]. Surprisingly, few papers have focused on techniques involving LSTM, and while some papers have mentioned using MFCC [2], [5], [8], [9], this has either mainly been with ML, or to say that MFCC is a significant downgrade from Mel-Spectrograms with DL networks.

It is the motivation of this paper to show that LSTMs, while certainly experiencing drawbacks, can be used as the starting point for powerfully performing deepfake detection models. Another motivating factor eventually became to prove that MFCC can still lead to strong testing results, and to not so readily disregard it for feature extraction and model training. In pursuit of this, a model was created in PyTorch using a set of two Bi-Directional LSTM layers, with dropout placed after each, and finishing off with a Linear Fully-Connected (FC) layer. The model works to learn features of Neural Network vocoders, a common component in many types of audio deepfakes [3], [6], [7], [10], [11]. It does so by taking MFCC data, processing it, and using it to train the model on its reduced features. The rest of paper's structure will go as follows: a deeper discussion of previous work in this area will be mentioned quickly, which will then lead into an explanation of the dataset, data processing, model architecture, and training strategy used, being concluded with an exploration of the experiment's results.

II. Related Works

Despite being an issue that has only recently garnered major attention, there's already been a decent number of attempts made to classify and detect audio deepfakes [4], [5], [7]. In fact, the major inspiration for this paper's research was one of these attempts [7]. The researchers in that paper utilized raw waveforms of regular human audio, as well as audio of six different Neural Network vocoders, instead of spectrum data like Mel-Spectrograms or MFCC. The model was a modified RawNet2 model with two different Linear FC layers constituting different functions. One was multi-class,

meant to designate between real audio and the specific type of vocoder used. The second was binary, meant to simply designate whether the audio was real or fake. The model was able to achieve an impressive EER of 0.0013 on regular data, as well as an EER of 0.0274 on data that had been processed to be harder to detect, such as adding crowd noises. However, the usage of a common classifying architecture, RawNet2 [2], [4], [7], shows that perhaps all architecture choices have yet been fully explored for the purposes of deepfake detection.

Furthermore, while there is focus on handcrafting DL models for deepfake classification, this has mostly been in the domain of various types of CNN models [1], [2], [4], [12], [13]. For example, in [12], a model with nine convolutional layers, four max pooling layers, ten MFM layers, and numerous batch normalization functions was created to test against data in the ASVspoof 2019 dataset. The model achieved very decent performance, with an EER of 0.0186 for one subset of the dataset, and an ERR of 0.0054 in another subset. However, this paper was written back in 2019, calling its results with today's newer audio deepfake capabilities into question.

Another example of CNN-type DL being used for deepfake detection was found in [13]. The researchers created three different models for the purposes of detecting deepfakes: a regular CNN model, a regular LSTM model, and finally a combined LSTM model. The purpose was to see what kinds of common DL models would work best when it comes to detecting whether video was AI-generated or not. The complete in-depth model architecture of each model was not fully explained, but the authors mentioned the CNN using three convolutional layers with sixteen, thirty-two, and sixty-four filters, a model with a CNN front-end and a two-layer one-hundred-twenty-eight-sized LSTM, and lastly a combined CNN-LSTM model. Additionally, they used a 0.001 learning rate and $1e^{-5}$ weight decay, with an Adam optimizer. Overall, one of the biggest issues was that the accuracy, F1-Score, and AUC were somewhat subpar. The CNN gave an accuracy of 0.755, with an F1-Score of 0.7559 and an AUC of 0.9302. The LSTM gave better accuracy metrics, but only by 0.02 points. The hybrid did the worst by far, losing about 0.10 points in almost every aspect, with an AUC of 0.8860. This shows that, even in those papers that do discuss LSTM and MFCC detection techniques [1], [2], [4], [12], [13], there is still much work to be down to prove that these techniques can actually lead to reasonable performance.

Finally, once again, while machine learning models, such as SVM, *have* been used in the past for deepfake detection, it has been noted in some of the papers [4], [5] that this method of detection was mainly popular around the beginning of the topic's research, and that they haven't scaled and evolved well with the increasing complexity involved in detecting AI-generated content. All in all, it has been shown that many models have either not used MFCC and LSTM, or if they have, they haven't done so with the impressive mid-to-high 90s accuracy, F1-Score, and AUC metrics that others have achieved with different models. Therefore, this paper introduces a classification architecture that *does* achieve these metrics, as well as hopefully serve as a starting point for further discussion and usage of such architecture for the design of deepfake detection models. With this brief description about the background of the subject finished, an explanation of the dataset used, as well as how the data was processed, can now be delved into.

III. Dataset & Data Processing

To begin, the dataset chosen for this experiment was one by the name of LibriSeVoc, which can be found in [14], and was discussed in [7]. It is comprised of hours of human audio (labeled gt), as well as audio in six different classes of AI vocoder, consisting of different vocoding techniques. Before mentioning the different types, it might be best to describe the purpose of a vocoder in the first place. Overall, there are two main types of audio deepfake: Text-to-Speech (TTS) and Voice Cloning (VC) [3], [4], [5], [6], [7]. TTS, as one might imagine, is an audio deepfake that takes text, and somehow transforms it into audio waves that should correspond to the voice of the user's choosing [3], [4], [5]. Meanwhile, VC involves using a base voice, perhaps from the user themselves, and transforming that voice to instead sound like someone else [3], [4], [5], [6], [7]. The main problem with both types is that before the process is done, the output is just audio data. Such data cannot alone create an actual audible representation of another person's voice. This is where the vocoder comes in. Its job is to transform that audio data into actual sound waves, which form actual listenable audio. These vocoders almost always use some sort of artificial intelligence themselves, and as mentioned earlier, are used in both major types of audio deepfake model [3], [4], [5], [6], [7], [10], [11].

In any case, the various exact vocoder techniques can be split first into the category of autoregressive models, including WaveNet and WaveRNN, and which probabilistically decide where to place waves depending on whatever it has learned from previous samples. They are often slower than models in the other two techniques, which could be considered downside compared to some other categories, but they still produce very decent audio comparatively [7]. The second main category is Diffusion, which instead of using a probabilistic model, uses a Markov chain, slowly de-noising the audio until it can be converted into a sample of audio, making a notably worse end-product than the previous category. WaveGrad and Diffwave are the two Diffusion models which the researchers chose to try and be wholistic with their model's detection capabilities, despite being more easily detectable to the human ear. Finally, there are Generative Adversarial Network-based (GAN-based) vocoders, which use DL to generate the correct audio signals with as little noise and signal as possible, trying to make it as similar to the human audio as it can while also making it sound like the user's chosen voice. This last category of vocoder has boasted some of the best quality and speed, including models like Mel-GAN and Parallel Wave-GAN, with Mel-GAN being a standoutin its difficulty being detected [7], [11]. WaveRNN and WaveNet were also standouts for this reason, and will be

discussed more in the results section.

However, while [7] used direct audio waves to feed into RawNet2, this paper takes a different approach. Instead, MFCC spectrogram data was utilized and processed to be fed into the model, with the librosa python library being especially useful for this purpose [15], [16], [17], [18], [19]. Practically speaking, MFCC are very similar to the Mel-Spectrogram. Mel-Spectrograms use all of the audio's features to create a representation of amplitude and frequency over time, doing so in what is called the "Mel-scale [17], [18], [19]." The Mel-scale is incredibly useful to look at, since it takes into account that humans can easily hear differences in lower frequencies, while they have a harder if not impossible time hearing differences in higher frequencies [19]. However, MFCC diverges in that, after creating audio data in the frequency dimension and in Mel-scale, it takes that data and transforms it into a time-based dimension using a function called 'Discrete Cosine Transform (DCT)' [16], [17], [18], [19]. This has the effect of making the overall data much more compact, with library function calls often giving MFCC around forty features to Mel-Spectrogram's one-hundred-twenty. It also tries to ensure that whatever data is left is the most important for feature extraction, hopefully speeding training up and letting the model focus on distinguishing features more so than with the Mel-Spectrogram. Again, this has not been greatly tested with DL models, and when it has, it's often been to say that MFCC performs far worse [2], [5], [8], [9], [16], [17], [18], [19]. It was the hope of this paper's model to use MFCC to help give efficiency and speed to the data processing pipeline, while also achieving the best class distinction possible.

The full data preprocessing pipeline occurred as follows: first, the LibriSeVoc dataset was loaded in, and a subset of the each class's samples was taken (four-hundred-seventy-five for the vocoder classes, and seven-hundred for the human class) [20], [21], [22], [23], [24]. The next step was to try and keep each dataset as equal as possible, since the model uses training, validation, and testing sets. The solution was to use the train_test_split() function from sklearn, which allows the user to split data in a way that is proportional to each class, and was extremely important in trying to ensure that neither the validation nor the testing data viewed an unbalanced portion of each class [25], [26], [27], [28]. Once the data splitting was finished, the training, validation, and testing dataloaders were created with a batch size of sixteen samples per batch, leading to thirty-four validation and testing batches, and one-hundred-fifty-six training batches. After this, there needed to be a way to ensure that extremely high values weren't viewed as overly significant, and extremely low values weren't completely disregarded [20], [21], [22], [29], [30], [31], [32]. This is where scikit-learn's Standard Scaler came in handy [31]. Along with using common NumPy functions like concatenate(), the scaler was fit on a group of a hundred different training samples, hopefully ensuring that the model could understand the data as wholistically as possible [20], [21], [22], [29], [30], [31], [32], [33], [34]. The whole sample was not used, and instead every seventh piece of the sample was used as input in the fitting process. Using NumPy's std() and mean() functions, as well as the scaler of course, a testing method was created to ensure the mean was close to zero, and the standard deviation close to one, with both functions giving successful results on every working model run [29], [30], [31], [32], [33], [34], [35], [36]. Finally, the last data preprocessing step occurs during training, where the sample is turned into an MFCC NumPy ndarray, and that ndarray is then transformed using the scaler [15], [16], [18], [20], [21], [22], [25], [31], [32].

IV. Model Architecture & Training

Now, there is enough background understanding to begin discussing the model's architecture. The overall model was kept simple to try and be as easy to understand as possible [23], [24]. It starts with a batch first Bi-Directional LSTM with one layer, turning the forty MFCC features into two-hundred-fifty-six hidden features [37]. After that, the output is fed into a dropout layer to make fifty percent of neurons inactive during training, with the goal being to curtail any overfitting [23]. Next, another LSTM layer was used, with an input of five-hundred-twelve features, and an output of one-thousand-twenty-four hidden features, adding another fifty-percent dropout afterwards [23], [24], [37]. The input of five-hundred-twelve may seem confusing at first, but this is simply because Bi-Directional LSTM's actually double the output features, making two-hundred-fifty-six turn into five-hundred-twelve. Finally, we use an FC Linear layer to turn the two-thousand-forty-eight features (again, because of Bi-directional LSTM's double feature output) into seven feature outputs, which corresponds to the sample's probability of being each of the seven classes [20], [21], [22], [23], [24], [37]. As for the loss function, numerous types were tested, and Cross Entropy Loss was found to be the best overall, giving the most stable results [38], [39]. Additionally, special weights were given to the loss function to prioritize classes that were noticed to be struggling more during the training process [38], [39], [40]. Assuming the classes are in the order of DiffWave, original audio, MelGAN, Parallel Wave-GAN, WaveGrad, WaveNet, and WaveRNN, the weights were the following: 0.866667, 1.3, 1.10, 1.05, 0.8167, 1.05, 0.8167. Lastly, additional testing found that the Adam optimizer [41] provided the greatest results for this task, being especially stable when paired with a learning rate (lr) of 0.0008.

```
# First LSTM layer for initial feature extraction
self.lstm_1: nn.LSTM = nn.LSTM(
    input_size=self.get_n_mfcc(),  # Number of MFCC features
    hidden_size=256,               # Number of features in hidden state
    num_layers=1,                  # Number of stacked LSTM layers
    batch_first=True,              # Input/output tensors have shape (batch, seq, feature)
    bidirectional=True,            # Bidirectional LSTM for better context
)

# By doing a 0.5 dropout, we make 50% of the neurons inactive during each training iteration
self.dropout1: nn.Dropout = nn.Dropout(0.5)

# Second LSTM layer for deeper feature extraction
self.lstm_2: nn.LSTM = nn.LSTM(
    input_size=512,                # 256 * 2 (bidirectional)
    hidden_size=1024,              # Number of features in hidden state
    num_layers=1,                  # Number of stacked LSTM layers
    batch_first=True,              # Input/output tensors have shape (batch, seq, feature)
    bidirectional=True,            # Bidirectional LSTM for better context
)

# By doing a 0.5 dropout, we make 50% of the neurons inactive during each training iteration
self.dropout2: nn.Dropout = nn.Dropout(0.5)

# Final fully connected layer for classification
self.fc: nn.Linear = nn.Linear(2048, len(self.get_classes()))  # 1024 * 2 (bidirectional)
return self
```

Fig. 1. The LSTM model

The training steps were standard as well, and for each batch of samples, we zero the gradients, forward pass, calculate the loss, backward pass, and update the weights [20], [21], [22], [23], [24]. The only other step added in during the experimenting process was the use of clipping the gradient norm, which is done after the backward pass [23]. Clipping the gradient norm helps with overfitting, and prevents wild and unpredictable changes. At the end of each training epoch, the model was set to evaluation mode, with a forward pass and loss calculation for each validation batch [20], [21], [22], [25], [26], [27], [28]. Once this was finished, the average loss of the validation batches was calculated, and an early stopping mechanism was put in place. To be specific, testing showed that by the time the average validation loss went lower than 0.4; this was near the best accuracy possible before overfitting. Therefore, whenever the average validation loss reaches this point, we end the training process and proceed to testing the final results. If early stopping does not activate, this training loop continues for a maximum of a hundred epochs. A discussion of the model's results is the next topic.

V. Results & Discussion

Overall, the final model results came out relatively decent. The first step, of course, was to model the training and validation loss and accuracy curves, with the purpose being to look for any major overfitting or undesirable effects [20], [21], [22].

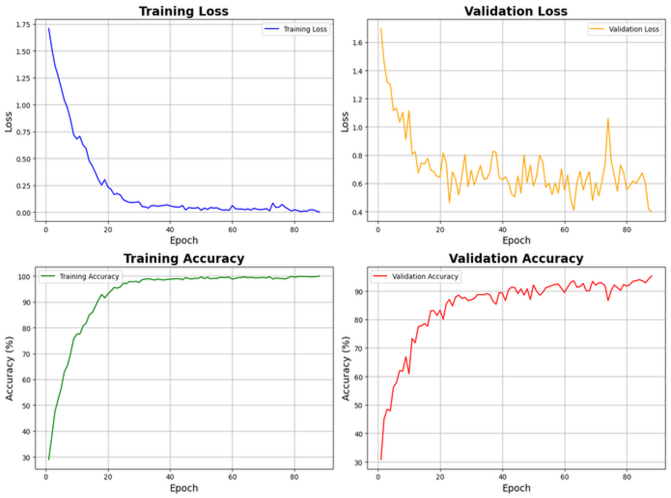

Fig. 2. Training and Validation Curves

As can be seen, nothing too out of the ordinary happened during the training process, except around near epoch seventy-five, where the validation loss suddenly jumped up an extreme amount [20], [21], [22]. Thanks to the early stopping, it seems the model finished before any major overfitting occurred with the training or validation loss, and the final validation accuracy was 0.953.

```
Epoch [88/100], Train Loss: 0.0009, Train Acc: 100.00%, Val Loss: 0.3974, Val Accuracy: 95.30%
  Per-Class Validation Accuracy:
    diffwave: 94.37% (67/71)
    gt: 86.67% (91/105)
    melgan: 94.37% (67/71)
    parallel_wave_gan: 98.59% (70/71)
    wavegrad: 100.00% (72/72)
    wavenet: 98.59% (70/71)
    wavernn: 98.59% (70/71)
```

Fig. 3. Final Validation Results

Much more important, of course, were the final testing results [20], [21], [22]. The testing process is practically the same as the validation process, but of course, we use never-before-seen testing batches instead. The results were decent, but there were some problems with the aforementioned MelGAN, WaveNet and WaveRNN vocoders. Parallel WaveGAN and WaveRNN did exceedingly well, and WaveNet still didn't underperform too badly. It should be noted that these results can change from model test to model test, but the normal range of testing accuracy is stable at ninety-three-to-ninety-six-percent accuracy, so a more 'worst-case scenario' result was shown to demonstrate its weakness.

```
============================== Evaluating Model on Test Set ==============================

Loading model from: /content/drive/My Drive/CYBR_4980_Project/Dataset_Extracted/LibriSeVoc_extracted/Deep_Fake_Detector_LSTM_V15.pth
Model loaded successfully.


Test Results:
  Average Loss: 0.4711
  Overall Accuracy: 94.18% (502/533)

Per-Class Accuracy:
  diffwave: 91.55% (65/71)
  gt: 89.52% (94/105)
  melgan: 88.73% (63/71)
  parallel_wave_gan: 100.00% (72/72)
  wavegrad: 100.00% (71/71)
  wavenet: 93.06% (67/72)
  wavernn: 98.59% (70/71)

------------------------- Model Evaluation Complete -------------------------
```

Fig. 4. Test Results

Additionally, to begin more wholistically understanding the results achieved with the model, a t-SNE plot was created to show where mistakes were occurring [20], [21], [22], [29], [30], [42]. As can be seen, most samples grouped up in their correct class, but there were notable issues with melGAN, WaveNet, and DiffWave ending up in the gt cluster. Original audio samples ending up in clusters like WaveNet, MelGAN, and Parallel Wave-GAN were also noticeable. However, it could be argued that false positives are a smaller issue when compared to false negatives, especially if used by an Intrusion Detection System (IDS) of some sort.

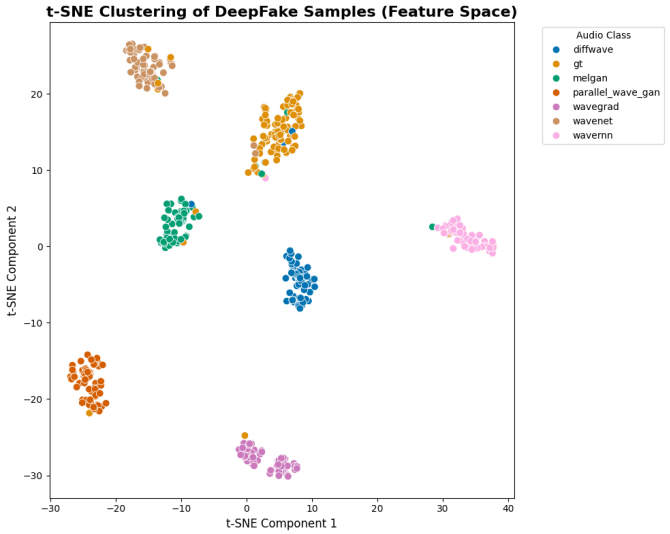

Fig. 5. t-SNE plot

While the cluster plots were decent for showing the overall improvement, as well as proving that the model did learn features specific to each class, confusion matrices were created to get an even more exact idea of what classes were causing issues for the LSTM model [20], [21], [22], [29], [30], [43]. As can be seen, WaveNet, MelGAN, and DiffWave once again show themselves as being the most major issues for the model. The model incorrectly guessed about four-to-seven percent as being human-created audio with all three. Still, each vocoder reached above ninety-percent classification accuracy as being *some* sort of vocoder, even if the exact vocoder type was incorrectly guessed.

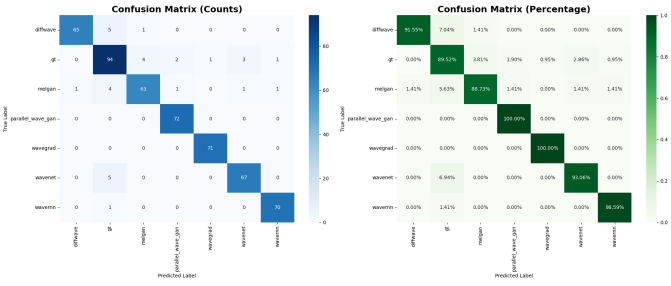

Fig. 6. Confusion Matrices

Furthermore, a classification report was created to show the user an in-depth calculation of the precision, recall, and F1-score for each of the classes [1], [2], [4], [5], [7], [20], [21]. [22], [29], [33], [45]. Again, very similar insights can be gleaned from the table, such as the fact that gt had a higher recall than precision. Like the previous graphs, MelGAN's recall was once again disappointing compared to the other classes, and DiffWave and WaveNet also had problems of course, but still got into the nineties in recall. As for F1-score, the precision/false positives downgraded gt into being the only class with a sub-ninety score, but all the others went above ninety, often being in the mid-to-high nineties range. In the end, Parallel Wave-GAN, WaveGrad, and WaveRNN all did extremely well, and even over ninety-percent of MelGAN was classified as a vocoder of *some* type, so it didn't do too poorly overall [1], [2], [4] [5], [7], [20], [21], [22], [29], [33], [43], [45].

	precision	recall	f1-score	support
diffwave	0.98	0.92	0.95	71.0
gt	0.86	0.9	0.88	105.0
melgan	0.93	0.89	0.91	71.0
parallel_wave_gan	0.96	1.0	0.98	72.0
wavegrad	0.99	1.0	0.99	71.0
wavenet	0.94	0.93	0.94	72.0
wavernn	0.97	0.99	0.98	71.0
accuracy	0.94	0.94	0.94	0.94
macro avg	0.95	0.94	0.95	533.0
weighted avg	0.94	0.94	0.94	533.0

Fig. 7. Classification Report

The final data collected to show model performance were EER and AUC statistics, chosen to be in lockstep with their common usage in other detection models [1], [2], [4],[5], [7], [8], [9], [46], [47], [48], [49], [50], [51], [52], [53], [54]. Compared to other EER results in the field of audio deepfake detection, which range from around 0.0013-0.43, and are often around 0.02-0.08, this Bi-Directional LSTM's AUC and EER seemed to be comfortably within this range. Unfortunately, it got nowhere near the 0.0013 EER that the original dataset creators got with their model [7], but 6.67-percent is not nearly as bad as other models have performed on similar tasks, and using MFCC didn't seem to destroy or massively harm the model's capabilities to tell classes apart from each other [1], [2], [4], [5], [7], [8], [9], [15], [16], [46], [47], [48], [49], [50], [52], [53], [54]. It should be mentioned that [54] was especially useful in helping to generate all of the EER statistics.

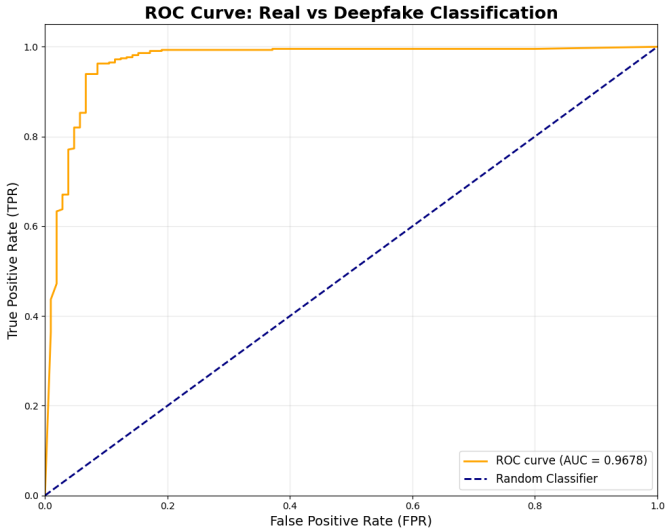

Fig. 8. ROC Curve

```
ROC Curve Statistics:
----------------------------------------
  Area Under Curve (AUC): 0.9678
  Equal Error Rate (EER): 0.0667 (6.67%)
  EER Threshold: 0.9948
----------------------------------------
```

Fig. 9. AUC and EER Statistics

VI. Drawbacks & Limitations

Before concluding this paper, it is important that the model's biggest drawbacks and limitations be expanded on. While the classification issues have already been mentioned, the Bi-Directional LSTM has a much more major problem: training and processing time. With only about three-thousand samples, each sample being six seconds in length, the model took around six-seven hours to stop training, finishing around the ninetieth epoch. The model *could* be trained around two times faster, but that would mean having to settle for a final accuracy of about eighty-six-to-eighty-eight-percent instead, a relatively steep price to pay. A naive solution could be to have the audio sample turned into MFCC data and transformed with the scaler only during the first epoch, saving it and using that processed data every other epoch, instead of repeatedly doing this inefficient process for every single iteration. Of course, this would obviously make the model less realistic, since company calls or audio data will never directly be scaled MFCC NumPy ndarrays. More efficiency will definitely have to be designed into the model itself to make it ready for production in any organizational setting. Additionally, as was noted by the authors in [7], vocoders are not a direct method for deepfake detection. While they are the last processing step, and there should be little reason to believe much would change, further experimentation needs to be done to see how the model operates with true deepfake audio.

Furthermore, with many AI models, both in the present and the near future, likely to create even more realistic audio, the 2023 datasets may already be going out of date [1], [2], [3], [4], [5], [7], [13]. As such, a future goal will be to create a dataset that uses deepfake samples from more recent AI models. Another point of concern is that the LibriSeVoc dataset should very much be considered a 'clean' dataset, meaning without background noise or poor-quality audio, a situation which may not always be the case in real life [7]. Over telephone or voicemail especially, a bad actor might be able to get away with adding such noise or otherwise creating poor-quality audio to mess with and confuse a detector. Such tactics have already been accounted for in [7], but this model has yet to implement them. Finally, the model noticeably struggled with GAN-type vocoders. Sometimes in testing, like in the figures shown, MelGAN specifically does worse than even the gt class, a disappointing result. Further effort will have to be put in to see what makes GAN-type vocoders so difficult, as well as to learn if there are any special features or additional models that could be used to improve performance.

VII. Conclusion

All in all, for being the first step on the part of the author to help solve the cybersecurity threat deepfakes pose, the results seemed promising. They *do* prove that, at the very least, LSTM can get better accuracy and EER than some previous papers have shown [13]. Hopefully, it can also provide some insight on how to begin approaching deepfake detection with LSTM, and lead to more novel methods of detection besides CNNs and pre-trained models [1], [2], [4], [5], [7], [13]. With time, it is likely that the model's issues of efficiency and accuracy can be improved. Although, LSTM's capability to truly become a production-ready detector, used in real-time scenarios to assist with phishing detection, is something that certainly needs much more exploration.

Acknowledgment

The author would like to thank Chengzhe Sun, Shan Jia, Shuwei Hou, and Siwei Lyu, who created the LibriSeVoc dataset, and were a major source of inspiration for creating this paper's model. Additionally, thank you to Changjiang Yang for the knowledge necessary to create an EER and EER threshold calculation for the LSTM model.

References

[1] J. Yi, C. Wang, J. Tao, X. Zhang, C. Y. Zhang, and Y. Zhao, 'Audio Deepfake Detection: A Survey', arXiv [cs.SD]. 2023 https://doi.org/10.48550/arXiv.2308.14970.

[2] Z. Almutairi and H. Elgibreen, "A review of modern audio deepfake detection methods: Challenges and future directions," Algorithms, vol. 15, no. 5, p. 155, May 2022. doi:10.3390/a15050155

[3] Z. Khanjani, G. Watson, and V. P. Janeja, "Audio Deepfakes: A survey," Frontiers in Big Data, vol. 5, Jan. 2023. doi:10.3389/fdata.2022.100106

[4] M. Li, Y. Ahmadiadli, and X.-P. Zhang, "A Survey on Speech Deepfake Detection," ACM Computing Surveys, vol. 57, no. 7, pp. 1–38, Feb. 2025. doi:10.1145/3714458

[5] B. Zhang, H. Cui, V. Nguyen, and M. Whitty, "Audio deepfake detection: What has been achieved and what lies ahead," Sensors, vol. 25, no. 7, p. 1989, Mar. 2025. doi:10.3390/s25071989

[6] D. Anikin, "Don't believe your ears: voice deepfakes," Kaspersky, https://www.kaspersky.com/blog/audio-deepfake-technology/48586/ (accessed Dec. 4, 2025).

[7] C. Sun, S. Jia, S. Hou, and S. Lyu, "AI-synthesized voice detection using neural vocoder artifacts," 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 904–912, Jun. 2023. doi:10.1109/cvprw59228.2023.00097

[8] A. Hamza *et al.*, 'Deepfake audio detection via MFCC features using machine learning', *IEEE Access*, vol. 10, pp. 134018–134028, 2022.

[9] A. S. Bin Saharom and F. Ehara, 'Comparative Analysis of MFCC and Mel-Spectrogram Features in Pump Fault Detection Using Autoencoder', in *2024 2nd International Conference on Computer Graphics and Image Processing (CGIP)*, 2024, pp. 124–128.

[10] Y. A. Li, A. Zare, and N. Mesgarani, 'StarGANv2-VC: A Diverse, Unsupervised, Non-parallel Framework for Natural-Sounding Voice Conversion', arXiv [cs.SD]. 2021.

[11] K. Kumar et al., 'MelGAN: Generative Adversarial Networks for Conditional Waveform Synthesis', arXiv [eess.AS]. 2019.

[12] G. Lavrentyeva, S. Novoselov, A. Tseren, M. Volkova, A. Gorlanov, and A. Kozlov, 'STC Antispoofing Systems for the ASVspoof2019 Challenge', *arXiv [cs.SD]*. 2019.

[13] F. T. Winata, N. J. Tanuwijaya, R. Setiawan, and R. Y. Rumagit, "Comparison of deepfake detection using CNN and Hybrid models," Procedia Computer Science, vol. 269, pp. 1556–1564, 2025. doi:10.1016/j.procs.2025.09.097

[14] C. Sun, LibriSeVoc Dataset, (May 02, 2023). doi: 10.5281/zenodo.15127251.

[15] librosa development team, "librosa.feature.mfcc," librosa, https://librosa.org/doc/main/generated/librosa.feature.mfcc.html (accessed Dec. 4, 2025).

[16] E. Deruty, "Intuitive understanding of mfccs," Medium, https://medium.com/@derutycsl/intuitive-understanding-of-mfccs-836d36a1f779 (accessed Dec. 5, 2025).

[17] librosa development team, "librosa.feature.melspectrogram," librosa, https://librosa.org/doc/0.11.0/generatedlibrosa.feature.melspectrogram.html (accessed Dec. 4, 2025).

[18] librosa development team, "librosa.stft," librosa, https://librosa.org/doc/main/generated/librosa.stft.html (accessed Dec. 4, 2025).

[19] L. Roberts, "Understanding the Mel Spectrogram — by Leland Roberts — Analytics Vidhya — Medium," Medium, https://medium-com.translate.goog/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53?_x_tr_sl=en\&_x_tr_tl=pt\&_x_tr_hl=pt-PT\&_x_tr_pto=tc (accessed Dec. 5, 2025).

[20] D. Bourke, "01. PyTorch Workflow Fundamentals," Zero to Mastery Learn PyTorch for Deep Learning, https://www.learnpytorch.io/01_pytorch_workflow/ (accessed Dec. 4, 2025).

[21] D. Bourke, "02. PyTorch Neural Network Classification," Zero to Mastery Learn PyTorch for Deep Learning, https://www.learnpytorch.io/02_pytorch_classification/ (accessed Dec. 4, 2025).

[22] D. Bourke, "03. PyTorch Computer Vision ," Zero to Mastery Learn PyTorch for Deep Learning, https://www.learnpytorch.io/03_pytorch_computer_vision/ (accessed Dec. 4, 2025).

[23] Kvpratama, "Audio classification with LSTM and torchaudio," Kaggle, https://www.kaggle.com/code/kvpratama/audio-classification-with-lstm-and-torchaudio/notebook (accessed Dec. 4, 2025).

[24] Saidrasool, "Speech recognition using LSTM: A step-by-step guide — by Saidrasool — Medium," Medium, https://medium.com/@saidrasool402/speech-recognition-using-lstm-a-step-by-step-guide-78e3ee7e7d5f (accessed Dec. 5, 2025).

[25] H. Shandilya, "Training Neural Networks with validation using pytorch," GeeksforGeeks, https://www.geeksforgeeks.org/machine-learning/training-neural-networks-with-validation-using-pytorch/ (accessed Dec. 3, 2025).

[26] M. A. I. Khan, "Training and validation data in pytorch," MachineLearningMastery, https://machinelearningmastery.com/training-and-validation-data-in-pytorch/ (accessed Dec. 3, 2025).

[27] S. Raheja, "Train-test-validation split in 2025," Analytics Vidhya, https://www.analyticsvidhya.com/blog/2023/11/train-test-validation-split/ (accessed Dec. 4, 2025).

[28] T. Segura, "'Train, Validation, Test Split' explained in 200 words.," Data Science - One Question at a time, https://thaddeus-segura.com/train-test-split/ (accessed Dec. 4, 2025). '

[29] NumPy Developers, "numpy.ndarray," NumPy, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html (accessed Dec. 4, 2025).

[30] NumPy Developers, "numpy.concatenate," NumPy, https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html (accessed Dec. 4, 2025).

[31] scikit-learn developers, "StandardScaler," scikit-learn, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html (accessed Dec. 4, 2025).

[32] H. Joshi, "Understanding Feature Scaling in Machine Learning: Techniques, Implementation, and Advantages," Medium, https://python.plainenglish.io/understanding-feature-scaling-in-machine-learning-techniques-implementation-and-advantages-fd9065a349aa (accessed Dec. 5, 2025).

[33] NumPy Developers, "numpy.ndarray," NumPy, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html (accessed Dec. 4, 2025).

[34] W3Schools, "NumPy Joining Array," W3Schools, https://www.w3schools.com/python/numpy/numpy_array_join.asp (accessed Dec. 4, 2025).

[35] NumPy Developers, "numpy.std," NumPy, https://numpy.org/doc/stable/reference/generated/numpy.std.html (accessed Dec. 4, 2025).

[36] Vishal, "Compute the mean, standard deviation, and variance of a given Numpy Array," GeeksforGeeks, https://www.geeksforgeeks.org/python/compute-the-mean-standard-deviation-and-variance-of-a-given-numpy-array/ (accessed Dec. 4, 2025).

[37] PyTorch Contributers, "LSTM," PyTorch, https://docs.pytorch.org/docs/stable/generated/torch.nn.LSTM.html (accessed Dec. 3, 2025).

[38] PyTorch Contributers, "CrossEntropyLoss," PyTorch, https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (accessed Dec. 4, 2025).

[39] PyTorch Contributers, "torch.nn," *PyTorch*, Jul. 25, 2025. https://docs.pytorch.org/docs/stable/nn.html\#loss-functions (accessed Dec. 07, 2025).

[40] Saturn Cloud, "Using weights in Crossentropyloss and BCELoss (pytorch)," Saturn Cloud Blog, https://saturncloud.io/blog/using-weights-in-crossentropyloss-and-bceloss-pytorch/ (accessed Dec. 4, 2025).

[41] PyTorch Contributers, "torch.optim," *PyTorch*, Aug. 24, 2024. https://docs.pytorch.org/docs/stable/optim.html (accessed Dec. 07, 2025).

[42] K. Erdem, "T-SNE clearly explained. an intuitive explanation of T-sne… — by Kemal Erdem (Burnpiro) — TDS archive — medium," Medium, https://medium.com/data-science/t-sne-clearly-explained-d84c537f53a (accessed Dec. 5, 2025).

[43] scikit-learn developers, "confusion_matrix," scikit-learn, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html(accessed Dec. 4, 2025).

[44] C. Y. Wijaya, "Breaking Down the Classification Report from Scikit-Learn - NBD Lite #6," Non-Brand Data, https://www.nb-data.com/p/breaking-down-the-classification (accessed Dec. 4, 2025).

[45] scikit-learn developers, "classification_report," scikit, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html (accessed Dec. 4, 2025).

[46] scikit-learn developers, "roc_auc_score," scikit-learn, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html (accessed Dec. 4, 2025).

[47] P. Belagatti, "Understanding the softmax activation function: A comprehensive guide," SingleStore, https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/#:~:text=The%20softmax%20function%2C%20often%20used%20in%20the,by%20the%20sum%20of%20all%20the%20exponentials (accessed Dec. 4, 2025).

[48] SciPy Community, "Softmax," SciPy, https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.softmax.html (accessed Dec. 4, 2025).

[49] J. Brownlee, "How to use ROC curves and precision-recall curves for classification in Python," MachineLearningMastery,

https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/ (accessed Dec. 4, 2025).
[50] A. Bhandari, "Guide to AUC ROC Curve in Machine Learning," Analytics Vidhya, https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/ (accessed Dec. 4, 2025).
[51] A. Yadav, "Multiclass receiver operating characteristic (ROC) in Scikit Learn," GeeksforGeeks, https://www.geeksforgeeks.org/machine-learning/multiclass-receiver-operating-characteristic-roc-in-scikit-learn/ (accessed Dec. 4, 2025).
[52] V. Trevisan, "Multiclass classification evaluation with ROC curves and Roc Auc," Towards Data Science, https://towardsdatascience.com/multiclass-classification-evaluation-with-roc-curves-and-roc-auc-294fd4617e3a/ (accessed Dec. 4, 2025).
[53] M. Alahmid, "FAR, FRR and EER with python," Medium, https://becominghuman.ai/face-recognition-system-and-calculating-frr-far-and-eer-for-biometric-system-evaluation-code-2ac2bd4fd2e5 (accessed Dec. 5, 2025).
[54] C. Yang, "How to compute equal error rate (EER) on ROC curve," Changjiang's blog, https://yangcha.github.io/EER-ROC/ (accessed Dec. 4, 2025).