Fall 2023

Liquid Tab

Nathan Hulet
nmh91@uakron.edu

Automatic Guitar Transcription: Liquid Tab

Nathan Hulet
Williams Honors College

ABSTRACT

Guitar transcription is a complex task requiring significant time, skill, and musical knowledge to achieve accurate results. Since most music is recorded and processed digitally, it would seem like many tools to digitally analyze and transcribe the audio would be available. However, the problem of automatic transcription presents many more difficulties than are initially evident. There are multiple ways to play a guitar, many diverse styles of playing, and every guitar sounds different. These problems become even more difficult considering the varying qualities of recordings and levels of background noises.

Machine learning has proven itself to be a flexible tool capable of generating accurate results in a variety of situations. To harness these benefits, a good program needs quality data and a model well suited for the task. The most promising models for automatic guitar transcription so far have been convolutional neural networks. These models are adequate, but they lack temporal context. A Liquid Time-constant Network is a type of recurrent neural network and therefore it retains a temporal state. By combining these approaches, the resulting model should prove itself as a flexible tool adept to many situations and playing styles.

TABLE OF CONTENTS

Page

INTRODUCTION

There are many more guitarists who can play music than can transcribe it, and even more experienced guitarists find it difficult to transcribe complex pieces. Even an imperfect tool to point out the appropriate notes from a recording would be an immense help to anyone trying to figure out a song. The purpose of this project was to experiment with combining a known approach to the problem with a promising new implementation of a recurrent neural network to create a versatile tool that anyone can utilize. A significant amount of the work to design this tool involved researching the current tools that exist and the machine learning methods and frameworks used to train the models. The goal of this project was to create a program that uses a command-line interface that could take in audio and output the song in guitar tablature notation. This tool was intentionally designed to be simple to allow for the future use of it in a larger project, like a website or app.

BACKGROUND

Machine learning is an excellent tool that has proven itself across the countless disciplines and fields it has been employed in. There are many different models and ways to implement machine learning, and each design has its own advantages and disadvantages. To understand what might be effective in each use case, it is important to start by understanding the nature of the problem at hand. In the case of automatic guitar transcription, it will help to break the problem down into the audio input for the model and the desired output of it.

All guitar recordings are simply recordings of the soundwaves produced by the instrument. All sound waves have a frequency. The higher frequency a sound wave has, the higher the pitch will sound to the listener. In music, the name of the note played is determined by the pitch. This means that the first step to discovering what note is played is to determine the frequency of the soundwave. Visually, this is difficult to determine, especially when multiple notes are played simultaneously. Below is the visual representation of a C chord played on a piano. This chord is produced by playing the C, E, and G notes at the same time.

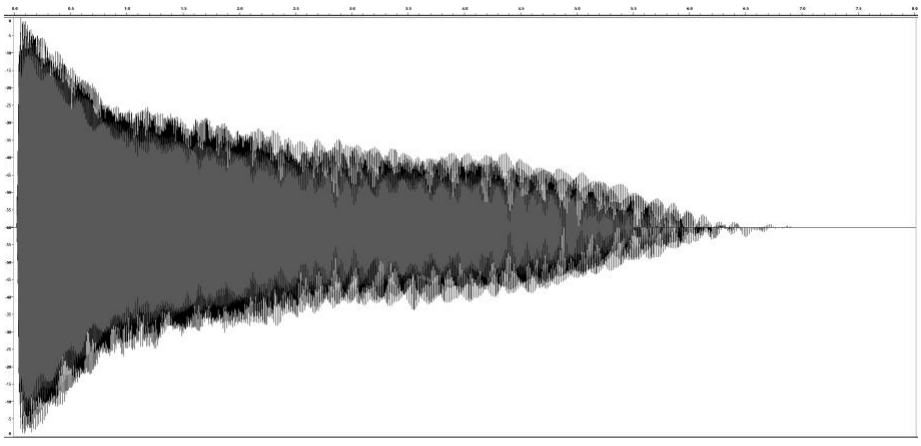

figure 1. *A visualization of a C chord on piano, comprised of C, E, and G notes* [1]

This format would not work well to figure out what the various frequencies involved are. However, since sound is a wave, there are many techniques that can be used to isolate the frequencies of it. A Fourier transform is a transform often used to convert a wave into a representation of its different frequencies. In particular, the closely related constant-Q transform (CQT) is well suited for musical applications. It works by separating the frequencies evenly in a way that directly corresponds to the notes being played [2]. This format is considerably better for the purposes of automated transcription. The same C chord as before is shown below after a CQT.

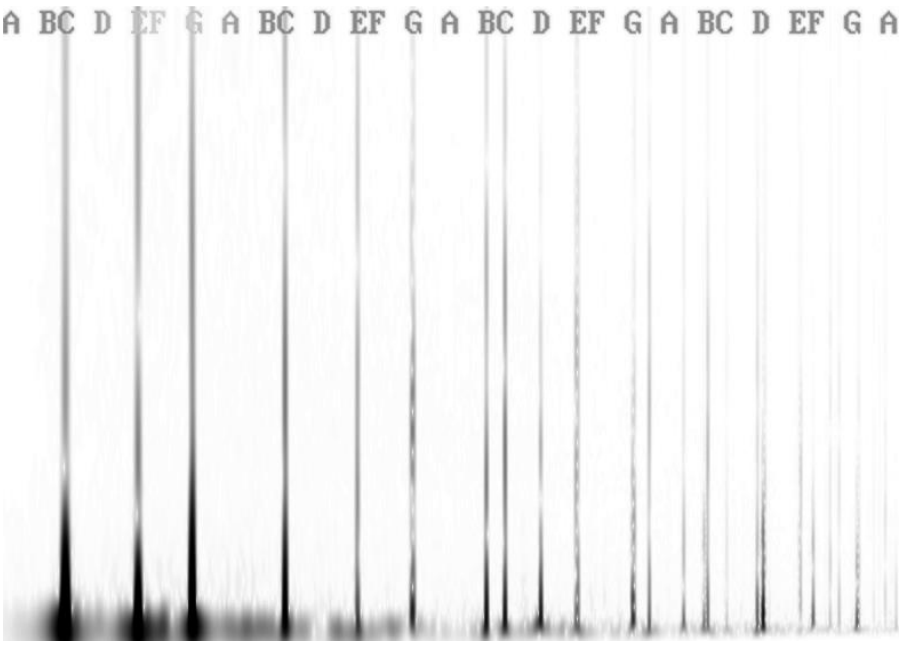

figure 2. *A visualization of a C chord after a constant-Q transform with note names added to the top* [3]

There are two primary choices for the output of the machine learning model. First, there is traditional music notation using a staff with notes where the note played depends upon its

height on the staff. The other option for guitar is tablature. This notation is well suited for fretted instruments since the lines represent the strings and notes are represented by numbers on the strings. These numbers show which fret to hold down at a given time [4]. This format makes playing much more intuitive for guitarists. It also makes transcription considerably simpler, especially since a computer program will already represent a note with a number.

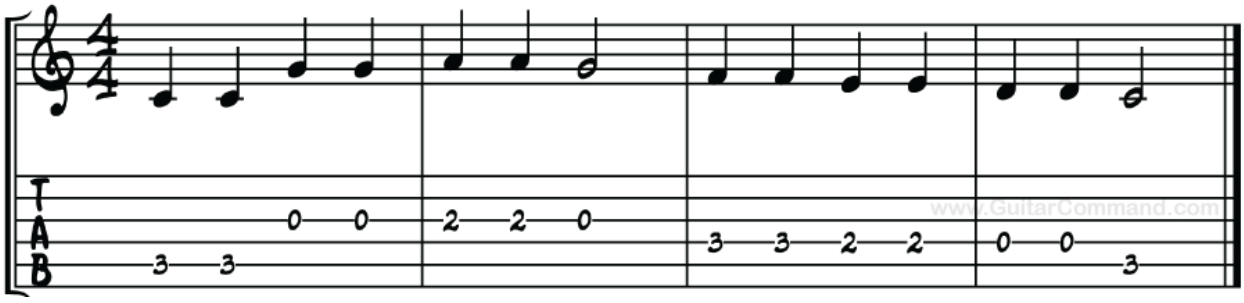

figure 3. *Traditional music notation with the corresponding guitar tablature below it* [4]

The most important part in creating a machine learning model is the data used to train it. If poor quality data is used, any model trained with it will learn to produce a mediocre output. To train a deep learning model, the model must be fed data, produce an output, and then that output needs to be measured against the ground truth. If the output is different, the parameters are changed slightly, and the cycle runs again. Over time, this should lead to a model that can accurately predict the output of any input data it is given. This all requires a dataset that contains the type of input needed and the ground truth data to measure the output against.

One promising model that has been used for automating guitar transcription is TabCNN [5]. It uses a constant-Q spectrogram image as input and outputs tablature. TabCNN uses a convolutional neural network (CNN) architecture. A neural network is a type of computer program designed to learn and make predictions based on input data. It is inspired by the way the

human brain works, with interconnected nodes or "neurons", that process information. A CNN is a type of neural network specially designed to process data grid-shaped data like an image. It uses convolutional layers to figure out what parts of the image have the most meaning.

A Liquid Time-constant network is an implementation of a recurrent neural network (RNN). RNNs are designed for processing sequential data like time [6]. This enables it to consider context and dependencies, making it suitable for tasks that involve time series analysis. Liquid Time-constant networks offer a high level of flexibility. Ordinarily, after a model is trained on data, the weights are set. This means the model can no longer change or learn. This causes problems where the model cannot adapt well to new data. One solution is to increase the quantity and variety of training data so that the model is more robust. However, gathering more data is difficult. Even with more data, it is impossible to prepare for everything. Liquid neural networks are more flexible since they can update their equations as they receive new input. This allows them to continue to learn even once they have been deployed.

DESIGN

For the design, I broke down the project into four parts: choosing the dataset and features, displaying the output, training the model, and creating the command-line interface tool. I decided to use the Python programming language to implement the entirety of this tool because it is well suited for data science and machine learning.

The dataset I chose to use to train the model is called GuitarSet [7]. Guitarset is a free-to-use dataset of richly annotated guitar recordings. It contains roughly 360 recordings played by six different players in several different styles and speeds. I'm using a Python library called "jams" to download, load, and validate the data. It provides a standard way to interact with richly annotated guitar data, allowing for this project to be easily expanded to new data in the future.

To train the model, a set of features must be extracted from the data for use in the training. I decided to use a constant-Q transform image as the only input to the model. For every song in the dataset, I will compute its CQT and then save the resulting images. The convolutional neural network will act as a feature extractor by taking these larger images and pulling out a smaller set number of features. This small set of features can then be used for the recurrent neural network. The network produces an output, and that output is measured against the ground truth to calculate the total loss. The loss is a number showing how bad the prediction was compared to what it should be. A loss of zero indicates a perfect prediction.

I have decided to implement the RNN by using an implementation of a Liquid Time-constant Network. There is a Python package called "ncps" that provides a standard interface for setting up the network with the leading machine learning frameworks [8]. I've decided to use the "pytorch" framework to train the model since it is easier to grasp for beginners and it is designed to work well with Python.

For the output, I have decided to have the network generate a jams file as well as a visualization. I chose to use jams files for two primary reasons: first, the training data uses jams files for input, so the model will be the most accurate when producing a jams file. The other reason for using a jams file is because of its versatility. It is straightforward to convert a jams file to a MIDI format, and many tools already exist to transform a MIDI file into guitar tablature or standard music notation. There are many challenges with converting a MIDI file into guitar tablature. By focusing on generating the jams file instead of the tablature, I will be free to use any of the other tools available to overcome these challenges. This will also let me easily swap out the tools if better ones become available.

I've decided to implement a simple user interface with the program running directly from the terminal. It will prompt the user to select a file by opening a file selector window, and then have them select a location for the result to be output to. By focusing on keeping the tool simple, I can work on the model now while keeping it easy to host on a website or computer application in the future.

IMPLEMENTATION

The first task in implementing this model was to build a framework to train it. Any framework would need to include a way to download the dataset, compute the CQT, split the data into a training and testing partition, instantiate the model, and finally iterate through the training cycle while saving at checkpoints. I found a framework called amt-tools that is designed to train machine learning models for automatic music transcription, and it already contained an instance of the TabCNN model [9]. I modified an instance of this project to add my own model to the framework for training.

For my own model, I started by modifying the TabCNN model so that I could use its output for my Liquid Time-constant network. The hidden layer for the RNN is persistent, so I kept it outside of the class to allow it to build off its experience. Finally, I connected my output from the RNN into a "Softmaxgroups" output layer that determines which fret (if any) was pressed down when a note is played. The definition of the RNN and a description of the model's layers are below.

```
units = round(self.fc_embedding_size * 1.25)
wiring = AutoNCP(units, self.fc_embedding_size)

class LiquidRNN(nn.Module):
    def __init__(self, wiring, input_size):
        super().__init__()
        self.rnn = CfC(input_size, wiring)

    def forward(self, x, hx=None):
        x, hx = self.rnn(x, hx)
        return x, hx
```

figure 4. *The definition of the RNN model.*

Layer Type	Parameters	Output Size
Input		192 x 180 x 1
Conv-Rectify	32 x 3 x 3	190 x 7 x 32
Conv-Rectify	64 x 3 x 3	188 x 5 x 64
Conv-Rectify	64 x 3 x 3	186 x 3 x 64
Pool-Max	2 x 2	93 x 1 x 64
dropout		
flatten		5952
dense		128
dropout		
LiquidRNN	128	128
dropout		
dense		126
reshape		6 x 21

table 1. *The neural network layers for detecting tablature from a CQT image. The final shape represents the 6 strings on a guitar and 21 frets*

To train the model quickly, I created a compute virtual machine using google cloud with a Nvidia Tesla P100 GPU. To train the model I used 6-fold cross validation. k-fold cross validation is a common practice for smaller datasets where the model is evaluated multiple times, giving increased confidence in the quality of the output the model will have on unseen input [10]. I choose to train my model with 1000 iterations, which resulted in 1000 iterations for each fold. I also set checkpoints to save the state of the model and evaluate the performance at every 100 iterations.

To better understand the RNN layer, I ran the training twice, and changed the wiring and neuron count in the RNN. Each model had a different sparsity level and number of hidden neurons. The first model had 218 neurons total and a 50% density level, and the second had 160 neurons and a 75% density level. To determine the wiring of the RNN layers, I used the

implementation of Neural Circuit Policies (NCP) in the “ncps” python package. NCP describes a 4 layer recurrent connection design that divides the neurons into sensory neurons for input, inter and command neurons, which generate an output decision, and motor neurons for the output itself [11]. The plot of the neurons and their connections for my RNN layer is below.

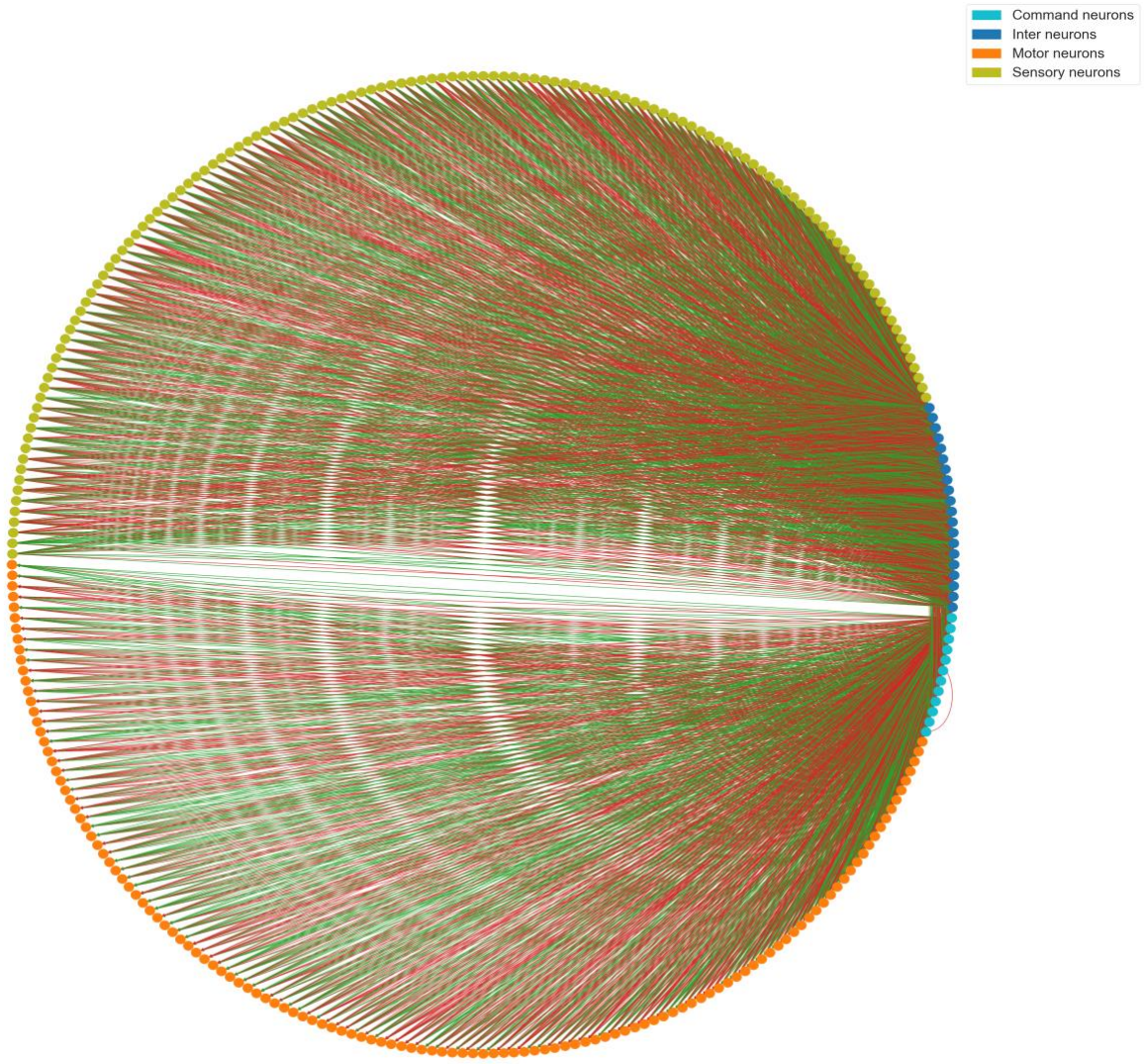

figure 5. *The neurons and their connections for the second RNN model, containing 12 command neurons, 20 inter neurons, 128 motor neurons, and 128 sensory neurons.*

Finally, I looked at the results and chose the model I thought would perform well for the cli tool. I implemented the tool using python and set it up to load the model and run a prediction on it. I also added the ability to convert multiple filetypes into the type that I used for the model. This allows any audio recordings or even videos with audio to be used as an input. The implementation of the transcribe function in the final cli tool is shown below.

```
# Load the model
model = torch.load(model_path, map_location=device)
model.change_device(gpu_id)
model.eval()
profile = model.profile

# Load in the audio and normalize it
audio, _ = tools.load_normalize_audio(audio_path, sample_rate)
data_proc = CQT(sample_rate=sample_rate,
                hop_length=hop_length,
                n_bins=192,
                bins_per_octave=24)

# Compute features
features = {tools.KEY_FEATS : data_proc.process_audio(audio),
            tools.KEY_TIMES : data_proc.get_times(audio),
            }

# Initialize the estimation pipeline
estimator = ComboEstimator([
    TablatureWrapper(profile=model.profile),
    StackedOffsetsWrapper(profile=model.profile),
    StackedNoteTranscriber(profile=model.profile)],
    )

# run predictions
predictions = run_offline(features, model, estimator)
stacked_notes_est = predictions[tools.KEY_NOTES]
```

figure 6. *The implementation of the transcribe function for the cli tool*

RESULTS

The training for the first model took about thirty hours to complete, and the second model took twenty-five hours. I used a tool called Tensorboard to show the progress of the training and to generate graphs to compare the evaluations for each fold. Overall, the training went well and ran without any intervention. The loss function for the training of both models look nearly identical. The graph below shows the loss steadily decreasing as the training progresses until it hovers around one. A graph like this is a good indicator that the training is going well.

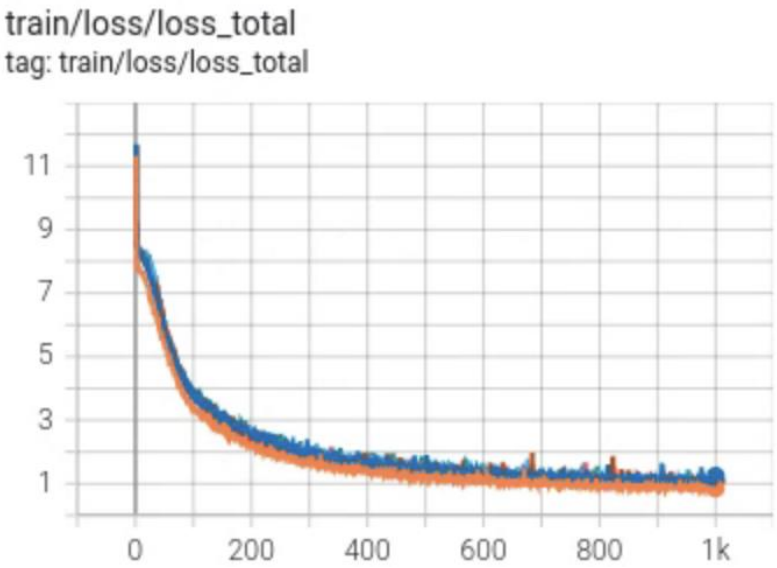

figure 7. *A graph of the loss total at each iteration during training. Each color represents a different fold.*

For validation, the model is given data is has not seen before, and the resulting output from the model is evaluated against the ground truth. Validation only happens when the instance of the model is saved at a checkpoint during the training and at the last iteration. Below is the loss total from the validation for every fold. The loss for the first run looks slightly lower overall.

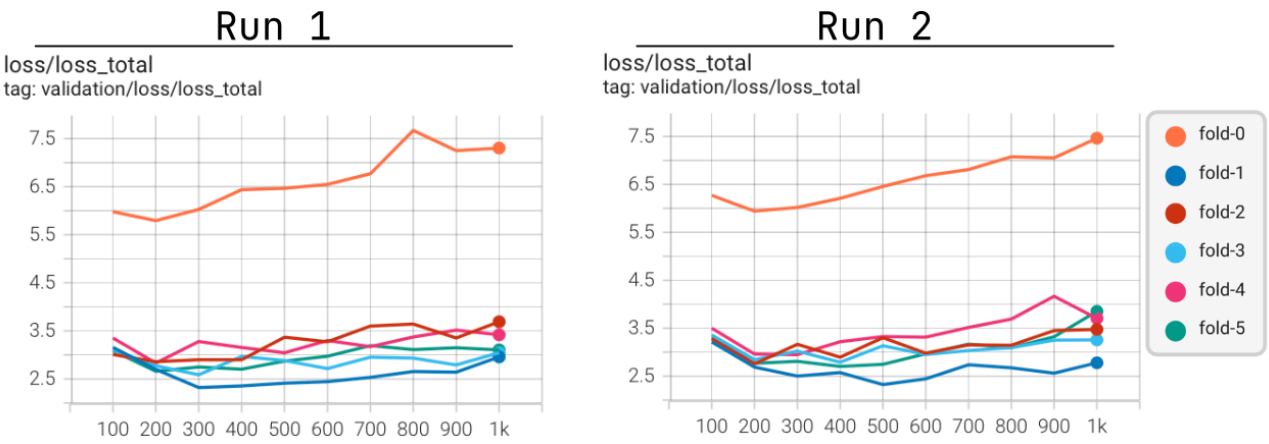

figure 8. *The graphs of the loss total at each checkpoint during validation*

One unexpected result of both runs was the large difference between the initial fold and the rest of the folds. Since the folds are randomly chosen based on a set seed, I changed the seed to see if this gap would disappear. Changing the seed removed the gap while the overall average results stayed relatively close to this run. It appears that the difference between fold 0 and the rest of the folds is simply due to the model performing poorly on the data in that fold.

The metrics I focused on for training and validation are accuracy, precision, and recall. Accuracy is a metric for the tablature generation itself. It is defined by the number of frames that contain a correct identification divided by the total number of frames. Precision is the number of correctly identified string and fret combinations divided by the total number of string and fret combinations that the model predicted. It is a measure of how often a prediction made is correct. Recall is the number of correctly identified string and fret combinations divided by the actual total number of string and fret combinations from the ground truth. In other words, this metric is a measure of how well the model does at predicting the same number of notes as the ground truth. The table below shows the average metrics for each run.

Metrics	Run 1 avg.	Run 2 avg.
loss total	3.95	4.01
accuracy	0.869	0.859
precision	0.757	0.724
recall	0.723	0.699

table 2. *The average results for all the folds of both runs.*

From these results, it is evident that the first run's predictions were better overall. This means that decreasing the number of neurons in the RNN had a negative effect on the performance. This table also shows the relation between a lower loss total and better predictions. Looking closer into accuracy of each fold reveals a similar trend of better results overall from the first run when compared to the second. It is also worth comparing the most accurate folds from each run.

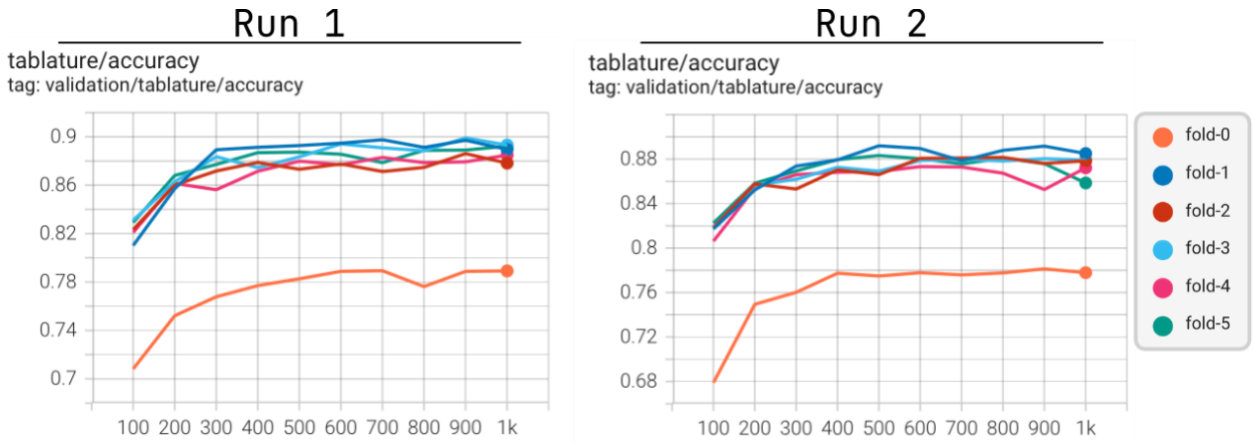

figure 9. *The graphs of the accuracy at each checkpoint during validation*

Metrics	Run 1 fold-3	Run 2 fold-1
loss total	3.05	**2.77**
accuracy	**0.893**	0.885
precision	**0.763**	0.745
recall	**0.740**	0.733

table 3. *The average results of the most accurate folds for each run. Bolded values represent which run performed better for the metric*

Again, these graphs show the first run holding a slight advantage over the second. Digging down into the most accurate folds for each run reveals that the validation loss total is not directly tied to better results like it seemed during the training. Overall, these results offer a good foundation for understanding each model better and selecting the fold and run to use for the final product.

The second fold from the first run had the highest accuracy out of any model trained. I decided to use this model in the final version of the cli tool. I recorded myself playing a simple version of the song Für Elise on my guitar. When I played it, I only ever played one note at a time, and never played any notes past the fourth fret. I also recorded myself playing four simple chords using only the first three frets. The resulting tablature estimations are below.

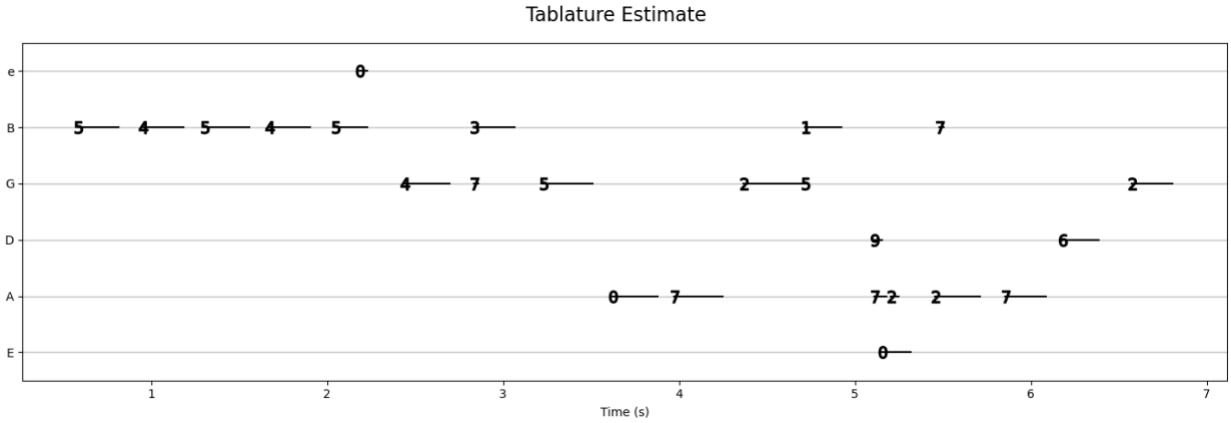

figure 10. *A tablature estimation of the song Für Elise recorded on an iPhone as a voice memo.*

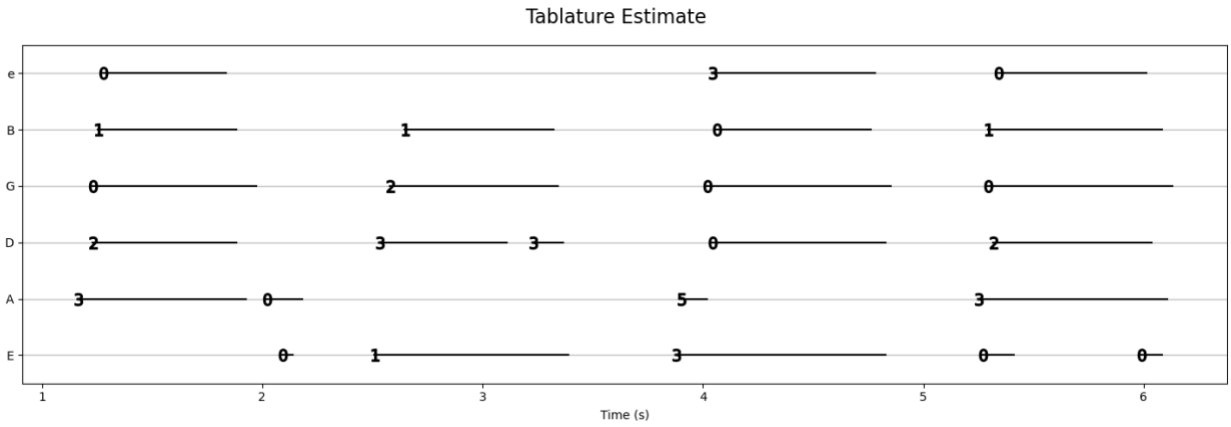

figure 11. *A tablature estimation of the chords C, F, G, and C recorded on an iPhone as a voice memo.*

These tablature estimations are better than I expected to achieve, although they are far from perfect. Both estimations include fingerings for notes that are much more difficult to play than what a human would intuitively write. In the Für Elise song, the model estimated the correct pitch for nearly every note, so the song would sound very similar. However, many of the frets it suggests are much higher up on the neck than I played, and so it would be much harder for a beginner to play. This is likely a result of the tool being trained on music that was often played higher than the first three frets. The tool also generated notes that were not played at all in the recordings. The chord chart was impressively accurate. It is easy to see when each chord was played, and the model even seems to have picked up that the notes on lowers strings were played slightly before the notes on the higher strings. The chords are slightly more accurate than Für Elise, but the fingering for the G chord is unusual, and extra notes are added after the strum occurred.

CONCLUSION

The goal of this project was to make transcribing guitar music simpler by creating a program that could take in a song and output the tablature notation. While the resulting tool I created is not perfect, it is useful, and it is something that I will depend on in the future while trying to figure out a new song. This project was very enjoyable to work on since it combined my interest in music with machine learning. Through this experience, I learned more about the nature of music, transcription, and designing and evaluating neural networks. The most valuable part of this experience for me was what I learned through evaluating each model's performance after designing them. This helped me to better understand what to look for in the results from a model and what to consider while designing a model like this in the future. From the results of this project, I am confident that future models will be capable of high-fidelity automatic transcription, and I am excited to continue working on ways to improve in this field.

FUTURE WORK

There is much room for future work to improve automatic guitar transcription. One of the easiest ways to improve the model would be to add more data. By increasing the amount of data, the model would be better prepared for different types of real-world scenarios, and it would be able to train longer, which should hopefully produce better results. Another way to improve the accuracy would be to introduce more features. Currently the only input to the model is CQT image of the audio. By introducing mores features, such as the time of each beat and the time signature of the music, the model could use the extra context to inform its predictions. Finally, more experiments done with this model's design, and they structure of the RNN would be helpful. It's clear that reducing the number of neurons hurt the performance, so it seems intuitive that increasing the count would have a positive effect.

The other side of improving this project would be to make the tool more accessible to guitarists. A good first step here would be support transcribing longer songs. Currently, all the notes of a longer song get squished together on the output, since it displays the tablature on one line. Another improvement would be adding beats and measures to the tablature to make it easier to read. Finally, designing a website or an app to use the tool would make it far more accessible for most people, since not many people are familiar with command-line programs.

REFERENCE LIST

[1] https://en.wikipedia.org/wiki/Constant-Q_transform#/media/File:C-major-piano-chord-waveform.png (accessed Nov. 15, 2023).

[2] "Constant-Q transform," *Wikipedia*, Apr. 02, 2022. https://en.wikipedia.org/wiki/Constant-Q_transform

[3] "Constant-Q transform," *Wikipedia*, Jul. 22, 2023. https://en.wikipedia.org/wiki/Constant-Q_transform#/media/File:CQT-piano-chord.png (accessed Nov. 15, 2023).

[4] G. C. Admin, "What Is Guitar TAB? A Guide To Reading TAB & Notation On Guitar," *Guitar Command*, Jun. 18, 2019. https://www.guitarcommand.com/what-is-guitar-tab/ (accessed Nov. 19, 2023).

[5] Andrew Wiggins and Youngmoo Kim, "Guitar Tablature Estimation with a Convolutional Neural Network", in Proceedings of the 20th International Society for Music Information Retrieval Conference, Delft, The Netherlands, Nov. 2019, pp. 284–291. doi: 10.5281/zenodo.3527800.

[6] R. Hasani, M. Lechner, A. Amini, D. Rus, and R. Grosu, "Liquid Time-constant Networks", *AAAI*, vol. 35, no. 9, pp. 7657-7666, May 2021.

[7] Qingyang Xi, Rachel M. Bittner, Johan Pauwels, Xuzhou Yeand Juan P. Bello, "GuitarSet". Zenodo, Aug. 20, 2019. doi: 10.5281/zenodo.3371780.

[8] "Atari Behavior Cloning - Neural Circuit Policies 0.0.1 documentation," *ncps.readthedocs.io*. https://ncps.readthedocs.io/en/latest/examples/atari_bc.html (accessed Nov. 19, 2023).

[9] F. Cwitkowitz, "Automatic Music Transcription (AMT) Tools," *GitHub*, Aug. 30, 2023. https://github.com/cwitkowitz/amt-tools/tree/master (accessed Nov. 19, 2023).

[10] J. Brownlee, "A Gentle Introduction to k-fold Cross-Validation," *Machine Learning Mastery*, May 21, 2018. https://machinelearningmastery.com/k-fold-cross-validation/

[11] M. Lechner, R. Hasani, A. Amini, T. A. Henzinger, D. Rus, and R. Grosu, "Neural circuit policies enabling auditable autonomy," *Nature Machine Intelligence*, vol. 2. Springer Nature, pp. 642–652, 2020.