Visual Detection of Guitar Fretboard and Subsequent Finger Position Overlay By Augmenting Computer Vision Methodologies with Deep Learning

Hyunbin Kim
Dept. of Business Administration
Seoul Nat'l University
hbkim96@snu.ac.kr

Jinsol Park
Dept. of Liberal Studies
Seoul Nat'l University
jinsolpark@snu.ac.kr

Sue Hyun Park
Dept. of Business Administration
Seoul Nat'l University
brokenopen10@snu.ac.kr

Abstract

The main objective of our project is to detect the fretboards of a guitar and visualize its strings and frets in order to create a computer vision-based tool for first-time guitar learners who have a hard time matching tabulature with specific positions on the fretboard. To do this, we use multiple classic computer vision methodologies such as: Canny edge detection, Hough line transformation, homography, and warping between two images. Additionally, we incorporate a deep learning model which we customized to find the bounding corners of a guitar's fretboard. Using a pre-recorded video as input, this project will work as a proof-of-concept that visual detection of the guitar fretboard and subsequent finger position overlay has the potential to be applied in real-time, therefore enabling usage as an improved medium of self-education.

1. Introduction

While the guitar is a very popular instrument, properly learning how to play it is a difficult, and in most cases, quite unpleasant task. This is in part due to how guitar scores are designed. The two most popular forms of guitar scores - tabulature and chord progression - are both intrinsically flawed. While tabulature is intuitively understandable and playable, it removes chord, and therefore hand/finger position data, making it hard for first-time guitarists to quickly ascertain exactly how they must hold the frets corresponding to the presented tabs. On the other hand, while chord progression scores preserve chord data, it neglects exact fret positions, requiring extensive knowledge of chords and their variants before becoming playable - which is, again, not considerate of first-time learners.

This is why most first-time guitarists resort to Youtube videos when learning songs. Neither tabulature nor chord progression is usable until the player has basic to intermediate knowledge of playing the guitar. However, video lessons are not ideal when learning an instrument, since videos do not wait for the learner; when learning guitar through Youtube videos, a substantial amount of time is spent pausing and rewinding the video.

These flaws demonstrate the need for an alternate medium of education. Some considerations based on what we've discussed above: for efficient and effective learning, this medium should be visual in nature, so that learners can *see* exactly where and how they must press the fretboard; these visual cues must be given in real-time, as the learner progresses through a song, *i.e.*, it must wait for the learner, unlike a recorded video lesson; and lastly, it must put emphasis on accessibility - it should be accessible at any time, anywhere, and preferably should not require extensive setup or modifications to the instrument in order to work. Based on these considerations, it would be reasonable to assume that an AR-based, real-time application that visually detects the fretboard should act as our preferred medium.

Therefore, for this project, we have established a proof-of-concept of an application capable of: visual detection of a guitar fretboard and its strings/frets; subsequent overlay of AR feedback on the corresponding area of the input video stream; and visual notification to the learner of correct/incorrect finger positioning - all in real-time. This substantially enhances our application's utility as a medium of education over existing methods.

2. Background and Related Works

2.1. Background

Although methodologies for guitar tracking have been subject to research, most of these are quite outdated, and rely entirely on classic computer vision - leading to dependencies in input, large variances in output, and less-than-optimal real-time detection capabilities.

Burns and Wanderley [1] presents a classic method for visually detecting and recognizing fingering gestures of the left hand of a guitarist in real-time, but their input data is produced by a camera mount on the guitar neck, which is not the desired input for our project.

Scarr and Green [5] proposes a method which takes the global view of a guitar fretboard as input; which is preferable since our application, in a real-world scenario, would most likely utilize the front-facing camera of a smartphone or a webcam as an input source. First, background subtraction and line detection is applied, and a cluster of lines with specific gradients is assumed to constitute the fretboard. A bounding rectangle is constructed to contain the cluster. Next, to obtain individual frets, the image bounded by the rectangle goes through line detection, where each cluster of approximately vertical lines is considered to be a fret. Next, the fretboard image is normalized so that the detected frets correspond to their expected locations in "fretspace". Finally, after edge detection on the normalized fretboard image, individual contours are filtered by the proposed criteria to identify the fingers. The contour information is then used to heuristically determine which frets are active in any given frame.

While individual finger positions were detected with a significant increase in accuracy compared to Burns and Wanderley's results, five major types of failures were spotted: incorrect neck bounds, camera angle error, joined fingers, incorrect linear transform, and finger not detected. These demonstrate the input-dependency and output-variance of classic computer vision-only pipelines.

Even though we recreated this pipeline to the best of our abilities, and additionally found a reference Github repository which improved upon this pipeline, we were unable to reach the speed and robustness that we desired. Even implementing edge-based tracking, proposed by Motokawa and Saito [3], did not work. Therefore, in our project, a need for non-classic computer vision methodology was recognized.

2.2. U-Net: Convolutional Networks for Biomedical Image Segmentation

Our largest failure point and processing speed bottleneck was observed when we were trying to locate the guitar fretboard within a given image. Since locating a fretboard is logically identical to *segmenting* the fretboard from the rest of the image, we decided to use deep learning to speed up this process. This was in part due to time constraints; we could not afford to spend too much time creating and evaluating feature sets and/or various classification techniques for an intermediary task. Using deep learning would enable us to, instead, create a singular training set that we could use to train a model - after which we could move on to post-processing and string/fret detection.

Ronneberger *et al.* [4] proposes an end-to-end, fully-convolutional network(FCN), designed for image segmentation purposes in the biomedical field. Named after its U-shaped architecture, shown in Figure 1, it largely consists of a contracting, downward path and an expanding, upward path, each forming the left and right halves of the architecture, respectively.

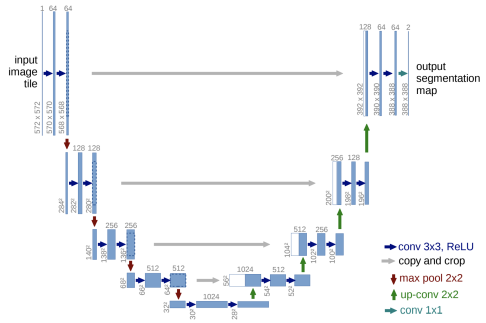

Figure 1. U-net architecture (example for 32×32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map.

The contracting path aims to get a context for the entire image, basically creating a coarse map of global and abstract semantic features, while the expanding path localizes and up-samples these results to get a finer, local, and detailed dense prediction. To make this possible, U-Net utilizes the skip architecture introduced in Long *et al.* [2], which concatenates the same-level cropped feature map from the contracting path to the up-convoluted feature map for each expanding step.

The improvements of U-Net over FCN, such as using ReLU as its activation function; an overlap-tile strategy of dividing the base image into overlapping tiles; use of reflection padding; a weight map derived from the ground-truth which supplements training; softmax outputs; and a loss function which couples cross-entropy and weight map loss; are beyond the scope of this course and project, but it is important to note that these improvements make possible more robust predictions from smaller training sets - ideal for our application, since we have to manually create our training set from scratch.

As a result, while U-Net might not be the best-performing network for our specific use case, due to 1) the abundance of tutorials and code implementations that can

be found online, and 2) our limited capability in creating our dataset, we believed that for the purposes of this proof-of-concept, U-Net was the most appropriate model to use.

3. Method

3.1. Dataset

For our dataset, we used a video of a bass being played in a controlled environment. We used a bass instead of a guitar because unlike a guitar, which has six comparatively thinner strings, a bass has four thick strings, which make for easier visual identification. We believe this is enough for the purposes of our project. Also, we used a pre-recorded video instead of a live video stream, since the real-time analysis capabilities of our software can easily be analyzed using the amount of frames processed per second. This removes the need for us to create dedicated frame buffer mechanisms and/or time keeping modules, which are beyond the scope of this course - instead letting us focus on the vision portion of our software.

Figure 2. Example frame of input video. Video was taken using a smartphone, with a frame size of 1920×1080, at 30fps.

3.2. Fretboard Corner Detection

To detect the four corners of the guitar fretboard for any given frame, which we later use as correspondence points for homography and warping, we used a combination of U-Net and classic computer vision techniques. This procedure is comprised mainly of two steps: 1) Segment the guitar fretboard from the rest of the image using U-Net, and 2) apply post-processing to segmentation results to detect the four corners.

3.2.1 Segmentation of Guitar Fretboard

There were largely two ways to set up our training dataset for U-Net. 1) Segment the four corners directly from the image, or 2) segment the fretboard from the rest of the image. While a specialized network with a better training environment could have facilitated the first option, considering that we were extremely unlikely to acquire noise-free results, we chose option 2, which has higher tolerance for noise, since it segments a larger, more densely connected portion of the image. An example of our training mask is shown below.

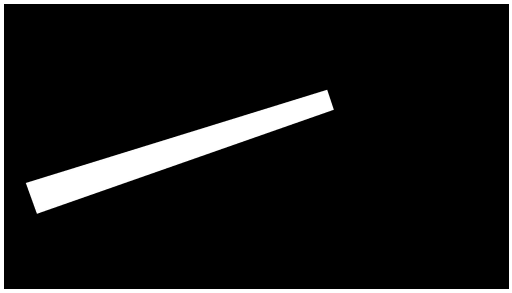

Figure 3. Example mask for frame shown in Figure 2. Created for every 5 frames of input video by manually clicking on the four corners of the fretboard and creating a polygon from registered clicks.

Since our input video was taken at 30 fps and was 12 seconds long, this means that we only have around 70 masks available for training and validation. While U-Net is robust for smaller training set sizes, this is an extremely small training set even by those standards. Therefore, we added in manual variations in the form of translations, rotations, and flips. Translations were added in manually, while rotations and flips were randomly applied when loading our model - using the probabilistic transform functionality provided within the Albumentations library.

The next step was to determine the appropriate hyperparameters for optimal results. The largest determining factor in both speed and accuracy turned out to be the downscale ratio; in accordance with intuition, a lower downscale, and thus a smaller input image led to lower accuracy, but longer training epochs were possible since our training batch could become larger, and vice versa. While our training was done on an Nvidia RTX 3080Ti with CUDA 11.3, which was more than capable of completing the task, limited VRAM meant we had to consider the trade-off between batch sizes (and thus speed) and accuracy.

To test the effects of larger epochs, we ran two training sessions at a lower ($1/8$) downscale, each with 40 and 2000 epochs, to see if we had any meaningful gains in accuracy.

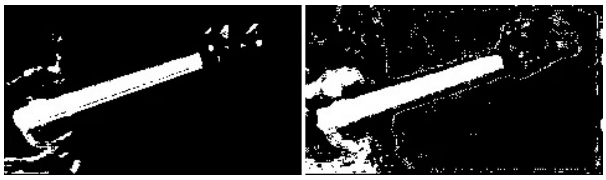

Figure 4. Results of training with 40 epochs (left) and 2000 epochs (right).

While at 2000 epochs, the fretboard itself was detected with less artifacts, a substantial increase in noise meant that

there was no meaningful increase in accuracy. Therefore, we could sacrifice the number of epochs and instead input a higher resolution image. In our final model, we opted for a $1/2$ downscale with just 30 epochs, which led to segmentation results with minimal artifacts.

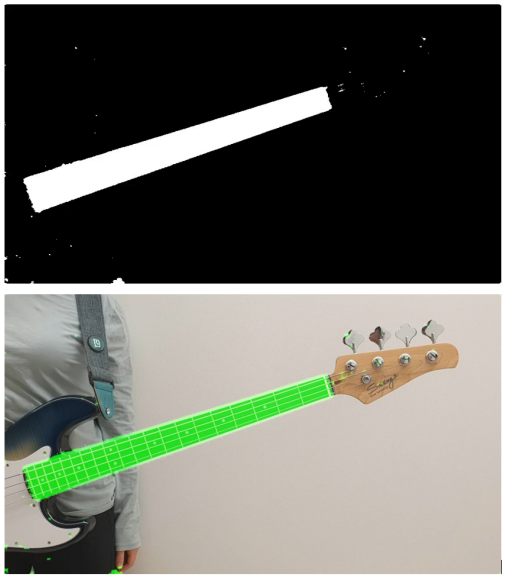

Figure 5. Results of training with 30 epochs and $1/2$ downscale. Depicted as output mask (above) and overlay on original frame (below).

One additional trade-off to note is processing speed. A larger input image means that inference takes a longer time to complete, due to convolutions being applied to a larger image. For just the inference portion of our algorithm, $1/8$ downscale led to about 40 frames processed per second (fpps), $1/4$ downscale led to 30 fpps, and $1/2$ downscale led to 14 fpps when done on a 3080Ti. We believe that 14 fpps is enough for a PoC of real-time processing, but also acknowledge that for lower performance machines, we might have to adjust the downscale ratio. However, considering that most modern smartphones have a dedicated silicon for DL, $1/2$ downscaling would be feasible in many cases, and $1/4$ downscaling will be enough for most cases.

We now apply post-processing to this result mask in order to refine the mask and extract the four corner points of the fretboard mask.

3.2.2 Corner Detection from Segmentation Results

The first step in our post-processing procedure is to apply morphological transformations to close any gaps in the fretboard portion of the mask and reduce noise in other parts of the image. While a morphological close operation was necessary when using $1/8$ downscaled inputs, for $1/2$ downscale, a solid detection of the fretboard meant we could skip this step and directly apply a morphological open operation with a 3×3 kernel.

Then, using the connectedComponentsWithStats function of the OpenCV library, we extract the largest connected component and delete everything else from the mask. This is a quick and reliable method of pruning all noise from the image, as long as our trained model works well enough to detect a solid rectangular area around the fretboard - which is the case here.

Afterwards, canny edge analysis and probabilistic Hough lines detection is applied. Since the results of a probabilistic Hough transform are output in decreasing accumulator order (according to OpenCV specifications) by taking the first two lines of the Hough transform results, we can get the two longest lines of our canny edge *i.e.*, the top and bottom edges of the fretboard - provided that we restrict the HoughLinesP function with appropriate minLineLength and maxLineGap values. For our implementation, we used a minLineLength of 100 and a maxLineGap of 5, which proved stable enough to consistently give us the two edges we need.

Figure 6. Results of canny edge analysis (above) and Hough lines detection (below).

Using the four endpoints we gained from HoughLinesP (two for each line segment), we can now calculate the equations for these two lines using basic cartesian arithmetic. We then drew these lines on a separate mask with the line function within OpenCV. These lines are drawn with a certain thickness (7 in our case), a value which we call noise tolerance. We then do a bitwise-and operation between this line-only mask and the connected component mask that we acquired earlier to get the portion of our mask correspond-

ing to the upper and lower edges of the fretboard. Notice here that the thickness of the line mask acts both as a buffer for errors incurred when drawing the lines (integer pixel coordinates are required when drawing a line, which means our cartesian equation is not accurately represented in our line mask), and for noise within the original mask (noise can erode the corners of our fretboard mask, so taking a couple more pixels extra reduces the possibility of such errors without negatively impacting results), hence why we call this thickness noise tolerance.

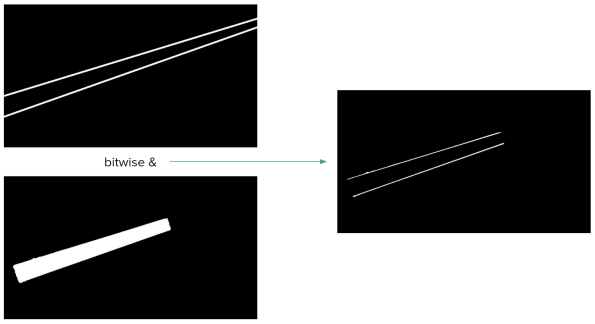

Figure 7. Depiction of how bitwise-and can give us the top and bottom edges of the fretboard.

Here, it is important to note that while Figure 7 shows two edges being analyzed, in our actual implementation, the edges are analyzed separately. Therefore, with one single edge defined, we can get the rightmost (max y-coordinate) pixel and leftmost (min y-coordinate) pixel of the edge, which we can then assume to be the right and left corners, respectively. Doing this for both edges gives us the four corners of our fretboard.

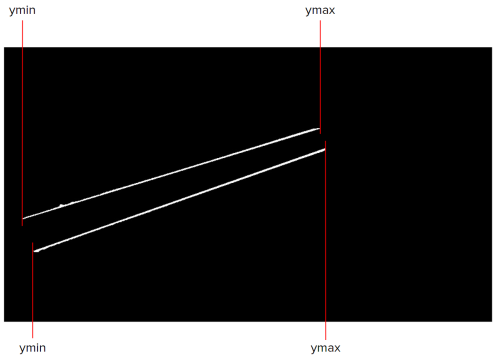

Figure 8. Using the rightmost and leftmost pixels as estimated corners. Two edges are simultaneously depicted for better understanding.

These coordinates are then multiplied by a factor of 2 (inverse to our downscale ratio of $1/2$) in order to relocate them in our original resolution image. An overview of our entire post-processing pipeline is given in Figure 9.

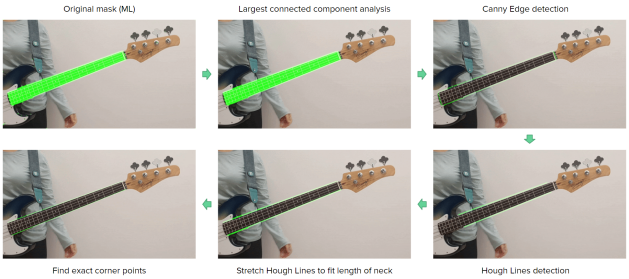

Figure 9. Overview of entire post-processing pipeline. Pipeline starts at top-left corner and advances clockwise.

The results of this pipeline are then handed over to the next step, which is string and fret detection. One thing to note is that for edge cases where the top and bottom edges are incorrectly detected, we keep the results of the previous frame.

3.3. String and Fret Detection

This section explains how we detected the guitar strings and frets for each video frame using the four corners of the fretboard provided by our deep learning model.

3.3.1 Base Image

Our original goal was to detect the strings and frets for each video frame, but failed to do this because 1) HoughLines from OpenCV didn't work as clearly and precisely as we expected, and 2) different optimal parameters were required for each video frame, which was impossible to tune. Instead, we kept a base image with hardcoded strings and frets, and warped this to each of the input video frames. Figure 10 shows the outer grid, strings, and frets displayed on the base image.

Figure 10. Base image with hardcoded grid, strings, and frets.

3.3.2 Finding Homography

To display this guitar grid for each video frame, we needed to find the homography between the base image and the video frame. Using the four grid corners given by the deep

learning model as correspondence points with the grid corners in the base image, we found the homography matrix. This procedure is illustrated in Figure 11.

Figure 11. The four correspondence points used to find the homography matrix. Four points from the harcoded base image (left), and four points from the deep learning results of the video frame (right).

3.3.3 Warping

Using the homography matrix, we warped each string and fret line of the base image onto the video frame. Throughout this process, there were some issues with the fret warping. For some inaccurate cases, the fret lines were inclined in a weird position, and the oscillations of fret positions between video frames were too big. Two optimizations were done to solve such issues. 1) To put the fret lines in the correct position, we interpolated the two endpoint of each fret so that the slope between them is perpendicular to the averaged slope of the two middle strings. This keeps the fret lines comparatively perpendicular to the strings, which is what we expect. 2) Second, to reduce the oscillation of frets between video frames, we averaged the currently warped fret with the previous two frets before displaying. This resulted in less movement of frets between video frames, making the overall resulting video less messier. The final result of warping for a sample frame is shown in Figure 12.

Figure 12. The result of warping the strings and fret lines of the base image onto a video frame with two optimizations on fret lines.

3.4. Fretting Position and Note Estimation

This section explains how we use the gathered fretboard grid to estimate fretting positions, and accordingly the methods to overlay feedback based on comparison with the input music.

3.4.1 Input Music Representation

In order to ease comparison between the input video and music, we hardcode information on the notes required to be played. The music we use is the C major scale, composed of notes (C, D, E, F, B, A ,G, C). For each note, we predetermine the index of string, fret number, and the designated finger that should press the fret. Figure 13 shows how the fretboard grid is indexed and how the first C note is represented as a tuple of ((1, 2), 0). In the input video, the player presses wrong positions for the notes F and B on purpose. Such construction of data requires our subsequent detection algorithm to distinguish which notes are being played properly and which are not.

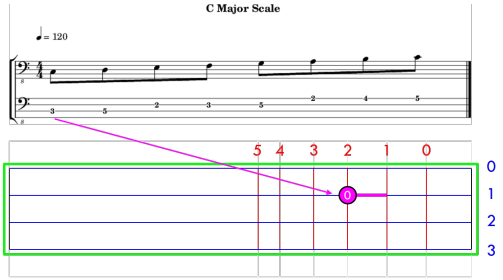

Figure 13. Bass tabulature for the C major scale and an overview of our generated virtual fretboard with string and fret indices. The corresponding index of the C note in the virtual fretboard is (1, 2). The index of the fretting finger is denoted as 0, which refers to the forefinger.

3.4.2 Fretting Position Estimation

To observe the player's fingering information, we first detect the fingertip positions and then estimate the nearest fret of each fingertip. We obtain fingertip coordinates for the index, middle, ring, and pinky finger by a lightweight open source real-time hand tracking library, MediaPipe Hands, details of which can be found at Zhang *et al*. [6]. Next, we estimate the "fretting position", referring to which string and fret a fingertip is located at, using the point of intersection of a string line and fret line as a proxy. If a fingertip lies closest to one intersection, we assume that the fingertip is on the fret characterized by the intersection point. Therefore, we obtain intersections of all string lines and fret lines, which are 28 in total, and for each fingertip, we select the candidate neighboring points. Given a fingertip coordinate as the center point, we extract the intersection coordinates whose x- coordinates lie within a deviation of e, as illustrated in Figure 14. Among the candidates, the intersection

that has minimum Euclidean distance from the fingertip is associated with the position of fingertip.

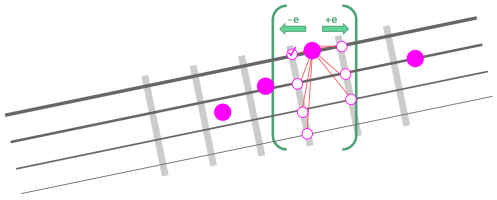

Figure 14. An abstract view of the fretboard and detected fingertips. The large pink circles denote detected fingertip positions; the small white circles denote neighboring string-fret intersections within a specified range of one fingertip position; the white circle with a checkmark denotes the selected intersection point that implies which fret the fingertip is near at.

3.4.3 Note Estimation

We compare the estimated fretting position with the ground truth information given by input music representation, all in sync with the input video. First, we extract which note should be played at each frame. The idea is to hardcode the index of frame at which transition of notes occur; beginning from the frame where the player just presses a fret to play a particular note X and ending at the frame where she starts to press the next fret, the frames in between the two timestamps are assigned note X. Next, we design a simple algorithm that while each frame is being read, if the ground truth fretting position is included among the 4 estimated fretting positions, the bass playing is judged to be correct. Note that the estimated fretting positions imply which frets the fingertips are near to, not that the player is actually pressing all the related frets.

4. Experiments

We tested the overall accuracy of our proposed methods by using a video of a person playing the C major scale on bass, as used above. Out of 8 notes in the C major scale, the player locates true fretting positions of 6 notes and false positions of 2 notes. In default, the displayed feedback provides visualization of the bounding box of the fretboard in green, the string lines in blue, and the fret lines in red. If the player is estimated to be pressing the correct fret at the proper timing, we display the correct region of string and fret and the name of the note in white.

As a result, summarized in Table 1, the program accurately determined all 6 notes that the player is playing correctly, and 1 out of 2 notes that the player is playing incorrectly. Figure 15 shows a partial sequence of the correct output.

One false positive instance occurred because we calculated the Euclidean distance in all directions from the center

		Actual	
		Positive	Negative
Predicted	Positive	6	1
	Negative	0	1

Table 1. The confusion matrix of the result

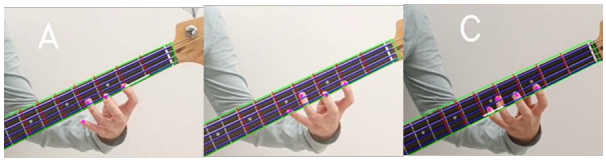

Figure 15. The result sequence of the last three notes of the C major scale. As the player is playing the correct A and C note, the program displays a white line indicating the correct fret region and the name of the note. For the B note in the middle, nothing is done because the detected fingers are out of position.

point instead of the orientation towards the correct region. From the viewer's perspective, the correct region where a finger should press the fret is the right side of the fret line. As shown in Figure 16, our algorithm outputs correct if the fretting position of a finger on the left side of the fret line corresponds to the true fretting position in the particular timing. This limitation can be improved by computing the Euclidean distance only on candidate neighboring intersection points directing towards the head of the bass.

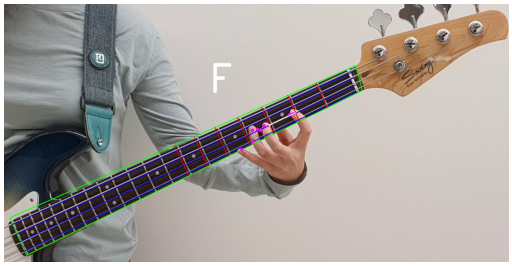

Figure 16. An error case where some random fretting position is considered correct. The player's middle finger is crossing the fret for the F note and is moving on towards the F# fret, but the algorithm detects that the finger is placed on the F fret.

Our proposed methods perform equally or better than previous works using deep learning to accurately detect the fretboard bounding box and intuitive computer vision methods for post processing and note estimation. To further broaden the scope of work, it is necessary to overcome two limitations: 1) single-view perspective and 2) lack of information on finger coordination. With a multi-view input that captures string movements, it would be possible to ascertain which finger is actually pressing the string. In addition, detecting patterns on finger coordination when a player plays a note or moves across frets would obviate the need to des-

ignate the fretting finger.

5. Conclusion

In this project, we managed to display the string and fret lines of a guitar (bass), and check if the fingers are properly placed for a particular note in a video. For this to work, we used a deep learning model (UNET) to get the four corners of the guitar grid in a video frame. Using those four corners as correspondence points, we found the homography between the base image and the video frame. Then, we warped the strings and fret lines hardcoded on the base image to the video frame. Using the string and fret line information, we use the MediaPipe library to detect fingertips, and checked if the fingers are pressing the appropriate string at a certain fret. To sum up, our entire process is a combination of classic computer vision methods (canny edge detection, homography), and deep learning methods.

Our project was done on a pre-recorded video due to hyperparameter issues, and long latency (model inferencing time, other algorithmic issues). Thus potential future work would be to optimize our algorithm and make it work for a real-time video input stream. Our current work processes our input video at a rate of about 8 frames per second, so with some optimization, extrapolation, and better-trained, further downscaled models with larger training datasets and more epochs, it is extremely likely that a real-time solution is indeed possible. Additional features such as checking finger position according to notes given with a consistent beat, or detecting the motion of the right hand, could be added. Another approach would be to incorporate the evaluation of audio, and make a multi-modal system for better accuracy.

Despite the problems we faced, our project delivers a proof-of-concept for a real-time application that can support a beginner guitarist's education in fretting. We hope that our project can encourage further development on real time guitar detection and finger overlay.

References

[1] Anne-Marie Burns and Marcelo M. Wanderley. Visual methods for the retrieval of guitarist fingering. *Proceedings of the 2006 conference on New Interfaces for Musical Expression*, pages 196–199, 2006. 2

[2] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015. 2

[3] Yoichi Motokawa and Hideo Saito. Support system for guitar playing using augmented reality display. *IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 243–244, 2006. 2

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, 2015. 2

[5] Joseph Scarr and Richard Green. Retrieval of guitarist fingering information using computer vision. *25th International Conference of Image and Vision Computing New Zealand*, pages 1–7, 2010. 2

[6] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. Mediapipe hands: On-device real-time hand tracking. *arXiv preprint arXiv:2006.10214*, 2020. 6