Using Microcontrollers to Classify Violin Playing

Ethan Lee[2], Suren Jayasuriya[1], Odrika Iqbal[1], Seth Thorn[1], Byron Lahey[1]
[1] *Arizona State University School of Arts Media and Engineering,* [2] *Pomona College*
(Dated: August 6, 2022)

ABSTRACT

While augmented violins are useful in studying the violin, they do not serve any practical purpose. With the introduction of online learning along with its challenges to violin pedagogy, we seek to garner a more accessible tool to aid online violin lessons. A step in the right direction, we created an image classification model which will determine whether a violinist is playing his violin in real-time, using the OpenMV camera + microcontroller. This is more practical due to its light weight and ease of use, along with its real-time application and ability to run on its own without a computer IDE.

I. INTRODUCTION

With the introduction of online learning due to the coronavirus pandemic, many facets of learning were altered while students remained at home. One aspect of education, music, was hit significantly harder than other subjects due to the necessity of precise audio queues and physical movement of a student's bow to teach one to produce a better sound from instruments such as the violin. Having experienced online music lessons for the better part 3 years now, I can tell you that online learning is not easy when studying a stringed instrument.

When learning that which requires physical movement, people often study what exactly creates the best output. For example, people have been studying the basketball jump shot for years, and we are still perfecting it. Athletes go back and review their gameplay which has been run through automated devices, receiving feedback on the arc of their shot or how aggressively they defended the ball, all for the purpose of outputting a better performance. This technology is amazing! While music is certainly not a sport (and is accordingly treated as such), why can't we strive for technology to help improve violin performance? If we can understand how the violin needs to be played, why can't computers pick up on those same ideas? For example, let us say that my violin teacher is playing his bow faster than I am while also using more bow; it is hard to see this when teaching an online lesson. Is there a way to spot this? We can not just stick an accelerometer on the violin – that would ruin the sound and change the feel of the violin entirely.

The first instinct would be to simply listen to the sound of the violin to hear what aspects of playing are going right – and this is a great option! After all, the goal is to produce the most optimal sound output. However, sound is not the most reliable. For one, it is hard to understand what exactly is causing a violinist to produce his sound; is his left hand flexibility creating this sound? Is he moving his violin a certain way? On top of this, any slight loss in audio would mean a loss in particular aspects of sound that make the violin sound how it does. Therefore, let us explore the use of visual devices with the violin.

II. RELATED WORK

FIG. 1. Example of augmented violin. Various technologies are spread across the body of the violin – the violin likely has a different feel and does not carry a similar weight to a traditional instrument.

While it is easy for us to look at a violinist and determine whether or not they are playing, computers have struggled to determine such. The immediate solution is to receive audio input, however feedback loops can create extra noise that disrupt the violin's sound. Other solutions have included augmented violins which contain sensors on or near the violin. While this solution does work, it is less practical.

Rather than using augmented violins (Figure 1), which do not produce natural sounds and are also not often used, the goal would be to be able to simply attatch a camera to a traditional/widely used violin. This can be used as a supplement to audio queues in case these audio queues are disrupted by feedback or other hindrances. On top of this, augmented violins are not widely available such as those in two studies which use augmented violins to improve intonation [2] and analyze bow strokes. [3]

Cameras using microcontrollers are not new technology either. One study used an embedded computer camera

controller for monitoring and management of image data on controlled lighting devices in order to save energy. [1]

The camera will make use of tensorflow, a free and open-source software library for machine learning and artificial intelligence. In this case, we will be using tensorflow to create an image classification model. Similar to the Fashion-MNIST dataset [4] (which is used for benchmarking machine learning algorithms), we will be classifying images into a machine learning model. However, we will be contributing different types of images and also applying the model to a real-time classifier using the OpenMV camera. Overall, the biggest change is the contribution of a new dataset complemented by changes in an image classification model to better suit the new data.

III. METHODS

A. Hardware

For our purposes, we will use the OpenMV camera. The OpenMV Cam is a small, low power, microcontroller board which allows you to easily implement applications using machine vision in the real-world. You program the OpenMV Cam in high level Python scripts (courtesy of the MicroPython Operating System) instead of C/C++. This makes it easier to deal with the complex outputs of machine vision algorithms and working with high level data structures. But, you still have total control over your OpenMV Cam and its I/O pins in Python. You can easily trigger taking pictures and video on external events or execute machine vision algorithms to figure out how to control your I/O pins.

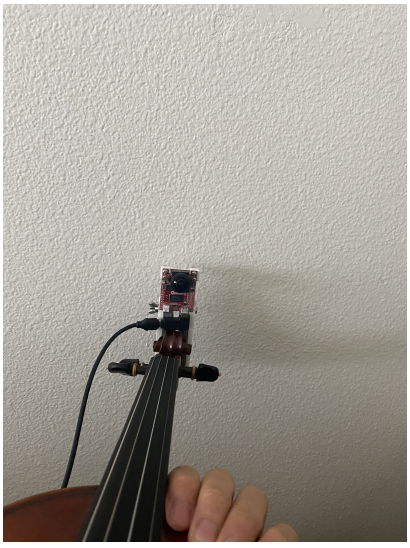

FIG. 2. Sample of camera position on the violin. Camera easily sits at the end of the scroll of the violin and does not disrupt play in any way. The camera holds very minimal weight while also proving to be quite effective.

In this study, the main focus of the hardware will be the use in implementing machine learning algorithms onto the camera. This way, the microcontroller can run on its own without the OpenMV IDE, which is what is used to test how well the machine learning algorithm is working. It is important that the camera can run on its own so that there is no software needed to be open on a computer while processing the output of the microcontroller.

The camera is attatched to a stand which has two clamps (Figure 3) – one which tightens above the scroll, and one below. The clamp is designed to hold the camera in place on the edge of the violin scroll as can be seen in Figure 2. The camera does not affect violin play much at all, as it weighs very little while still holding a STM32H743II ARM Cortex M7 processor running at 480 MHz with 32MBs SDRAM + 1MB of SRAM and 32 MB of external flash + 2 MB of internal flash.

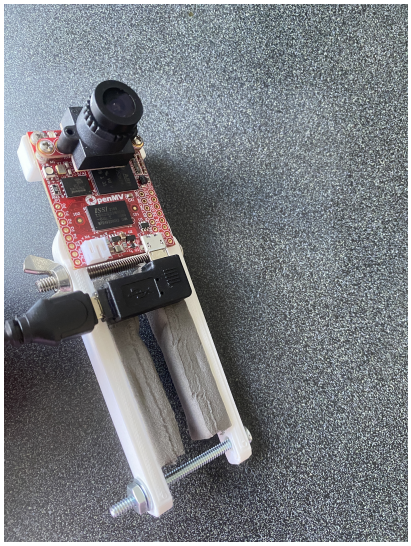

FIG. 3. Image of OpenMV camera/microcontroller. The microcontroller is attached to a stand which attatches to the scroll of the violin – two screws on the top and the bottom of the stand can tighten around the scroll to provide stability to the camera.

B. Neural Network

We begin by using the openMV camera to simply take 200x200 images of: playing the violin with the bow, resting with the violin on the shoulder, plucking the string of the violin, and another category 'unsure' for any other images not associated with playing the violin. Following this, I placed each image into respective folders dependent on which facet of violin playing is associated with the image. This is done so that I can efficiently label each image with a '0' for playing with bow, a '1' for resting with the violin on the shoulder, '2' for plucking the string, and a '3' for miscellaneous images.

FIG. 4. Grid of images captured for the 4 class image classification model. Each image is labeled with the action in regards to violin playing. Grid captured using the Model Maker designed by the tensorflow authors.

Due to a retrospectively weak model, I also created a second folder with the same inputs, but without pizzicato and miscellaneous images to create a two-class image classification model which is simpler and easier to manage.

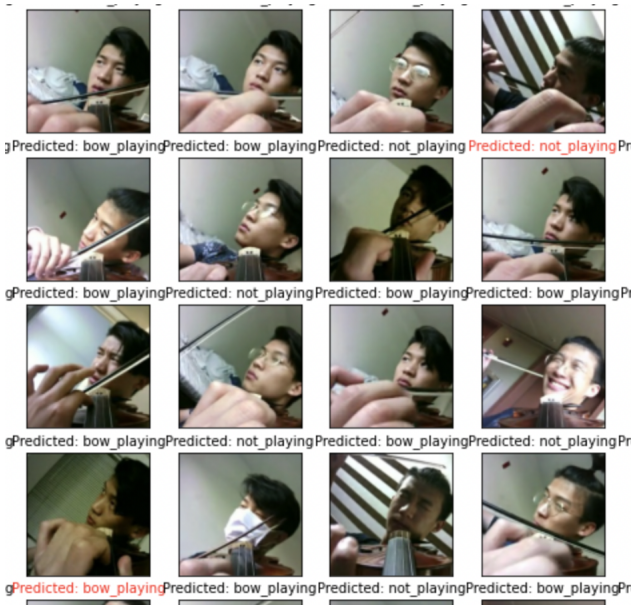

FIG. 5. Visualization of validating the 2 class model. Red color indicates that the prediction was incorrect, while black text indicates that the prediction was correct. As can be seen, the prediction accuracy is quite high. Grid taken using the Model Maker designed by the tensorflow authors.

Since the images are separated into folders based on class, we can easily assign integer labels to each image. After that, we join all these folders together to create a dataset that can be used for training. However, we want to set aside some images for validation as to test that our model is working. For this, we randomly set aside one third of the images, along with their labels, for testing purposes.

These 200x200 images are represented as matrices with RGB values [0, 255] for each pixel of the image; the data is represented as numbers. Using these matrices, the next step is to build and compile our model. For this, we will use a simple Dense model. Fully connected layers of a neural network are defined using the Dense class. We can specify the number of neurons or nodes in the layer as the first argument, and specify the activation function using the activation argument. We can make use of the Sequential() API which will simply arrange the basic building blocks of neural networks in Keras in sequential order so that data flows from one layer to another. In order to condense our complex image data for use in a simple dense model, we need to alter our data since it is too difficult to work with. Our complex multidimensional data needs to be flattened to get the single-dimensional data as output. For that it is needed to create a deep neural network by flattening the input data into single-dimensional data as can be seen in Figure 6.

```
model = keras.Sequential([
    Flatten(input_shape=(200, 200, 3)),
    Dense(64, activation='relu'), #input layer
    BatchNormalization(),
    Dense(2, activation = 'softmax') #output layer
])
```

FIG. 6. Code run in google colab to build layers of a neural network for an image classification model.

Next, we add our input layer. Our input layer has 64 units which use the 'ReLU' activation function. The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It is useful because it helps to prevent the exponential growth in the computation required to operate the neural network. If it scales in size, the computational cost of adding extra ReLUs increases linearly.

Alongside our input layer, we must also add our output Dense layer. For this, we would output a layer with 2 different outputs in the case of our 2 class image classification. In Figure 6, we also make use of the 'Softmax' activation function – Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. The main advantage of the function was that it is able to handle multiple classes.

Next, let us add batch normalization into our model by adding it in the same way as adding a Dense layer.

BatchNormalization() is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer allows every layer of the network to do learning more independently. This solves a major problem called internal covariate shift. Internal Covariate Shift is the change in the distribution of network activations due to the change in network parameters during training. In neural networks, the output of the first layer feeds into the second layer, the output of the second layer feeds into the third, and so on. When the parameters of a layer change, so does the distribution of inputs to subsequent layers. These shifts in input distributions can be problematic for neural networks, especially deep neural networks that could have a large number of layers. So, we add a layer, BatchNormaliztaion(), to mitigate this effect.

Now that we have built our model, let us compile our model. In compiling our model, it is important to use an optimizer. Optimizers are Classes or methods used to change the attributes of your machine/deep learning model such as weights and learning rate in order to reduce the losses. The loss function is the function that computes the distance between the current output of the algorithm and the expected output. It's a method to evaluate how your algorithm models the data; reducing loss is very important in creating a strong model. In our case, we will be using the Adam optimizer simply because it is generally better than every other optimization algorithm, has faster computation time, and requires fewer parameters for tuning.

```
lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.0000006,
  decay_steps=STEPS_PER_EPOCH*1000,
  decay_rate=1,
  staircase=False)

model.compile(tf.keras.optimizers.Adam(lr_schedule),
              loss= SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

FIG. 7. Code run in google colab to compile layers of a neural network for an image classification model.

As can be seen in Figure 7, there is a variable 'lr_schedule' – this variable takes place of the learning rate. Generally, the default learning rate is .01; a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train. In our case, our learning rate will start on the order of magnitude of 10^{-6} but will slowly decrease; this is done so that the model does not learn too quickly as to not jump to conclusions about each image, as the backgrounds of a lot of the images can be similar (along with the person playing the violin remaining the same throughout this study thus far).

We are also making use of SparseCategoricalCrossentropy() to compute the crossentropy loss between the labels and predictions. Also called logarithmic loss, crossentropy loss predicts class probability and compares it to the actual class desired output 0 or 1. A score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. Generally, SparseCategoricalCrossentropy is more efficient when you have a lot of categories; this model has been optimized for a 3 or 4 class model.

As for the accuracy metric, it is used simply to visualize how well the model is doing, as opposed to looking at other metrics such as loss. We train our model in 40 epochs (training cycles), where the accuracy slowly increases into 90 percent accuracy.

C. Quantization

Now we have a compiled model! This model is a tensorflow model in the format .h5 which is incompatible with our OpenMV camera, but that is soon to change. By importing lite, we can convert our tensorflow model into a tensorflow lite model, reducing the size of the model from 80 MB to 30 MB. Because of this, the model is faster. TensorFlow Lite provides a set of tools that enables on-device machine learning by allowing developers to run their trained models on mobile devices including the OpenMV camera. It is simply a lighter version of TensorFlow, intended to provide the ability to perform predictions on a pretrained model.

Once our .tflite model is created, we can also perform post-training quantization on our model. Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. We are essentially rounding our numbers into int8 as opposed to float32 numbers, which saves a lot of space but should slightly decrease the accuracy. Due to the constraints of our OpenMV camera, we use integer-only quantization to quantize our .tflite model. The only difference is that the input and output tensors are in integer-only format.

Now, we can look into our quantized .tflite model along with our original .tflite model to classify images of violin playing.

IV. RESULTS

It is important to evaluate how effective our model is on a dataset separate from the training/validation images. On top of this, it is also important to see the differences in accuracy between the tensorflow models and the smaller tensorflow lite models.

As can be seen in Figure 8, both .tflite models do not have an accuracy close to 90 percent. It seems that there may have been a high chance of overfitting, but let us get into that later. The most interesting result is that our quantized model has a similar accuracy to the original

.tflite model. Sometimes, the quantized model has a better accuracy than the original model! While the model is not working perfectly, this can be a common occurrence due to the two models being so similar.

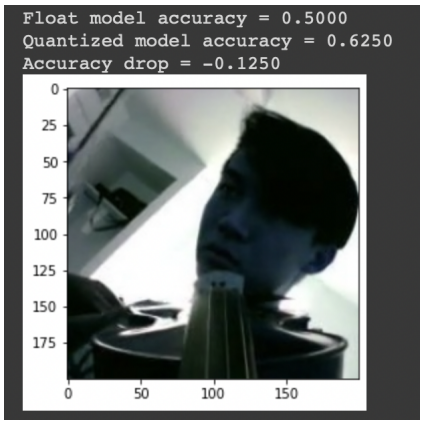

FIG. 8. Code output of testing accuracy for tflite models. Image captured as output to ensure accurate testing.

For practical use, the quantized model is very much more useful. Since it has a smaller file size, it runs much more smoothly on the camera, bolstering a high 14 frames per second while running on the OpenMV camera. That being said, the original .tflite model ran on 4 fps, which is really not going to be useful.

V. DISCUSSION

So the name of the game is creating a smaller file. There are a few ways to do this. One way is to convert the image data into greyscale. Initially, I had been doing this since our .tflite file was too big to even fit into the camera. However, since we were able to obtain larger storage in the microcontroller, I have been processing data on the RGB scale. On top of this, we are able to add more input layer units onto the dense model. Adding more input layer units creates a bigger file as well, as then the neural network trains on more units of data.

However, one issue with using the simple Dense model is that it is likely not complex enough for the problem at hand. Since each image is pretty similar to each other given that the camera is placed in a similar area, a lot of the images look the same. This can be seen in figures 4 and 5 as well – a lot of the images have a similar look. Some sort of feature detection (detecting the bow or the bridge of the violin) would be extremely helpful in determining if a violinist is playing his violin. This would be able to take my input data and map it into sort of a feature space. Then, we would be able to use this multi-dimensional representation of my data to get a prediction through the dense layer; dense layers are often the last step in a multi-layer model.

FIG. 9. Portion of output of model.summary() in python. Parameters provide a description of how complex image data is which is being processed.

Looking at figure 9, we can see that there are over 7 million values to represent our given 200x200 data. Because of this, there is a good chance that the model will vastly overfit due to memorizing training data; the model would run into difficulty classifying test data. So, to compensate, the model would just memorize the answer rather than learning from it.

To fix this, I attemped to use different models that I had found online: one made by Tensorflow itself, and one by a google colab user. However, these models did not fit our specified requirements. For one, most of these models were too big to even fit onto the microcontroller. This is an easy fix, as we can reduce the number of neurons in the neural network as to preserve space; we can also quantize our models to reduce space. This however created a second issue in that the OpenMV camera crashes when using these models. There is no good explanation for this issue, as the IDE does not return a reason for crash, but anything more than a simple dense model seems to struggle on the camera.

Another issue with the model is potential overfitting. As mentioned before, there is a high chance that the model is memorizing the dataset rather than learning about it; it is not adaptable. Overfitting is a concept in data science, which occurs when a statistical model fits exactly against its training data. When this happens, the algorithm unfortunately cannot perform accurately against unseen data, defeating its purpose. This is because the model is too fitted for the training data, and so it can not provide an accurate idea of what a general dataset should output.

One good way to prevent overfitting is to avoid randomly selecting a validation dataset. While this seems counter intuitive, it is much more effective in validating how well the model is performing. We do this by setting aside a dataset taken on a certain day, and using it as our validation dataset while data taken other days are used as training data. This makes it easier to validate whether the model is truly able to validate a general dataset. This is because it is possible that the model is memorizing specific features of the image that do not pertain to what is being classified. However, if the validation data is completely different, then we can better gauge the accuracy of the model; we are essentially testing the model on new, unseen data, and therefore more applicable.

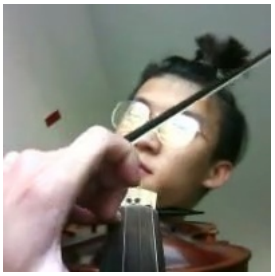

FIG. 10. Image of violinist playing pizzicato. Image appears as if the violinist is playing the violin with his bow, but he is plucking the string.

On top of this, the dataset is not very diverse. For one, most images look pretty similar as mentioned before, but there is another underlying problem. The entire dataset contains images of the same person playing the violin; there are no images of other people playing the violin. There are also other factors: all images were taken in the daytime, different violin scrolls can yield different rotations of the camera angle, and possibly others as well. In future work, it is important to create a more diverse dataset with more violin players with different lighting and backgrounds. The difference can even be as simple as someone getting a haircut – that could make a big difference!

FIG. 11. Image of violinist bouncing his bow off the string. While it appears the violinist is playing the violin, his bow is not visibly on a string.

Another aspect to note are the nuances associated with playing the violin. For one, pizzicato can come in many different forms: there is the traditional way to pluck the string using the right hand and index finger, but other violinists like to rest their thumb on the fingerboard or use their middle finger; it is all up to preference. On top of this, it is very difficult to distinguish pizzicato from playing with the violin bow (as can be seen in Figure 10). It is also possible to pluck the string with the left hand; if the left hand is down but the bow is not on the string, does this qualify as playing the violin?

Another more important nuance is the bouncing of the violin bow off the string. Often, violinists will bounce their bow off the string to get a bouncy sound, or to create a light sound when playing fast. However, when the bow is off the string, intuitively I am not playing the violin, but sometimes this is not the case. Take this violin performance in Figure 11, where the violinist's bow is clearly not on the string. If there were an OpenMV camera, ideally it would classify this posture as not playing the violin, but in this performance the performer is in the middle of playing violin notes.

Overall, the model has many issues and nuances but can be resolved with time.

VI. FUTURE WORK

The most immediate purpose of the image classification model is to detect whether the violinist is playing. It will be added to a system that uses a microphone to process live acoustic violin input that is digitally processed and passes through actuators and speakers installed at the base of a violin, inside a shoulder rest, that actuates the violin body. Depending on the processing, sometimes this system will produce undesirable feedback due to the closed loop. The image classification model will detect whether the violinist is in a position that would suggest intentional actuation of the instrument by bowing or plucking. Normally, the presence of acoustic signal would suffice to provide that information, but the actuation and feedback make that technique insufficient. The recognition of bowing or plucking can also be used to change the kinds of processing done to the acoustic violin signal, which extends and differentiates the performance possibilities.

[1] Adam, G. K., Kontaxis, P. A., Doulos, L. T., Madias, E.-N. D., Bouroussis, C. A., and Topalis, F. V. (2019). Embedded microcontroller with a ccd camera as a digital lighting control system. *Electronics*, 8(1).

[2] Pardue, L. S. and McPherson, A. (2019). Real-time aural and visual feedback for improving violin intonation. *Frontiers in Psychology*, 10.

[3] Rasamimanana, N. H., Fléty, E., and Bevilacqua, F. (2006). Gesture analysis of violin bow strokes. In Gibet,

S., Courty, N., and Kamp, J.-F., editors, *Gesture in Human-Computer Interaction and Simulation*, pages 145–155, Berlin, Heidelberg. Springer Berlin Heidelberg.

[4] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747.