PlayByEye: Transcription Piano Music Using Visual Recognition with Machine Learning

Aviv Khavich, Jeremy Kritz, Ridhima Sakhuja

***Abstract*—The purpose of our project is to transcribe piano music to MIDI files using only visual data. The process of piano music transcription can often be very lengthy and tedious. Applications have been developed in recent years to automate music transcription utilizing both audio and video information. However, our approach is novel in that it is based entirely off of machine learning from end to end in detecting which keys are played. Our approach also makes it possible to not only transcribe music from a pre-recorded video, but to also transcribe music in real-time from a live video feed. Each of the constituent steps achieved strong results, but further changes need to be made to make the end-to-end application product ready. Future extensions of this project could involve augmenting the application's transcriptive ability with audio data to further increase the accuracy of the model, applying such technology in the robotics field, and using the transcribed MIDI information for music prediction and synthesis.**

***Index Terms*—Convolutional Neural Networks, Difference Images, Homography, MIDI Files, Music Transcription, Semantic Segmentation, U-Net**

I. Introduction

PIANO transcription is the process of transcribing piano music into an alternate representation such as a MIDI file or sheet music. Musicians often use some of form of piano transcription when writing music. Representations such as MIDI files and sheet music are often useful for encoding various features of the music being played that may be standardized in audio feed. Such features include tempo, pitch, duration, and velocity of the music notes being played.

Traditional methods of music transcription typically involve the use of audio feeds to analyze various features of the music, such as pitch detection and tempo. While very experienced musicians can often transcribe music by ear, traditional methods of transcribing music can be both difficult and extremely tedious. Allowing it to be done simply and automatically with visual input can save an immense amount of time.

There has been extensive research into multi-pitch estimation and audio signal processing for music transcription, but less so in the field of computer vision. Previous works have tackled the problem with traditional computer vision approaches, such as Hough transforms and hard coded preprocessing of the images (Akbari et al. 2018). Instead, we take a novel approach to detecting a keyboard by utilizing state-of-the-art machine learning and computer vision techniques, including semantic segmentation, which is the sector of image classification wherein images are classified on a per-pixel basis, and classic convolutional networks. Semantic segmentation is often used in autonomous driving, biomedical imaging, and other applications.

While electronic keyboards can directly transcribe music in real-time, traditional pianos cannot, and our approach can allow transcription to be done both in real-time and post hoc with a pre-recorded video. The computer vision approach also does not suffer from the drawback of auditory noise, which is often present in settings where music is played. For this project, we chose to focus solely on a visual-based approach as a proof of concept. Even so,a hybrid model that includes both audio and video inputs would certainly improve accuracy.

A possible extension of our project could be to develop a method for taking into account audio signal processing to improve the models accuracy. The results of our project could also help in developing a feature for piano music similar to auto-complete for text messaging. The current transcription model only analyzes when a note is pressed or unpressed, but another potential extension would be training the model further to recognize the velocity of notes being played and encoding that into the output MIDI file as well. Our project would not only make piano transcription easier and efficient, but also has applications in aiding the disabled (for example, deaf musicians) as well as in robotics, since in the future, the process could conceivably be reversed to allow a robot to physically play a piano from MIDI

input.

A. Pipeline Components

Our pipeline is composed of four basic steps which are described below.

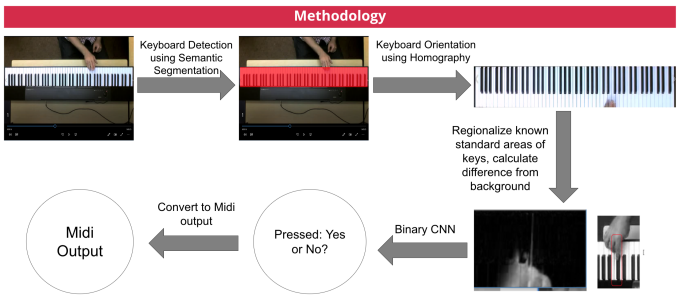

Fig. 1. The image above shows the pipeline for our project.

1) Semantic Segmentation:
The first and most complex step of our pipeline is the initial semantic segmentation of the keyboard in the frame. Semantic segmentation involves identifying classes in an image and assigning a per pixel class to segment the image into relevant components. In this case, we are searching for the quadrilateral of the piano keyboard in a video or webcam feed.
2) Homography and Regionalization:
The second step of our pipeline utilized homography and regionalization to separate the keyboard into key regions. We used homography to transform the detected keyboard contour into a normalized front-facing rectangular shape. Since most piano keyboards have a standardized layout with 88 keys starting with the same musical note, normalizing the keyboard in a video frame enables the piano keys to be in known locations for the next steps of the pipeline. The initial frame of the video, where the piano is visible with no notes played, is used as the background image. Once the keyboard is normalized, it is then regionalized into 88 separate rectangles, one for each key. Each key is subtracted from the background image to obtain the absolute difference image. Each difference image is then fed into a convolutional neural network for the next step of our pipeline.
3) Per-Key Convolutional Neural Networks:
The key difference images are inputted into a simple, fast, and relatively shallow convolutional neural network. The network analyzes the inputted key regions and outputs a binary classification to indicate if the key in question is pressed or not in the frame. In this step, there are two separate convolutional neural networks–one each for black and white keys respectively.
4) MIDI File Conversion:
The outputs of the convolutional neural network will then be mapped back to the keys, and each note will be tracked for the duration that it is pressed. When a key is pressed or unpressed (changes state), this change is transcribed to the output MIDI file.

Because the 88 images classified in each frame are so small, and the network is so shallow, the application easily classifies each frame within 30-50 milliseconds, the amount of time it takes to be real-time at 20-30 frames per second.

II. Methodology

A. Semantic Segmentation

The main component of real time semantic segmentation in our project is a neural network that we must train to identify a keyboard in an image. Such neural networks are typically known as fully convolutional neural networks. In order to train one, we needed two sets of images: the images we were classifying, and masks which contain the pixelwise classification information. We gathered our training dataset from four main sources: the largest portion was from a video dataset provided to us from a similar study (Akbari et al. 2018) that attempted to tackle the same problem by using classical computer vision techniques. We also gathered images of pianos from libraries like ImageNet and OpenImages, and frames from videos that we had recorded of ourselves playing the piano. After gathering the images, we wrote a Python script to manually create the masks for each image by clicking the corners of region of the keyboard in the image, or discard the image if the keyboard is absent, obscured, or invalid for any other reason. Once we had selected and saved the keyboard region in each picture, the script saved the original image, a gray scaled training image, and a labelled image consisting of a black polygon, to indicate the location of the keyboard, on a white background.

Semantic segmentation networks typically are composed of convolutional layers to encode/downsample the image, deconvolutional layers to decode/upsample the image, and skip connections in between. These skip connections prevented us from utilizing the standard Sequential Keras model, so we chose to implement our neural network using the Keras Functional API, which allowed us to use Python to develop non-sequential networks. Some of the neural networks commonly used for semantic segmentation include FCN-8 (an implementation of the FCN paper as an addendum to a pretrained VGG16 network), U-Net (a similar network typically used for biomedical imaging), ENet (a lightweight and fast, but less accurate network), and ICNet (an image cascade network used for high resolution images). After testing out the various architectures, we decided to use the U-Net model as the main neural network for semantic segmentation component of the pipeline, as it achieved the best results. We utilized binary cross entropy loss with the Adam optimizer.

Binary cross entropy loss is defined as follows

$$L = \frac{-1}{n}\sum_{i=1}^{n} y_i * ln(p_i) + (1 - y_i) * ln(1 - p_i)$$

where n is the number of pixels, y_i the class of the pixel, and p_i the probability that the pixel is in the class. With a binary classifier that determines whether the pixel in question is a piano or not,y_i either 1 or 0. p_i is the output of sigmoid layer at the end of the neural network and is a number between 0 and 1. This loss function is typically used for binary classifiers over average mean squared error (ASME), which is

$$L = \frac{1}{n}\sum_{i=1}^{n}(p_i - y_i)^2$$

because ASME overpenalizes errors compared to binary cross entropy. We experimented with different datasets and learning rates to achieve the highest accuracy and lowest loss. Our final learning rate was 1e-6. Further analysis of the results we had achieved with the U-net neural network are included in the Experimental Results and Discussion Sections. Once the image is classified, we used cv2.threshold() to make the mask binary. We then utilized cv2.findContours() to detect the largest contour in the image, which is likely to be the keyboard. Finally, we used cv2.approxPolyDP() to approximate the four corners of the detected polygon. These four points are used to calculate the homography in the next step.

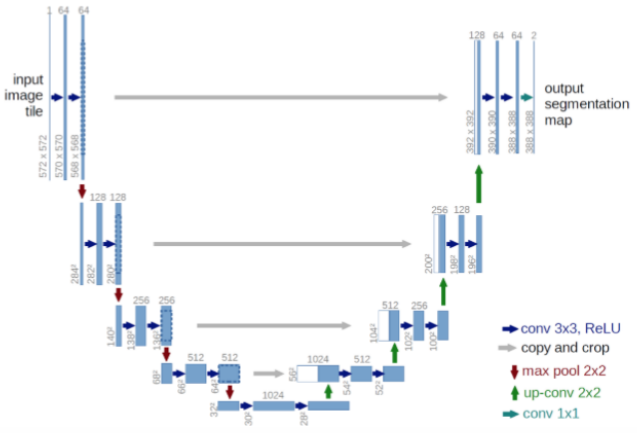

Fig. 2. The U-Net architecture. It involves four skip connections between the encoder and decoder, and differs from other semantic segmentation architectures in that these tensors are concatenated rather than added (Ronneberger et al.2019).

B. Homography and Regionalization

Since piano keyboards typically have a standard format of 88 keys, normalizing the keyboard orientation in each image is beneficial for training the convolutional neural networks further in our pipeline. The piano images derived from the results of the semantic segmentation component will virtually always be a parallelogram. Once the keyboard was located in an image, we transformed the keyboard into a normalized parallelogram using a classical computer vision technique called homography. Homography is the transformation between one set of 2D points in an image to another set of 2D points in an image. Typically, when a 3D object is imaged through the image formation pipeline, a 3x4 camera matrix M is used to transform the 3D world point into a 2D camera point. That is,

$$\begin{bmatrix}\tilde{x}\\ \tilde{y}\\ \tilde{w}\end{bmatrix} = \begin{bmatrix}m_{11} & m_{12} & m_{13} & m_{14}\\ m_{21} & m_{22} & m_{23} & m_{24}\\ m_{31} & m_{32} & m_{33} & m_{34}\end{bmatrix}\begin{bmatrix}X\\ Y\\ Z\\ 1\end{bmatrix}$$

where the pixel coordinates (x, y) of the point are $x = \tilde{x}/\tilde{w}$ and $y = \tilde{y}/\tilde{w}$. Then, for points p_1 and p_2 in images 1 and 2 of a point P, $p_1 = M_1P$ and $p_2 = M_2P$. However, if we assume that all the

points P we are imaging lie on a planar surface, we can set the world coordinates of that planar surface to be $Z = 0$, that is, the XY plane. This allows us to remove the Z column of the camera matrices, which turns them into 3x3 invertible matrices H_1 and H_2. We can then get P from p_1 by inverting H_1, that is, $P = H_1^{-1}p_1$. Thus, points p_1 and p_2 can be related by the equation

$$p_2 = H_2(H_1^{-1}p_1)$$

We then define the 3x3 matrix $H = H_2 * H_1^{-1}$ to be our homography matrix. We now have a relation between two imaged points that can be written as

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{w}_2 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Multiplying the matrices, we obtain the following equations:

$$\tilde{x}_2 = h_{11}x_1 + h_{12}y_1 + h_{13} \quad (1)$$

$$\tilde{y}_2 = h_{21}x_1 + h_{22}y_1 + h_{23} \quad (2)$$

$$\tilde{w}_2 = h_{31}x_1 + h_{32}y_1 + h_{33} \quad (3)$$

Since $x_2 = \tilde{x}_2/\tilde{w}_2$ and $y_2 = \tilde{y}_2/\tilde{w}_2$, it follows that

$$x_2 = \frac{h_{11}x_1 + h_{12}y_1 + h_{13}}{h_{31}x_1 + h_{32}y_1 + h_{33}} \quad (4)$$

$$y_2 = \frac{h_{21}x_1 + h_{22}y_1 + h_{23}}{h_{31}x_1 + h_{32}y_1 + h_{33}} \quad (5)$$

Multiplying by the denominators,

$$x_2(h_{31}x_1 + h_{32}y_1 + h_{33}) - h_{11}x_1 - h_{12}y_1 - h_{13} = 0 \quad (6)$$

$$y_2(h_{31}x_1 + h_{32}y_1 + h_{33}) - h_{21}x_1 - h_{22}y_1 - h_{23} = 0 \quad (7)$$

This allows us to construct a system of linear equations $Ax = 0$, where

$$x = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{bmatrix}^T$$

and A is a $2N$x9 matrix, with N being the number of point correspondences. Every two rows of A are

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_2x_1 & x_2y_1 & x_2 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & y_2x_1 & y_2y_1 & y_2 \end{bmatrix}$$

Given 4 point correspondences, the matrix A can be solved through Singular Value Decomposition, such that $A = U\Sigma V^T$. The last column of V^T is the solution to the equation, which gives us our homography matrix H. This matrix can then be multiplied by any point in image 1 to apply the given transformation to image 2. This is achieved through cv2.warpPerspective(). Since our piano can be assumed to be a planar surface, and since we find four points outlining its shape through semantic segmentation, we can correspond those four points to a normalized rectangle, and thus transform our quadrilateral polygon to that rectangular shape.We can then approximate the orientation of the rectangle to determine which side is the top by searching for darker pixels. If darker pixels are overwhelmingly concentrated on the left, right, or bottom of the rectangle, we can assume the orientation to be incorrect. This is because in a normal piano, darker pixels should be concentrated near the upper portion of the keyboard due to the black keys.

To implement homography in our project, we used methods from OpenCV. More specifically, we used cv2.convexHull() to force convexity on our four detected corners, cv2.findHomography() to find a perspective transform between the corners and the set of points [0, 0], [0, 80], [624, 80], [624, 0] (a 624x80 pixel rectangle), and cv2.warpPerspective() to apply the derived transform, the calculations of which are shown in the previous subsection, on the video frame.

Four homographies are calculated, one for each rotation of the four corner points of the front-facing rectangle. As we desired the orientation with the black keys on top, we needed a way to quickly approximate which orientation was the correct one. We solved this by averaging the pixels in the upper 40 pixels of the rectangle. In the correct orientation, this region had the heaviest concentration of black pixels due to the black keys. Thus, the minimum of the four averages was likely to be the correct orientation. The image below shows an example of the image outputted after applying the homography and orientation.

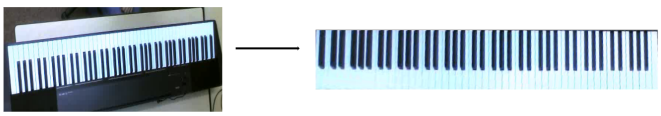

Fig. 3. By applying homography, we can transform the keyboard, which is at an angle and slanted (left), into a normalized front-facing rectangle (right).

Following homography, the next step was to regionalize the keyboard into 88 separate rectangles–one for each key in a standard piano keyboard. These positions were hard-coded based on the target normalized rectangle we determined.

C. Per-Key Convolutional Neural Networks

We initially intended to simply feed the raw image of the keys into the two convolutional neural networks to classify them. However, after some experimentation, we observed that the networks performed rather poorly, and that we often ourselves could not tell when a key was pressed or not. Instead, we opted to follow Akbari et al.'s example and utilize difference images. This way, variance in the quality, lighting, angle, etc. of a key would not affect accuracy of classification, as it would be compared to a baseline background image and be clearly different.

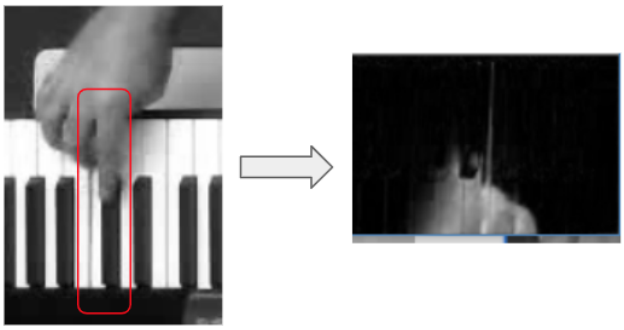

Fig. 4. The above image is an example of the effectiveness of difference imaging.

For this step, we wrote a Python script that allowed us to go through every frame of a video. We initially took a frame with no keys being pressed, clicked the corners of the keyboard, which performed a homography and normalized the keyboard. This was our background image for labelling the dataset. We then went through the video and picked out interesting frames where keys were pressed. The script would cycle through the white and black keys and we manually labeled each one as pressed or unpressed. The program would then save the absolute difference image between the frame and the background image for each key. This was our dataset for the key-wise networks.

We utilized two separate convolutional neural networks to determine the status of keys–one for white keys and one for black keys. At the beginning of each video, semantic segmentation is used to locate the keyboard in the image and find the homography transform, which is saved. This initial frame with a blank keyboard is also saved as a background image. Then, assuming the camera is immobile throughout the video, each subsequent frame is subjected to the same homography. The keyboard is subtracted at each frame from the background to calculate the absolute difference image. Each regionalized key is then fed into one of the two networks according to its color. The input shape to the white key classifier was 22x80, while the input to the black key classifier was 22x60. Both the white and black key networks had identical architectures: a convolutional layer followed by a maxpool layer, followed by a batch normalization layer. This is then repeated a second time. The output of the second batch normalization is then flattened and fed into a dense layer outputting to the single output neuron, which has a sigmoid activation. Once again, we utilized binary cross entropy loss with the Adam optimizer. We utilized batch normalization as opposed to dropout as the former has been proven to be more effective following convolutional layers, due to the fact that activations in a convolutional layer are highly correlated as well as having relatively few parameters.

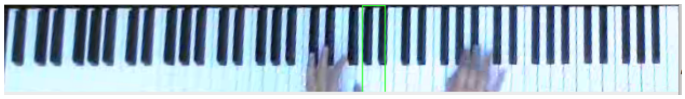

Fig. 5. Screenshot from our initial labeling script.

Fig. 6. Screenshot from our adjusted labeling script utilizing difference images.

```
model = Sequential()
model.add(Conv2D(8, kernel_size=(3, 3), padding='same',
                 strides=(1, 1), activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(16, kernel_size=(3, 3), padding='same',
                 strides=(1, 1), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(1, activation="sigmoid"))
```

Fig. 7. The architecture of each key classifier.

D. Transcribing to MIDI

Once we know when a key is pressed or not, it is trivial to analyze the time frames at which the key is initially pressed, and when it is no longer pressed, i.e. state change. We achieved this with a running array during the video which kept track of which keys were pressed. When a key was added or removed from the array, the appropriate message (”note_on” or ”note_off” respectively) was encoded into the MIDI track corresponding to that note. The time of the last change was also tracked in order to encode the delta time of each message. We constructed the MIDI file using the Mido Python library.

III. Experimental Results

A. Semantic Segmentation

A significant component of our project was using semantic segmentation to identify a keyboard a given set of video frames. As discussed in the Methods section, we had labeled the video frames to create masks, where the keyboard was denoted as a black rectangle and the rest of the image was white. We then used these masks to train our implementation of the U-Net architecture. The following images are the initial results from the convolutional neural network.

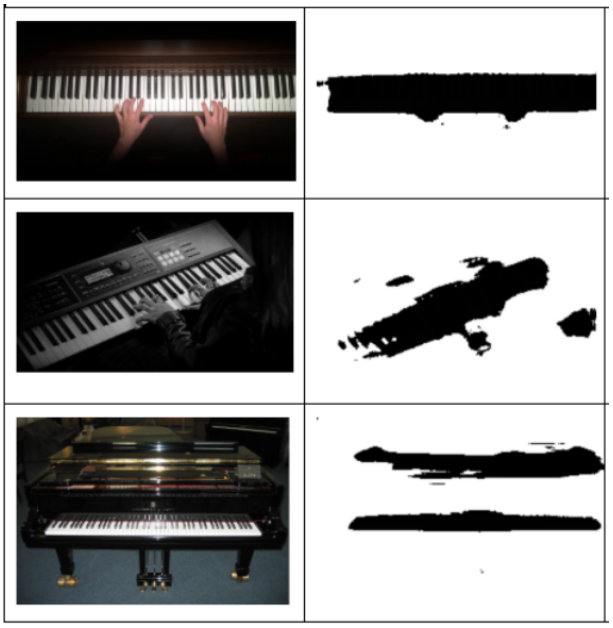

Fig. 8. Initial results from our semantic segmentation neural network

While the results in the top row seem to be good, since the keyboard is accurately identified, the results in the middle row seem to indicate that there is some degree of overfitting, as regions that were not part of the keyboard are also labelled in black. The keyboard in the last row was not identified correctly, which could have been due to the light reflection seen in the original keyboard image. In general, as the vast majority of training was performed on overhead shots of keyboards, that is what our network excelled at. In the future, we would aim to expand the dataset significantly with more videos of our own to avoid this problem. However, for testing purposes and for our live demonstration, we limited ourselves to the kinds of overhead shots provided to us by the Akbari et al. study.

The original results were from the neural network after it was trained on the dataset without any image augmentation. In order to improve the accuracy of the neural network, we used Keras’s ImageDataGenerator to create a generator to augment the dataset. We utilized random zooms, rotations, brightness changes, shear transforms, and vertical flips to augment the dataset and improve its generalizability, as well as varying learning rates. On our sixth training attempt, we achieved a network that we felt was consistent enough to move on with for demonstrative purposes. The results of the semantic segmentation neural network after augmenting the training set are shown below.

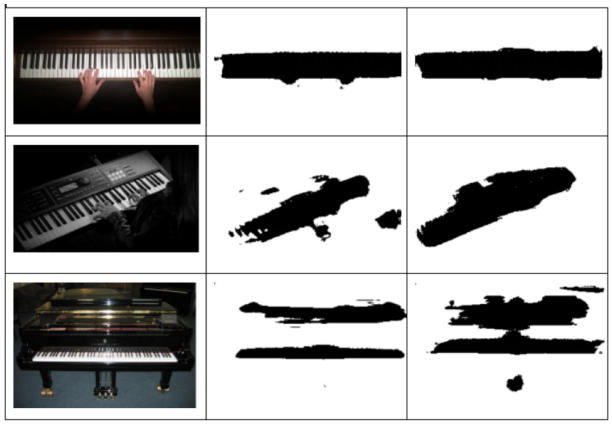

Fig. 9. Comparison of semantic segmentation neural network results: the original images are shown in the leftmost column, the initial mask results are in the middle column, and the improved results are in the rightmost column.

The image above shows a comparison of the results received previously and after we expanded the training dataset. The left most column shows the original images, and the middle column shows the initial results. The results after we expanded

the dataset are shown in the rightmost column. The results in the top row seem to be approximately the same. However, in the middle row, the results seem to be less overfitted as only the keyboard region is labelled as black. The last row results have improved a little, but are still not very accurate, which could be due to the nature of the image.

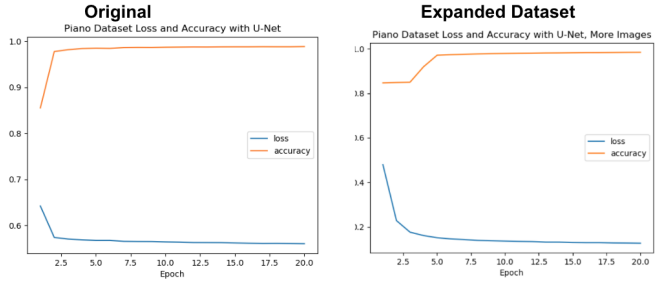

Fig. 10. Graphs comparing results of original semantic segmentation and segmantic segmentation with expanded dataset. Orange line shows accuracy and blue line shows binary cross entropy loss.

The graphs above show loss and accuracy with neural network both before and after expanding the dataset. The loss and accuracy for the expanded dataset takes slightly longer to "max out" due to the more complicated dataset.

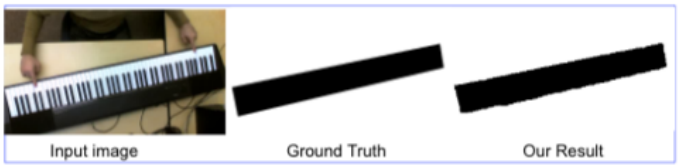

Fig. 11. Semantic Segmentation results from final version of neural network.

We trained our network on Akbari et al.'s training set and tested on their test set. On the test videos, we achieved a 98.6% test accuracy rate and a precision of 94.7%. Our recall rate was 98.5% and the F-Score was 96.5%. We consider these to be very successful results for our U-Net implementation.

B. Homography and Regionalization

In order to standardize our data, we had used homography to normalize the orientation of the keyboards in the video frame. Afterwards, we divided the keyboard into 88 separate rectangles for the key classification datasets. While the actual process homography and regionalization is always correct in theory, it is highly dependent on accurate corner approximation with cv2.approxPolyDP(). Before using OpenCV's approximation method, we attempted to write our own to approximate corners. However, this achieved poorer results. Even though our semantic segmentation was often close to 100% in accuracy, the ridges and irregularities that did occur could often throw off results.

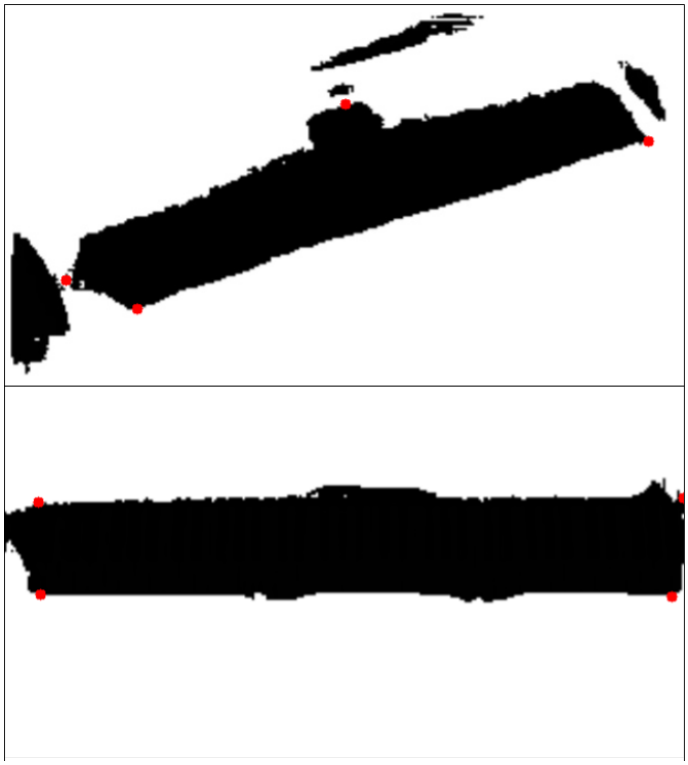

Fig. 12. An example of a poor corner approximation (top) and a good approximation (bottom) with cv2.approxPolyDP(). The poor result also had an imperfect semantic segmentation, which led to the error. This illustrates how key an accurate semantic segmentation was to the pipeline.

C. Per-Key Convolutional Neural Networks

Our key-wise networks achieved fairly good results considering the relative shallowness and small input size. In testing, the white key CNN achieved an accuracy of 87.3%, a precision of 79.1%, a recall of 94.6%, and an f-score of 86.2%. Likewise, the black key CNN achieved an accuracy of 87.6%, a precision of 68.9%, a recall of 97.7%, and an f-score of 80.8%. While accuracy was strong, around 90% for both, the networks suffered most from poor precision.

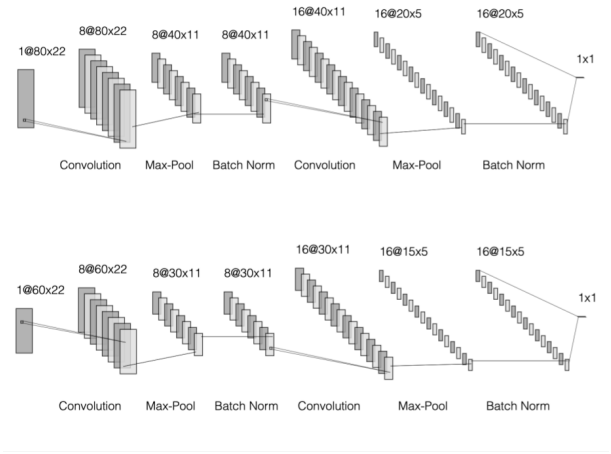

Fig. 13. The architecture of our two key classifiers.

D. MIDI Output

Our final output from the pipeline is a MIDI file encoding the time at which any note is pressed or unpressed. Due to the multi-layered complexity of the pipeline, the results are not perfect, but we consider them to be very successful as a proof of concept. As seen below, despite some irregularities, the shape of the output MIDI file very closely approximates the input test video's ground truth MIDI file (ground truth provided by Akbari et al.).

Fig. 14. The ground truth of what was played in the video, with time as the horizontal axis and notes as the vertical axis.

Fig. 15. The output of the pipeline.

IV. Discussion

A. Sources of Error

One source of error for us as of now has been overfitting in the semantic segmentation network. Choosing a network that properly balances performance and inference speed is a challenge. As we intended the final result of this project to be real-time classification, we chose to go for a relatively light network, and although it is quite accurate, even more accurate, but slower results might be achieved with a deeper network.

Our dataset consists of relatively very similar frames, but we expect that even better results could come through a properly variant dataset.

However, in practice, as we tested on videos that largely consisted of overhead shots similar to our training set, this error was not noticeable in testing. Even when the segmentation is mostly accurate, however, small regions of false positives can occur, and corner detection becomes increasingly difficult in these situations. As such, corner approximation for the homography became our largest source of error. The result would be a distorted keyboard, as shown below.

Fig. 16. The above image shows an example of a distorted keyboard when corner approximation is imperfect.

Fig. 17. The above image shows the results of semantic segmentation on the keyboard shown previously. The region on the right impairs corner approximation as it is included in the overall piano contour.

Even when the keyboard is slightly distorted, this creates significant issues, as our approach detects pressed keys in hard-coded rectangular regions. In the above example, the first 8 keys would be set into 8 separate regions based on an actual keyboard layout, but these regions are not representative of where the keys actually are in frame.

This error can be rectified in two ways. One is a more accurate semantic segmentation network, although we believe we have come close to the upper limit of what could be expected with a relatively fast architecture like U-Net. The more pertinent fix would be a smarter corner approximation algorithm.

Using Hough lines on the binary mask would be an example, as well as developing an algorithm to take into account how much of the area of the approximated quadrilateral is not keyboard pixels. We made attempts at such an algorithm during the process, but they were too slow for final implementation. As such, we relied on OpenCV for this, but our needs require a much more accurate point correspondence for acceptable homography.

The second largest source of error came from the relatively low precision of the key classifiers. While accuracy was at around 90% for both, the white key classifier had a precision of only about 80%, and the black key classifier 70%. This meant that we experienced many false positives during classification. The bulk of this error, in our opinion, arises from an incomplete training dataset which underfitted to undesirably include things like fingers occluding keys, and noise from surrounding keys being pressed. Some of this cross-correlation between nearby keys is unavoidable, but our networks' performances could certainly be improved.

When these two sources of error are compounded throughout the pipeline, the final result can be jittery and off by a note or two in the final result. However, we see this as a successful proof of concept that a machine learning approach from end to end is both possible and desirable, and believe that with sufficient work, this approach could be comparable to or better than others.

B. Transition to Live Demonstration

We also performed a live demonstration of this pipeline utilizing a purchased webcam and keyboard. Some sources of error arose when transitioning to a live demo. All three neural networks used in our project were trained with data taken from our training set of videos. Live demos had the camera in a slightly different location, and had significantly different lighting, with shadows and different levels of light on different areas of the keyboard. This led to semantic segmentation being somewhat worse than in the training videos, but not to a very large extent. Mostly, the error came from the aforementioned corner approximation, and the problem with the key regions not matching up, as described above, often occurred. However, as a whole, our live demonstration still generally approximated the shape of our input, showing that our approach has strong potential for generalizability.

A graphical interface was also developed for this demonstration to visually show which keys were being pressed throughout the duration of each run.

C. Extensions

For future work and possible extensions, a machine learning approach to better corner detection, rather than the existing functions in OpenCV, could possibly cut down on error. Alternatively as mentioned earlier, that an algorithm taking into account proportional keyboard pixels within a quadrilateral would achieve a much more accurate corner approximation.

Limiting the size of the key regions may improve resistance against false positives from surrounding keys. However, this needs to be balanced so as not to remove information necessary to classification. This would also be highly dependent on accurate homography.

In addition to homography, further corrective affine transforms could be employed to ensure a normalized rectangle. Corrections for fisheye distortion in camera lenses would also improve accuracy.

More videos from multiple angles, as well as more varied keyboards in training would improve the semantic segmentation capability of our network.

Additionally, we believe it still may be possible to train the key classifiers on raw images rather than difference images. If this were achieve with a sufficient accuracy, the immobilization constraint on the camera could be lifted, and we could use semantic segmentation in each frame rather than only once per run. This was our original approach, until we found that the key classifiers were not as accurate as we would have liked. Such an approach would also be slower, as semantic segmentation would take up most of each frame's computation. This would also allow us to expand to mobile devices, which are much shakier than a camera on a tripod, but are more realistic for everyday use of such an application. The models could be converted for mobile use using Tensorflow Lite, although it remains to be seen how fast this would be.

A potential extension for this project would be to not only use visual recognition for transcribing piano music, but also audio recognition, such as multi-pitch estimation. Using both audio and visual features to determine the notes being played would

potentially help improve the models accuracy. Furthermore, while there have been applications that have used either computer vision or audio signal processing techniques to transcribe music, there have been little to no such applications that have combined both audio and visual processing. This is a potential area for the field to expand to.

As mentioned before, an additionally potential extension is that was not in the scope of our project would be to predict future notes or to auto-complete a song similar to how the auto-complete feature works in text messages. This would be unrelated to vision, and would be achieved through a recurrent neural network or a one-dimensional convolutional neural network. In a similar vein, music recognition from the MIDI output could be implemented, in an approach similar to Shazam and other music recognition applications.

Reversing the pipeline would also be useful in controlling a robot arm with inputted MIDI information. The MIDI input would provide the keys to be pressed, at which point the camera would semantically segment the image, perform the homography, regionalize the keyboard, locate the appropriate key region, and press the key.

D. Cost Analysis

Our project is entirely software based, whether running on post hoc recorded videos or in real time. In either case, one would only need to download the application, and either select a video to analyze, or use the camera on the device to record in real time. Developing the project was also entirely free, involving only software. We developed the pipeline and trained the networks on Google Colaboratory, a free online Jupyter notebook provided by Google which also provides free computational resources, including CPU, GPU, and even experimental TPU. Colaboratory allows us to interface with our Google Drive as a file system, where we stored our datasets. All of this project was done entirely within this free framework.

To run this application, needless to say, requires a piano or electric keyboard, a webcam, and a tripod or other type of stand to keep it steady. For our live demonstration, we used an Alexis 88-key full-size beginner keyboard costing $209.99. This project works irrelevant of the piano being used, provided it has 88 keys following a normal black-and-white color scheme. To capture video of the piano, we used a Logitech desk mounted tripod, and a Logitech 1080p widescreen webcam, costing $19.98 and $66.65 respectively.

If this project were to be made into a product, however, it would not make sense to sell any of the related hardware, and the release would exclusively be the software.

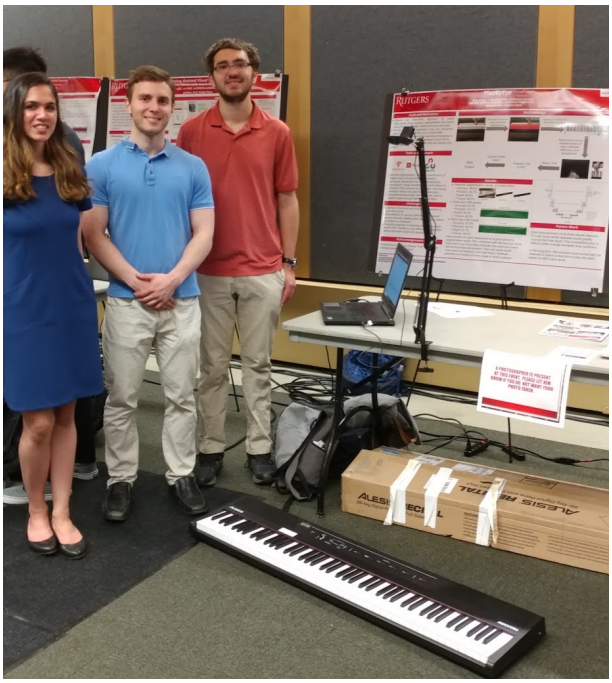

Fig. 18. Our setup for the live demonstration, including our webcam, clamp, and keyboard.

V. Conclusion

Using the discussed pipeline and methodology, we were able to develop an application that can transcribe piano music into a MIDI file both from a post hoc video and in real time using a web camera and a keyboard. In the end, we achieved a high accuracy in each constituent step, which resulted in fair performance overall. Our application has significant implications for the music transcription field, as our approach can potentially make piano transcription easier and more accurate for musicians. We continue to believe an end-to-end machine learning approach is the most desirable for future applications.

VI. Robotics and Society

In the last couple years, there have been many real world robotic systems that are related to making piano music transcription more automated using machine learning, computer vision, and audio signal processing. The following three papers are discussed in this section: claVision: Visual Automatic Piano Music Transcription, Key Detection for a Virtual Piano Teacher,and Observing Pianist Accuracy and Form. We also discuss the use of semantic segmentation in autonomous driving, as a major component of our project pipeline was semantic segmentation.

A. claVision: Visual Automatic Piano Music Transcription

The first paper, claVision: Visual Automatic Piano Music Transcription, is based off a research program that was conducted at Simon Fraser University. The paper introduces a novel vision based system called claVision that performs automatic piano music transcription through the analysis of videos that show a person playing a keyboard. The purpose of claVision was to aid musicians who have hearing disabilities in transcribing music. For this reason, claVision is one of the first systems that can transcribes music without relying on audio data. Instead of analyzing the audio of the videos, the software visually analyzes the music played in the videos by tracking the persons hand and which keys are being pressed, and then converts the transcripted music into a midi file and sheet music. Capabilities have been developed for the system to transcribe music over both pre recorded videos of piano music as well as real time processing of piano music. The claVision software is able to achieve an accuracy rate of approximately 95% in transcribing piano music over various different speeds and complexities and sustains a relatively low latency time. The following image is a screenshot from sheet music claVision produced from transcribing Twinkle Twinkle Little Star(Akbari et al. 2015).

B. Key Detection for a Virtual Piano Teacher

The second paper, Key Detection for a Virtual Piano Teacher, discusses an augmented reality based approach to a virtual piano teacher application. The purpose of such an application is to enable users to learn how to play the piano through highlighting and

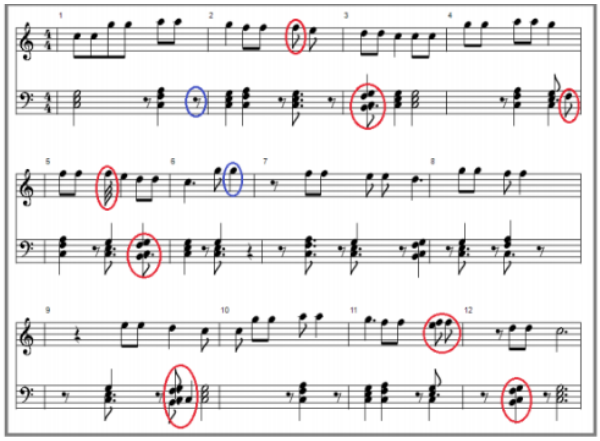

Fig. 19. Twinkle Twinkle Little Star transcription

fingering information overlaid on the keyboard in a computer monitor, and therefore allowing users to see exactly which notes they need to play. Unlike the claVision software which processes keyboards through video feeds, this application exclusively requires real time detection of the keyboard and the piano keys. Currently, the virtual piano teacher only supports the teaching of chords and scales, as well as basic finger techniques catered towards novice piano players. Even so, the following application introduces a novel approach to teaching beginners how to play the piano. Traditional methods of learning to play the piano generally include either taking lessons or learning from books and/or the internet. Although it may not be very useful for more advanced piano players due to what the application currently supports, the virtual piano teacher could potentially provide a more efficient and easier way to teach fundamental practical and theoretical skills that are required for playing(Goodwin et al. 2013).

C. Observing Pianist Accuracy and Form with Computer Vision

The third paper, titled Observing Pianist Accuracy and Form with Computer Vision, describes an alternate piano tutoring system that observes the user playing a piano and gives the user feedback about their technique and accuracy. The primary objectives of this project were to identify which notes are being pressed at a given time and to determine which finger is pressing the given notes. Like our project's pipeline, the pipeline for this particular project also included a convolutional neural network that took in both audio and visual input for

Fig. 20. Virtual Piano Teacher

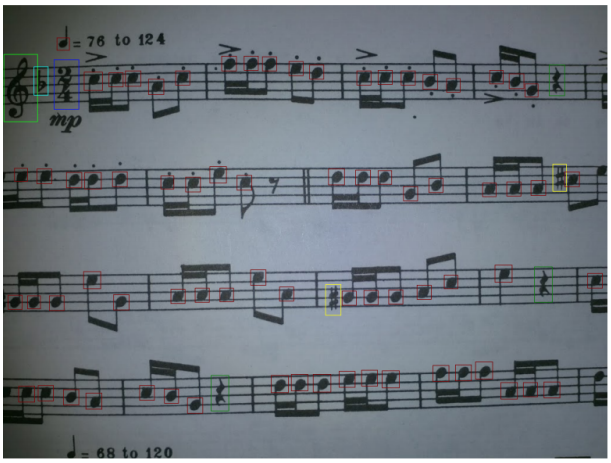

Fig. 21. The following image is an example from paper discussed.

determining notes and fingers.
As with the previously discussed paper, the purpose of such a project is to make learning the piano more widespread, convenient, and affordable. Traditional method of learning the piano typically involve one on one interaction with a tutor. This can typically be very costly and time consuming. With interactive piano systems such as this one and the previous one, learning to play the piano can potentially become easier and more affordable for larger groups of people. Such initiatives could potentially be extended to other instruments, and make learning to play music in general more cost efficient and widespread.
One of the biggest differences between this project and the previously discussed project is that the former project enables users to play along with the application as it highlights the keys that need to be played. On the other hand, this project provides feedback about the user's musical accuracy and technique(Lee et al. 2019).

D. Semantic Segmentation in Autonomous Driving

An important usage of real-time semantic segmentation similar to our implementation is used in autonomous driving technology, which is being developed by a number of leading technology companies. To make sense of what a car sees, semantic segmentation is used to separate the camera image into relevant regions corresponding to objects in view, such as people, other cars, the road, signs, and so on. One such system is described in the paper Speeding up Semantic Segmentation for Autonomous Driving. As you can see below, the systems required to make autonomous driving safe and effective are quite a bit more complex than a binary classification of whether a pixel is part of a keyboard, but the underlying concept and technology are the same. This example uses

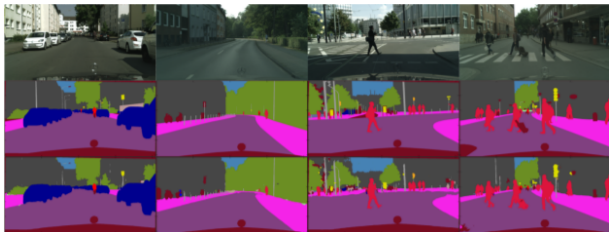

Fig. 22. Examples of Semantic Segmentation for Autonomous Driving. Ground truth above results

the Cityscapes dataset, which is used for semantic segmentation, with pixels classified into the classes flat, nature, object, sky, construction, human, and vehicle. We used this dataset very early on when testing which semantic segmentation architecture made sense for this project.

Like with U-net, their proposed system uses "skip-connections", or pipelines that bring data from early layers directly to later ones, just as can be seen in the diagram of U-net's architecture. This helps to combine general feature detection, such as recognizing a person or vehicle, which typically happens at higher layers, with accurate per-pixel classification of small features, which happens in lower layers.

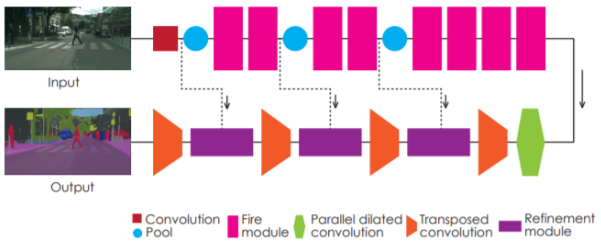

Fig. 23. Architecture for Semantic Segmentation for Autonomous Driving.

APPENDIX A
TRAINING CODE FOR SEMANTIC SEGMENTATION

This code was run in Google Colaboratory, a free online Jupyter notebook with allocated GPUs.

```
###TRAINING CODE###
import numpy as np
import cv2
from glob import glob
import os
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.preprocessing.image import ImageDataGenerator
from keras import backend as keras

image_dim = (160, 288)

def adjustData(img,mask):
  img = img / 255
  mask = (mask == 0)
  return (img,mask)

def trainGenerator(batch_size,train_path,image_folder,mask_folder,
                   aug_dict,
                   image_color_mode = "grayscale",
                   mask_color_mode = "grayscale",
                   image_save_prefix  = "image",
                   mask_save_prefix  = "mask",
                   flag_multi_class = False,num_class = 2,
                   save_to_dir = None,target_size = (256,256),
                   seed = 1):
    '''
    can generate image and mask at the same time
    use the same seed for image_datagen and mask_datagen to ensure
    the transformation for image and mask is the same
    if you want to visualize the results of generator,
    set save_to_dir = "your path"
    '''
    image_datagen = ImageDataGenerator(**aug_dict)
    mask_datagen = ImageDataGenerator(**aug_dict)
    image_generator = image_datagen.flow_from_directory(
        train_path,
        classes = [image_folder],
        class_mode = None,
        color_mode = image_color_mode,
        target_size = target_size,
        batch_size = batch_size,
        save_to_dir = save_to_dir,
        save_prefix  = image_save_prefix,
```

```
        seed = seed)
    mask_generator = mask_datagen.flow_from_directory(
        train_path,
        classes = [mask_folder],
        class_mode = None,
        color_mode = mask_color_mode,
        target_size = target_size,
        batch_size = batch_size,
        save_to_dir = save_to_dir,
        save_prefix  = mask_save_prefix,
        seed = seed)
    train_generator = zip(image_generator, mask_generator)
    for (img,mask) in train_generator:
        img,mask = adjustData(img,mask)
        yield (img,mask)

def unet(pretrained_weights = None,
        input_size = (image_dim[0],image_dim[1],1)):

    inputs = Input(input_size)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(pool1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv4)

    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)

    up6 = Conv2D(
```

```
    512, 2, activation = 'relu',
    padding = 'same',
    kernel_initializer = 'he_normal')(
        UpSampling2D(size = (2,2)
        )
    (drop5)
    )

merge6 = concatenate([drop4,up6], axis = 3)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(merge6)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(conv6)

up7 = Conv2D(
    256, 2, activation = 'relu',
    padding = 'same',
    kernel_initializer = 'he_normal')(
        UpSampling2D(size = (2,2))
        (conv6)
    )

merge7 = concatenate([conv3,up7], axis = 3)

conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(conv7)

up8 = Conv2D(
    128, 2, activation = 'relu',
    padding = 'same',
    kernel_initializer = 'he_normal')(
        UpSampling2D(size = (2,2))
        (conv7)
    )

merge8 = concatenate([conv2,up8], axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                kernel_initializer = 'he_normal')(conv8)

up9 = Conv2D(
    64, 2, activation = 'relu',
    padding = 'same',
    kernel_initializer = 'he_normal')(
        UpSampling2D(size = (2,2))
        (conv8)
    )
```

```
    merge9 = concatenate([conv1,up9], axis = 3)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(merge9)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv9)
    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
                   kernel_initializer = 'he_normal')(conv9)
    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

    model = Model(input = inputs, output = conv10)

    model.compile(optimizer = Adam(lr = 1e-6),
                  loss = 'binary_crossentropy',
                  metrics = ['accuracy'])

    #model.summary()

    if(pretrained_weights):
        model.load_weights(pretrained_weights)

    return model

training_dir = '/content/drive/My Drive/Colab/Dataset/train/'

data_gen_args = dict(rotation_range=45,
                     shear_range=0.5,
                     zoom_range=[1,1.25],
                     vertical_flip=True,
                     brightness_range=(0.5, 1.5),
                     fill_mode='nearest')

gen = trainGenerator(16, training_dir, 'image', 'label', data_gen_args,
                         target_size=image_dim, save_to_dir = None)

print("done generating array")
model = unet()

model_checkpoint = ModelCheckpoint(
    '/content/drive/My Drive/Colab/data/unet_piano_2.hdf5',
    monitor='loss',
    verbose=1,
    save_best_only=True
    )

model.fit_generator(gen, steps_per_epoch=100, epochs=20,
                    callbacks=[model_checkpoint])
```

APPENDIX B
TRAINING CODE FOR KEY CLASSIFIERS

```
import numpy as np
import cv2
from glob import glob
import os
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.preprocessing.image import ImageDataGenerator
from keras import backend as keras
import sys

train_dir = '/content/drive/My Drive/Colab/Dataset/keys/white'

input_shape = (80, 22, 1)

model = Sequential()
model.add(Conv2D(8, kernel_size=(3, 3), padding='same',
                 strides=(1, 1), activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(16, kernel_size=(3, 3), padding='same',
                 strides=(1, 1), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(1, activation="sigmoid"))
model.summary()
model.compile(optimizer = Adam(lr = 1e-4),
              loss = 'binary_crossentropy', metrics = ['accuracy'])

batch_size = 16

# this is the augmentation configuration we will use for training
train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=15,
        shear_range=20,
        horizontal_flip=True,
        #brightness_range=(0.1, 1.5),
        fill_mode='nearest')

train_generator = train_datagen.flow_from_directory(
        train_dir,  # this is the target directory
        target_size=(input_shape[0], input_shape[1]),
# all images will be resized
```

```
        batch_size=batch_size,
        color_mode = 'grayscale',
        class_mode='binary')

model_checkpoint = ModelCheckpoint(
    '/content/drive/My Drive/Colab/data/white_keys.hdf5',
    monitor='loss',verbose=1, save_best_only=True)
model.fit_generator(train_generator, steps_per_epoch=20,
    epochs=300, callbacks=[model_checkpoint])
```

APPENDIX C
FINAL PIPELINE CODE

```
import time
import cv2
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as keras
from glob import glob
from itertools import combinations
import os
from mido import Message, MidiFile, MidiTrack, second2tick

keras.clear_session()

testing_dir = '/content/drive/My Drive/Colab/Dataset/test'
semseg_model_dir = "/content/drive/My Drive/Colab/data/unet_piano_5.hdf5"

video_dim = (640, 360)
prediction_dim = (288, 160)

semseg_model = load_model(semseg_model_dir)

white_key_model_dir = "/content/drive/My Drive/Colab/data/white_keys2.hdf5"
black_key_model_dir = "/content/drive/My Drive/Colab/data/black_keys2.hdf5"
white_key_model = load_model(white_key_model_dir)
black_key_model = load_model(black_key_model_dir)

cap = cv2.VideoCapture(os.path.join(testing_dir, "V1.wmv"))
fps = cap.get(cv2.CAP_PROP_FPS)
print(fps)
frame_delta = 1./fps

frame_arr = list()
image_arr = list()

while(cap.isOpened()):
      # Capture frame-by-frame
      ret, frame = cap.read()
      if ret:
        img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        frame_arr.append(img)
        img = cv2.resize(img, (288, 160))
        img = np.reshape(img, img.shape + (1,))
        img = img/255
        image_arr.append(img)
      else:
        break
```

```
cap.release()
image_arr = np.array(image_arr)

best_homography = None
background = None
should_flip = False

max_area = 0
background_index = 30
while max_area == 0:
  start = time.time()
  predictions = semseg_model.predict(np.array([image_arr[background_index]])
    , verbose=1)
  print(time.time() - start)

  ret, thresh = cv2.threshold(predictions[0], 0.5, 255, cv2.THRESH_BINARY)
  cv2.imwrite(os.path.join(testing_dir, "thresh.png"), thresh)

  im2,contours,hierarchy = cv2.findContours(thresh.astype(np.uint8), 1, 2)

  #find largest shape in image
  for cont in contours:
    if cv2.contourArea(cont) > max_area:
        cnt = cont
        max_area = cv2.contourArea(cont)

  if max_area > 0:

    epsilon = 0.02*cv2.arcLength(cnt,True)
    approx = cv2.approxPolyDP(cnt,epsilon,True)
    comb_approx = combinations(approx, 4)
    max_area = 0
    best_comb = None
    for comb in comb_approx:
      hull = cv2.convexHull(np.array(comb))
      if len(hull) == 4:
        hull_area = cv2.contourArea(hull)
        if hull_area > max_area:
          max_area = hull_area
          best_comb = comb

    resize_ratio = np.divide(video_dim, prediction_dim)
    best_comb = np.multiply(resize_ratio, best_comb)
    homography_points = [np.array([[624, 0],[0, 0],[0, 80],[624, 80]]),
                         np.array([[0, 0],[0, 80],[624, 80],[624, 0]]),
                         np.array([[0, 80],[624, 80],[624, 0],[0, 0]]),
                         np.array([[624, 80],[624, 0],[0, 0],[0, 80]])]
```

```
    min_average = None
    for normalized_points in homography_points:
      homography, mask = cv2.findHomography(np.array(best_comb),
                                            normalized_points)

      keyboard = cv2.warpPerspective(frame_arr[background_index],
                                     homography, (640, 360))
      keyboard = keyboard[0:80, 0:624]
      #analyze homographies for concentration of black pixels
      average_pixels = np.average(keyboard[0:40, 0:624])
      if min_average is None or average_pixels < min_average:
        min_average = average_pixels
        best_homography = homography
        background = keyboard

    flip = cv2.flip(background, 1) #check horizontal orientation
    if (np.average(background[0:80,0:20]) > np.average(flip[0:80,0:20])):
        background = flip
        should_flip = True

  background_index += 1

cv2.imwrite(os.path.join(testing_dir, "back.png"), background)

#PER KEY CLASSIFICATION

mid = MidiFile()
track = MidiTrack()
mid.tracks.append(track)

PRESSED = 0
UNPRESSED = 1
DEFAULT_TEMPO = 500000
pressed_notes = list()

last_time = 0
current_time = 0
for idx, img in enumerate(frame_arr):
  if idx > background_index:
    start = time.time()

    keyboard = cv2.warpPerspective(img, best_homography, (640, 360))
    keyboard = keyboard[0:80, 0:624]
    if should_flip:
      keyboard = cv2.flip(keyboard, 1)
    diff = cv2.absdiff(background, keyboard)

    white_keys_regions = list()
    black_keys_regions = list()
    for i in range(52): #white keys
```

```
    start_horz = 0
    end_horz = 0

    if i == 0:
        start_horz = 0
        end_horz = 21
    elif i == 51:
        start_horz = 602
        end_horz = 623
    else:
        start_horz = i*12 - 5
        end_horz = (i+1)*12 + 4

    key_region = diff[0:80, start_horz:(end_horz + 1)]
    key_region = np.reshape(key_region, key_region.shape + (1,))
    key_region = key_region/255
    white_keys_regions.append(key_region)

for i in range(-1, 7): #black keys (occur in 8 cycles of 5)
    if i == -1: #only one key here
        start_horz = 2
        end_horz = 23

        key_region = diff[0:60, start_horz:(end_horz + 1)]
        key_region = np.reshape(key_region, key_region.shape + (1,))
        key_region = key_region/255
        black_keys_regions.append(key_region)
    else:
        for j in range(5):
          start_horz = 0
          end_horz = 0
          if j == 0:
              start_horz = i*85 + 22
              end_horz = i*85 + 43
          elif j == 1:
              start_horz = i*85 + 39
              end_horz = i*85 + 60
          elif j == 2:
              start_horz = i*85 + 58
              end_horz = i*85 + 79
          elif j == 3:
              start_horz = i*85 + 72
              end_horz = i*85 + 93
          elif j == 4:
              start_horz = i*85 + 86
              end_horz = i*85 + 107

          key_region = diff[0:60, start_horz:(end_horz + 1)]
          key_region = np.reshape(key_region, key_region.shape + (1,))
          key_region = key_region/255
```

```
                black_keys_regions.append(key_region)

    white_predictions = white_key_model.predict_classes(
                        np.array(white_keys_regions))
    black_predictions = black_key_model.predict_classes(
                        np.array(black_keys_regions))

    white_index = 0
    black_index = 0
    black_pos = [1, 3, 6, 8, 10] #where black keys lie in the 12-note cycle
    changed = False
    for note in range(21, 109): #range of piano key MIDI values
      note_pressed = False
      if (note % 12) in black_pos: #check position in octave
        if black_predictions[black_index][0] == PRESSED:
          note_pressed = True
        black_index += 1
      else:
        if white_predictions[white_index][0] == PRESSED:
          note_pressed = True
        white_index += 1

      time_delta = int(second2tick((current_time - last_time),
                        mid.ticks_per_beat, DEFAULT_TEMPO))
      if changed:
        time_delta = 0 #so multiple notes can be pressed and unpressed at once
      if note_pressed:
        if note not in pressed_notes:
          changed = True
          pressed_notes.append(note)
          track.append(Message('note_on', note=note, time=time_delta))
      else:
        if note in pressed_notes:
          changed = True
          pressed_notes.remove(note)
          track.append(Message('note_off', note=note, time=time_delta))
      if changed:
        last_time = current_time
  current_time += frame_delta
  #print(time.time() - start)

for i, track in enumerate(mid.tracks):
    print('Track {}: {}'.format(i, track.name))
    for msg in track:
        print(msg)
mid.save(os.path.join(testing_dir, 'piano.mid'))
```

ACKNOWLEDGMENT

We would like to thank Professor Dana for her continuous support and guidance, as well as Professor Godrich and the Department of Electrical and Computer Engineering.

REFERENCES

[1] Akbari, M., Liang, J. and Cheng, H. (2019). A real-time system for online learning-based visual transcription of piano music.

[2] People.eecs.berkeley.edu. (2019). [online] Available at: https://people.eecs.berkeley.edu/ jonlong/long_shelhamer_fcn.pdf [Accessed 9 Mar. 2019].

[3] Ronneberger, O., Fischer, P. and Brox, T. (2019). U-Net: Convolutional Networks for Biomedical Image Segmentation. [online] arXiv.org. Available at: https://arxiv.org/abs/1505.04597 [Accessed 9 Mar. 2019].

[4] Akbari, M. (2019). claVision: Visual Automatic Piano Music Transcription. [online] Nime2015.lsu.edu. Available at: https://nime2015.lsu.edu/proceedings/105/0105-paper.pdf [Accessed 8 May 2019].

[5] Ieeexplore.ieee.org. (2019). Key detection for a virtual piano teacher - IEEE Conference Publication. [online] Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=6727030 [Accessed 9 Mar. 2019].

[6] R. Hartley and A. Zisserman. Multiple View Geometry in Computer Vision. Cambridge University Press, second edition, 2003.

[7] Zhao, H., Qi, X., Shen, X., Shi, J. and Jia, J. (2019). ICNet for Real-Time Semantic Segmentation on High-Resolution Images. [online] arXiv.org. Available at: https://arxiv.org/abs/1704.08545?fbclid=IwAR01PKvAbCAQ1YvjT5pnez1dEqH3RxZA0j4jmTbdMQz0Aj1mReBSUlT_NmE [Accessed 8 May 2019].

[8] Treml, M., Arjona-Medina, J., Unterthiner, T., Durgesh, R., Friedmann, F., Schuberth, P.,Mayr, A., Heusel, M., Hofmarcher, M., Widrich, M., Nessler, B., Hochreiter, S. Speeding up Semantic Segmentation for Autonomous Driving. [online] Availible at: https://openreview.net/pdf?id=S1uHiFyyg

[9] Lee, J., Doosti, B., Gu, Y., Cartledge, D., Crandall, D. and Raphael, C. (2019). Observing Pianist Accuracy and Form with Computer Vision - IEEE Conference Publication. [online] Ieeexplore.ieee.org. Available at: https://ieeexplore.ieee.org/document/8658842 [Accessed 8 May 2019].