PyPool - Pool Aim Bot

Rohin Meduri
University of Washington
Seattle, WA 98195
rmeduri@uw.edu

Anirudh Prakash
University of Washington
Seattle, WA 98195
anirudhp@uw.edu

Andriy Sheptunov
University of Washington
Seattle, WA 98195
andriy99@uw.edu

Caelen Wang
University of Washington
Seattle, WA 98195
wangc21@uw.edu

1 Introduction

The game of pool, also known as billiards or snooker, is commonly played around the world. The game consists of a rectangular table with 6 pockets, and 15 balls (plus cue ball) that players would like to send ("pot") to the pockets by hitting shots using their cue sticks. In 8-ball pool, there are 4 ball types: solids (solid-colored, numbered 1-7), stripes (partially solid-colored and partially white, numbered 9-15), the 8-ball (solid black, numbered 8), and the cue ball (solid white). There are 2 players, each with a style of ball that they would like to pot by hitting the cue-ball, while only potting the 8-ball on the last shot.

We present an application to find all possible shots in a game state that would pot balls of the desired style (solids or stripes). This system takes as input an RGB image of the pool table from the player's perspective and outputs that image with a visual overlay that indicates at which angle to hit which ball into which pocket. We imagine this being used as an Augmented Reality (AR) system, which informs the player of the best shots available. We implement this project using Python, mainly utilizing the OpenCV and PyTorch frameworks.

2 Related Work

There has been work done in the past for applying computer vision to the game of pool.

Jebara et al. (1997) [1] worked on probabilistic computer vision methods to help play billiards and give rough lines for shots on a wearable AR system. Table detection was done by RGB clustering using Expectation Maximization (EM), before using an EM algorithm to find the table contour lines. Balls and pockets were detected using symmetry transforms.

Uchiyama and Saito (2008) [2] worked on building an AR system for 9-ball pool, which is different than 8-ball pool (there is only one striped ball). To find the four corners of the table, they used RGB values and inner angle with a threshold set by experimentation to mask the table, then finding contours, using Mata's method to find line segments, and clustering into four sides. Balls were found using RGB values on the table area as well. They were classified by calculating RGB distance to predetermined ball colors and voting method.

Vachaspati [3] worked on a system to track ball paths in online videos of 9-ball pool games. Table detection was done through a Hue, Lightness, Saturation (HLS) filter–which we ended up doing as well for reasons described later–followed by convex hull and Hough transform, clustered into four groups. Projection was done with a homography.

Bauza et al. (2016) [4] worked on shot calculation for 8-ball pool, much like our project. However, they used a stationary camera angle on a top-down view, thus making the algorithm simpler. Ball detection was done using HoughCircles and identifying by color.

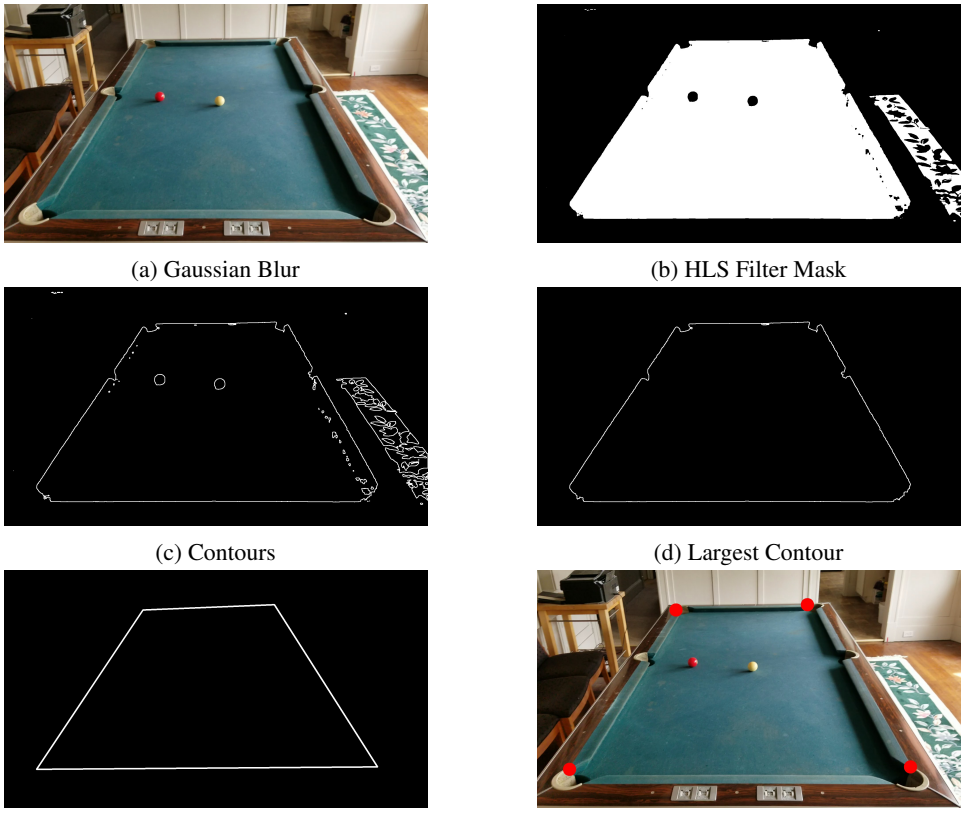
Kopeć (2018) [5][6] worked on using computer vision on mobile videos to determine the unevenness of a pool table. Table detection was done through an HLS filter, much like Vachaspati, and Canny edge detection, followed by opening, closing, and convex hull. Ball detection was done with OpenCV's SimpleBlobDetector and tracking. K-means clustering was used to find the four sides (lines) of the table, and then the four intersection corners of the rectangle.

Our project differs from the existing works because we use a neural network for ball classification, which produces better accuracy. We also use different methods for finding the table corners, dynamic camera angles, and display the visual overlay of the shot directly on the image.

3 Methodology

3.1 Finding Table Polygon and Corners

Our first step is to find the table in the image, as we only want to focus on the game state. This can be used for knowing where to find the balls and how to project the table later on.



(a) Gaussian Blur

(b) HLS Filter Mask

(c) Contours

(d) Largest Contour

(e) Approx Polygon

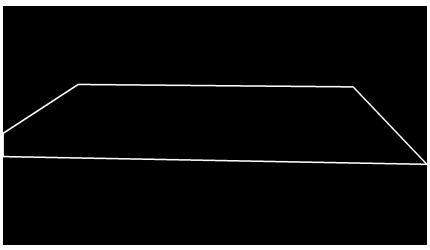
(f) Corners

Figure 1: Finding the Table

We apply a 5×5 Gaussian blur to smooth/reduce image noise, as shown in Figure 1a. We then convert the image to Hue-Lightness-Saturation (HLS) and filter by certain values (in this case, shades of green; see Experimental Results section 4.1). We use HLS over RGB because in HLS, H accounts for hue, so we simply have to set viable low and high values; L accounts for lightness so lighting does not affect the filter too much. This produces a table mask, as shown in Figure 1b. Notice that this does not always find only the table; in the case of 1b, we also find part of the green carpet, which will be taken care of later. All this is done in `find_table_corners.hls_filter` (see Appendix).

Next, we take the mask from 1b and run OpenCV's `findContours`, which joins all points along boundaries having the same hue (in this case, white), producing something like Figure 1c. We then find the largest of these contours by area (measured by `cv2.contourArea`), and this is always the contour around the table, as in Figure 1d. Notice that this is our way of getting rid of noisy contours: the carpet, the balls, and others. We now have a rough outline of the table, but we want to be able to form a nice quadrilateral around it. This is done using `cv2.approxPolyDP` to produce Figure 1e. This approximates a polygon using the Ramer-Douglas-Peucker (RDP) algorithm, which finds fewer points to approximate the curve (contour). All this is done in `find_table_corners.contour_poly`. We should note that we attempted to use other methods for this step, such as convex hull, Hough lines, and k-means, some of which was done in related works, but there were issues with noise and insufficient line detection/clustering; `cv2.approxPolyDP` was the most successful for our purposes.

The final step is to find the corners from the approximated polygon. Fortunately, `cv2.approxPolyDP` returns the points needed to make the polygon, which for our purposes, in an ideal situation, would correspond directly to the corners of the table, as are drawn in red in Figure 1f.



(a) 5-Sided Polygon

(b) 5 Corners

(c) 4 Corners

Figure 2: Only 3 Corners Visible in Image

However, `cv2.approxPolyDP` does not always find only 4 corners. In cases where one of the four corners of the table is not in the image, `cv2.approxPolyDP` will find a 5-sided polygon, with three vertices being the 3 corners in the image, and the other 2 being along (or close to) one edge of the image, as is seen in Figure 2a, which leads to 5 corners instead of four (Figure 2b). So we take the midpoint between the edge points (in `find_table_corners.get_corners`) and treat that as the 4th corner. This is okay to do because in most cases, the corner is not that far off of the image, and we find reasonable 4 corners (Figure 2c) that we can use for later steps.

3.2 Ball Detection and Classification

In the game of pool, the player needs to identify the locations of all the balls on the table, as well as their types, in order to decide the best shot. Our system performs this through localization and classification of the pool balls.

3.2.1 Localization

Taking the HLS mask and corner information from the previous step, we perform a bit-wise masking operation to produce a constrained HLS mask (Figure 3a) that only focuses on the table region. By property of HLS, the pool balls should be filtered out by the mask, leaving their positions as black circles in the HLS mask. This reduced image simplifies the task of ball detection by presenting

only relevant information. We use `cv2.HoughCircles` which applies the Hough Gradient method to detect circles within the HLS mask, effectively localizing the pool balls in the original image (outlined in light green in Figure 3b). For specific Hough Circles settings, see Experimental Results section 4.2. For the implementation of the localization process, refer to "Ball Detection - Hough Circles" in the Appendix.

(a) Constrained HLS Mask

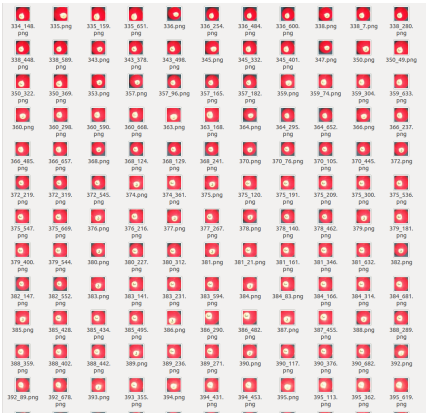
(b) Detected Balls

Figure 3: Ball Localization

3.2.2 Dataset

Using the above ball detection method, we construct a dataset of pool ball images. We record videos of each individual ball, and extract information frame-by-frame (Figure 4a). A tight bounding box is computed from the detected ball within each frame, and is saved as a 40×40 image. Due to inconsistent video lengths, some classes produce less images than others. To resolve this class-imbalance, we sample images uniformly at random and perform a random rotation to produce exactly 1000 images per class (Figure 4b). The final dataset contains 1000 images per class (Cue-Ball, 1-Ball, ..., 15-Ball) summing to a total of 16000 images, which is randomly divided into an 80/20 train-test-split. For specific online data augmentation during training, see Experimental Results section 4.3.

(a) 3-Ball Video Frame



(b) 3-Ball Dataset

Figure 4: Dataset Construction

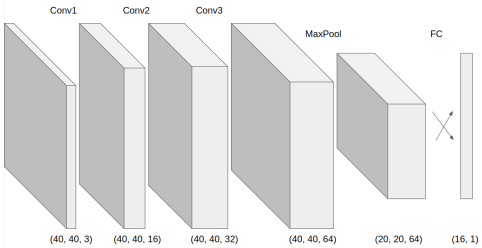
3.2.3 Classification

Using the pool ball dataset, we train a convolutional neural network for pool ball classification. We propose a simple network architecture (Figure 5) which is suitable for the complexity of the dataset. The network is fully convolutional, with a fully connected last layer for the purpose of producing 16 class probabilities. Each convolution layer is batch normalized to prevent exploding/vanishing gradients. Rectified Linear Unit (ReLU) activation is used to introduce nonlinearity. Following the 3 convolutions, max pooling is performed to shrink the feature map by a factor of 0.5 to only preserve relevant information. A dropout layer limits neuron activation by probability of 0.1, which simulates biological neuron activation and produces better generalization. The convolved feature maps are then

flattened into a linear feature space, containing 16 neurons representing the class probability output upon log-softmax activation. Finally, the class label can be extracted by taking the index of the largest class probability. For specific network hyperparameter settings, see Experimental Results section 4.4. For the implementation of the localization process, refer to "Ball Classification - Convolutional Neural Network" in the Appendix.

Layer	Input Dimensions	Output Dimensions	Normalization/ Transformation	Activation
Convolution	(40, 40, 3)	(40, 40, 16)	Batch Norm	ReLU
Convolution	(40, 40, 16)	(40, 40, 32)	Batch Norm	ReLU
Convolution	(40, 40, 32)	(40, 40, 64)	Batch Norm	ReLU
Max Pooling	(40, 40, 64)	(20, 20, 64)		
Dropout (10%)	(20, 20, 64)	(20, 20, 64)	Flatten	
Fully Connected	20 * 20 * 64	16		Log Softmax

(a) Model Architecture



(b) Model Diagram

Figure 5: Convolutional Neural Network

3.3 Projection using Homography

To facilitate shot calculations, we project the pool table to a 2D overhead view using homography. Since the dimensions of a pool table are fixed, we simply compute a projection matrix between the player-view corners and the overhead-view corners of a rectangle with the aspect ratio of a pool table. We first standardize the ordering of the player-view corners to correspond with that of the overhead-view corners. We order the corners clockwise starting from the top-left corner (tl, tr, br, bl). We define tl such that the orientation of the pool table is "upright" (shorter sides on top and bottom).

To produce this ordering, we first choose tl to be the higher of the two leftmost player-view corners, and bl to be the lower. This leaves two corners that need to be ordered and two ways to order them. We choose the ordering that minimizes the sum of the length of the tltr side and the length of the blbr side, as in Figure 7; the other way of ordering the corners would result in blbr and tltr being the diagonals of the quadrilateral, which would be longer than the side lengths, as in Figure 6.

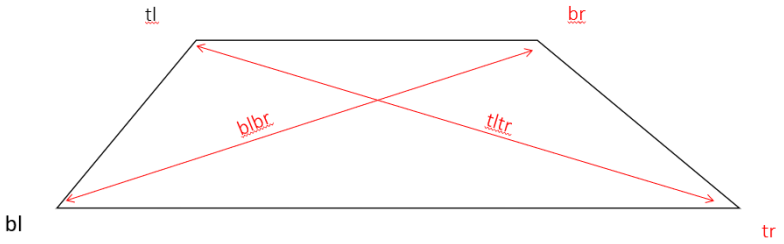


Figure 6: Incorrect selection of br and tr: results in blbr and tltr being longer

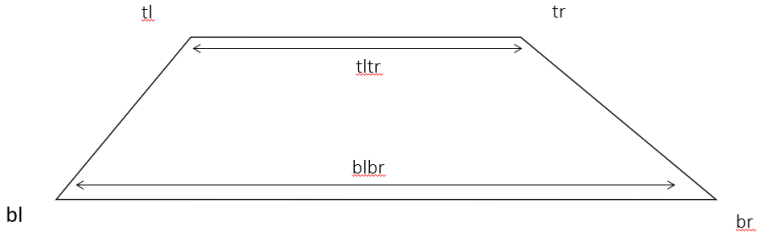


Figure 7: Correct selection of br and tr: results in blbr and tltr being shorter

This algorithm can result in a "sideways" rather than "upright" orientation of the pool table. In these cases, we check that tltr is shorter than tlbl, and rotate the corners if it is not (Figure 8).

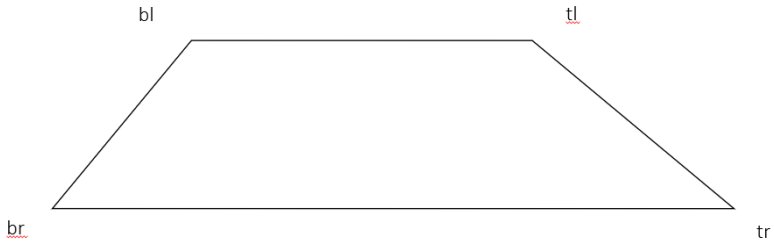


Figure 8: Rotated the corners of Figure 7 to get an "upright" orientation

Once the corners are ordered, we compute a homography between the player-view corners and the corners of an upright rectangle with the aspect ratio of a pool table (Figure 9). We then use this homography to project coordinates to/from the 2D overhead view for later use below. For the implementation of the projection process, refer to "Projection Using Homography" in the Appendix.

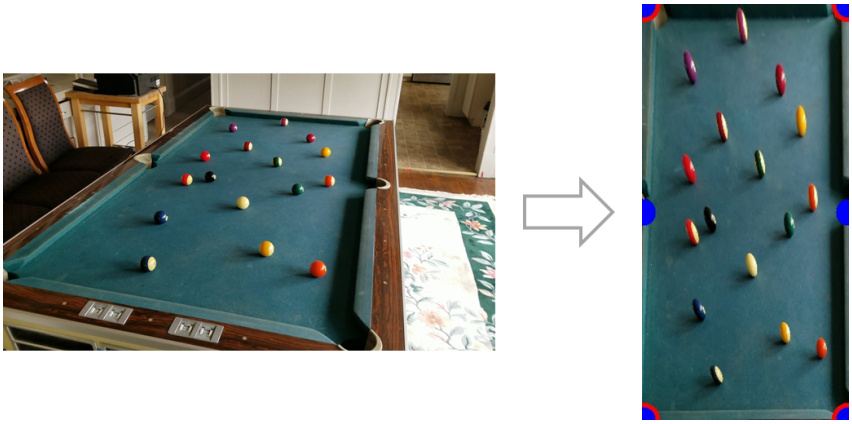


Figure 9: Visualization of the projection to a 2D plane using a homography. The image on the right shows corners in red and pockets in blue.

3.4 Shot Finding

The algorithms used to find angles at which to hit the cue largely depend on the goal of the players - potting some specific ball (e.g., the "7" ball), potting any ball of a certain class ("stripes" or "solids"), hitting the cue in a way that makes it difficult for the opponent to pot balls of their class, potting the largest number of balls in one shot, etc. Since the main focus of our project is to create an augmented view that assists a player in a practical way, we use a simple approach in detecting shots that go directly from the cue to a target ball and into a pocket. While there are more complex shots (bounce, spin, jump, etc.), our goal is to identify shots that would commonly occur in typical pool games, and are realistic for a typical novice player to hit.

In order to find shots, we iterate over each (pocket, ball) pair and check if the ball can be pocketed by a direct shot from the cue, based on projected coordinates from above (the pockets are the corners and the midpoints of the two longer sides of the table - see Figure 9). We accept only shots that have the target ball within a threshold ϵ from the line passing through the cue and the pocket. In other words, we calculate the shortest distance between the center of the ball and the line through the center of the cue and the center of the pocket using the point-to-line distance formula:

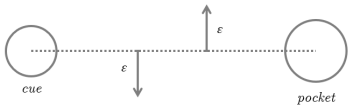
$$d(\text{ball}, \text{cue}, \text{pocket}) = \frac{|(y_{\text{cue}} - y_{\text{pocket}})x_{\text{ball}} - (x_{\text{cue}} - x_{\text{pocket}})y_{\text{ball}} + x_{\text{cue}}y_{\text{pocket}} - x_{\text{pocket}}y_{\text{cue}}|}{\sqrt{(y_{\text{cue}} - y_{\text{pocket}})^2 - (x_{\text{cue}} - x_{\text{pocket}})^2}}$$

We filter only balls such that $d \leq \epsilon$. Note that despite assuming that our balls are point-masses (and thus doing all calculations using the centers of the balls), varying ϵ allows us to simulate shots that hit the ball slightly off-center. The larger we set ϵ, the more off-center we allow the ball to be hit.

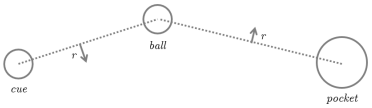
We then eliminate shots that intersect stray balls on their shot path, assuming the shot goes through the two segments $\langle \text{cue}, \text{ball} \rangle$ and $\langle \text{ball}, \text{pocket} \rangle$. We define stray balls as those whose shortest distance to either of the above segments doesn't exceed a constant r, i.e., $d(\text{stray}, \text{cue}, \text{ball}) \leq r$ or $d(\text{stray}, \text{ball}, \text{pocket}) \leq r$. r is chosen to roughly match the typical radius of a ball in the source footage, ensuring that stray balls stay outside of one ball radius of the shot path. Eliminating shots that intersect stray balls like this prevents dirty shots that may inadvertently pocket other balls, or put the board into an unpredictable state for the player. Weakening this restriction by calculating trajectories for stray balls is excessive, since we would also have to account for the changing trajectories of the cue and the target ball from hitting stray balls.

To maintain simplicity and high performance, we assume perfect momentum conservation, frictionless surface, and arbitrary travel speed; stray balls have arbitrarily complicated trajectories in this model, so it's not worth it to consider their trajectories, or the trajectories of balls they hit on their way. We simply filter out shots that hit stray balls. We find that even with these "strong" restrictions on direct shots, typical layouts of balls on the pool board result in several possible shots.

Figure 10 shows the conditions that must hold for a shot to be generated by our algorithm. For detailed implementation of our shot-finding algorithm, refer to "Shot Finding" in the Appendix.



(a) We search for target balls within a distance ϵ of the segment passing through the cue center and the pocket center.



(b) We search for stray balls within a distance r of the segments passing through 1) the cue center and the target ball center, 2) the target ball center and the pocket center. The algorithm discards shots that hit stray balls in their path.

Figure 10: Valid Shot

Once the set of all valid (ball, pocket) shots has been found, we apply the reverse of the homography (i.e. multiply by the inverse of the homography matrix) performed in section 3.3 to the ball and pocket center coordinates. This projects them back into the bounds of the input image, after which we display the shot segments to the user (Figure 11). As an alternate, less cluttered mode of display, we may filter only the "easiest" shot which has the minimal distance from ball to pocket.

Figure 11: Output of the system on simpler test cases where the player is potting solids

4 Experimental Results

Figure 12: Output on a more complex test case where the player is potting solids. Note that the blue and green balls are not detected, presumably because they are similar in color to the table.

4.1 HLS Filter Tuning

Optimal Mask Thresholds:

$$\begin{aligned} \text{Hue} &: [65, 150] \\ \text{Lightness} &: [0, 200] \\ \text{Saturation} &: [10, 255] \end{aligned}$$

We experimented to set low and high values for H, L, and S, to filter the table. Hue controls the color scale, where lower values correspond to warmer colors (red, orange, yellow), while higher values correspond to cooler colors (green, blue, purple); in this case, 65 is yellow-ish green and 150 is blue-ish green, so almost all hues of green are in the filter range. Lightness controls how dark the color is, where low corresponds to dark (black) and high corresponds to light (white); in this case, we filter all values but the lightest, which are close to white. Saturation controls how much of the color is present, where low corresponds to grayer and high corresponds to the color itself; in this case, we filter all but the grayest saturation values.

So in the end, we are able to filter all shades of green with nearly all lightness and saturation, to make sure our system is not dependent on the lighting situation and such. It is important to note that the filter values were determined by experimentation, with the main intention to make sure the entire table was filtered, including the shadows (for the purpose of finding table corners). Because of this, similar colors like the green and black balls are inadvertently masked as well, as if they are part of the table. This causes those balls to be undetected, as in Figure 12 (as they are not black circles in the table mask - from section 3.2.1).

4.2 Hough Circle Tuning

Optimal Parameters:

$$\begin{aligned} \text{minimum distance between circle centers} &: 30 \\ \text{edge gradient threshold} &: 100 \\ \text{accumulator threshold} &: 20 \\ \text{minimum circle radius} &: 10 \\ \text{maximum circle radius} &: 50 \end{aligned}$$

We make the assumption that no balls are touching and there are no occlusions. Without this assumption the minimum circle distance should be decreased. This parameter should also be adjusted

according to the camera distance from the table. Since circle detection only happens on the HLS mask, the edge gradient is not that important since there will be a clear edge between the white table region and the black circle regions. The accumulator threshold controls how many circles are detected. This parameter heavily correlates with how the HLS mask is defined. If the mask doesn't produce clear circles for every ball, accumulator threshold should be decreased to be more generous about what constitute as a circle. The minimum/maximum circle radius directly depends on how far the camera is from the table, and should be adjusted accordingly.

4.3 Data Augmentation

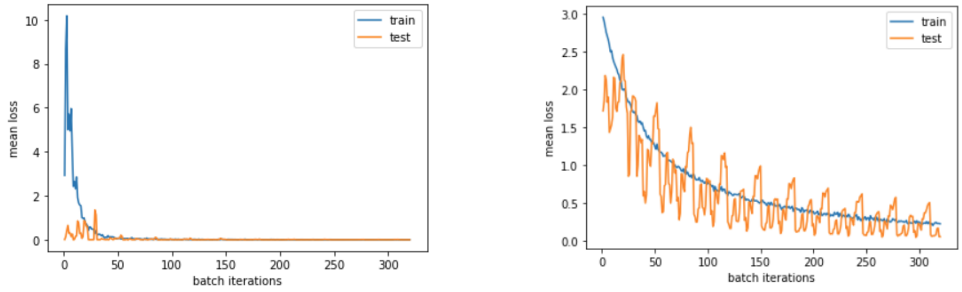
Online data augmentation is performed during training to introduce variations to the dataset, in order to produce better generalization. The transformations applied to the training data are `RandomRotation(degrees = 360)`, and `ColorJitter(brightness = 0.2)`. These transformations are ideal since they directly reflect the possible variations in unseen data. The balls on the table can be in any orientation and have a variety of brightness values due to camera angle and lighting conditions. Transformations that alter hue should not be used since color is a defining characteristic of the pool balls. Similarly, flip/reflection transformations are not ideal since they alter the numbers on the balls in a way that does not reflect real-world examples. Furthermore, channel-wise normalization is performed to produce more stable learning results.

4.4 Model Hyperparameter Tuning

Optimal Hyperparameters:

$$
\begin{aligned}
\text{epochs} &: 10 \\
\text{train batch size} &: 400 \\
\text{test batch size} &: 99 \\
\text{optimizer} &: \text{Adam} \\
\text{loss function} &: \text{cross entropy} \\
\text{learning rate} &: 1e{-}5 \\
\text{weight decay} &: 5e{-}4
\end{aligned}
$$

Train Loss: 0.199506, Test Loss: 0.207677, Accuracy: 3111/3176 (98%)



(a) Learning Rate 1e-3

(b) Learning Rate 1e-5

Figure 13: Training Plots

The train and test batch sizes are determined according to the amount of train/set data produced by the train-test-split. Model convergence occurs rather quickly if the learning rate is high. A lower learning rate produces a model with more consistent results in the final application. Therefore the learning rate is adjusted to encourage thorough exploration of the problem space, which allows optimization to find a good minimum for the categorical cross entropy loss function. Since the model is complex, weight decay (along with intrinsic model layers, i.e. pooling, dropout) is used to combat overfitting. The final model strikes a good balance between train and test loss, which produces good accuracy and generalization.

5 Conclusion and Future Work

Our application can detect and visualize pool shots for a specified side (stripes or solids) in live frame-by-frame inference. Some of the limitations of our system include failing to identify certain balls that are of similar color to the table (blue, green, black), and requiring almost the entire table to be in view to get accurate results. The biggest takeaway from our work is how parameters from one module can greatly influence the results further down the line in a multi-stage pipeline.

The most pressing future work is to refine the detection algorithm to correctly detect and classify all balls, perhaps by refining the HLS filter and model parameters. Furthermore, our system represents balls as point-masses, having all shots go through the center of the target ball. Ideally, we want the cue ball to collide at a particular spot on the perimeter of the target ball to yield the desired angle. The shot calculations must be amended to account for this slight difference in trajectory. We should also be able to find shots that involve bouncing the cue ball off the side of the pool table, and optimizing shot selection based on the difficulty of the shot/where the cue ball is likely to end up. Running frame-by-frame inference can be computationally costly and produce inconsistent/"jumpy" results, so we should implement tracking for better stability. Finally, we envision this system being integrated into an AR solution that could assist players in playing pool in real time.

References

[1] http://www.cs.columbia.edu/~jebara/papers/TR-439.pdf

[2] https://www.hindawi.com/journals/ahci/2008/357270/

[3] https://pranj.al/poolvision.pdf

[4] http://kastner.ucsd.edu/ryan/wp-content/uploads/sites/5/2014/03/admin/pool-aid.pdf

[5] https://www.youtube.com/watch?v=8AhsOAz9RSU

[6] http://lukaszkopec.com/files/pydata-pool.pdf