HoopNet - Tracking Basketball Shots With Deep Convolutional Neural Networks

Sammy Zhang
swzhang@mit.edu

Xinyu Lin
linx3@mit.edu

Bryan Chen
bchen96@mit.edu

***Abstract*— In this paper, we explore a new method to count the number of shots made in a clip of a basketball game. We present a model that first detects and draws a bounding box around the basketball hoop in a particular frame of a video clip by applying the Tiny-YOLOv3 architecture. We then feed the bounded box into a simple CNN that we built and experimented from scratch to classify whether or not a basketball shot is being made. We manually labeled our dataset, which was comprised of frames from 2019 NBA Finals Games 1 and 2, both taking place on the Toronto Raptors home court. Both hoop detection and shot classification perform with high accuracy on testing data from other Finals Games taking place on the Raptors home court. However, the combined pipeline does not generalize well to non-Finals games or to games that do not take place on the Raptors home court due to the lack of variety in our training data.**

1. Introduction

In many professional basketball games, the number of points scored by each team and each player are tracked by human scribes. However, this manual tracking system is tedious and redundant. We present HoopNet, a deep convolutional neural network that tracks the number of shots made in a video clip of a basketball game. Due to the limited time and compute resources, we will not distinguish between two pointers, three pointers, and free throws. We will also not distinguish between the team or the player that made the shot (though this is feasible using the results from Basketball Player Tracking by Acuna [3]. Additionally, due to the lack of manual labeling resources, we will only be training, validating, and testing our model on games taking place at Toronto Raptor's home court.

To accomplish this, we separate our model into two parts: 1) hoop detection and 2) shot classification. In the first part, we use tiny-YOLOv3 [2] to detect the basketball hoop in a frame. For any particular frame of the video, the model will draw a bounding box around the hoop if it is visible in that frame. After detecting the hoop, we crop out the rest of the image and send the part of the image inside the bounding box to the second part of the model to classify if a shot is made in that particular frame. In the second part, we used a CNN built from scratch.

2. Related Works

In the past years, many papers have been published regarding players and ball movement tracking in basketball games. In particular Wu L. et. al. created a hierarchical basketball detection algorithm and a motion tracking algorithm based on the adaptive object model [10]. In addition, Yoon Y. et. al. proposed techniques to track and identify players under poor camera angles and low-quality videos [11]. As a potential extension to our current paradigm, we could also experiment with these models and combine ball movement tracking to enhance our predictions. Additionally, another possible extract is to keep track of individual player scores through player tracking [11] and player motion [3].

Besides YOLO, there are many other object detection architectures such as R-CNN (Regional-based Convolutional Neural Networks) [8] and DPM [7]. R-CNN uses an object proposal algorithm called selective search to reduce the number of bounding boxes that are fed to the classifier to close to 2000 region proposals. These patches are then fed into a CNN, followed by a SVM to predict the bounding box of each patch. On the other hand, the DPM (Deformable Parts Model) recognizes objects with a mixture of graphical model of deformable objects. It contains 1) a coarse root filter that defines a detection window that approximately covers an entire object, 2) multiple part filters that cover smaller parts of the object and 3) a spatial model that scores the locations of the part filters. However, Tiny-YOLOv3 [1] outperforms these models in terms of speed and is more feasible with our limited computing resources due to its small architecture.

Hongyi Zhang et. al. studied the uses of a new data augmentation called mixup that could be used beyond ERM to improve the robustness of a model [4]. The idea behind mixup is that a linear combination of two inputs should result in a linear combination of their outputs. Honyi et. al. found that incorporating mixup into their model allowed their models to be more robust to adversarial attacks, but resulted in lower standard accuracy on the regular test set. However, using mixup, their model was able to better learn the boundary between multiple classes, resulting in a more robust model. Mixup would be a data augmentation step that feeds data into the model and generates an expected output given the input. We explain the algorithm more thoroughly below.

3. Our Model

3.1. Hoop Detection

3.1.1. Dataset

We used VLC Media Player to split a short NBA clip into .png files for every three frames. Over 7,000 frames were generated from a 15-min highlight video. Each frame is 1280x720 pixels. We then used labelImg, shown in Figure

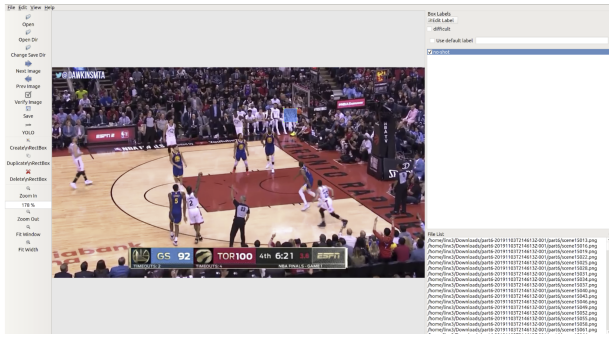

Fig. 1. labelImg Labeling Interface: The blue bounding box around the hoop is visible.

1, to manually label each frame as follows:

(class)(x-center)(y-center)(width)(height)

The classes are shot vs. no-shot. Out of over 7,000 frames, 275 labelled images were created. The ratio of shots to no-shots is approximately 1:1. This manual labelling took around 4 5 hours. Note that for the purposes of hoop detection, we can ignore the class of each label since we only need the location and size of the bounding box. The class is only relevant in part II when classifying between shot vs. no-shot. The reason that we labeled class and bounding box together is to reduce the time of manual labeling.

3.1.2. Network Architecture

We used the tiny-YOLOv3 architecture. Tiny-YOLOv3 is a simplified version of YOLOv3, which has a much smaller number of convolution layers. This means tiny-YOLOv3 does not need to occupy a large amount of memory, reducing the need for hardware. This also speeds up detection, but sacrifices some detection accuracy. We chose to use YOLO because of its fast unified architecture. The base YOLO model processes images in real-time at 45 frames per second. This is advantageous for real-time hoop detection and point tracking during a live game. Additionally, YOLO learns very general representations of objects and outperforms other detection methods including DPM [7] and R-CNN [8] in various domains such as natural images and artwork [1].

As shown in Figure 2, the tiny-YOLOv3 architecture consists of six convolutional layers each followed by a max-pooling layer, four more convolutional layers, one YOLO layer, three more convolutional layer, one YOLO layer, and two fully connected layers. Each YOLO layer contains anchors, which are initial sizes (width, height) that YOLO uses to get an approximation for the object's size and shape. YOLO then predicts the off-sets from anchors rather than predicting the bounding box from scratch. Anchor boxes are generated by clustering the dimensions of the ground truth boxes from the original dataset to find the most common shapes/size. Some hyperparameters of the architecture include IoU threshold, learning rate, momentum, weight decay, focal loss gamma, etc.

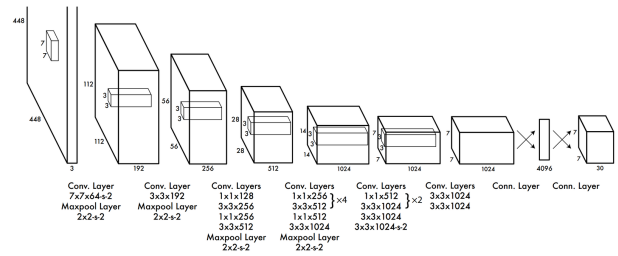

Fig. 2. tiny-YOLOv3 convolutional layers

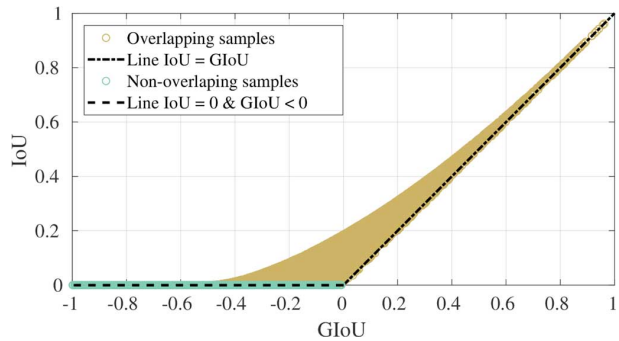

Fig. 3. tiny-YOLOv3 convolutional layers

3.1.3. Loss Function

The loss function of tiny-YOLOVv3 is the sum of coordinate loss, confidence loss, and classification loss. However, since the model is only detecting one class, the hoop, classification loss can be ignored.

Coordinate loss is defined by GIoU [9] and calculated as follows:

$$IoU = \frac{B \cap G}{B \cup G}$$

$$GIoU = IoU - \frac{C(B \cup G)}{C}$$

where C is the smallest closed convex object containing the prediction box and ground truths, B refers to the predicted bounding boxes, and G refers to the ground truths. According to these equations, when the overlap between B and G is large, both GIoU and IoU will be close to 1. If there exists no overlap between B and G, IoU will be close to 0 while GIoU will be less than 0 and approach -1 as the distance between B and G increases. Thus, the range of IoU is [0,1] while the range of GIoU is (-1,1]. Figure 3 shows the relationship between IoU and GIoU.

Recall that in a neural network, any given loss function must be differentiable to allow for backpropagation. One advantage of defining coordinate loss in terms of GIoU is that in cases where there is no intersection, IoU has no value and thus no gradient. On the other hand, GIoU is always differentiable. The coordinate loss is defined as:

$$Coord_{loss} = 1 - GIoU$$

The confidence loss is defined as follows:

$$Conf_{loss} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(C_i - \hat{C}_i)^2] + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} [(C_i - \hat{C}_i)^2]\} \quad (1)$$

To better understand this loss function, recall that YOLO divides the input image into an SxS grid. If the center of an object is within a grid cell, that grid cell will be responsible for detecting the object. Each grid cell predicts B anchor boxes and the confidence score for these boxes. The confidence scores reflect how confident the model is that the box contains an object. Formally, confidence is defined as:

$$Pr(obj) * IoU_{pred}^{truth}$$

This confidence score is also known as objectness. In YOLOv3, the objectness is predicted using logistic regression.

Now, we dig deeper into the definition of the confidence loss. $\sum_{i=0}^{S^2}$ iterates through all grid cells that YOLO divides the image into. $\sum_{j=0}^{B}$ sums the losses for each anchor box for a particular grid cell. C_i and $\hat{C}_i$ represent the true and predicted objectness respectively.

Note that we not only need to train the network to detect an object if there exists one in a cell; we also need to punish the network if it predicts an object in a cell where there isn't any. Thus, we make use of the masks 1_{ij}^{obj} and 1_{ij}^{noobj}. 1_{ij}^{obj} is 1 when there is an object in cell i, otherwise 0. 1_{ij}^{noobj} is 1 when there is no object in the cell i, otherwise 0. The hyperparameter λ_{noobj} regularizes the penalty of detecting an object while there isn't any.

3.1.4. Experimental Setup

We split our dataset into 80% training and 20% validation. Recall that the dataset consists of frames from 2019 NBA Finals Games 1 and 2, both taking place on the Raptors home court. Prior to training, the data was augmented with shear, translation, rotation, cropping, etc. to prevent overfitting as shown in 4. We experimented with various hyperparameters such as learning rate, GIoU threshold, weight decay, and etc. We used SGD momentum as our optimizer. All training was completed on AWS p2.xlarge instances, which provides up to 15 NVIDIA K80 GPU's and 64 vCPU's.

3.1.5. Results and Discussion

We experimented with various training hyperparameters including learning rate, GIoU threshold, and momentum. Due to noise in validation GIoU loss throughout the epochs, the loss for each epoch in 5, 6, and 7 was averaged over the past 10 most recent epochs.

As shown in 5, a learning rate of 10^{-4} converges to a significantly lower loss over 80 epochs than a learning rate of 10^{-2} or 10^{-3}. This is because a larger learning rate tends to jump over the minimum and might not converge to a small enough value. On the other hand, the choice of GIoU threshold does not influence the loss significantly. Recall that GIoU threshold dictates the GIoU cutoff for a positive prediction. As shown in 6, different thresholds, 0.1, 0.2, 0.3, 0.4, and 0.5 all converge to approximately the same loss over 80 epochs. This means most of the GIoU's of the predicted bounding boxes to the truth are above 0.5. Lastly, we also experimented with different choices of momentum since we used SGD momentum as our optimizer. As evident in 7, a momentum of 0.95 converges to the smallest loss over 80 epochs. However, its loss is relatively similar to that of 0.98, 0.99, and 0.995 than that of 0.90.

Fig. 4. Sample augmented training batch

We tested our hoop detection model on various test data. Recall that the model was trained using data collected from the 2019 NBA Finals Games 1 and 2, which both took place on the Toronto Raptors home court against the Golden State Warriors. However, we generated three categories of testing data: 1) 2019 NBA Finals Game 5 that also took place on the Toronto Raptors home court against the Warriors 2) a regular season game that took place on the Raptors home court against the Utah Jazz 3) a regular season game that took place on the Los Angeles Lakers home court against the Lakers. We found that our model is able to draw bounding boxes around data from test category 1 with a loss of around 1.84, which is quite accurate compared to the loss. The model is also able to draw bounding boxes from test category 2 with a loss of around 2.03. On the other hand, the model is only able to detect hoops in around 25% of the data from test category 3, which took place on the Lakers home court. This is because YOLO might use the colors and shapes of logos near the side line directly below the hoop to determine the exact location of the hoop. Since the Raptors home court and Lakers home court have very different sideline logos, YOLO

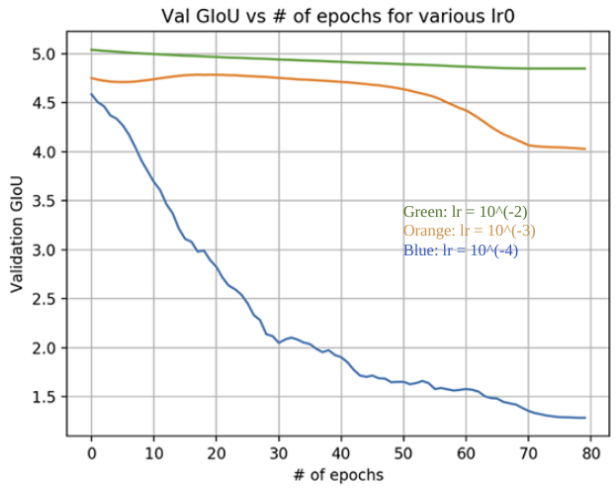

Fig. 5. Validation GIoU loss with different learning rates over training epochs

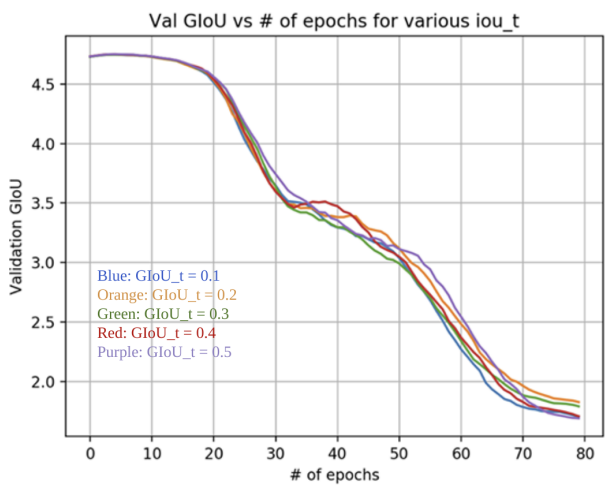

Fig. 6. Validation GIoU loss with different IoU thresholds over training epochs

has difficulty detecting the hoop. However, this problem can be fixed by adding new variety of data to the training set. Instead of using labeled frames of games taking place on the Raptors home court, we must add those taking place on other courts if we want the model to generalize well.

Additionally, we attempted transfer learning with our hoop detection model to help lower the loss. We used the pre-trained weights of classic Tiny-YOLOv3, which had been trained on over 80 classes of objects, for our first convolutional layers, and only apply gradient updates to the later fully-connected layers of the architecture. This resulted in a high loss of around 4.46. This is because the shape of the hoop is not similar to any of the 80 classes of object our transferred weights were trained on. As a result, the model performed poorly since it could not have gradient updates to the convolutional layers.

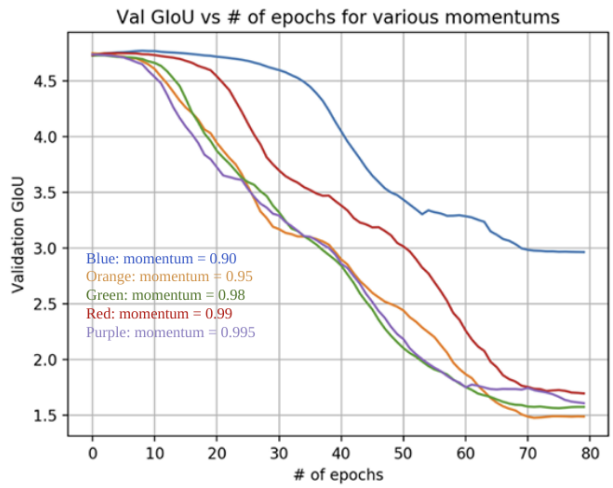

Fig. 7. Validation GIoU loss with different momenta over training epochs

3.2. Shot Classification

3.2.1. Dataset

We had two iterations of datasets for our classification model. In our original dataset, we took screenshots of a basketball game where shots were both being made and being missed. This data that we collected was generally centered around the hoop, but contained a fair amount of surroundings. The reasoning behind this was twofold. We could both make the model more robust by zooming into these pictures by small amounts, and we could account for predictions that could be partially incorrect in our first hoop-detection phase. Figure 8 shows an example of these images. The images were rescaled to 256x256 pixels and flipped horizontally to double the size of the dataset. With these data augmentations, we ended up with 402 images of no shots and 210 photos of images with shots, for a total of 612 images at a roughly 2:1 ratio of no shots to shots. This collection was entirely manual, as there do not exist any public datasets on basketball shots. The collection took over 3 hours to collect all of the photos from a video online through screenshots.

The second dataset that we used was more focused on the basket. Using the segmented data collected for YOLO, we curated new images with just the hoop in the frame. Figure 9 shows an example of these images. Since the new data included just the hoop, the images generated were now much smaller, so we resized them to 32x32 pixels and flipped them horizontally to double the dataset. We ended up with 324 images with shots and 226 images with no shots for a total of 550 photos at a roughly 7:5 ratio of shots to no shots. We used small-angle rotations in these photos to encourage robustness, and that is a trait we describe in the upcoming sections. This data collection took about 3 to 4 hours but resulted in both data that we could use for YOLO in the previous section as well as for our classifier.

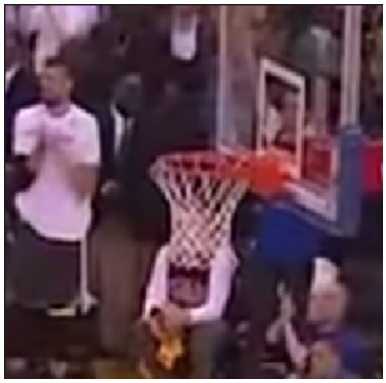

Fig. 8. Original Classification Dataset

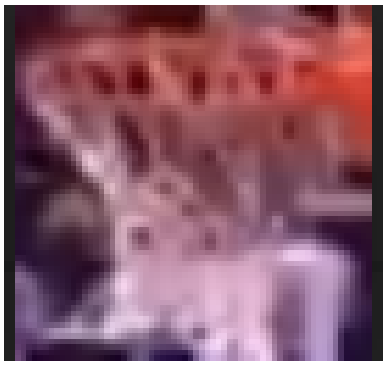

Fig. 9. Smaller Classification Dataset

We chose not to flip the images vertically or do large data augmentations because we figured the images that we would be passing the model would not undergo these transformations. Although we do want to create a more robust model, we wanted to train a model that could perform on the real-life data that we would give it, rather than data that was obviously augmented.

3.2.2. Network Architecture

We created the model itself with a goal of keeping it shallow and narrow. This would allow the model to both be trained faster and to run faster so that we would be able to classify shots in near real-time. Furthermore, we did slight hyperparameter tweaking of the convolutional layers, the number of nodes per layer, and the dropout percentages before we ended up choosing one final model for all of the datasets, keeping these hyperparameter numbers constant as well.

The network architecture is as follows. We used a simple 6-layer network with 3 convolutional layers and 3 fully connected layers. Our convolution sizes were 32, 64, and 64, respectively, with kernels (2,2), (3,3), and (2,2). We included dropouts between these layers of 0.25 and 0.2. Afterwards, we flattened the model and applied fully connected layer with 64 nodes, dropout of 0.4, then two fully connected layers of size 32 and 1. All activations were ReLU excluding

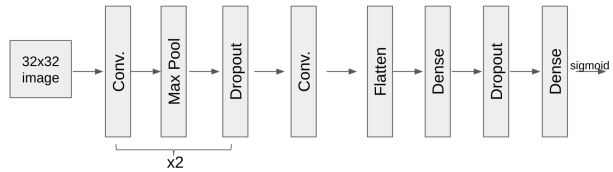

Fig. 10. CNN Architecture

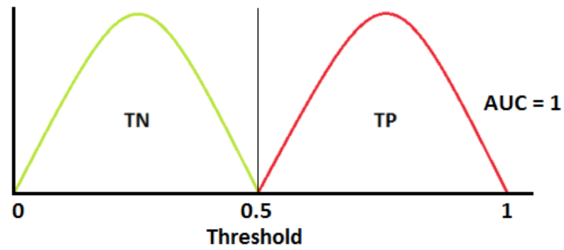

Fig. 11. Threshholds of TP and TN

the output layer, which had a sigmoid activation. We used Adam as the optimizer and binary crossentropy as the loss function. We used AUC as a measure of how well the model performed. Figure 10 shows an outline of the architecture. With this model, we were able to make predictions in less than 20ms per frame, making the model able to perform in real-time if necessary.

3.2.3. Initial Results

Original Dataset:
We used AUC as a measurement of how well the model performed on the data since it was imbalanced for this original dataset. AUC refers to the Area-Under-the-Curve measurement. The curve refers to a curve on the graph where the Y-axis is the true positive rate (TPR) and the X-axis is the false positive rate (FPR). The formulas for each are as follows:

$$TPR = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{TN + FP}$$

In a perfect model, the TPR and FPR will be completely disjoint, as shown in figure 11. This disjoint is equivalent to the graph in figure 12, where the red line shows TPR and FPR. The measurement AUC comes about because it is the area underneath this red curve. This means that an AUC of 1 is a perfect separator, and an AUC of 0.5 is equivalent to random guessing.

We found that with our CNN model, we were able to achieve an AUC of 0.774 and a loss of 0.782 after tweaking some hyperparameters for the model. We believe that the AUC was low because the images contained extra noise that wasn't relevant. For instance, if the hoop and ball took up 1/6th of the image, then the image contains a lot of unnecessary information that the model might be focusing

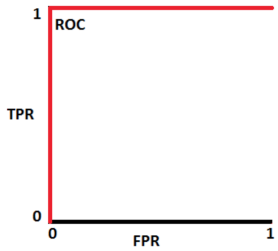

Fig. 12. Perfect AUC Graph

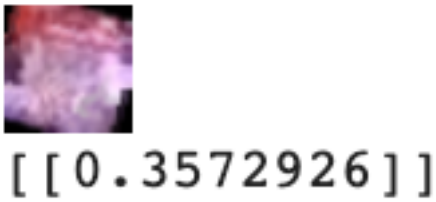

Fig. 13. Mixup Example with Image and Associated Label

on, which would cause it to perform poorly. Furthermore, since there was very little data, with just over 600 images, our model wasn't able to learn well. Since the dataset was both small and inaccurate and data collection would be time-consuming, we decided to use more specific data.

Segmented Dataset:
We continued our measures in AUC to properly compare the model performance on the two datasets, even though the data is more balanced in this new dataset. We found that our model was able to achieve a 0.914 AUC after tweaking hyperparameters. We believe that this AUC is much higher because it avoids the excess data that the original dataset included. These images contain only the information that is necessary to determine if a shot has been made, and this allows the model to perform much more strongly. We then do an initial phase of trying to improve the robustness of this model by adding small rotations to each original image. This augmentation results in a total of 825 images, with a 6:5 ratio of shots to no shots. Using this new dataset on the same model as before, we achieved an AUC of 0.934 and a loss of 0.255, which is a substantial improvement. We believe this improvement comes primarily due to the increase in data, as we increased the data by 50%. Even though we added rotations, these rotations were pretty small and would therefore cause the model to be slightly more robust than before but have more data to learn off of.

3.2.4. Improving Robustness

After performing the simple robust improvement for the new dataset, we wanted to further challenge the robustness of the model. We used mixup for this task. The motivation behind mixup states that a linear combination of inputs should result in a linear combination of outputs, and with our image dataset, an image that is 90% shot and 10% no shot should still be classified as a shot. Mathematically,

$$x^* = \lambda x_i * +(1-\lambda)x_j$$
$$y^* = \lambda y_i * +(1-\lambda)y_j$$

where $x_i, x_j \in X$ are the datapoints, $y_i, y_j \in Y$ are the labels, and $\lambda \in [0,1]$ is a value drawn from the beta distribution. This linear combination of points allows the model to learn the boundaries of the classes more smoothly because mixup creates images that traverse the transition between different classes. This traversal blurs the interpretation of images for people while creating a more robust model. Figure 13 shows an example input image, x^*, and its associated output, y^*. Notice that the image is a combination of both classes, a shot and no shot. Furthermore, the image contains an original image x that is rotated. We leave in these rotated images to increase the overall robustness of our model training.

To create this mixup, we adopted code from [5]. We altered the code to work with our binary classification model as well as our model parameters. This code allowed us to create a data generator that found images of different classes (in our case, shot versus no shot), and combine them with some λ probability drawn from a beta distribution.

Using mixup, we were able to get an AUC value of 0.948 and a loss of 0.436 on the original test set. These values have a higher AUC and higher loss than our previous model, and even with the higher loss, we expect them to perform more strongly on the actual training data due to the adversarial training. We test the results of the mixup model on the final test set further on.

3.2.5. Results

A summary of the results obtained with the classifier is in the table below. These are the top-performing values of our model given the same architecture as described above. Our AUC plots are given in figure 14.

Dataset	AUC	Loss
Original Data	0.774	0.782
Segmented Data	0.934	0.255
Segmented Data + Mixup	0.948	0.436

We decided to take the best models trained on segmented data and segmented data with mixup for our final testing.

3.2.6. Discussion

We note that for our classification models, although the tasks are simple and the images contain exactly the information we want, there are three large drawbacks. First, all of our data comes from one video source, primarily due to the time constraint of collecting data ourselves. This means that our model might not be able to perform as well on outside games

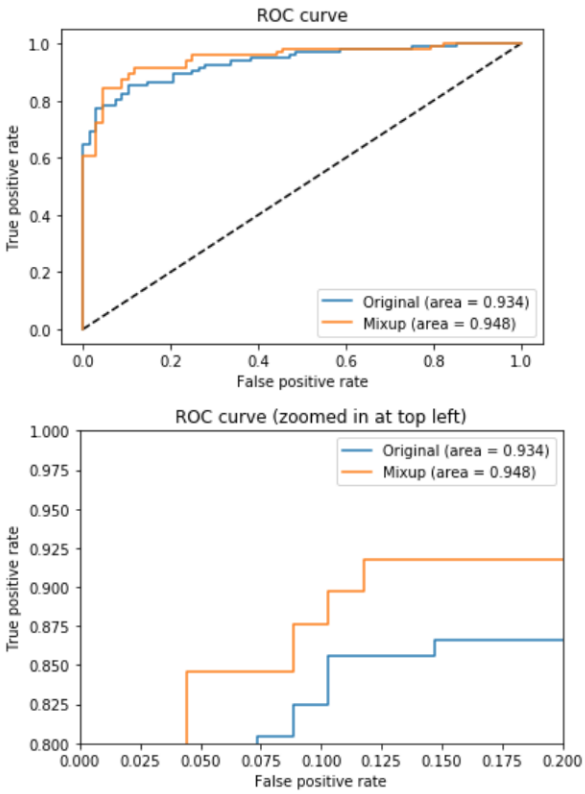

Fig. 14. AUC Plots for our best models

due to court changes, lighting changes, audience colors, or other variables outside of our control. Additionally, since we only collected images from one video clip, we have a small amount of labeled data. For classification tasks, it is standard to have thousands of labeled data pieces, as many models improve with more data. Finally, our data labels are fairly balanced, which is beneficial if we were to expect an equal number of shots and no shots per a set of frames. While this works for our classifier performance, we find that this has large impacts on the application of our classifier in real testing data.

We notice that the mixup model has more data to train with than our regular model since we are able to combine a large number of data points together to generate new data. While we expect our mixup model to perform much more poorly in classification compared to our regular model, since models that have undergone adversarial training are generally poorer-performing, our mixup model is very close in AUC scores. We believe that this is primarily due to the data increase.

4. Merging Hoop Detection And Shot Classification

4.1. Test Data and Results

We generated three categories of testing data: 1) 2019 NBA Finals Game 5 that also took place on the Toronto Raptors home court against the Warriors 15 2) a regular season game that took place on the Raptors home court against the Utah Jazz 16 3) a regular season game that took place on the Los Angeles Lakers home court against the Lakers 17.

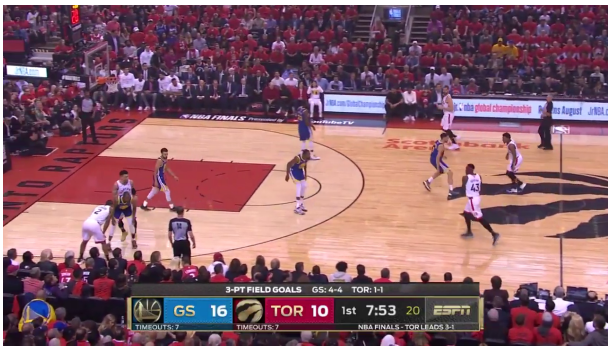

Fig. 15. 2019 NBA Finals Game 5 Raptors home court

Fig. 16. Raptors vs Jazz 2019 Regular Season Raptors home court

Fig. 17. Raptors vs Lakers 2019 Regular Season Lakers home court

The following table shows the percentage of false positive and false negative on various test datasets. The combined model uses the optimal hyperparameters determined independently for hoop detection and shot classification. A false positive is defined as a shot being predicted while no shot is actually being made. A false negative is defined as a no-shot being predicted while a shot is actually being made.

Dataset	False Positive	False Negative
Category 1	0.18	0.009
Category 2	0.654	0.04
Category 3	0.702	0.02

Fig. 18. Example of shot in testing pipeline

Fig. 19. Example of no-shot in testing pipeline

4.2. Final Models Results and Discussion

Since our code for the two models were created separately, we had to create a pipeline for passing in the test data and pipelining the model results into an interpretable output. Our Tiny-YOLOv3 model inputs a frame of the game. It outputs (x_1, y_1) and (x_2, y_2) coordinates for the top-left and bottom-right corners of the bounding box. We then take these coordinates, rescale them to the dimensions of our original input photo, and crop out the bounding box in the original photo and resizing it to 32x32 pixels, resulting in a small square that we can now pass into the classification model. We create an array of testing data using all of these images, and we predict their classes using both classification models. We use a simple threshold of 0.5 to output the classes.

Since our testing data was unlabeled, we needed a more efficient way of viewing the outputs beyond comparing original test images to the expected class output. We decided to use our original full-sized test images and color the border of these images red or green depending on the classifications, no shot or shot, respectively. This way, we were able to quickly scroll through the images and see the predictions that our model made. Examples are shown in 18 and 19

The test error is significantly lower in NBA Finals Game 5 relative to the other two categories. This is due to the lack of diversity of training data. Recall, that the training data comprised of only data from NBA Finals Games 1 and 2. As mentioned earlier in the discussion of hoop detection, YOLO performed poorly the regular season game that took place on the Los Angeles Lakers home court. This is because YOLO might use the colors and shapes of logos near the sideline directly below the hoop to determine the exact location of the hoop. As shown in 15 and 16, the Raptors home court and the Lakers home court have very different sideline logos. As a results, YOLO performed poorly in detecting hoops on non-Raptors home courts. This causes a high test error in Category 3.

As mentioned earlier in the discussion of hoop detection, YOLO performed well on data from test Category 2 (a regular season game that took place on the Raptors home court vs. Utah Jazz). However, the test error is still high because out shot classification model did not perform well. Although both games took place on the same court, the background audience of the two games are significantly different. As shown in 15, most of the background audience wore red; on the other hand, in 16, the background audience wore a mix of colors. For the perspective of the camera angle, the hoop lays above the background audience. Since our training data were from Finals Games 1 and 2, our shot classification model would be vulnerable to the difference in background noise in a finals game vs. a regular season game. As a result, there is a high test error in Category 2 despite the correct YOLO detections. This problem can also be fixed by using a more variety of training data.

Furthermore, as shown in 4.2 Test Data and Results, the false positive rate is a lot higher than our false negative rate in all three categories of data. This is because there are significantly more frames with no-shots than frames with shots. To fix this imbalance, we can adjust the proportion of images with shots to no-shots in our training data (currently as roughly 1:1.2) to reflect the proportion in the test data.

5. Future Work

We have many ideas for extensions of the project. First, we can integrate our two stages into a single program that takes a video as input and outputs an integer representing the number of shots made in video. We can also include a feedback that considers the temporal motion [13] of the ball or an aggregation method that takes considers predicted outputs over a distribution. These methodologies can increase the accuracy in shot classification since we are not longer looking at just one frame to determine whether a shot is being made or not.

Due to the limited time of the project, we also did not get to explore different model choices for both stages (e.g. R-CNN, DPM, LSTM). To better tune our hyperparameters, we can also apply hyperparameter evolution [12]. In the next iteration, we will also include more efficient and more varied data. While mixup offered promising results from training, the application performed poorly.

Lastly, we can combine with state-of-the-art player tracking and gesture recognition to track scores for individual players. A more ambitious goal would be to track other game statistics such as assists, rebounds, fouls, and etc.

References

[1] Joseph Redmon and Santosh Divvala and Ross Girshick and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv preprint arXiv:1506.02640*, 2015.

[2] Redmon, Joseph. *"https://github.com/pjreddie/darknet"*

[3] Acuna, David. "Towards Real-Time Detection and Tracking of Basketball Players using Deep Neural Networks." (2017).

[4] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[5] Zhang, Chengwei *"https://github.com/Tony607/keras_mixup_generator"*.

[6] Zhang, Chengwei *"https://www.dlology.com/blog/simple-guide-on-how-to-generate-roc-plot-for-keras-classifier/"*.

[7] Felzenszwalb, Girshick, McAllester and Ramanan, "Object Detection with Discriminatively Trained Part Based Models," PAMI 2010

[8] Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Conference on Computer Vision and Pattern Recognition, pp. 580–587. IEEE (2014)

[9] Rezatofighi, H., Tsoi, N., Gwak, J., Sadeghian, A., Reid, I., Savarese, S.: Generalized Intersection over Union. In: The IEEE Conference on Computer Vision and Pattern Recognition. IEEE (2019)

[10] Yoder J.W., Johnson R. (2002) The Adaptive Object-Model Architectural Style. In: Bosch J., Gentleman M., Hofmeister C., Kuusela J. (eds) Software Architecture. WICSA 2002. IFIP — The International Federation for Information Processing, vol 97. Springer, Boston, MA

[11] Y. Yoon et al., "Analyzing Basketball Movements and Pass Relationships Using Realtime Object Tracking Techniques Based on Deep Learning," in IEEE Access, vol. 7, pp. 56564-56576, 2019. doi: 10.1109/ACCESS.2019.2913953

[12] Fernando C., Banarse D., Blundell C., Zwols Y. "PathNet: Evolution Channels Gradient Descent in Super Neural Networks," in CoRR 1701.08734, 2017.

[13] Ramzy M., Rashed H., Sallab A., Yogamani S. "RST-MODNET: Real-time Spatio-temporal Moving Object Detection for Autonomous Driving", 2019.