CS 230 - Final Report
Wyatt Pontius, Kylan Sakata, Pablo Santos

NBA 2K19 DeepBaller: A NN-Controlled Real-Time Video Game AI

Project Motivation:

NBA 2K19 is a basketball simulation video game in which the player takes control of a virtual basketball player on the court. With eSports - and the 2K League in particular - becoming increasingly relevant, we were surprised to see that no attempts to create an automated bot to play 2K existed. Given we have no source code from the 2K environment, the entire process must take place only using the visual output from the game, which is exactly how a human would play the game.

At first, 2K is a daunting game, with a plethora of visual outputs and a high number of controller inputs required to play the game. We sought to simplify the problem through a number of methods: first, setting up a constant game mode of 3 on 3 basketball with a static background; second, training an object detection network to identify features on the screen before feeding the visual output from the Xbox to the network which plays the game; and third, narrowing down the controls necessary to play the game to the core controls in order to reduce the complexity of potential neural network outputs.

Another challenge with playing a video game such as 2K is that anything must be done in real-time for the game to be played successfully. This required us to carefully consider all system design decisions such that we could obtain the visual output from the Xbox, process it and perform object detection, feed this information as inputs to the neutral network which plays the game, perform the network operations to obtain outputs, then send this controller input back into the Xbox without any noticeable delay, repeatedly for the duration of a game.

Problem Statement:

The purpose of this project was to create a system - based on neural networks - that could allow a computer to play NBA 2K19 on Xbox without any source code or pre-coded understanding of the game. The system was to be designed in such a way that the computer could play the game in real-time, taking visual outputs from the game and making decisions as a human would without any noticeable lag. For this to be feasible, the system was composed of two neural networks working in tandem: one which takes the visual output and condenses it into a manageable amount of information and another which takes this information then decides which action to perform on the Xbox.

CS 230 - Final Report
Wyatt Pontius, Kylan Sakata, Pablo Santos

Data Description:

Unfortunately, there was no existing dataset - cleaned or otherwise - which could be used for this project. As a result, we collected all of the data used in the project.

The first phase of data collection consisted of collecting screen captures of frames from gameplay which featured some or all of the elements important to playing the game - the player, teammates, opponents, ball, hoop, and meter - and hand-labeling them using a Yolo Annotation tool (https://github.com/ManivannanMurugavel/Yolo-Annotation-Tool-New-). Our dataset for the object recognition portion of the project consisted of approximately 2500 images and accompanying label text files with coordinates of the elements of interest within each image. The coordinates were stored in a format that could be used by Darknet to train the YOLOv3-tiny network for object recognition.

The second phase of data collection consisted of simultaneously recording frames - with the data stored as object coordinates on screen, so the size of the data would be more manageable than a massive array of pixels - as well as controller inputs. In practice, 10 times per second, the coordinates of all elements of interest on screen were recorded along with the buttons being pressed on the Xbox controller. As we were using a time-sequence network, we ended up storing one second's worth of coordinates for each set of controller inputs, resulting in a (10 x 45) array of coordinates along with a (1 x 9) array of movement controls and a (1 x 5) for each 0.1 seconds of gameplay. The 9 movement choices correspond to 8 cardinal/ordinal directions and no movement, while the 5 action choices correspond to pass, shoot, jump, call for screen, and no action.

Network Architecture:

In searching for an object recognition network, we decided to go with a preexisting network instead of designing our own due to the efficiency of existing options. However, only a subset of existing solutions could work for our purposes, as we needed something capable of processing a high number of frames per second, limiting our options. In the end, we settled on the YOLOv3-tiny network (https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg) due to its combination of high speed and accuracy. As described in the data section, we hand-labeled our own corpus of image data and then trained the network via Darknet, built in C on a Windows desktop. We tweaked the learning rate to minimize the amount of time required to achieve a suitable amount of loss, however, we did not change any other aspects of the network due to its efficiency in its default state.

We built the decision-making network ourselves, applying somewhat standard architecture choices for a time-dependent analysis such as this. The input to the network is a (10 x 45) array representing the coordinates of elements of interest on screen for 10 frames over the course of a second. This input is passed through two GRU layers each consisting of 256 nodes. After these layers, the network is split into two branches so that the movements and actions could be predicted separately. Each of these branches consisted of a GRU layer consisting of 256 nodes, followed by a fully connected layer of 256 nodes with softmax activation, and finally a regression layer with categorical cross entropy loss, using the Adam optimizer. We then merge the outputs of these branches back together. We then used np.argmax to determine the optimal movement and action choice, before relaying these choices back to the Xbox as virtual controller input. Since the movements and actions are somewhat dependent on each other, we didn't want to separate networks, but they need to be executed separately: hence the two branches.

Presentation & Analysis of Results:

Having painstakingly labeled a set of around 2500 images, we were rewarded with strong results from our object recognition network.

In a dataset of 100,000 frames, there were only 30 images with double detections, indicating that our network is highly discerning and not prone to identifying elements that are not in fact on the screen. On the other hand, qualitatively, the network does quite well in identifying the important elements on screen, succeeding in nearly all frames. The exception

to this is frames in which one element is almost entirely obfuscated by another, in which case the network only recognized the more visible element. This occurs semi-regularly - whether the ball disappears due to being held close to a player's body, or an opponent heavily covering a teammate on defense - and thus it is very difficult to automatically calculate any meaningful quantitative statistics, such as the percentage of the frames all elements are being correctly recognized.

Having successfully collected the coordinates of important elements on screen, we then fed these as inputs into the decision-making network described previously. The performance of this network is difficult to quantitatively assess, so a qualitative analysis was performed which consisted of i) sanity checking the results to ensure the network was not simply predicting "do nothing" all of the time and ii) watching gameplay of DeepBaller. The decision-making network successfully predicts that nothing should be done when there appear no important elements on screen (such as when there is a menu screen), though the results are more sporadic when the game is occuring (i.e when there are important elements detected on screen). Having watched an entire game's worth of gameplay, it is clear that the network has learned substantially how to play in various situations. DeepBaller has the teammate inbound the ball to the player, has the player dribble the ball up the court, and is capable of calling for screens and passing depending on the positions of the elements on screen. When conditions are correct, DeepBaller is even capable of scoring, which requires not only initiating but also timing a shot (holding a button down) and is difficult even for human players. On defense, however, DeepBaller struggles massively. Instead of either switching to a defender who is not guarding the opponent with the ball (known as "off-balling") or staying in front of the opponent with the ball, the network instead moves the player out of bounds, which is useless. Interestingly, the network does recognize when shots from the opponent are imminent and attempts to block them by pressing the block button, but this is largely ineffective due to the poor positioning of the player.

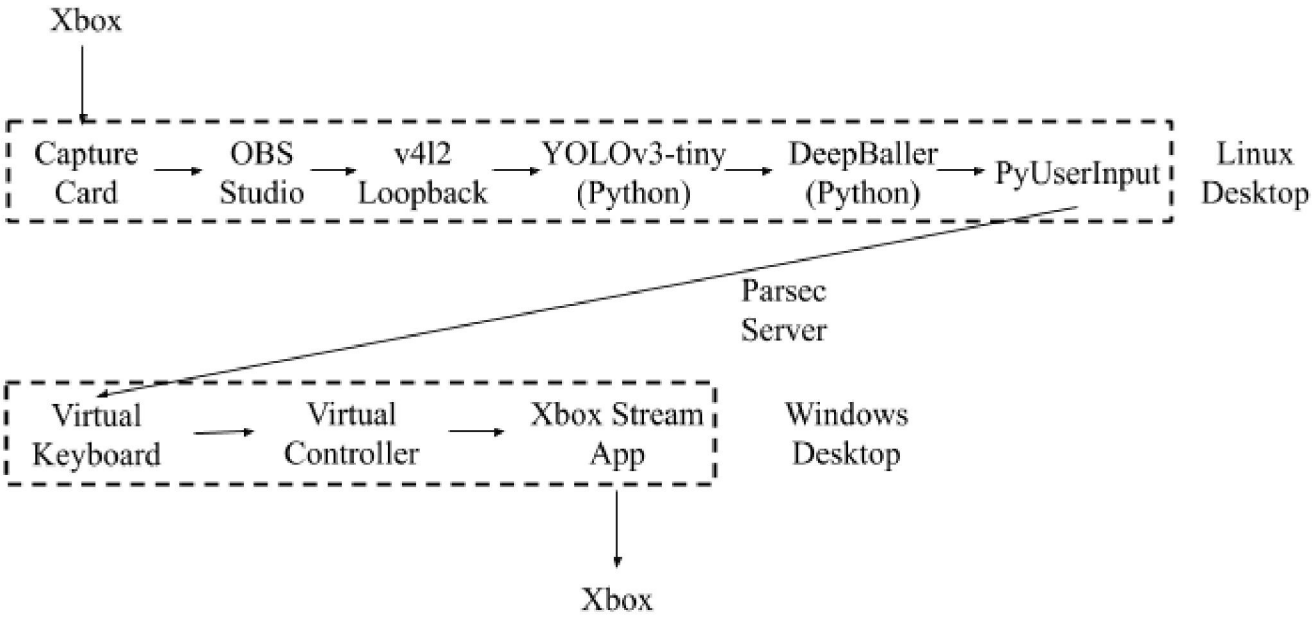
With under two hours of training data collected from different contributors, it is remarkable that such a level of performance has been achieved. With more training data - likely a factor rather than an order of magnitude, even - it is clear that the network would be able to learn more situations and behave more appropriately than it does currently. Furthermore, by only having training data from one human contributor - who is likely to have a play style which the network can learn more easily than cobbling together play styles from different human contributors - it is possible that the performance of the network could be improved with less data.

Given the difficulty of system engineering associated with this project, the amount of time we were able to dedicate to hyperparameter tuning was small. With some network tuning, such as adjusting the length of the memory for the

GRU layers or tweaking the network architecture slightly, it is also possible that we could achieve performance improvements over our current implementation.

Project Discussion and Insights:

Though this project did involve quite a bit of deep learning, there was a greater amount of engineering work involved to create the system as a whole, of which the neural networks were only a part. To provide further depth to this claim, a diagram showing the flow of information in the project is shown below:



We have created a system composed of three separate pieces of hardware - an Xbox, a Linux desktop, and a Windows desktop - that operates in a closed loop, in real-time, without user intervention. The process of designing such a system - and actually getting everything to communicate properly, with essentially zero lag - was incredibly arduous. The fact that the system is able to comprehend what is happening in the game and make a sensible decision at such a high level, even capable of scoring, is remarkable.

Code:

Github: https://github.com/wyattpontius/cs230

CS 230 - Final Report
Wyatt Pontius, Kylan Sakata, Pablo Santos

Contributions:

The contributions for this project were incredibly balanced, with all team members actively contributing portions to each major component of the project. Below we've described areas where each of us contributed a little bit more:

Wyatt Pontius:

- Labeled frames for training data
- Led design of system
- Built Darknet on Windows
- Designed decision-making neural network
- Allowed use of Linux PC and 1080Ti GPU

Kylan Sakata:

- Labeled frames for training data
- Allowed use of Xbox One
- General Python development/consultation/debugging
- Research/review of GitHub repositories to leverage in our system

Pablo Santos:

- Labeled frames for training data
- Built Darknet on Windows
- Contributed to coding python scripts to implement player agent training and testing
- Gathered hours of training data
- Allowed use of personal Windows PC and 1070Ti GPU

References:

1) https://github.com/ChintanTrivedi/DeepGamingAI_FIFA
2) https://github.com/Sentdex/pygta5