Object Detection for the HoloLens Project

# Background

## PROJECT FLOW

The original goal of this project was to create a game that can be run on the computer & the Microsoft HoloLens to measure a difference in playtime and to see a runtime improvement. The game we chose was a statistical overlay for blackjack which would detect playing cards using a variant of the YOLO (You Only Look Once) machine learning object detection algorithm and then display statistics for the next move. A GitHub repository [1] containing a Jupyter Notebook for training the playing cards was used. The repository was responsible for obtaining & formatting the data needed to initialize training on the darknet network [2]. More on Training in the next section.

At the start, we decided on YOLOv2 because of its compatibility with Unity 2019 (Unity version required for the HoloLens 1) and the need for an ONNX file. We found an existing repository [3] which handles YOLOv2-tiny ONNX files and running interference[[1]](#footnote-0) on them. In order for the YOLOv2 repository to work for our needs, we need to have it detect playing cards. This is done by loading a trained ONNX file into the project. However, we could not convert the darknet WEIGHTS file into an ONNX file. To convert from a YOLOv2 WEIGHTS file to ONNX, we used this repository [4] but due to software versioning issues (the tool was made in 2017), this did not produce a result. With research, no other tools seemed to exist for the YOLOv2 conversion. We then decided to try YOLOv4. YOLOv4 was trained with the built-in darknet network successfully. The resulting WEIGHTS file was converted to ONNX via another tool [6]. This conversion was successful and a valid ONNX file was converted.

Once we had a valid (or so we thought) ONNX file, we then used this file in this repository [5] to load the ONNX file into Unity. Unity recognized the file, but gave errors saying the array sizes do not match. We thought this could be because the ONNX file was not created correctly, this assumption turned out to be true, and we had to stop the project here. Using an ONNX analyzer tool, we found out that the output vector was not the same as the output vector in the accompanying WEIGHTS file.

## ONNX

ONNX stands for Open Neural Network Exchange, which is a neural network model in a convenient all-in-one file to interface between different programs (Unity in our case). There are multiple ways to convert the WEIGHTS file into an ONNX file for use with Unity. You can use PyTorch[[2]](#footnote-1) and convert the model to a PyTorch model, then directly to an ONNX file. Another option is to use different framework, like OpenCL

# Training

The Jupyter notebook used for training, called creating-playing-cards-dataset, was extremely helpful in creating the dataset needed. First, a video of each card was created with varying levels of exposure. This was to give more variance to the dataset and increase accuracy. Once the videos were taken, they were loaded into the notebook which cuts out the card from its background, draws a bounding box around the card number, and then superimposes them with other cards onto a generated background to create a “scene”. These scenes were then converted from the Pascal VOC format to xml & text files for use as inputs to the darknet network for training. 

One issue we had was getting the cutout algorithm to recognize the cards, the solution was to increase the lighting by a factor of 2x to get the desired result. Another issue we had was creating bounding boxes for the face cards (King, Queen, Jack), we believe the algorithm did not work well with the box surrounding the card itself (shown here in Figure with the red arrow). Solution for that was to continually take more videos until the bounding box algorithm found the boxes (this took a while, but we eventually got some results).

Once we had the dataset that darknet expects, we moved onto training on the network from the generated scenes. At first, we used YOLOv2 but due to some issues, we ended up training full on YOLOv4. For YOLOv4, you need the following information to train:

* A custom configuration (.cfg) as a base to start with, modified according to AlexeyAB [2].
* An obj.names file containing the location of the scene files with <object-class> <x\_center> <y\_center> <width> <height> outlining the locations of the bounding boxes.
* An obj.data file containing the location of the names file & how many classes.

Training is started using ./darknet detector train data/obj.data yolo-obj.cfg yolov4-tiny.conv.137, which will train on your GPU. After 100 iterations, you want to change the command to use your created WEIGHTS file instead of yolo4-tiny.conv.137 so that the darknet training program can calculate the mAP & average loss when training.

# Testing

Currently, the detection works via a webcam & the built-in darknet detector demo [2]. This demo runs interference on the input webcam frames, and then draws bounding boxes on the output video feed frames. Prerequisites for the demo:

* Darknet [2] must be compiled.
* A trained weights file must be generated.
* A webcam is preferred but the detection does work on static images.

To run the demo:

1. Plug in the webcam into the host computer.
2. Position the webcam vertically above the cards to be detected and make sure the scene has plenty of light.
3. Navigate to the darknet folder containing the build files.
4. Run the following command from windows PowerShell:

**darknet** detector demo .\data\obj.data .\cfg\yolov4-tiny-obj3.cfg .\backup\yolov4-tiny-obj3\_20000.weights -c 0

**.\data\obj.data** → location of the custom object data file.

**.\cfg\yolov4-tiny-obj3.cfg** → location of the custom object configuration file.

**.\backup\yolov4-tiny-obj3\_20000.weights** → location of the trained WEIGHTS file.

**-c 0** → Use system webcam with ID=0

1. Observe the output in the video feed window with the bounding boxes drawn on. The identified cards will also appear in the command line window.

# Future for the Project

This project was interesting and had a lot of challenges. I’d say in the future, we need to get the integration between the HoloLens & the game logic program in Unity for the project to be fully successful. Ideally, a set of group members to possibly take on this project should:

* Successfully output an ONNX file for use in Unity.
* In Unity, recognize playing cards from the ONNX model.
* Integrate the object recognition code [5] and our blackjack statistics project
* Run the game on the hololens.
* Perform testing to see if the project saves battery life.

# References

| [1] | geaxgx and dsousadev, "GitHub," 2019. [Online]. Available: https://github.com/geaxgx/playing-card-detection. |
| --- | --- |
| [2] | AlexeyAB and pjreddie, "GitHub," 2021. [Online]. Available: https://github.com/AlexeyAB/darknet. |
| [3] | wojciechp6, "GitHub," 2020. [Online]. Available: https://github.com/wojciechp6/YOLO-UnityBarracuda. |
| [4] | wiibrew, 2017. [Online]. Available: https://github.com/wiibrew/pytorch-yolo2. |
| [5] | keijiro, "GitHub," 2021. [Online]. Available: https://github.com/keijiro/YoloV4TinyBarracuda. |
| [6] | Tianxiaomo, "GitHub," 2020. [Online]. Available: https://github.com/Tianxiaomo/pytorch-YOLOv4. |

1. Input an image into the neural network and receive an output. [↑](#footnote-ref-0)
2. An open-source machine learning framework that interfaces between a neural network and your system’s GPU (Graphics Processing Unit). [↑](#footnote-ref-1)