2020

Computer Vision Gesture Recognition for Rock Paper Scissors

Nicholas Hunter
The College of Wooster, nhunter20@wooster.edu

Computer Vision Gesture Recognition for Rock Paper Scissors

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in Computer Science in the
Department of Mathematical & Computational Sciences at
The College of Wooster

by
Nicholas Hunter
The College of Wooster
2020

Advised by:

Dr. Mircea Ionescu

THE COLLEGE OF
WOOSTER

Abstract

This project implements a human versus computer game of rock-paper-scissors using machine learning and computer vision. Player's hand gestures are detected using single images with the YOLOv3 object detection system. This provides a generalized detection method which can recognize player moves without the need for a special background or lighting setup. Additionally, past moves are examined in context to predict the most probable next move of the system's opponent. In this way, the system achieves higher win rates against human opponents than by using a purely random strategy.

Acknowledgments

First, I would like to thank my advisor Dr. Ionescu for his invaluable help with this project. I would also like to thank all of the Computer Science professors who have taught me so much during my time at Wooster. To all my friends, thank you for your help and moral support over the years. To my mother, father, and brothers I love you all greatly. I know this is just the start of something bigger, but words cannot express my gratitude for all the ways you've helped me get to this point.

Contents

Abstract v

Acknowledgments vii

Contents ix

List of Figures xi

List of Tables xiii

CHAPTER PAGE

1 Introduction 1
- 1.1 Rock Paper Scissors . . . 1
- 1.2 Project Purpose . . . 4

2 Machine Learning Background 6
- 2.1 Machine Learning . . . 6
- 2.2 Supervised & Unsupervised Learning . . . 7
- 2.3 Image Classification & Object Detection . . . 8
- 2.4 Convolutional Neural Networks . . . 9
- 2.5 YOLO (You Only Look Once) . . . 12
  - 2.5.1 Introduction . . . 12
  - 2.5.2 YOLO Algorithm . . . 13
  - 2.5.3 Non-Maximum Suppression . . . 15
  - 2.5.4 Transfer Learning . . . 17
  - 2.5.5 Performance Metrics . . . 18

3 Related Work 22
- 3.1 rps-cv Project . . . 22
- 3.2 Gesture And Activity Classification . . . 24
  - 3.2.1 Two Stream CNNs . . . 25
  - 3.2.2 Skeletal Tracking . . . 26
  - 3.2.3 YOLO for Gesture Recognition . . . 27

4 Methods 29
- 4.1 Algorithm Selection . . . 29
- 4.2 Data Collection . . . 31
- 4.3 Data Annotation . . . 33
- 4.4 Training . . . 36

5 Gameplay Prediction 38
5.1 Background . . . 38
5.2 Goals . . . 40
5.3 Recurrent Neural Networks (RNNs) . . . 40
5.4 Methods . . . 42
5.5 Prediction Results . . . 45
6 Software 48
6.1 Inference . . . 48
6.2 User Interface . . . 49
6.3 Application Architecture . . . 51
6.4 Software Installation and Delivery . . . 52
7 Results 55
7.1 Data Augmentation Parameters . . . 55
7.2 Training Length . . . 58
7.3 Application Testing . . . 59
8 Conclusion 60
References 62

List of Figures

Figure		Page
1.1	Rock Paper Scissors Hand Gestures	2
2.1	From Stanford CS231n Winter 2016 lecture 8 slides [17]	8
2.2	Filter Convolution Example [22]	9
2.3	Convolution Visualization with Stride of 2 [18]	10
2.4	Convolution Layer Example [18]	12
2.5	YOLOv3 Network Architecture [24]	12
2.6	Intersection Over Union [26]	14
2.7	Intersection Over Union [26]	15
2.8	The YOLO system [25]	15
2.9	Example precision-recall curve for a fixed IOU threshold and varying confidence threshold [2]	20
3.1	rps-cv hardware [10]	23
3.2	Two-Stream CNN Architecture [29]	25
3.3	DeepPose Examples [32]	26
3.4	YOLO Activity Classification [28]	27
4.1	Sample Rock Gestures	31
4.2	Sample Paper Gestures	32
4.3	Sample Scissors Gestures	33
4.4	Example frame in Labelbox	35
5.1	Unrolled RNN [20]	42
5.2	Prediction Model Architecture: Method 4 with 50 LSTM Units	44
5.3	LSTM Prediction Model Accuracy	45
6.1	Application User Interface	49
6.2	Image Pipeline	54
7.1	mAP vs Saturation	56
7.2	mAP vs Exposure	57
7.3	mAP vs Hue	57
7.4	mAP vs Number of Training Iterations	59

List of Tables

Table Page

5.1 Model Training Length Results 46

7.1 Hue Saturation Exposure Model Tuning Results 56
7.2 Model Training Length Results 58

CHAPTER 1

Introduction

1.1 Rock Paper Scissors

Rock Paper Scissors is a simple two person hand game. The game can be played with one or multiple rounds. Both players begin by saying "Rock, Paper, Scissors" while moving their fist in an up and down motion. The players then simultaneously play, or throw, one of the three hand gestures while saying "Shoot!". Each hand gesture (rock, paper, or scissors) beats one other gesture. If both players play the same one, the result is a tie game. Rock beats scissors, scissors beat paper, and paper beats rock. The idea behind these rules is that rocks make scissors blunt, that scissors cut paper, and that paper covers rock. Games like Rock, Paper, Scissors are typically described as finger-throwing or finger-flashing games [27] [7]. Players 'throw' their chosen gesture simultaneously during gameplay.

While the hand gestures can be represented in slightly different ways, rock is represented by a fist. Paper is represented by a flat hand, and scissors is represented by a horizontal peace sign.

Rock Paper Scissors is a fundamentally simple game, which can be played anywhere using only hands. This sets it apart from other decision making games like coin flipping or drawing straws, both of which require specific objects to play.

Like all decision making games, it is important to clearly define fixed rules before starting play. With Rock Paper Scissors, one of the most important aspects is deciding how many rounds need to be won to win the entire game.

Figure 1.1: Rock Paper Scissors Hand Gestures

Rock Paper Scissors is actually one of the oldest decision making games we know of in human existence [7]. The game is also known as Jan-Ken-Pon Rochambeau, which presumably led to some people calling the game Roshambo. Evidence of a similar hand gesture game has been found dating back to around 2000 B.C.E. in Egypt [7]. In this game, two players would simultaneously show differing numbers of fingers. By 200 B.C.E, the concept of three distinct hand gestures emerged in eastern Asia. Players used various gestures representing competing objects or entities. Some examples are:

- Snake, frog, and slug

- Elephant, human, and ear-wig
- Chief, gun, and fox
- Tiger, village chief, and the village chief's mother

With the elephant, human, and ear-wig, the elephant crushes the person, the human crushes the ear-wig, but the ear-wig crawls up the elephants trunk and either eats its brain [27]. With the tiger, village chief, and village chief's mother, the chief is beat by his mother.

Rock Paper Scissors is often seen as a trivial children's game, but the game has developed further cultural significance within the last three decades. In 1995 two brothers Doug and Graham Walker wanted to create a website, and being fans of Rock Paper Scissors, created worldrps.com as a bit of a joke. They called their project the World Rock Paper Scissors Society. On the website, they developed overly complicated rules and regulations for the game, and wrote an entirely fictitious history for Rock Paper Scissors [27]. What began as a joke between two brothers became an internet phenomenon over the next 10 years and turned into "a unique viral experiment" according to Doug Walker.

Some see the World Rock Paper Scissors Society as a parody poking fun at other organized sports, but some see it as a testament to nostalgia and a game that transcends geographic and cultural barriers. The group began to hold championships complete with referees in traditional black and white striped uniforms. Microsoft and Yahoo were at one point corporate sponsors, and the 2007 World Rock Paper Scissors Society championship was televised on ESPN [27]. The winner even got $10,000.

The expanding cultural phenomenon around Rock Paper Scissors has developed beyond an internet parody and has led to several academic research papers in the areas of psychology, economics, and more specifically game theory. Rock Paper

Scissors has also been used in Biology as a way of examining competing strategies. While many people assume Rock Paper Scissors is a fair and random game, this belief rests upon an assumption that humans can behave randomly. As we will examine later in this paper, humans are a little more predictable that we would like to think.

1.2 Project Purpose

The main purpose of this project was to create an application utilizing computer vision to play Rock Paper Scissors against. A secondary goal was to explore concepts of computer vision, machine learning, and human-computer interaction (HCI). Machine learning is used here both to extract gesture information from images, and also to improve the computer's win rate by predicting patterns in opponent play.

It should be noted that other projects have taken on these same goals, but have taken different technological approaches. These differences are most significant with regards to computer vision algorithm choice. Some of these projects will be discussed later in this paper. For now however, a further goal was to have this application run without the need for a specialized environment. This entails using a normally positioned webcam under normal lighting conditions.

The project is fundamentally comprised of multiple different problems within two main areas: gesture detection and opponent prediction. The application must first detect whether or not a gesture is present in an image. Next, it must determine which gesture is present. In order to predict the opponent's next play, a strategy examining past plays must be employed. Choosing randomly is obviously a strategy, but to achieve better win rates some form of opponent prediction must be used.

While the software we developed could be considered impractical or frivolous by

some, we think it serves a purpose as a relatable and interactive way to demonstrate concepts of machine learning to those without any formal knowledge of the subject.

CHAPTER 2

Machine Learning Background

2.1 Machine Learning

Computer programming is often first presented as creating sets of instructions for a computer to follow [30]. The idea that computers follow strict sets of logical rules is a powerful one, but as technology progresses, applications are making more decisions based on data as opposed to hardcoded rules.

Data can be represented in infinitely many ways, but some are easier for computers to fully interpret. Structured data is formatted and organized in a way that makes it easy to process, search, and analyze. For example, data that could be entered into columns and rows of a spreadsheet or database would be classified as structured data. Unstructured data covers all other data that has no defined data model [21]; for example, text, pictures, and videos. While they can be represented in a structured manner for our devices to display and transmit, the distinction is about the difference between the information a human would interpret while reading text or looking at an image, and what a computer would interpret while displaying them.

Throughout the history of computing, there has been constant effort applied to making computers progressively more like humans. An early example of this

is Alan Turing's paper from 1950 titled "Computing Machinery and Intelligence" where Turing presented the idea of "The Imitation Game". The game involves three participants. One participant is a human judge who electronically chats with the other two participants via a computer terminal. Of the other two participants, one is a human and one is a computer pretending to be a human [4]. A computer system wins the game if the human judge cannot consistently tell the other human and computer apart.

Before diving into Machine Learning, it's important to define what it means for a computer to learn. Tom Mitchell's 1997 book Machine Learning includes a straightforward and frequently used definition:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. - Mitchell [19]

2.2 Supervised & Unsupervised Learning

Supervised learning is a subset of machine learning, and relies on a labeled dataset. Connecting to Tom Mitchell's definition, a supervised learning algorithm improves its performance on some task using a labeled dataset as experience. Labeled data includes both input data as well as expected outputs. Supervised learning can be used to both classify inputs into categories or to predict values. For supervised learning applications, the task is to map inputs to the correct output.

Unlike supervised learning, unsupervised learning deals with unlabeled data without any expected outputs. A common use for unsupervised learning is clustering, where algorithms find hidden patterns within datasets. This difference leads to very different use cases.

2.3 Image Classification & Object Detection

Image classification is the process of assigning one or more classes to an input image based on the main objects present in the image [33]. For example, given a labeled dataset containing images of dogs and cats, an image classification algorithm can be trained to classify images as either containing a dog or a cat. Since a labeled dataset is required to train an image classifier, image classification is a supervised learning problem.

Localization is the task of finding the location of objects in an image. This is usually done with rectangular bounding boxes represented by a center coordinate combined with height and width values [6]. Localization algorithms can either be used for single or multiple objects. Generally speaking, Object Detection combines classification and localization for multiple objects in an image [6].

Instance Segmentation goes further than object detection by attempting to create segments with each known object on a pixel by pixel basis [33]. These segmentations can be represented using polygons forming the shape of the detected object, or as previously mentioned using groups of pixels.

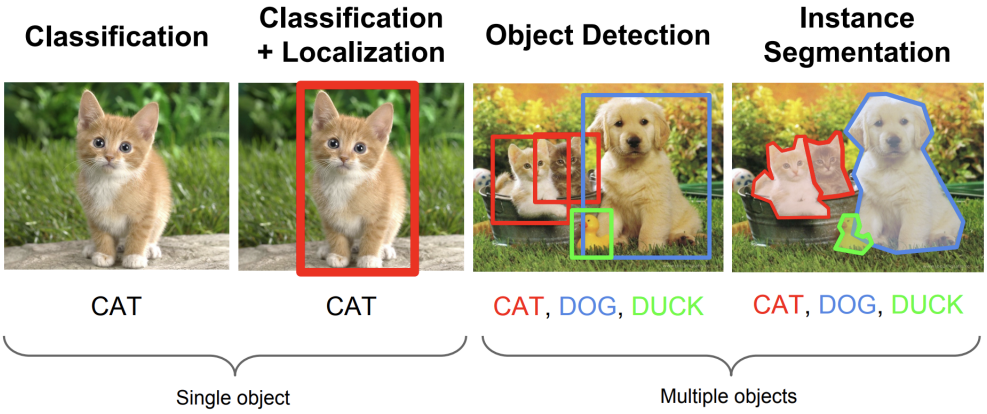

Figure 2.1: From Stanford CS231n Winter 2016 lecture 8 slides [17]

2.4 Convolutional Neural Networks

Convolutional Neural Networks, or CNNs, are frequently used for image classification and object detection tasks. This section will examine how CNNs operate, and provide some simple examples.

CNNs almost exclusively process square images for efficiency and consistency. Most images have to be reshaped to the CNNs input dimensions prior to being processed by the network. Computers represent images as collections of pixel values. A pixel is the smallest element in an image, and is usually represented by 1, 3, or 4 color components. These individual color components are also called channels. Greyscale images are an example of single channel images. RGB (Red Green Blue) is often used to represent color images, but for printing purposes CMYK (Cyan Magenta Yellow Black) is also frequently used. For these three types of images, each color component of a pixel has an intensity value within a certain range.

One of the main components of CNNs are filters or kernels. Filters are $n \times n$ matrices where n is an integer. They usually have small sizes like 3×3 or 5×5 [18]. Filters allow CNNs to detect features like edges, or curves.

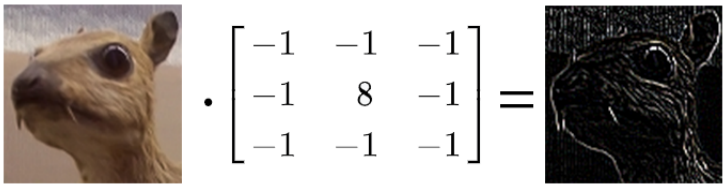

Figure 2.2: Filter Convolution Example [22]

To understand how Convolutional Neural Networks function, one must understand the mathematical operation called convolution. 2D convolution involves moving a filter over a 2D array of data and continually calculating the dot-product with the filter and a subset of the data. Movement is required because images are

orders of magnitude larger than the filters. If we have a 3×3 filter, the convolution process will begin by computing the dot product of the filter with the top-left 3×3 section of the image. The result of the dot product operation is a single number that is stored in a new array. The convolutional filter then shifts to the right. The distance of each shift is called the stride. A stride of 2 means the filter will move two units in the desired direction. When the filter has passed across the entire width of the image matrix, it then moves back to the left and down by the stride value. See Figure 2.3 for a visualization. In this way, the filter is passed across the entire image, and a new "filtered" image is generated. Padding, or surrounding the original matrix with zeros, may be necessary to ensure the filter can cover all of the data [18]. A padding size of 1 adds a row or column to each side of the original matrix, increasing its width and height by 2.

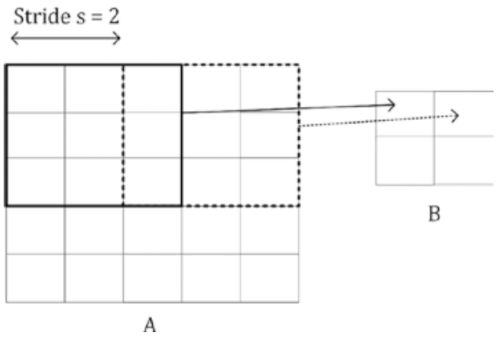

Figure 2.3: Convolution Visualization with Stride of 2 [18]

Depending on the stride and padding values, convolution can output a smaller matrix or one of the same size [18]. If the filter is $n_K \times n_K$, the input matrix is $n_A \times n_A$, stride is s, and padding size is p, then the resulting dimension after convolution will be:

$$\lfloor \frac{n_A - n_K + 2p}{s} \rfloor + 1$$

CNNs are generally made from stacks of three main layer types: convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply the previously discussed process. Pooling layers however, are used to further reduce dimensionality. Max pooling is the most common pooling operation [18]. Pooling also relies on a $n_K \times n_K$ region and a stride, but the stride is usually equal to n_K for pooling. Max pooling simply outputs a matrix with the maximum values from each region. Let's use a simple concrete example where our region is 2×2 and stride is 2. A will be our max pooling input, and B will be our resulting matrix.

$$A = \begin{bmatrix} 1 & 2 & 4 & 1 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 0 \\ 0 & 2 & 7 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 4 \\ 2 & 7 \end{bmatrix}$$

Going back to convolutional layers, each layer typically applies multiple filters or kernels. Since we are no longer dealing with simple matrices, we will be using the term tensor from here out. An example of a convolutional layer with n_c filters can be seen in Figure 2.4. Additionally, convolutional layers are always followed by pooling layers [18].

CNNs begin by detecting lower level features like edges and curves. That information is then used by subsequent convolutional layers to detect higher-level features. Finally, fully connected layers composed of artificial neurons are activated by preceding layers. These neurons learn to connect features from convolutional layers with output classes. Generally speaking, CNNs are composed of a number of convolutional and pooling layers followed by a number of fully connected layers that eventually output class predictions.

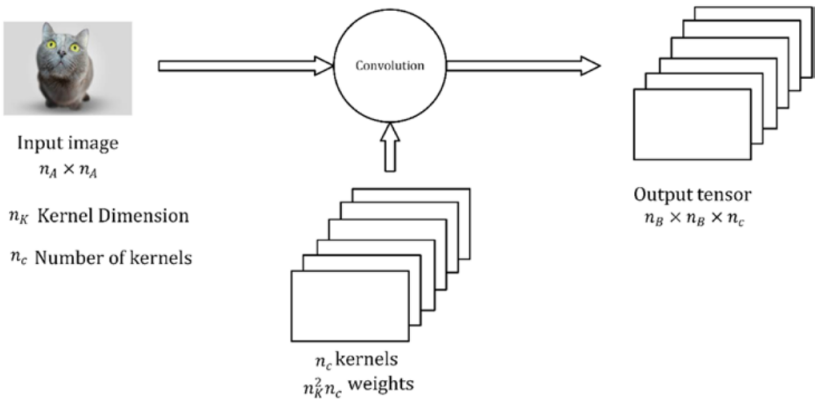

Figure 2.4: Convolution Layer Example [18]

2.5 YOLO (You Only Look Once)

2.5.1 Introduction

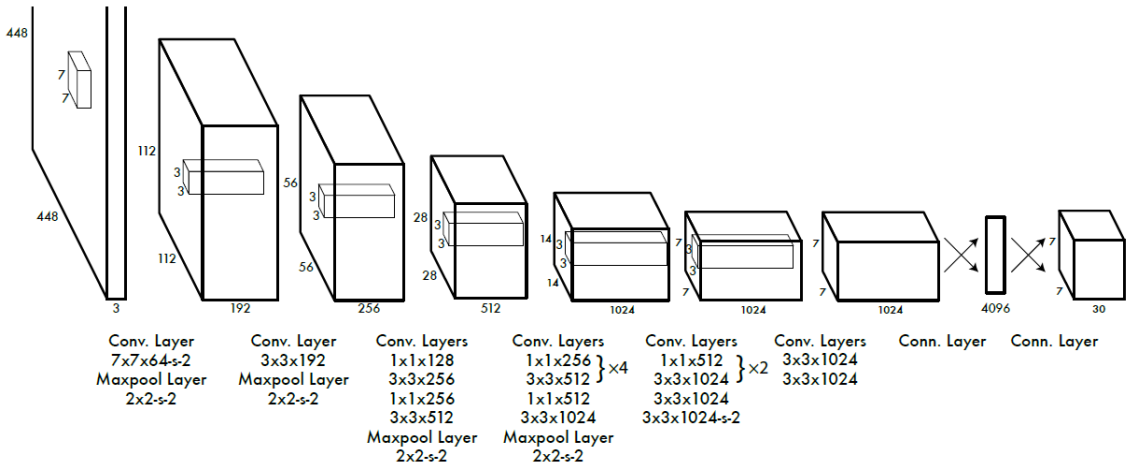

Figure 2.5: YOLOv3 Network Architecture [24]

YOLO is a prominent object detection algorithm. It was first presented in a 2016 paper titled "You Only Look Once: Unified, Real-Time Object Detection" by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi [25]. The work was a joint effort between the University of Washington, the Allen Institute for AI, and the Facebook AI Research team. Since publishing their first paper, there have been

two significant revisions resulting in YOLO-9000 and YOLOv3 [24]. While we used YOLOv3 for this project, we will first try to understand the original algorithm before addressing modifications that were later made.

Prior to YOLO, most accurate object detection algorithms used an approach based on repeatedly running an image classifier over portions of an image. Earlier systems like DPM (deformable parts models) used sliding windows to run over the image [25]. These systems classified each section and ran post-processing to remove duplicates. Later systems like R-CNN (Region Convolutional Neural Networks) used region proposals first, as opposed to a sliding window [25].

Systems like DPM and R-CNN are fundamentally pipelines containing multiple algorithms an image is run through. This results in multiple passes through an image classification network. While these algorithms can achieve relatively high accuracy, a large tradeoff exists with processing speed.

YOLO sets itself apart by implementing a unified approach to object detection with a single convolutional neural network. YOLO (You Only Look Once) gets its name from the fact that an image is only passed through the network once, as opposed to looking at multiple regions in separate passes.

2.5.2 YOLO Algorithm

YOLO begins by segmenting an input image into a grid of S x S cells. Each cell in the grid is responsible for predicting B bounding boxes. Each bounding box is comprised of five different numerical predictions: x, y, w, h, and confidence [25]. All five values are represented as floating point numbers from 0.0 to 1.0. Here, the x and y represent the center of the predicted bounding box, and their values are relative to the bounds of the grid cell that the point (x,y) falls into. The variables w and h represent width and height respectively relative to the entire image.

With YOLO, confidence is defined as:

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}} [25]$$

Pr(Object) is the model's confidence a box contains an object. Initially, the YOLO model only cares if an object exists and does not consider object classes. IOU stands for intersection over union, and is a measure that compares predicted boxes with ground truth bounding boxes generated by humans as part of the data labeling process [26]. The intersection is the area of overlap between predicted and ground truth bounding boxes. The union is the combined area of both boxes. Therefore, IOU is calculated by dividing the intersection by the union. A visual representation of this can be seen in Figure 2.6. IOU values approaching 1 mean the predicted box is extremely close to the ground truth box. As IOU decreases so does the models localization accuracy.

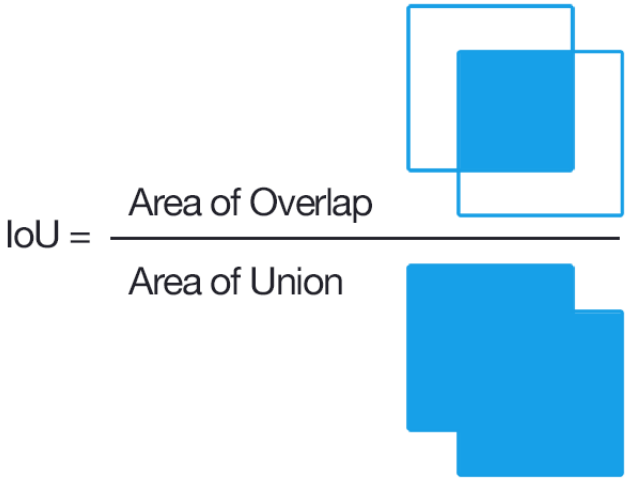

Figure 2.6: Intersection Over Union [26]

Each cell in the S x S grid additionally predicts C conditional class probabilities (where C is the number of labeled classes). These probabilities are conditional on an object being centered in the cell, so Pr(Class$_i$ | Object) [25].

Figure 2.7: Intersection Over Union [26]

After the image is divided into grid cells, and each grid cell predicts B bounding boxes and C class probabilities, it outputs predictions. YOLO predictions are formatted as a S × S × (B * 5 + C) tensor.

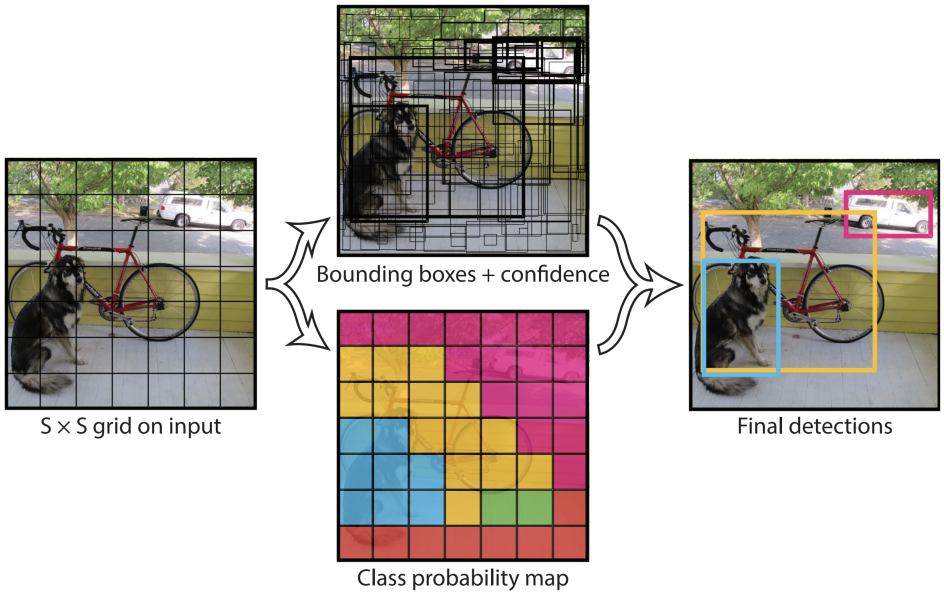

Figure 2.8: The YOLO system [25]

2.5.3 Non-Maximum Suppression

A process called non-maximum suppression (NMS) can be applied to the outputs of an object detection algorithm to filter predicted boxes and remove presumed

duplicates. The authors of YOLO note this process is not necessary, unlike for R-CNN or DPM, but that it results in a slight accuracy increase for YOLO [25].

NMS Input:

- B - a list of proposed boxes (height, width, and center position, object class)
- S - a list of corresponding confidence levels
- N - the NMS overlap threshold

NMS Output:

- D - A list of filtered proposals along with their corresponding confidence levels

NMS Algorithm:

1. First, the proposal with the highest confidence value is moved from B to D which will hold the final accepted proposals.
2. Next, the accepted proposal in D is compared with every proposal in B. If the Intersection Over Union (IOU) for the accepted proposal and a proposal in B is greater than the NMS threshold, then that proposal is removed from B. This removes overlapping boxes with lower confidence levels irregardless of class (assuming the IOU value crosses the NMS threshold).
3. Once again the proposal with the highest confidence value is selected from B and moved into D.
4. Like in step 2, this proposal is compared with all remaining proposals in B while eliminating proposals from B that exceed the IOU threshold.
5. This process of selecting the highest confidence proposal and comparing it to all proposals in B is repeated until B is empty.
6. The algorithm then returns D which contains all accepted proposals.

[13]

Since the NMS algorithm does not consider object classes, setting an appropriate NMS threshold is dependent on the specific object detection use case. Setting the threshold too low could result in unintended rejections for nearby objects. However, if the threshold is set too high, the same object may be detected multiple times. Any application that needs to detect overlapping objects, or count objects needs to consider these factors.

Luckily, our application of NMS is less sensitive since most people only use one hand at a time when playing Rock, Paper, Scissors. In our application, it doesn't matter if the same object is detected multiple times as long as the class predictions are all correct.

2.5.4 Transfer Learning

Transfer learning is the process of using knowledge learned from performing a task to improve learning a new but related task [31]. In this way, valuable learned information can be applied to many similar tasks. Transfer learning is related to how humans learn. For example, take the task of driving. Learning to drive safely requires some amount of time spent learning. Imagine after learning to drive in one specific car, you now have to drive a different car. The inputs you receive in both cars, like visual cues from outside and gauges, are likely very similar. Driving either car requires similar outputs, and performance metrics would remain the same (don't crash, obey the law, etc.). With all the driving knowledge learned from driving the first car, driving another similar car should require much less learning.

With YOLO, input and output types remain the same no matter what type of objects the system is trained to detect. While it is absolutely possible to train a YOLO network to detect custom objects from scratch without transfer learning,

this would take significantly more time. Instead, convolutional weights from the Darknet-53 model are used as a basis to extract features from an image.

As stated in the original YOLO paper, the convolutional layers of the original YOLO network were trained on the ImageNet dataset which contains 1000 object classes[25]. These convolutional layers were pre-trained for around one week in the development of YOLO. YOLOv3 uses a different convolutional neural network structure than YOLO or YOLO9000. This new network called Darknet-53, created alongside YOLOv3, uses 53 convolutional layers and was also trained using the ImageNet dataset [24].

For the purpose of this project, a YOLOv3 network was trained to detect three distinct classes corresponding to the three hand gestures used while playing Rock Paper Scissors. However, when examining the original YOLO paper and the YOLOv3 paper, it becomes obvious that the YOLO object detection system is traditionally used with a higher number of classes. The YOLO papers include benchmarks using the COCO and Pascal VOC 2007 datasets [25]. COCO is an acronym for "Common Objects In Context" and contains 80 different object classes (COCO). The Pascal VOC 2007 dataset contains 20 different classes [11]. Both are large scale datasets used for object detection and segmentation problems. All iterations of YOLO require changing model configuration parameters depending on the total number of classes. Specific changes made to accommodate three classes are covered in the Methods chapter.

2.5.5 Performance Metrics

While IOU, as described in section 2.5.2, is an important metric for comparing individual predictions to ground truth bounding boxes, metrics like Mean Average Precision (mAP) are used to evaluate information retrieval systems [3] and object detection systems [11].

First it is important to understand precision and recall. Precision is a measure of the "false positive rate" [2] [8].

$$\text{precision} = \frac{|\text{retrieved and relevant at rank n}|}{|\text{retrieved at rank n}|}$$

Within the field of object detection, rank is usually represented by an IOU value (like 0.5) or a confidence value from a detector system. Only detections at or above a chosen rank are considered "retrieved". The denominator here is simply the number of objects detected by the system for a given class. A detection is relevant if the class prediction matches the ground truth label.

Recall evaluates the ability of a system to retrieve relevant information [8] and is a measure of the "false negative rate" [2].

$$\text{recall} = \frac{|\text{retrieved and relevant at rank n}|}{|\text{retrieved}|}$$

Here the numerator remains unchanged, but the denominator represents the number of ground truth objects for a given class.

A high precision rate (close to 1.0) means a system is returning primarily relevant or correct results. A high recall rate (also close to 1.0) means the system is returning almost all of the objects for the given class.

For example, imagine a hypothetical object detection system trained to detect cars and trucks. Given an image known to contain 10 cars and 4 trucks, the system predicts 9 cars with IOU values at or above a given threshold value. If the system correctly detected 7 cars, and incorrectly detected 2 trucks as cars, the precision for detecting cars would be $\frac{7}{9}$, and the recall would be $\frac{7}{10}$.

A precision-recall curve is calculated for a specific class by varying the confidence threshold for detections (different from IOU) and calculating precision and recall at each threshold [2]. IOU examines localization accuracy, but most models output

a confidence percentage for each detection being correct. While IOU remains unchanged, the detection confidence threshold is varied. Here, a detection must be over both the IOU threshold and the confidence threshold to count as a detection. These values are traditionally plotted as scatterplots with precision on the y-axis and recall on the x-axis. AP (Average Precision) is then calculated as the mean precision value taken at a set of equally spaced recall values. For the Pascal VOC Challenge, 11 recall values were used from $[0, 0.1, \ldots, 1.0]$ [11]. Precision-Recall curves often contain some wiggles, so an interpolation process can be used to smooth the curve out as is done with PASCAL VOC[11].

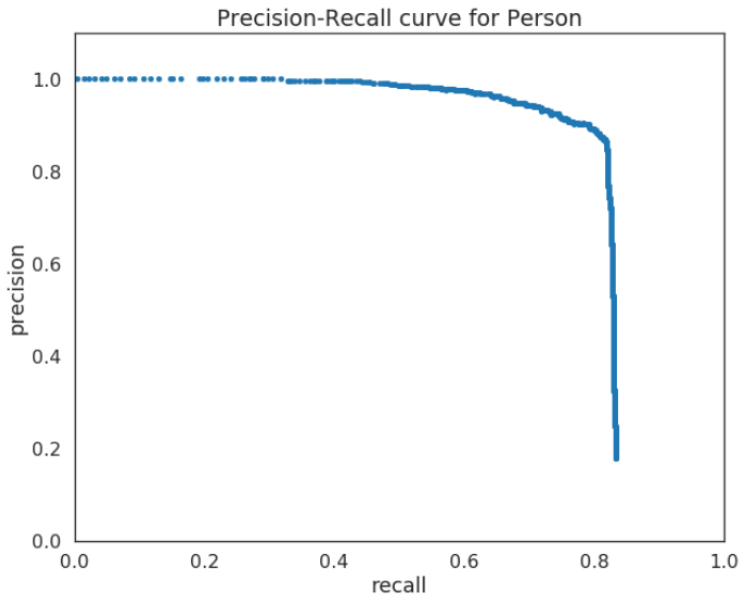

Figure 2.9: Example precision-recall curve for a fixed IOU threshold and varying confidence threshold [2]

Different object detection competitions use different IOU threshold values. PASCAL VOC uses one IOU value of 0.5, which they admit "was set deliberately low to account for inaccuracies in bounding boxes in the ground truth data, for example defining the bounding box for a highly non-convex object, e.g. a person

with arms and legs spread, is somewhat subjective" [11]. Additionally, a set of varying IOU values can be used and averaged later in the mAP calculation process.

mAP is calculated by taking the mean of AP for all object classes and IOU thresholds. In the case of PASCAL VOC2007, with a single IOU threshold value of 0.5 and 20 classes, mAP is calculated as the mean of AP for all 20 classes. The COCO 2017 challenge used 10 IOU threshold values with 80 classes [2]. For COCO 2017, 10 different precision-recall curves would be calculated for each class. From each curve an AP value is then calculated. mAP is then the mean of all calculated AP values.

CHAPTER 3

Related Work

3.1 rps-cv Project

One important decision made during the course of this project was choosing the detection algorithm. Initially, our plan was to use some form of image classification algorithm. Image classification is the practice of classifying an entire image into one or many distinct categories.

One project that took this image classification approach is rps-cv (Rock Paper Scissors Computer Vision) by Julien de la Bruère-Terreault which was published in The MagPi Magazine Issue 74 [10]. This project uses a Raspberry Pi computer, a camera, a 3D printed stand with integrated lights, and a green surface that serves as a photographic background. For image classification, this project uses a Support Vector Machine (SVM) that classifies images as either "Rock", "Paper", or "Scissors". SVM's are a class of machine learning models.

The Raspberry Pi is a small single-board computer with moderate processing capabilities. This was likely an influential factor in the choice of algorithm and hardware setup. Their custom camera and light stand helped to ensure as much consistency as possible when collecting training data and when using the application. Due to these factors, the pictures they captured from the Raspberry Pi camera look

extremely evenly and consistently lit. Using a fixed camera position made it possible to ensure captured photos only contain the background and a single hand.

Having a highly structured and consistent environment has a number of advantages. First of all, the captured frames do not contain any extraneous objects or information. Since only a single hand is present in each image, it's much easier to remove the image background and focus entirely on the hand gesture.

When deciding on an approach for our project, we also began by taking an image classification approach. Without having the same consistent setup as with the rps-cv project, and even using a more complex convolutional neural network, the results from our initial training was unsatisfactory. While we cannot speak directly to the feasibility of using image classification for gesture detection, without specialized hardware and a consistent photographic environment, we decided to explore additional options.

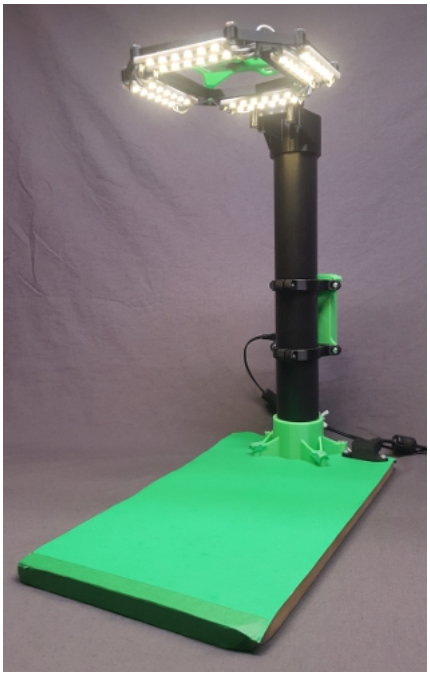

Figure 3.1: rps-cv hardware [10]

3.2 Gesture And Activity Classification

Before examining related work in gesture recognition, it is important to understand the distinction between gestures and activities in the context of video and image classification. Gestures, by definition, involve some movement or positioning of ones body, and are a form of non-verbal communication. Activities on the other hand can include multiple people and their environment.

Some examples of gestures are:

- Waving
- Thumbs up
- Pointing
- Sign language

Some examples of activities are:

- Shaking hands
- Opening a door
- Taking an object
- Reading a book

There is significant overlap between gesture recognition and activity recognition. Gesture recognition is primarily used as a way to interact or communicate with computers, while activity recognition can also be used in applications like video surveillance.

Object detection algorithms have not traditionally been used to detect gestures and human actions. As a paper published in 2018 titled *YOLO based Human Action Recognition and Localization* mentions, there are two main methods for classifying activities in a video stream [28].

3.2.1 Two Stream CNNs

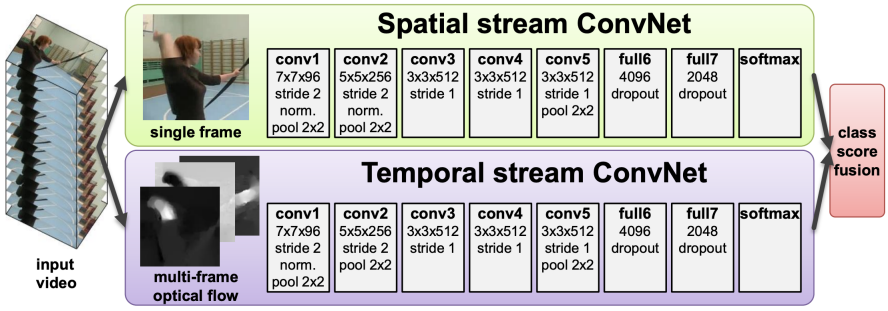

Figure 3.2: Two-Stream CNN Architecture [29]

One method for classifying gestures and human activities utilizes two-stream Convolutional Neural Networks (CNNs). With this method, two parallel CNNs are used for spacial and temporal data.

Temporal data is captured across multiple frames using optical flow algorithms. This data helps to convey directionality and magnitude of movements over time. Optical flow displacement between several frames can then be stacked and transformed into images representing the location, direction, and intensity of motion. Depending on the implementation, the vertical and horizontal components of optical flow vectors can be split into separate frames [29].

In parallel, a spacial stream is passed still images. This stream performs standard image classification, since certain actions may be associated with certain objects. At the end of this two stream process, the class scores from both CNNs are fused. This method simply takes two different approaches to gesture classification with the view that the two models will complement each other and improve overall accuracy. An example of this two-stream architecture can be seen in Figure 3.2.

3.2.2 Skeletal Tracking

Another popular method for gesture/activity classification is skeletal tracking or pose estimation. While many systems like the Microsoft Kinect and Leap Motion rely on stereo cameras for 3D depth information and more accurate pose estimation, some systems like DeepPose function with a single 2D image [32]. Pose estimation systems are more complicated, so for the sake of this comparison, they map points in 2D or 3D to specific parts of the body like wrists, elbows, fingers, or noses. Pose estimation is a very computationally expensive process, and a separate model is required to classify activities from the positional pose data. Some example DeepPose results can be seen in Figure 3.3.

Figure 3.3: DeepPose Examples [32]

3.2.3 YOLO for Gesture Recognition

Another option for gesture and activity recognition is to solely use an object detector. In *YOLO based Human Action Recognition and Localization* Shubham Shinde et.al used YOLO (v1) for activity classification [28]. For their dataset, they used the LIRIS Human Activities dataset which contains 367 actions within 167 videos.

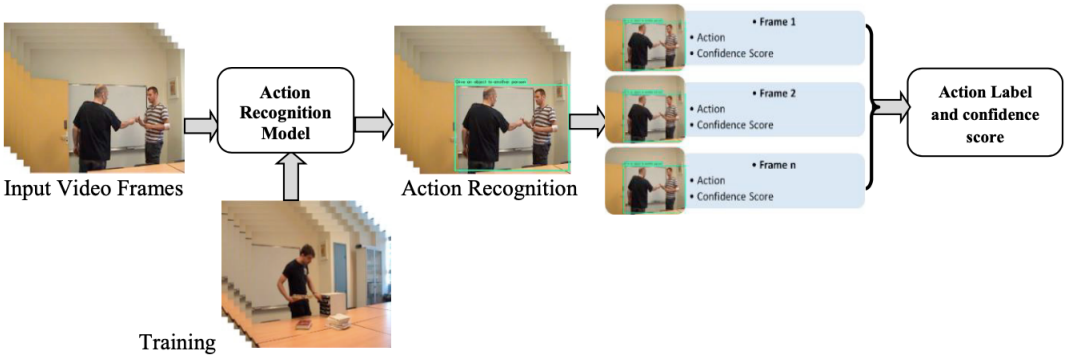

Figure 3.4: YOLO Activity Classification [28]

The authors first split the LIRIS dataset into 109 training videos and 58 testing videos. Next, individual frames were chosen from the training set and labeled with any applicable actions. These images and their accompanying labels were then used to train their YOLO model. Some examples of action classes are:

- Discussion between two or more people
- Give an object to another person
- Handshaking
- Typing on a keyboard

While their methods are slightly unclear, their system runs a series of frames through YOLO and looks at repeated detections across multiple frames. An overview

of their system architecture can be found in Figure 3.4. One of their key findings is that activities can often be detected from a single frame [28].

Their findings, and specifically a mean precision value of 89.88% was very promising. While detecting actions in video is different from detecting gestures for Rock Paper Scissors, actions involving multiple people would seem to be more variable than three hand gestures. Therefore we thought YOLO would have a good chance of being effective for our use case.

CHAPTER 4

Methods

4.1 Algorithm Selection

As previously mentioned, one of the goals of this project is to explore new forms of human-computer interaction, or HCI. A Google search for "Rock Paper Scissors game" reveals many different websites that allow you to play the game online. Some pit players against a virtual or computerized opponent, while others allow two human players to play each other remotely. One example of these websites is rpsgame.org. It allows to remote players to compete with each other. After both players connect to the game, each player clicks on an image of a rock, paper, or scissors. When both players have selected their move, the result for the round is displayed to both players along with their opponent's chosen move.

Having buttons to click provides a simple and intuitive user interface that allows users to use the application in a web browser with very minimal computational resources. However, this approach results in a completely different way of playing the game. Clicking one of three buttons on a screen is a completely separate action compared to actually playing Rock Paper Scissors physically in person. The rps-cv project discussed in Chapter 3 represents a middle ground where hand gestures are used, but where a custom apparatus is needed. For that reason, we wanted to

create a way to play the game virtually while returning to the original method of gameplay. This led us to our decision to use images from conventionally positioned webcams.

Laptop and Desktop webcams are traditionally positioned to show the upper torso and head of the person in front of them. Their orientation and focal length by nature result in a good deal of potentially inconsistent background information. Apart from their displays, most computers do not produce any significant light either. This led us to look for a computer vision algorithm that can function well in an inconsistent physical environment with different lighting conditions.

This goal of allowing players to use natural physical gestures combined with our limited attempt to use image classification led us to examine object detection algorithms. Unlike image classification where classes are determined by the entire image, object detection is the process of identifying areas of an image with specific objects [6]. We theorized that object detection algorithms would perform better with images that contain large amounts of data that is irrelevant in determining player gestures since they can focus on smaller regions. This is much like how a human would play by watching their opponent's hands and identifying the gesture.

There are many different types of Object Detection algorithms, but we decided to use YOLOv3. While other families of algorithms are generally more accurate, like R-CNNs, at the time of our research YOLO models presented an excellent balance between accuracy and performance [6]. Additionally, YOLOv3 has become extremely prominent in the field of computer vision which has resulted in significant third party support of the model.

4.2 Data Collection

We began the data collection portion of this project by taking a series of short videos. In each video, one subject simulated multiple games of Rock Paper Scissors by making one of the three hand gestures at a time. Videos were captured using a variety of cameras in different physical environments with varying lighting conditions and with several different people. Data was collected primarily with a 720p MacBook FaceTime camera, and a 1080p Logitech C920 USB webcam.

Figure 4.1: Sample Rock Gestures

For each of the three gestures (Rock, Paper, or Scissors) we tried to collect diverse training data. While we found the Rock gesture to be relatively more consistent (Figure 4.1), both Paper (Figure 4.2) and Scissors (Figure 4.3) resulted in people using many different hand orientations. Please note that for the figures these gestures

Figure 4.2: Sample Paper Gestures

were cropped as squares from larger images, and that the location of the crop is separate from the annotated bounding boxes covered in Section 4.3.

After capturing multiple different video clips with several subjects, frames were extracted as JPEG images. We extracted five images per second of video using a command line application ffmpeg. An example frame extraction command is listed below:

```
ffmpeg -i video.mov -vf fps=5 -qscale 0 thumb%04d.jpg -hide_banner
```

By default, ffmpeg outputs images that are noticeably lower quality than the corresponding source videos. Setting qscale to 0 keeps the quality at a similar level, and results in larger output file sizes.

Figure 4.3: Sample Scissors Gestures

4.3 Data Annotation

Over the course of this project, over 5500 unique frames were extracted to be labeled. A cloud based application Labelbox [16] was used to store our dataset of images, as well as to label them. Labelbox supports using multiple datasets for a single project, so we created several to correspond with how the data was captured and how it should be used. All images with the exception of 450 images in a validation dataset should be considered part of the training dataset. For each individual image, we used Labelbox to draw bounding boxes around any Rock, Paper, Scissors, gestures present in the image. The following classifications were applied on an as needed basis to exclude images from the main training dataset:

- Blurry
- No Gestures

- Other

Since our images were captured from videos, by nature some frames were too blurry to recognize any gestures, or didn't contain any of the specific hand poses and were classified respectively. Images with recognizable but blurry gestures were additionally classified as "Blurry" to keep our training dataset as clean as possible. The "Other" skip classification was used to exclude images for other miscellaneous reasons like poor lighting.

Each bounding box was drawn to include the entire hand stopping at the wrist. As previously mentioned, each box was classified as one of the three Rock Paper Scissors gameplay poses. See figure 4.4 for an example image from our dataset with bounding boxes as displayed by the Labelbox application. Here yellow represents Scissors and purple represents Rock.

After labeling all of the images in Labelbox, the labels were exported in a JSON format from Labelbox. This JSON file contained hyperlinks to each image hosted by Labelbox, as well as label information. Rectangular bounding box labels are exported from Labelbox as a string representing the class and four (x,y) coordinates corresponding to each corner of the label's box.

The official YOLOv3 darknet [23] implementation by Joseph Redmon requires that each image in the dataset have a corresponding .txt file containing one label per line. Each row is in the format of class x_center y_center width height. Classes are represented numerically with each number corresponding to a line in a separate classes text file. x_center is the center of the bounding box along the x-axis as floats from 0.0 to 1.0. This value was found by finding the center of the box along the x-axis and dividing this by the width of the image. The same process was followed for y_center using the y-axis and dividing by the image height. width and height are also floats from 0.0 to 1.0 representing the ratio of box width/height to image width/height.

Figure 4.4: Example frame in Labelbox

Due to the data format discrepancies between Labelbox and darknet, a preprocessing script was written in Python to download the images and correctly format the labels for training. This script automatically excludes images with a skip classification, translates bounding box formats, and creates text files for each image containing their respective label data. The preprocessing script has two modes to be used with training data or validation data. Validation data was exclusively used to evaluate the performance of different models after training, and was never used in the model training process.

Our final dataset contains 3112 training and 783 testing images. While we did our best to generate diverse training data, one area we could have improved in was balancing out our dataset. Our final dataset was 39% Scissors, 37% Rock, and

25% Paper. Rock naturally appeared the most in our training videos, but our initial models struggled with detecting scissors, so we created additional data to help rectify that issue. For our validation set however, we took care to ensure an even balance with 155 images for each class for a total of 465.

4.4 Training

Initially we began by attempting to run darknet YOLO on a 2018 Macbook Pro. After some testing it became apparent that a GPU with CUDA support would be needed for training YOLO. This led us to set up a dedicated Ubuntu 18.04 LTS workstation with a NVIDIA GTX 1060 6GB GPU.

Unlike some other machine learning frameworks, darknet uses text based configuration files to specify model and training parameters. Several different configuration files for YOLOv3 come preconfigured with darknet. For this project we copied yolov3.cfg and first adjusted the network resolution from 608x608 px to 416x416 px for performance reasons. Next, we followed instructions from AlexeyAB's darknet fork [1] to modify the configuration file for our custom dataset. Default YOLOv3 configuration files assume 80 different classes, but for this project we used three classes. Several changes were made to create a new configuration file to support three classes.

Darknet configuration files used for YOLO are comprised of many sections. Each section begins with a string enclosed in brackets like [convolutional]. Darknet configurations begin with a [net] section used to specify training and inference parameters. For example, this includes batch size, data augmentation parameters, and input dimensions. Each following section in the file represents a layer in the neural network. This standardized format allows any Darknet model to be run with a configuration file and model weights.

While darknet configuration files have an argument for image rotation, this seems to be currently unimplemented for training. However, darknet does support saturation, exposure, and hue augmentation.

To select augmentation values, we began by training a baseline model with all values set to zero. We then trained three models with varying saturation values, three with varying exposure values, and three with varying hue values, for a total of 9 additional models. Each parameter was set to 0.5, 1.0, and 2.0. Each of the nine models had only one augmentation parameter not set to zero. After training all 10 models, each one was tested against our validation set to calculate mAP and other performance metrics. Darknet does not have any functionality to compute these metrics, so AlexeyAB's popular darknet fork was used for all model validation [1].

When training a machine learning model, an important decision is how long to train the model for. Not training enough will result in worse precision, recall, and IOU metrics. Training for too long can overfit the model and decrease its performance on new unseen data. To determine a good stopping point for training, we trained a model for 12000 iterations. For YOLOv3, 2000 iterations per class is commonly recommended [1]. Darknet automatically saves model weights every 1000 iterations, so after training for 12000 iterations we tested each set of weights against our validation set. For our final model, we selected the weights with the highest mAP score.

CHAPTER 5

Gameplay Prediction

5.1 Background

If you Google "How to win at Rock Paper Scissors" there is a high likelihood you'll end up on a page that cites a paper published in 2014 titled *Social cycling and conditional responses in the Rock-Paper-Scissors game* [34]. This study was conducted at Zhejiang University in China and was published in the prestigious Nature journal. It has since been referenced in numerous news articles, blogs and videos.

The study's authors, Zhijian Wang, Bin Xu, and Hai-Jun Zhou, recruited 360 students at their university from various departments to participate in their laboratory experiment [34]. The participants were grouped into 60 disjoint populations of size 6. Each of the 60 populations then played a total of 300 Rock Paper Scissors rounds. Each experimental group played their games with a fixed payoff value a. Losing a round resulted in a score of 0, winning a round resulted in a score of a, and for a tie both players received a score of 1. Values used for a were 1.1, 2, 4, 9, and 100 with 12 population groups assigned each value [34]. To incentivize players, they were paid at the end of the experiment in cash proportional to their accumulated payoffs.

The aim of this study was to examine how humans make decisions in a competitive environment. The Nash equilibrium has frequently been applied to Rock

Paper Scissors, and as a model "predicts that players randomize completely their action choices to avoid being exploited" [34]. This assumes each game is completely separate, and that a players action in one game should not effect their action in another game. On the other hand, evolutionary game theory generally predicts the presence of cyclic motions [34], thereby assuming some form of dependence between games. Wang et. al. designed their laboratory experiment to determine if evolutionary game theory is an appropriate framework to use in this context, and if so to examine how humans make decisions involving cyclic motions.

When analyzing their data, they excluded one group with $a = 1.1$ as their behaviors differed from all other groups. Across these 59 remaining groups, the authors found that "the probabilities that a player adopts action R, P, S at one game round are, respectively, 0.36 ± 0.08, 0.33 ± 0.07 and 0.32 ± 0.06 (mean ± s.d.)" [34]. This first piece of data is compatible with the Nash equilibrium framework, as it doesn't suggest dependence relationships between games. However when examining a players actions across two consecutive games, Wang et. al. discovered some correlations. For example, in their analysis, they found that for all population groups, from one game to the next, players were more likely to repeat the same action as opposed to switching gestures. This effect diminished as payoffs increased [34].

One key development of this study was the authors Conditional Response Model. Very simply put, this model suggests that players consider their past performance when deciding on their next move. This model is sufficiently mathematically complex to the point where it was not considered for our project. However, the authors suggest that their findings can be "regarded as an extension of the win-stay lose-shift (also called Pavlov) strategy" [34].

All of this evidence suggests that humans do not play Rock Paper Scissors completely randomly according to the Nash equilibrium, and therefore their

previous choices may be slightly exploitable. However, it is very important to note that this study found "persistent cyclic motions at the population level (but not at the individual player level)" [34].

We believe that population level patterns present the potential to improve gameplay performance against an individual human. This assumption only suggests that computers may be able to gain a slight edge against individuals by learning from a population level dataset of games.

5.2 Goals

While the main area of this Independent Study is around Computer Vision, it seemed fitting to utilize machine learning to try and give the computer an edge against players without explicitly breaking the rules. The goal of this section was to train a model to perform better than random for predicting future moves when dealing with population level data. This means the model needs to be more than 33% accurate in predicting a players next move over a number of unseen games.

5.3 Recurrent Neural Networks (RNNs)

Feedforward neural networks pass data through the network from layer to layer without any loops. If you imagine the network's inputs on one side, and outputs on the other, data always travels in the same direction [12]. Convolutional neural networks like those used in all iterations of YOLO are examples of feedforward neural networks.

On the other hand, Recurrent Neural Networks (RNNs) do contain loops within the network [20]. These loops allow for information to persist. Traditional feedforward networks, like Convolutional Networks, rely on fixed-size vectors for

their inputs and outputs, which is a limitation [14]. RNNs make it possible to work with sequences of data which is important in many domains like Natural Language Processing (NLP).

Language processing is a commonly used example for RNNs. Imagine you wanted to build a system to predict the next word in a sentence. A traditional neural network could be trained to learn which words most frequently follow other words, but it would be unable to include context or information from previous words. Sentences are simply an ordered sequence of words, but as we know they can vary in length. Predicting the next word in a sentence is known as a many-to-one prediction problem, because multiple words are being used to predict the one word which follows. Each word can be thought of as a time step in the sequence, and should ideally add to the context of the sentence.

Take the sentence "the clouds are in the *sky*". In this example, our network is trying to predict the word "sky" with the input "the clouds are in the". This is an example of a short term dependency [20]. Here it is fairly easy to see how predicting the last word is dependent on the context provided by the word "clouds". This dependency is short term because there is a small gap between the relevant information and where it was needed. Traditional RNNs are good at using short term memory, but as the gap between relevant information and where it is needed grows, they become much less effective.

Long Short Term Memory (LSTM) networks are a special form of RNN and were first introduced in 1997 to solve some of the issues with long term dependencies [20]. They are now a widely used form of RNNs and are very effective across a large set of problems. LSTMs are fairly complicated, and could be the subject of an entire Independent Study project, so fundamentally they use internal gates to decide what information should be retained and what information should be forgotten [20]. This is an intuitive idea because as humans our minds decide to prioritize certain pieces

of information over other information deemed less important. Within an LSTM layer of a neural network, the number of LSTM units determines how many LSTM cells data passes through. While we usually use loops to think about RNNs, in reality these loops can be unrolled into multiple connected units all with the same structure but different weights [20]. An example with t units can be seen in Figure 5.1.

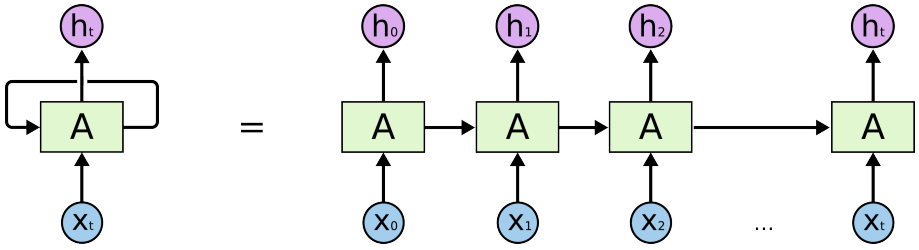

Figure 5.1: Unrolled RNN [20]

While using a LSTM network may not be the most effective for predicting future moves for Rock Paper Scissors, it is an approach that has been used successfully in the past by others in the same problem space [15].

5.4 Methods

We began the prediction element of this project by searching for a dataset of Rock Paper Scissors games. Through searching online, we found a blog that contained a downloadable dataset with over 120,000 Rock Paper Scissors games and over 445,000 individual throws [9]. This data originated from the roshambo.me database, but we were not able to source the dataset from them directly.

To begin with, we converted the dataset from an Excel file format to a Comma-Separated Values (CSV) format. The dataset contains four columns: game_id, game_round_id, player_one_throw, and player_two_throw.

The Python Pandas module was used to load this entire dataset. The data was then grouped by game_id, before being separated into training and testing sets. 90% of games were used for training and 10% for testing. Scikit learn was used to create this train/test split, and a consistent random seed was used. As a result, we were able to keep the training and testing sets identical across multiple training sessions to ensure consistent model comparisons.

We augmented each round with a value representing who won the round. We will call this the game result column. A loss for player1 is represented as 0, a tie as 1, and a win for player1 as 2. Each players throw was also recoded from 1-3 to 0-2 with 0 representing Rock, 1 as Paper, and 2 as Scissors. All values were kept between 0 and 2 to to ensure consistency when converting integers to categorical variables.

Each game naturally has a variable number of rounds depending on how long the users played for. Every game with more than three rounds was additionally subset into multiple games of incremental length to provide more training data. Each shorter subset starts with the same gameplay sequence but contains one fewer rounds. For each game in the dataset with n rounds, the first n-1 rounds are used as the input sequence, and then player1's move in the last round is the output to predict. Games with fewer than three rounds were excluded completely, to ensure all network inputs contained at least two time steps.

The Keras machine learning framework was used for training and model inference. While training each model, the batch size was set to 1. This was because batches need to have the same dimensions (or the same number of time-steps/rounds for each game). A bucketing process where games of the same length are grouped together could have been implemented, but we decided this would add unnecessary complexity to this project. In our implementation, the training data is looped

through one game at a time. Keras then fit each game individually to continuously improve the model.

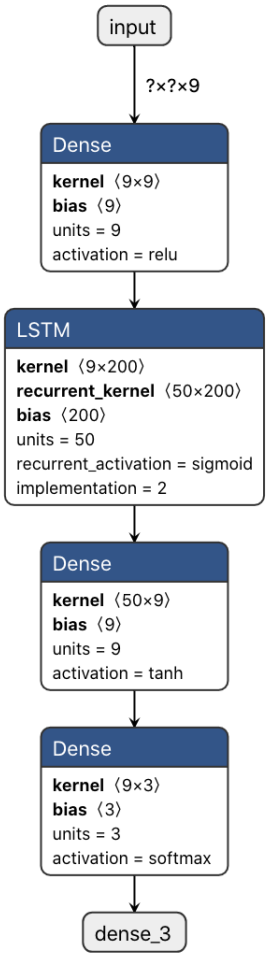

Figure 5.2: Prediction Model Architecture: Method 4 with 50 LSTM Units

A total of 28 different prediction models were trained. Four methods were used, and for each method 7 models were trained with 1, 3, 5, 10, 20, 50, and 100 LSTM units. Each method varied in the columns of data passed to it.

- Method 1 was only passed sequences of player1's throws.
- Method 2 was passed sequences of both player's throws.
- Method 3 was passed player1's throws and game results.

- Method 4 was passed both players throws and game results.

All models were identical except for input dimensions and the number of LSTM units. The general network structure can be seen in Figure 5.2.

As discussed previously, we decided not to use batches with our data, so we could not use any evaluation methods provided by Keras. Instead, for each model we looped through each game in the test set and used the model to predict an outcome. Our reported accuracy is the number of games in the test set where the output was correctly predicted divided by the total number of games in the test set.

5.5 Prediction Results

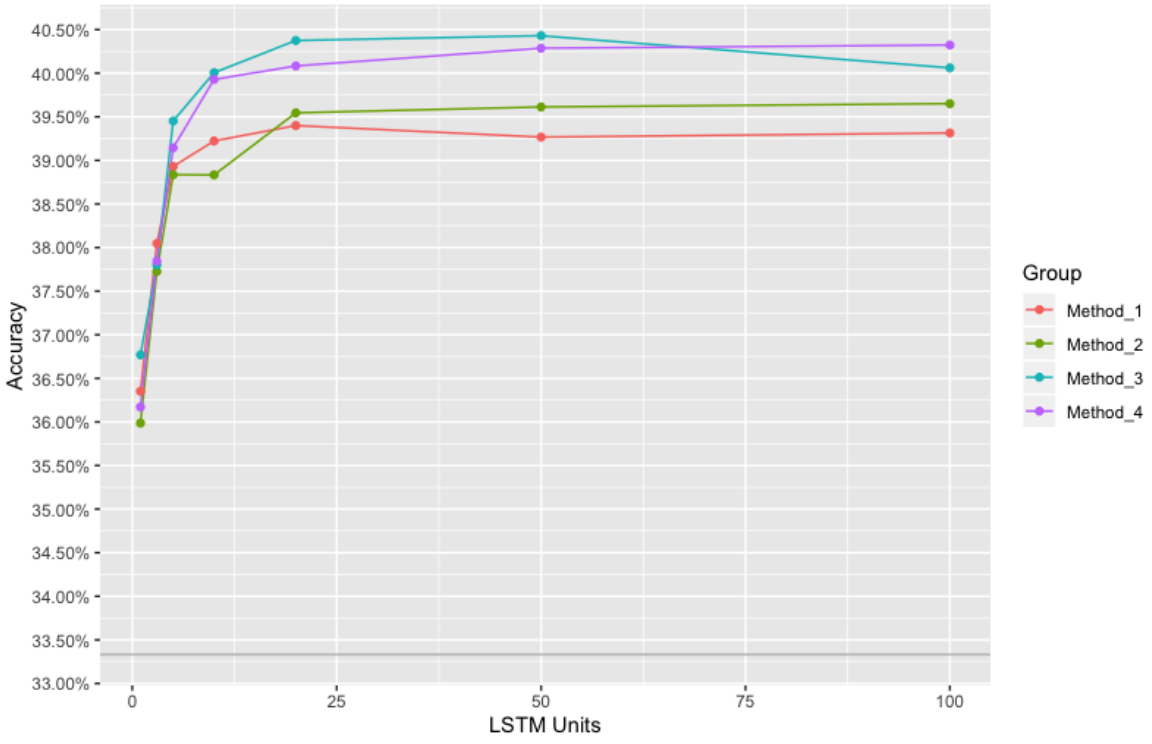

Figure 5.3: LSTM Prediction Model Accuracy

Method	LSTM Units	Accuracy
Method 1	1	0.3635
	3	0.3805
	5	0.3893
	10	0.3922
	20	0.3940
	50	0.3927
	100	0.3931
Method 2	1	0.3597
	3	0.3772
	5	0.3884
	10	0.3883
	20	0.3955
	50	0.3961
	100	0.3965
Method 3	1	0.3677
	3	0.3780
	5	0.3945
	10	0.4000
	20	0.4037
	50	0.4043
	100	0.4006
Method 4	1	0.3617
	3	0.3784
	5	0.3914
	10	0.3992
	20	0.4008
	50	0.4029
	100	0.4032

Table 5.1: Model Training Length Results

For our LSTM gameplay prediction models, we can see in Table 5.1 that Method 3 (player1's throws and game results) with 50 LSTM units had the highest predictive accuracy on the test set. However, both Method 3 and Method 4 performed quite similarly. As discussed in Section 5.1 some research has found evidence of a "win-stay lose-shift" population level strategy. Since both Method 3 and Method 4 include who won each game, it makes sense that these two models would perform

better. Our final accuracy of 0.4043 is similar to an accuracy of 0.3873 achieved by Paul Klinger in his RPS-RNN project with a simpler RNN model [15].

While 40% prediction accuracy may not seem like much, it should help the computer to gain an edge over players. This is definitely an area of this project that could be expanded upon. We were unable to find any relevant research papers on Rock Paper Scissors prediction, so how well a model can perform in this problem space remains a somewhat open question.

CHAPTER 6

Software

6.1 Inference

One of the early concerns when developing the software portion of this project was determining how the YOLOv3 model would be run. The darknet framework [23] is written in C, and as part of the build process compiles a darknet executable. This executable is used for several tasks including training and different inference demos. Inference is the process of passing an input through a machine learning model and receiving the predicted outputs. The darknet executable demo can take still photos, video files, and webcam streams as inputs. Using a webcam with the desktop training computer made it possible to process frames at the native camera speed of 30 frames per second without any bottlenecking.

On a 2018 Macbook Pro without CUDA support, darknet is forced to use the CPU for inference as opposed to a GPU which is significantly more efficient. Additionally, under macOS, darknet cannot be compiled with multithreaded support without installing additional compilers. Running on the 2018 Macbook Pro 2.6 GHz 6-Core Intel Core i7, in one single threaded test with YOLOv3 at 416x416 px an image took 16.45 seconds to process. Compiling darknet with OpenMP for multithreaded support brought that number down to 4.73 seconds for a single image. Compared

to 0.033 seconds with a CUDA capable GPU, running YOLO on the CPU didn't seem practical for this project.

After some research, we discovered that the OpenCV computer vision library supports inference for YOLOv3 models using the same weights and model configuration files. On the same MacBook Pro we found OpenCV could process frames on the CPU at around 3 to 4 frames per second (fps). Even conservatively choosing 3 fps, 1/3 of a second is a massive improvement over 4+ seconds with the darknet library. Based on these results, and with OpenCV being easier to install on various platforms, the decision was made to use the OpenCV library with Python 3.7.

6.2 User Interface

Figure 6.1: Application User Interface

While we initially tried to develop the software user interface using OpenCV, this proved to be too restrictive as OpenCV is meant for image and video processing.

For example, at the time of our research, buttons were only supported in C++ for OpenCV and were unsupported in the OpenCV python library.

After trying different graphical user interface libraries written for Python, we decided to use the Kivy framework. This choice proved to be a good one, and the resulting application relies heavily on Kivy for the interface and gameplay timing logic. Kivy provides multiple different types of widgets for creating user interface elements. For this application, images, buttons, and text were primarily used. Kivy additionally supports multiple layout configurations which were used to group elements together. As seen in figure 6.1, the application has a large live camera view on the left, and a sidebar on the right.

Due to slower YOLOv3 CPU processing speeds, a decision was made not to have a live display of images processed through the network. As a result, the application displays a live feed of images from the ImageProvider class. This reduces most lag in the application which was frustrating to users during testing.

In the sidebar, there is a large "Play" button at the top which initiates a new game of Rock, Paper, Scissors. Timers are used to display sequential text over the camera feed, and to begin detecting objects. After the first game is played, the score is displayed underneath the play button. After each round, a cropped image containing the user's detected gesture is shown alongside text containing the predicted gesture (Rock, Paper, or Scissors). The computer's choice for that round is displayed in the same manner. Finally, there is a "Something Wrong?" button at the bottom of the sidebar which is always present after the first round. This button is intended to be used after the application detects the player's gesture incorrectly. Clicking on it will delete the last round from the game history, recalculate the score, and save the entire image the incorrect detection was based on. If a Labelbox API key and dataset ID have been configured, the image is then uploaded to Labelbox.

Images that are saved locally, or uploaded to Labelbox, can then be manually labeled and used as additional training data.

6.3 Application Architecture

The image pipeline in this application was designed with a Producer-Consumer model to ensure modularity. The ImageProvider class continually captures images using the OpenCV library and places them into a queue. From there, frames can be consumed by different parts of the application. Before a game begins in the application, all frames are consumed by the Application class and displayed by Kivy. To reduce resource consumption, images are only processed through the YOLOv3 network when necessary to detect gestures during gameplay. The ImageProcessor class handles all object detection processing in a separate thread. When network processing is enabled, frames are consumed from the ImageProvider class, processed through the YOLOv3 network, and placed into a processed frames queue. A parallel queue of object detections containing box location, class, and detection confidence is also used. The application class then consumes from both parallel queues. When one or multiple objects are detected in a frame, the object with the highest confidence level is chosen. From there, Rock, Paper, Scissors gameplay logic is used to determine if the player or computer won. A cropped version of the image which was used to detect the last gesture is then displayed to allow the user to verify the detection was correct. An overview of this image pipeline can be seen in Figure 6.2.

For choosing the computer's next play, a weighted random strategy is used for the initial round. This is done using numpy and a probability distribution from the previously mentioned Wang et. al. study [34]. From the weighted random selection, the computer plays the opposing gesture that beats it.

After the first round, the players throw history and win history are passed to the LSTM prediction network. This network predicts the players next move at the beginning of every round, and chooses the appropriate play to win against the players predicted play.

6.4 Software Installation and Delivery

This application was primarily developed to run on Mac and Linux computers. Kivy is a multi-platform user interface framework that supports mobile operating systems like iOS and Android, but after some research we determined that getting OpenCV and Kivy to work together on mobile would require a significant amount of time.

From a software standpoint, our software requires Python3 and pip to install python packages. A webcam is the only requirement which may not be part of every computer's hardware. After downloading the application code, which is hosted on Github at https://github.com/nick-hunter, we recommend creating a Python virtual environment to run the application in. Virtual environments help isolate python packages to keep them from interfering with each other. Full installation instructions are included in the application's Readme.md file.

One of the most significant python packages this application requires apart from Kivy is OpenCV. OpenCV is primarily written in C++ and needs to be compiled before being used. While OpenCV includes Python bindings, which were used in this project, compiling OpenCV is a process that varies by operating system and platform. For this project we chose to install OpenCV from a python package called OpenCV-Python. It should be noted that while this project utilizes the entire open-source OpenCV code base, it is released by a third party developer. These OpenCV-Python modules contain a precompiled compiled version of OpenCV

which makes the installation process incredibly easy. Python packages that contain compiled code are known as wheels, and multiple versions need to be compiled and released for their respective platforms.

During the development of our software we ran into significant challenges when trying to package our software into an executable. While the application runs without any issues when started from a Terminal/command line window, we ran into camera permission issues under macOS. Since macOS version 10.14, Apple has required that applications explicitly request media permissions from the user. OpenCV determines whether or not this check is required through the use of a build time flag. When OpenCV is being compiled, the authorization request code is only included if the macOS version where compilation is occurring is greater or equal to 10.14. After much troubleshooting it was discovered that OpenCV-Python compiles all macOS wheels using macOS 10.9 in their continuous integration system. Due to this fact, the media capture authorization code is being left out. Interestingly, this does not cause issues when running the application from the Terminal, since the Terminal seems to catch applications that attempt to open camera devices and sends its own media capture authorization request to the operating system.

We opened an issue with the developer who supports the OpenCV-Python, and are waiting for the problem to be resolved. One possible solution would be to compile additional macOS versions under macOS 10.14 or newer. Currently we are unable to package the application to deliver as an executable using existing tools.

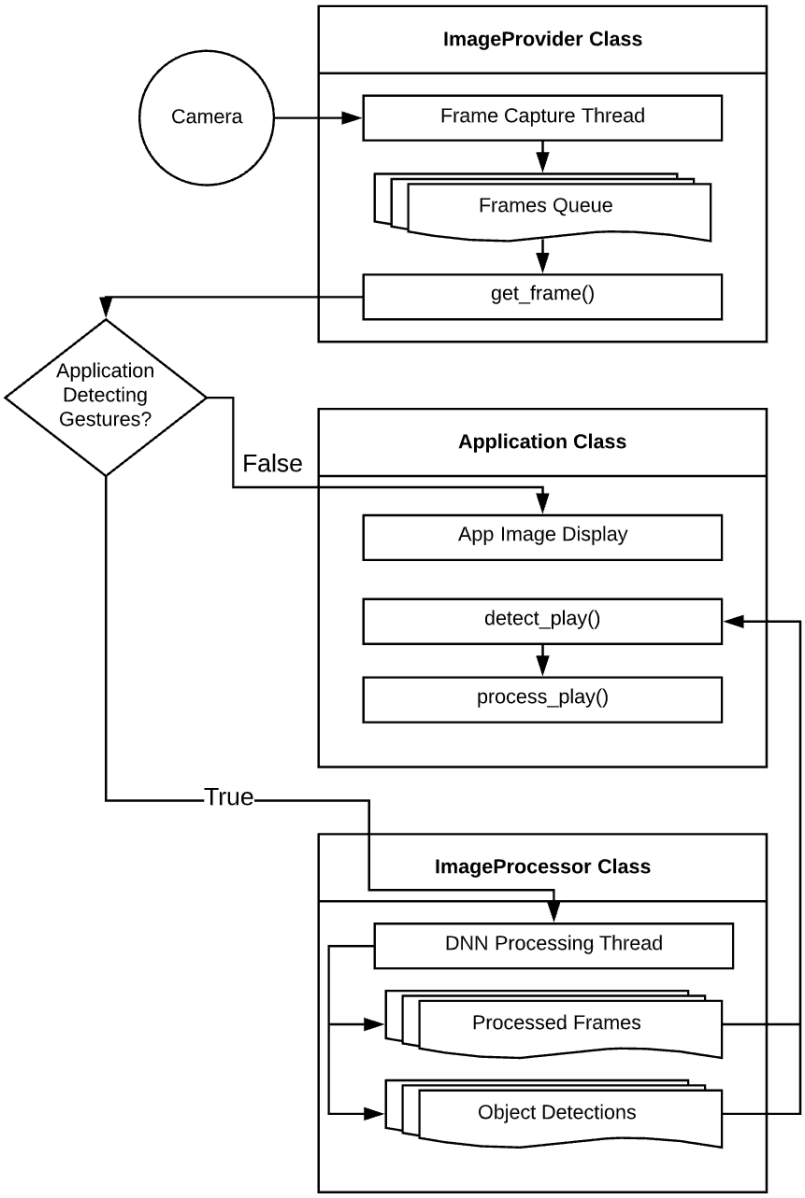

Figure 6.2: Image Pipeline

CHAPTER 7

Results

7.1 Data Augmentation Parameters

When discussing results for this project, we will primarily focus on how our trained YOLOv3 model performs while detecting the three Rock Paper Scissors gestures.

To recap, we initially trained 10 different models with the same training and testing sets, but with different training image augmentation parameters. It is important to note that these models were not trained on our final dataset and had only 2592 images for training and 684 images for testing. All models were trained for 2000 iterations. The models were then all tested against our validation set which contains 465 images never used during the training process. Of the 465 images, 155 were of each gesture class. The results of the model validation can be seen in Table 7.1. For mAP (Mean Average Precision) and precision, the numbers following the @ symbol are the confidence thresholds.

As you can see, adjusting the image augmentation parameters resulted in less than a one percent difference in mAP@0.5 across all the models. We obviously can't compare our results for a model with only three classes against models with 20+ classes, but a mAP of 0.95 is extremely good to begin with.

When looking at Figures 7.1, 7.2, and 7.3, we only noticed a linear relationship

Model Number	mAP@0.50	precision@0.25	Saturation	Exposure	Hue
1	0.96343	0.91	0	0	0
2	0.9553	0.9	0.5	0	0
3	0.953866	0.91	1	0	0
4	0.957415	0.91	2	0	0
5	0.959481	0.93	0	0.5	0
6	0.955541	0.91	0	1	0
7	0.959593	0.92	0	2	0
8	0.962081	0.91	0	0	0.5
9	0.960492	0.91	0	0	1
10	0.956138	0.91	0	0	2

Table 7.1: Hue Saturation Exposure Model Tuning Results

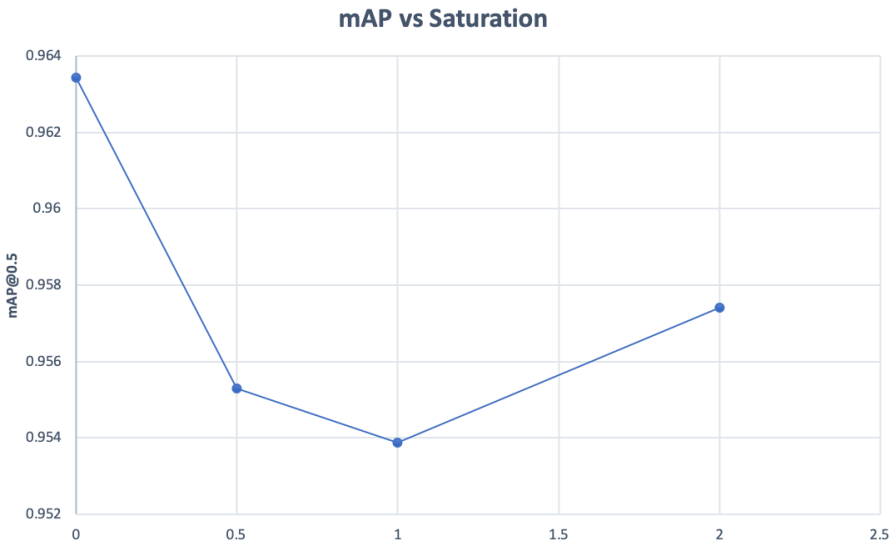

Figure 7.1: mAP vs Saturation

with hue. Because differences in mAP were overall very small, and because saturation and exposure lacked linear relationships, we do not feel comfortable drawing any conclusions about data augmentation parameters besides hue. While differences attributed to changes in the hue parameter are likewise small, mAP does seem to decrease as the hue value is increased. For all of these reasons, we decided to use the original augmentation values provided with the darknet framework going forward. In the original YOLOv3 configuration file, saturation and exposure

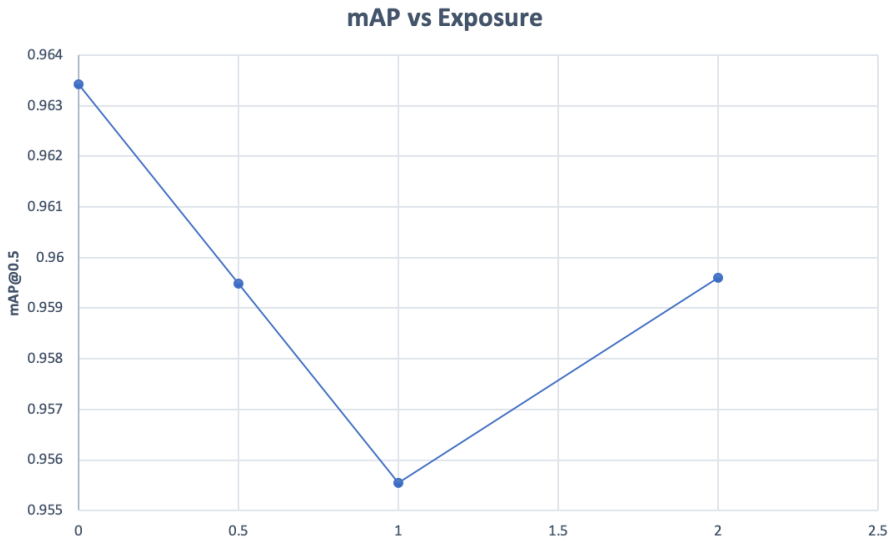

Figure 7.2: mAP vs Exposure

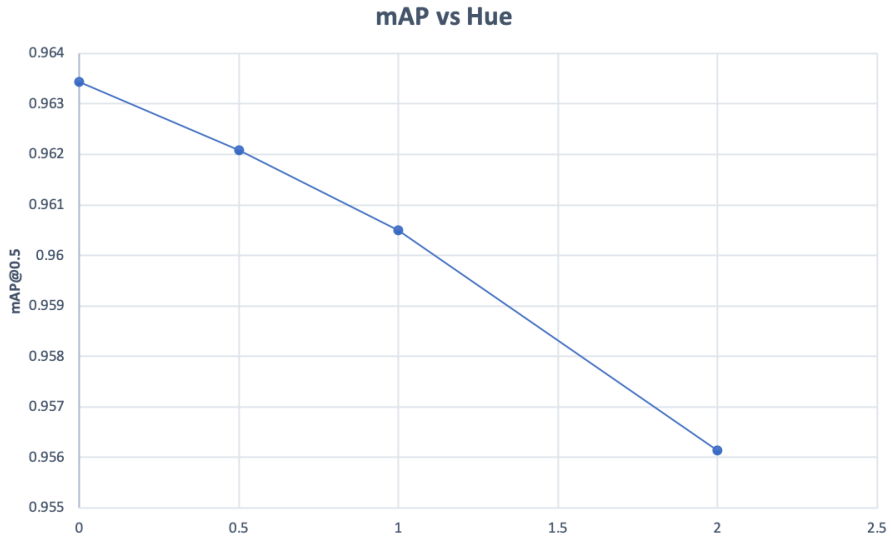

Figure 7.3: mAP vs Hue

are set to 1.5 while hue is set to 0.1 [23]. Having hue at 0.1 is in line with our results however.

7.2 Training Length

For our next experiment we wanted to determine how many iterations our model should be trained for. AlexeyAB's extremely popular darknet fork suggests training for 2000 iterations per object class, but not less than 4000 iterations in total [1]. Using our final dataset (3112 training and 783 testing images) we trained a model for 12,000 iterations. Darknet automatically saves weights every 1000 iterations during training, so we were able to compare validation performance for 12 different network weight files.

Iterations	mAP@0.50	precision@0.25	recall@0.25
1000	0.800347	0.77	0.83
2000	0.96239	0.92	0.95
3000	0.962061	0.90	0.96
4000	0.970745	0.93	0.97
5000	0.949873	0.91	0.96
6000	0.957989	0.93	0.95
7000	0.953966	0.92	0.96
8000	0.954929	0.89	0.95
9000	0.956902	0.89	0.96
10000	0.962979	0.91	0.97
11000	0.959202	0.91	0.97
12000	0.962001	0.91	0.97

Table 7.2: Model Training Length Results

Looking at Table 7.2 it is clear that this model performs best at 4000 iterations. This shows us that training for longer does not always result in increased performance. Overfitting can occur when a model is trained for too long and its performance on unseen data decreases. Overfitting refers to a model being too closely fit to its training set to where it loses generalizability.

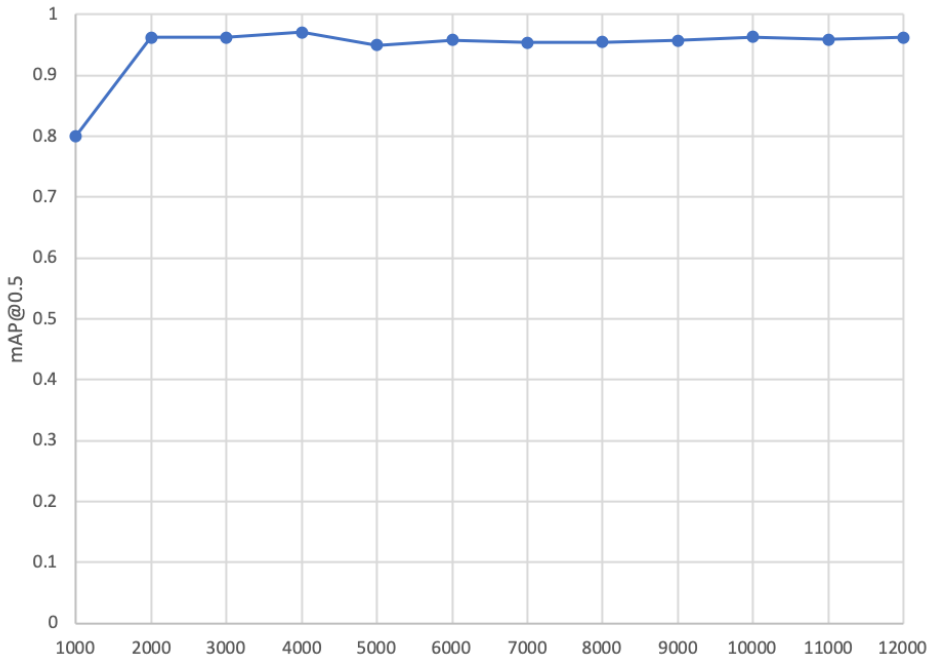

Figure 7.4: mAP vs Number of Training Iterations

7.3 Application Testing

In addition to the two previous tests using the validation dataset, we performed real world testing with the final software and final model. 99 individual games were played, with Rock Paper and Scissors each being thrown 33 times. Out of the 99 throws, 97 were detected correctly for an accuracy of 97.98%. While accuracy is a slightly different measure than precision, this value is in line with our final model validation results. Scissors was the only gesture misclassified in this test. One game Scissors was classified as Rock, and the other time it was classified as Paper.

CHAPTER 8

Conclusion

This project successfully demonstrated that YOLOv3 can accurately classify simple hand gestures like those used in Rock Paper Scissors. We have tested this both on a validation dataset and under real world conditions using our software. Furthermore, the application performs in a wide range of conditions, and fulfills all of our initial goals. Specifically, the application performs without the need for a specialized environment or hardware, and operates with a high degree of accuracy.

With a computer running macOS or Linux and a webcam, you can now play a more natural and unrestricted game of Rock Paper Scissors. While we accomplished our initial goals, there are many ways this project could be expanded upon. First of all, the application still requires the user to press play before a game. This was done to save computational resources and reduce complexity, but having a way to start a game with a gesture would be more intuitive and would provide a better user experience.

Additionally, while we collected and labeled a relatively large amount of data, it was from only about eight different people. To create a more generalizable model, training data from more people would be useful. Having a larger proportion of diverse skin tones in the dataset would also be very beneficial.

Other object detection models could also be examined. YOLOv3 is still extremely popular in 2020, but since its release in 2018, newer object detection models have been released. YOLOv3 is still a rather computationally expensive model, which

makes realtime processing nearly impossible without a CUDA capable NVIDIA GPU. It would be interesting to see if other more efficient models can achieve similar performance metrics on the same dataset. This would make developing a mobile Rock Paper Scissors game easier.

Finally, despite significant effort, we were unable to create a packaged version of the software for distribution. Hopefully this will become possible in the future with changes to the OpenCV-Python package.

Once again, a massive thank you to everyone who helped make this completed project a reality, and to the open source community as a whole. This project relies on countless open source components, so in that spirit our application has been released under the MIT License. Additionally, thank you to the Labelbox team for the free educational license that helped make this project possible.

References

1. AlexeyAB. darknet fork, 12 2019. URL https://github.com/AlexeyAB/darknet. 36, 37, 58

2. Timothy C Arlen. Understanding the map evaluation metric for object detection. Medium.com. xi, 19, 20, 21

3. Steven M. Beitzel, Eric C. Jensen, and Ophir Frieder. *MAP*, pages 1691–1692. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_492. URL https://doi.org/10.1007/978-0-387-39940-9_492. 19

4. Jason Bell. *Machine Learning: Hands-On for Developers and Technical Professionals*. John Wiley and Sons, 2014. 7

5. Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Improving object detection with one line of code. https://arxiv.org/pdf/1704.04503.pdf, August 2017.

6. Jason Brownlee. A gentle introduction to object recognition with deep learning, 2019. URL https://machinelearningmastery.com/object-recognition-with-deep-learning/. 8, 30

7. Rodney P. Carlisle, editor. *Encyclopedia of Play in Today's Society*. SAGE Publications, Inc, 2009. 1, 2

8. Ben Carterette. *Precision and Recall*, pages 2126–2127. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_5050. URL https://doi.org/10.1007/978-0-387-39940-9_5050. 19

9. Justin Collier. How to win at rock paper scissors (using data from 120,000 games), May 2016. URL https://justincollier.com/life-hacks/how-to-win-rock-paper-scissors/. 49

10. Julien de la Bruère-Terreault. rps-cv. https://github.com/DrGFreeman/rps-cv, August 2019. xi, 22, 23

11. Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010. ISSN 0920-5691. doi: 10.1007/s11263-009-0275-4. URL http://dx.doi.org/10.1007/s11263-009-0275-4. 18, 19, 20, 21

12. Tushar Gupta. Deep learning: Feedforward neural network. https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7, Jan 2017. 47

13. Sambasivarao. K. Non-maximum suppression (nms). https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c, Oct 2019. 17

14. Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015. 48

15. Paul Klinger. Rps-rnn. https://github.com/PaulKlinger/rps-rnn, July 2019. 49, 53

16. Labelbox. Labelbox. URL labelbox.com. 31

17. Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Lecture 8: Spatial localization and detection. Stanford, February 2016. xi, 8

18. Umberto Michelucci. *Advanced Applied Deep Learning: Convolutional Neural Networks and Object Detection*. apress, 2019. xi, 9, 10, 11, 12

19. Tom Mitchell. *Machine Learning*. McGraw Hill, 1997. 7

20. Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 2015. xi, 47, 48, 49

21. Devin Pickell. Structured vs unstructured data – what's the difference? https://learn.g2.com/structured-vs-unstructured-data, Nov 2018. 6

22. Michael Plotke. Filter image examples. URL https://en.wikipedia.org/wiki/Kernel_(image_processing). xi, 9

23. Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016. 34, 38, 57

24. Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL http://arxiv.org/abs/1804.02767. xi, 12, 13, 18

25. Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL http://arxiv.org/abs/1506.02640. xi, 12, 13, 14, 15, 16, 18

26. Adrian Rosebrock. Intersection over union (iou) for object detection. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/, November 2016. xi, 14, 15

27. Katharine Schwab. A cultural history of rock-paper-scissors, December 2015. URL https://www.theatlantic.com/entertainment/archive/2015/12/how-rock-paper-scissors-went-viral/418455/. 1, 3

28. Shubham Shinde, Ashwin Kothari, and Vikram Gupta. Yolo based human action recognition and localization. *Procedia Computer Science*, 133:831 – 838, 2018. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2018.07.112. URL http://www.sciencedirect.com/science/article/pii/S1877050918310652. International Conference on Robotics and Smart Manufacturing (RoSMa2018). xi, 24, 27, 28

29. Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos, 2014. xi, 25

30. Dale Stokdyk. What is computer programming and how to become a computer programmer, June 2018. URL https://www.snhu.edu/about-us/newsroom/2018/06/what-is-computer-programming. 6

31. Lisa Torrey and Jude Shavlik. Transfer learning, 2009. 17

32. Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Jun 2014. doi: 10.1109/cvpr.2014.214. URL http://dx.doi.org/10.1109/CVPR.2014.214. xi, 26

33. Sik-Ho Tsang. Review: Deepmask (instance segmentation). https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339, December 2018. 8

34. Zhijian Wang, Bin Xu, and Hai-Jun Zhou. Social cycling and conditional responses in the rock-paper-scissors game. Nature Scientific Reports, 2014. 45, 46, 47