Automated Vertical Jump Height Estimation

Richie Harris

*Abstract*—A system has been developed for automatically estimating vertical jump height from a video recording. It uses the TensorFlow Object Detection API to extract time series data from the video, maximum likelihood estimation to find the takeoff and landing frames of the jumper, and kinematics to calculate the height of the jump. Four algorithms for finding the maximum likelihood estimate were implemented and compared. The system has achieved an average unsigned error of 3.7% from the true jump height for a set of 32 videos.

# Introduction

Vertical jump height is a common athletic performance metric [1], but obtaining accurate measurements can be challenging. One common approach is to use a *Vertec Vertical Jump Tester* [2], but this equipment can cost hundreds of dollars [3]. There are also smartphone applications available that measure vertical jump height using video, but they are not automated [4]. The user must manually scroll through each frame in the video to identify the takeoff and landing frame.

To obtain an accurate vertical jump height estimate without human input, a new method was developed that uses image processing and statistical modeling to estimate vertical jump height from a video recording.

# System Design

## Overview

Figure 1 shows the block diagram for the system.

A screenshot of a cell phone

Description automatically generated

*Figure 1 System block diagram*

Each frame of the video clip is processed by the TensorFlow Object Detection API. This returns the vertical position of the jumper’s feet for each video frame. Next, maximum likelihood estimation is performed on the feet position data to extract the frame that the jumper first leaves the ground (takeoff frame) and the frame that the jumper makes contact with the ground after being airborne (landing frame). This information is used in a kinematics equation to estimate the vertical jump height.

## Video Processing

The TensorFlow Object Detection API was used to extract the vertical position of the jumper’s feet in each video frame. The TensorFlow Object Detection API is an open source framework built on TensorFlow that makes it easy to deploy object detection models [5]. It uses convolutional neural networks to perform object detection and recognition [5]. It provides several models that have been pre-trained on the COCO dataset. The *ssd\_resnet\_50\_fpn\_coco* model was used for this project due to its relatively high accuracy score [5]. The API can detect many different objects, including humans, and indicates a detection by creating a bounding box around the object, shown in figure 2.

A person standing posing for the camera

Description automatically generated

*Figure 2 Example human detection for a single frame using the TensorFlow Object Detection API*

Only the vertical position of the bottom of the box, corresponding to the jumper’s feet, is used by the system. The position is measured in pixels from the top of the video frame. Figure 3 shows an example time series graph for the position of a jumper’s feet.

A screenshot of a social media post

Description automatically generated

*Figure 3 Example time series extraction of the jumper’s feet position for the entire video*

These positions were stored in an array for the next step in the system.

## Maximum Likelihood Estimation

Maximum likelihood estimation (MLE) is a method of estimating the parameters of a statistical model given a set of observations by finding the parameters that maximize the likelihood of the observations given those parameters [6]. For this project, the model was the piecewise function shown in equation 1.

(1)

In equation 1, is the vertical position of the jumper’s feet in pixels and is the frame number from the beginning of the video. is the mean position before takeoff, is the mean position after landing, is the vertex of the parabola, is the stretch factor of the parabola, is the takeoff frame, and is the landing frame.

This model represents the expected behavior of the jumper. Before the jump, the position of the feet should be constant. During the jump, there is constant acceleration, so the position of the jumper’s feet is a second order polynomial. After the jump, the position should again be constant.

Note that and will not necessarily be the same because if the person jumps forward, from the perspective of the camera it will appear that the vertical position of their feet is different in the frame. This can be seen to some degree in figure 3, although in some cases the difference is larger.

Equation 1 can be reduced to four model parameters because and can be found by taking the mean of all observed points before and , respectively, and the parabola can be uniquely described by the three points: (, ), , and (, ). The four model parameters are shown in equation 2.

(2)

Four different algorithms were implemented for finding the maximum likelihood estimate as part of an iterative development approach. The algorithms aim to find the model parameters that minimize the sum of the squared error between the observed data, , and the model estimate, , at each frame in the video, as shown in equation 3.

(3)

Each algorithm is described in more detail in the sections below.

## Brute Force Algorithm

The most straightforward solution is a brute force approach, whereby every possible set of parameters is tested, and the combination that yields the minimum error is returned. Through experimentation, the runtime for optimizing for all four parameters was extremely long and therefore deemed an unreasonable solution. To make this approach feasible, it was assumed that was equal to the minimum point in the observed data. This left and left to solve for and led to more reasonable runtimes. This assumption proved to be reasonable because the accuracy results were still high. However, even with the assumption, the runtimes were still relatively long.

## Dynamic Programming Algorithm

To reduce the runtime, a dynamic programming algorithm was implemented. Dynamic programming is a technique that solves an optimization problem by breaking it down into simpler subproblems and utilizes the fact that the optimal solution to the overall problem depends on the optimal solutions to the subproblems [7].

To implement this approach, the algorithm first optimizes for , and then optimizes for . This algorithm also used the assumption that was equal to the minimum point in the observed data.

While optimizing for , the algorithm assumes is such that and is the same, creating a symmetric parabola. Once is found, it optimizes for using the solved value of .

The accuracy of this algorithm was slightly worse than brute force, indicating that it did not find the optimal and in all cases. This is because this MLE problem does not fully meet the requirements needed for a dynamic programming algorithm to find the optimal solution. The subproblems of finding and were not fully independent because the shape of the parabola depends on both and .

## Gradient Descent Algorithm

To find a faster algorithm than brute force that still finds the optimal solution, a gradient descent algorithm was implemented. Gradient descent minimizes a function by iteratively moving in the direction of the negative gradient [8]. For this project, the function that is being minimized is equation 3. In this version of the algorithm, it was also assumed that was equal to the minimum point in the observed data.

To find the direction in which to move next, the errors of the surrounding points in the solution space were calculated, and the direction with the lowest error was chosen. The next iteration moved one step in this direction, as shown in figure 4.

A close up of a screen

Description automatically generated

*Figure 4 The gradient descent approach*

In figure 4, the red boxes indicate the current position of and , the blue boxes indicate locations where the error was calculated, the grey arrows represent possible directions to move, and the black arrow represents the direction that was taken.

## Removing Assumptions

After observing the accuracy and runtime results for the three algorithms just described, the gradient descent algorithm was determined to be the most promising. To expand on the previous implementation, the assumption that was equal to the minimum point in the observed data was removed.

The new algorithm starts by calling the previous gradient descent algorithm to initialize a starting point for and that is close to the expected optimal point. Then, this initial estimate is refined. Like before, errors for the surrounding points in the , solution space are calculated. The difference is that now a second gradient descent optimization is performed to find the optimal position for the given point in the , space. The process is shown in figure 5.

A picture containing clock

Description automatically generated

*Figure 5 Gradient descent without assuming (h,k) is the minimum point in the observed data*

With the optimal for each point, a new minimum can be reached. This algorithm incorporates insights from the dynamic programming approach to remove the assumption and improve upon the previous gradient descent approach. Specifically, instead of performing a higher dimensional gradient descent over , , , and , the optimizations for and are separated from the optimizations for and .

## Kinematics

After and are found, they are converted time using the video frame rate. The jump height is then calculated using equation 4, which is derived from kinematics.

(4)

# Peformance

A dataset of videos was collected to test the system. The chosen performance metrics were the unsigned percent error between the true jump height and the system’s estimate, and the runtime in seconds. Percent error was averaged over the entire dataset, and the runtime was the total time to make an estimate for every video in the dataset.

## Dataset

The dataset consisted of 32 videos recorded at 60 frames/sec with a smartphone. The jumper was the same in each video, and the height of the jump was varied. The phone was held still during the recording to achieve the best results.

The takeoff and landing frames for each video were manually identified by watching the video frame by frame. These frames were converted to time, and equation 4 was used to obtain the ground truth jump heights for each video in the dataset. These were compared to the system’s estimates.

## Experimental Results

The average percent error for the entire dataset and the total runtime for the entire dataset for each algorithm are shown in figure 6.

|  |  |  |
| --- | --- | --- |
| Algorithm | Average % Error | Total Runtime |
| Brute Force | 4.984 | 21.672 |
| Dynamic Programming | 6.047 | 0.805 |
| Gradient Descent | 4.984 | 0.622 |
| Gradient Descent without assumption | 3.700 | 2.491 |

*Figure 6 Average percent error and total runtime results for the entire dataset*

For reference, and example MLE using gradient descent without the assumption is shown in figure 7.

A close up of a map

Description automatically generated

*Figure 7 An example MLE using gradient descent without the assumption*

# Discussion

## Analysis of Results

The results show continuing improvement as the algorithm was iterated upon.

The brute force method had significantly higher runtime than the other implementations, which was expected because it performed an exhaustive search of the solution space while the other methods used more sophisticated ways of converging.

The dynamic programming method had significantly better runtime. This makes sense because when searching the two dimensional solution space for and , brute force searched everywhere while dynamic programming did not. If there are possible locations for and , brute force will have a runtime proportional to . Since the dynamic programming approach optimizes for first, and then , it will have a runtime proportional to .

However, the error was worse than the brute force method because the MLE problem does not fully meet the requirements needed for a dynamic programming algorithm to find the optimal solution. The subproblems of finding and were not fully independent because the shape of the parabola depends on both and .

The gradient descent method and brute force method had the same average percent error, indicating that they found the same, optimal solution for each video. This means that the error function shown in equation 3 was likely concave because no local minima were encountered. The runtime for gradient descent was better than for dynamic programming, indicating that even fewer search points were tested. This makes sense because the gradient descent method continually moves closer to the optimal point, while the dynamic programming method does not.

Without the assumption that the vertex of the parabola was the minimum of the observed data, the algorithm achieved better accuracy. This indicates that points with lower error could be found if all parameters could be optimized for. However, the average percent error was only slightly better, so the assumption that was made for the first three algorithms was reasonable.

With an error of less than 5%, the system will be an average of slightly less than one inch away from the true height for jumps of 20 inches. For comparison, the *Vertec Vertical Jump Tester* measures vertical jumps to within 0.5 inches [3].

## Limitations

The main limiting factor for the accuracy of the system is the frame rate of the video. With 60 frames/sec, 40 frames airborne corresponds to a jump height of 21.45 inches, while 41 frames airborne corresponds to a jump height of 22.54 inches. Thus, the frame rate quantizes the possible jump height estimates that the system makes.

The statement at the end of section A must be qualified because it assumes that the ground truth jump heights are close to the true jump heights. However, there is likely some difference.

## Future Work

Collecting a dataset with a higher frame rate could significantly improve results by increasing the resolution of jump height estimates. High-end smartphones can record slow motion video with 240 frames/sec [9]. With 240 frames/sec, 160 frames airborne corresponds to a jump height of 21.45 inches, while 161 frames airborne corresponds to 21.72 inches. If a higher frame rate is not available, an interpolation scheme could be applied to obtain better resolution as well.

When collecting new data, it would be desirable to use a more rigorous method of obtaining the ground truth jump heights. For example, ground truth jump heights could be obtained using a *Vertec Vertical Jump Tester* while the videos are being recorded.

##### Acknowledgments

Thanks to Professor Tsai of Harvey Mudd College for guidance over the course of the project.

##### References

1. "NFL Combine Workouts." *NFL Draft 2020*, www.nfl.com/combine/workouts.
2. "Vertical Jump Test." *Breakthrough Basketball*, www.breakthroughbasketball.com/fitness/vertical-jump-test.html.
3. "Vertec." *Power Systems*, www.power-systems.com/shop/product/vertec.
4. "What's My Vertical." *The Hoops Geek*, [www.thehoopsgeek.com/measure-vertical-jump-iphone/](http://www.thehoopsgeek.com/measure-vertical-jump-iphone/).
5. "Speed/accuracy trade-offs for modern convolutional object detectors." Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K, CVPR 2017
6. Balaban, Jonathan. "A gentle introduction to maximum likelihood estimation." *towards data science*, towardsdatascience.com/a-gentle-introduction-to-maximum-likelihood-estimation-9fbff27ea12f.
7. "What is Dynamic Programming?" *educative*, [www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2G1pAq0OO0](http://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2G1pAq0OO0).
8. "Gradient Descent." *ML Glossary*, ml-cheatsheet.readthedocs.io/en/latest/gradient\_descent.html.
9. "How to Shoot Stellar Slow Motion Video on your Phone ." *Wired*, www.wired.com/story/how-to-shoot-slo-mo-video/.