Quantitative Behavior Tracking of *Xenopus laevis* Tadpoles for Neurobiology Research

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by

Alexander Hansen Hamme

Annandale-on-Hudson, New York
December, 2018

Abstract

Xenopus laevis tadpoles are a useful animal model for neurobiology research because they provide a means to study the development of the brain in a species that is both physiologically well-understood and logistically easy to maintain in the laboratory. For behavioral studies, however, their individual and social swimming patterns represent a largely untapped trove of data, due to the lack of a computational tool that can accurately track multiple tadpoles at once in video feeds. This paper presents a system that was developed to accomplish this task, which can reliably track up to six tadpoles in a controlled environment, thereby enabling new research studies that were previously not feasible.

Acknowledgements

I would like to thank my advisers; Sven Anderson, Arseny Khakhalin, and Keith O'Hara, for the mentorship and guidance they have given me throughout the course of this project's development. I would also like to thank my friends for helping make my time here at Bard a wonderful experience, and my family for always being there when I needed them.

Contents

1 Introduction

1.1 Animal Models in Neurobiology

Animal models are an essential part of neurobiology research because certain systems in their brains can be analogous to the human brain. This provides an opportunity for researchers to study the structure and development of the brain, as well as certain neurological disorders that affect humans, without needing to perform experiments on people. While rats and mice are the most widely used animal models in neuroscience research (Ellenbroek and Youn 2016), the amphibian *Xenopus laevis* species, commonly known as the African clawed frog, has also proven to be useful as an alternative animal model (Lee-Liu et al. 2017, Pratt and Khakhalin 2013, McQuillan 2017).

1.2 *Xenopus laevis* Tadpoles

While the adult frogs of *Xenopus laevis* have been used for biomedical research (Lee-Liu et al. 2017), the tadpoles are more typically used for neuroscience research, due to the potential to study aspects of their brains at different stages of development (Pratt and Khakhalin 2013, McQuillan 2017). One of the behaviors exhibited by *Xenopus* tadpoles that is of special interest to researchers is the escape reflex that occurs when they are startled. This behavior, also known as the "c-start" (shown in Fig. 1), can be used as an indication of whether the tadpole's brain is functioning properly, because if it is determined that healthy tadpoles will generally startle in response to an artificial stimuli, then this stimuli can be used in experiments to determine whether certain neurochemical drugs have affected their ability to perform startle behavior (Sillar and Robertson 2009). This in turn is a direct indication of whether their cognitive systems are functioning properly, which provides the opportunity to pinpoint which drugs might affect certain systems in the tadpole's brain. In addition to experimenting with neurochemicals, studying the startle response can also provide insight into the development and working structure of the tadpole brain, because the behavior is not exhibited until the tadpoles reach a certain stage of development. Therefore,

an exhaustive comparison between the brain structures of tadpoles before and after the point in time when the startle reflex is developed might reveal the exact brain region and structure that is responsible for this behavior.

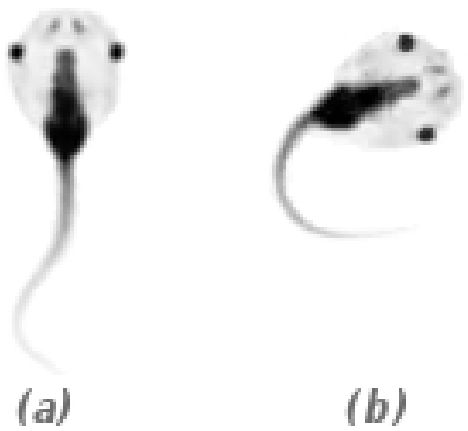


Figure 1: Xenopus laevis tadpole. (a) normal swimming behavior, (b) "C-start" response

In addition to C-start responses, Xenopus tadpoles exhibit a variety of other swimming behaviors, including path planning, collision detection and avoidance, and possibly even schooling behavior. Quantitatively analyzing all of these behaviors and how certain chemicals might affect them could give valuable insights into their cognitive systems (Pratt and Khakhalin 2013), however to date there has been no readily available computational tool that can reliably record the movement patterns of *Xenopus laevis* tadpoles.

1.3 Statement

In this study, a computational application was developed to approach this problem. The completed system uses a trained convolutional neural network to detect tadpoles in video frames, and then maps these detections to distinct tadpole trajectories using identity retainment and trajectory prediction algorithms. These concepts will be introduced in the Background section, and the specific implementations used in this system will be discussed in the Methods section. The final program successfully tracks multiple Xenopus laevis tadpoles at once and records their movement data, enabling new research experiments that were not feasible before.

2 Background

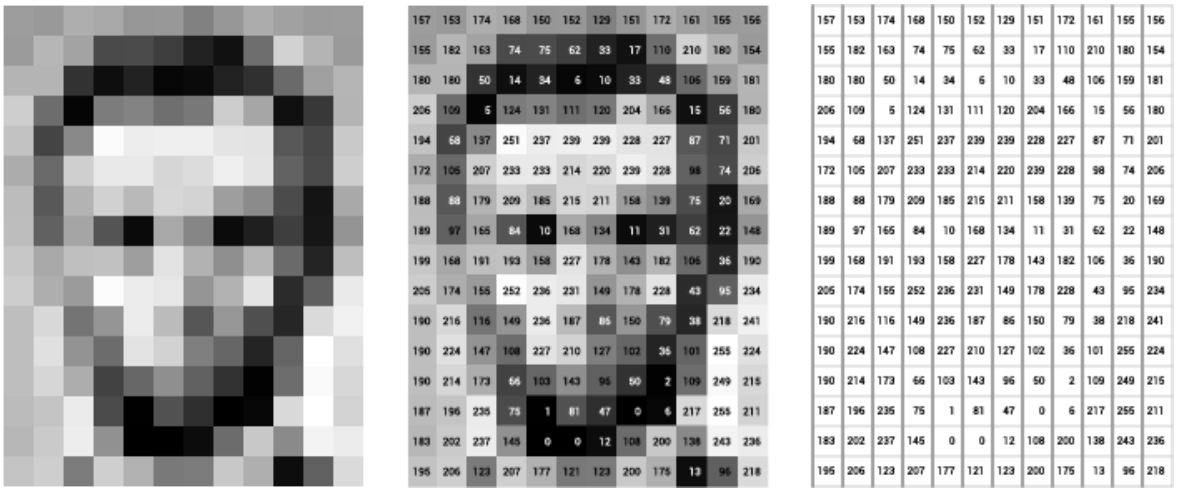
This section introduces key concepts, algorithms, and areas of computer science that are essential for understanding the components of this tracking system. Specifically, the fields of machine learning and computer vision, as well as the concepts of object detection, object tracking, individual identity retainment, and trajectory prediction are discussed.

2.1 Computer Vision

The field of computer vision is an area of computer science that strives to give computers a visual understanding of the world like that of humans. For the average person, the act of visually identifying objects of interest in the surrounding environment comes with ease, but transferring these abilities to machines is a surprisingly difficult task.

Images are represented using pixels. In color images, each pixel is an array of three numbers between 0 and 255, which correspond to the amount of Red, Green, and Blue combined to create a single additive color value. For gray-scale images, each pixel is only one value, between 0 (black) and 255 (white). To the computer, therefore, an entire image is just a multidimensional array of numbers. This concept can be visualized with a low resolution gray-scale image, as shown in Fig. 2.

Consequently, using computers to analyze images actually entails developing algorithms to analyze *integers*, with the goal of finding meaningful patterns or connections within the images of interest. This is often a difficult task, because mapping meaningful relationships between visual patterns and arrays of pixel numbers can be extremely complex.



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Figure 2: Pixel breakdown of a small gray-scale image of Abraham Lincoln (Levin n.d.) with dimensions 12x16. This means there are $12 * 16 = 192$ pixels that make up the image, with each having a value between 0 (black) and 255 (white). Computers "see" images as the series of numbers on the right.

Consider the following hypothetical scenario: there exists a computer that can understand spoken instructions, and a researcher wants to teach it how to determine whether there are people in each of a large album of pictures. The first two images which the researcher must use to begin teaching it to recognize people are shown in Fig. 3.

Figure 3: Two sample images to be passed to the computer. Instructions must be given to teach it what distinguishes people from other objects in the images.

The researcher tries to develop a set of rules to teach the computer to distinguish people from other objects. Color is not a useful attribute, because hair, skin, and clothing can be many different colors, and the images are gray-scale. Size is also irrelevant, because the size of a person within an image changes based on their distance to the camera. For instance, the people in the first image are much smaller than those in the second. A seemingly logical pattern to search for would be the shape of a human being: a circle for the head, above a broader rectangular shape, the torso. As it turns out, this heuristic does work some of the time, but unfortunately it is not accurate enough to be reliable. For example, in the images shown in Fig. 3, searching for the circle-above-rectangle shape leads to the false detections shown in Fig. 4.

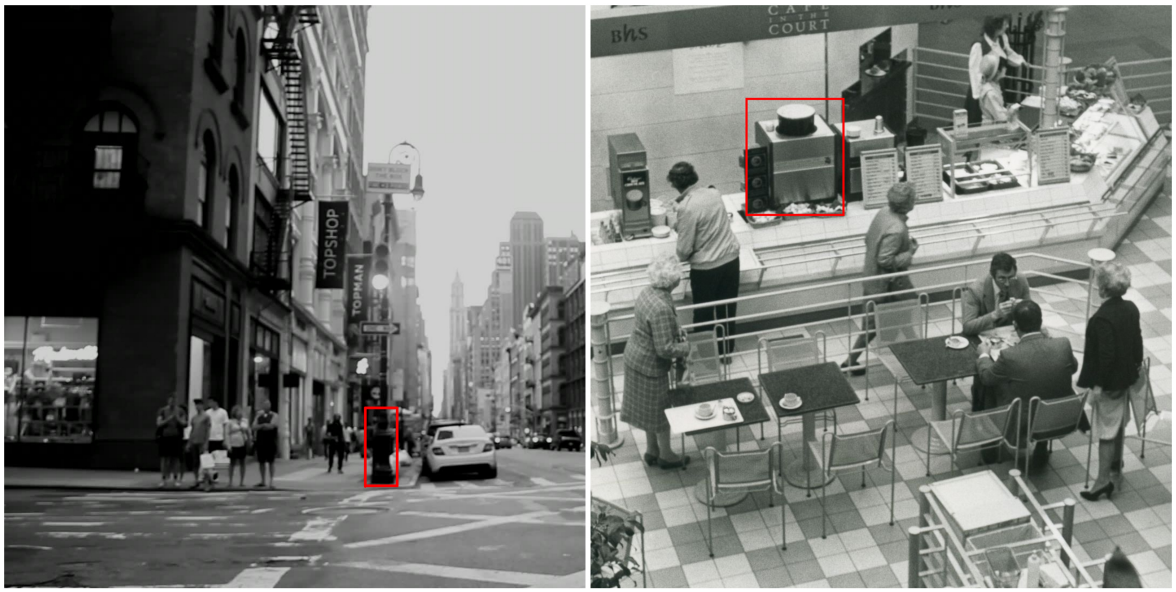


Figure 4: False people detections that would be made in the above images using a simple circle-above-rectangle shape detection algorithm.

Ultimately, a researcher in a situation like this would have a very difficult time trying to provide a complete set of rules that distinguish people from the myriad of other everyday objects that can appear in images as well. This is why perhaps the biggest challenge in computer vision is designing algorithms that allow computers to recognize the same kinds of patterns in images that our brains naturally find in our surrounding environments. Areas like this are where the field of machine learning has made huge improvements.

2.2 Machine Learning

Machine learning is a subset of the widely-encompassing field of artificial intelligence. Machine learning strives to give computers the ability to learn and improve their ability to solve specific problems without being given explicit instructions by human programmers. The learning process, usually called "training", requires the input of large quantities of real-world data to the computer, which it then analyzes algorithmically to find patterns and improve its ability to make new predictions. When training has finished, the computer produces a trained "model", which can be used to make future predictions on data outside the training set. In the case of images, a computer is given hundreds, thousands, or even millions of images that are labeled with the correct names of objects within them, and the machine learning algorithms find hidden patterns within the image data that allow its final model to recognize and classify these objects correctly.

An important axiom of machine learning is that the quality of the data provided for training directly affects the computer model's accuracy and ability to make predictions. Training the model on data that is lacking or unrepresentative of the real-world situation will generally result in it exhibiting poor performance.

2.2.1 Deep Learning

Deep learning is a subset of machine learning that can be applied to far more complex problems than simple machine learning algorithms can handle. It involves the use of neural networks, computational structures that are loosely inspired by the biology and structure of the human brain. These artificial neural networks are made up of layers of connected units, often called neurons, as shown in Fig. 5.

In the brain, neurons receive input signals from their dendrites and send output signals along their axon, which branches out and connects to the dendrites of other neurons with synapses. Each neuron has a certain potential to "fire", or become activated, depending on the strength of incoming signals from other neurons coming in along its dendrites. If the combined input signals are above the neuron's activation threshold, it will "spike" (another word for fire), sending a new signal along its axon to all the other neurons it has outgoing connections

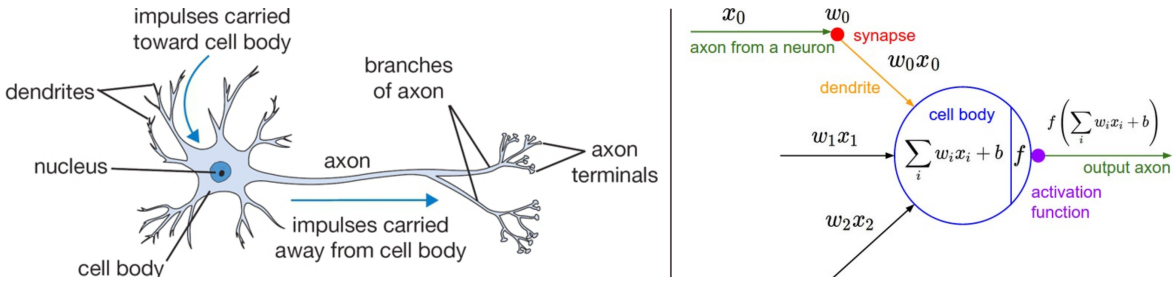


Figure 5: Comparison between a biological neuron structure (left) and its mathematical counterpart used in neural networks (right). Diagram from Stanford's CS231n Course slides (Stanford University 2018).

to. The strength of synapses do not stay static, but instead change frequently, growing stronger or weaker especially when learning is taking place. These changes in the strength of a synapse between two neurons are primarily determined by the frequency with which it is fired, i.e. the frequency that the output neuron activates its connection to the receiving neuron. As a result, over time, important connections between neurons are reinforced, and unused connections become weaker or are even removed completely.

In the computational model of the neuron, the neuron has similar input and output connections to other neurons. Each input signal, e.g. x_0, from another neuron carries a signal to the synapse. This signal strength is then multiplied by the strength (or "weight") of that particular synapse, which in this case is w_0. The resulting signal strength that the neuron actually receives is $w_0 * x_0$. Similarly to the biological neuron, the synaptic strengths between neurons change with activity, and thus control the the strength of each neuron's influence on another. The weighted signals from all incoming neuron are summed together with a constant b, using the main cell body function $\sum_i w_i x_i + b$, and if this sum is above the neuron's threshold, it will fire. The activation function f is used to model the neuron's firing rate, which corresponds to the frequency of spikes along its axon (Stanford University 2018).

Combining Neuron Units to Create Networks

In the creation of a neural network, layers of artificial neurons are stacked together to create layers, and connections are created between adjacent layers of neurons, which act as the dendritic and axonal connections shown in Fig. 5. During training, the network is presented

with input data, on which it slowly learns how to give correct outputs using a process called backpropagation (Stanford University 2018). Backpropagation entails comparing the output the network produced on the given data with the output it was *supposed* to produce, and then using the amount of error (typically called "loss") to modify the weights of the connections between the units in the network (i.e. the synaptic strengths between neurons). This process is run starting from the final layers and then backwards through the hidden layers, which is why it is called **back**propagation. Over time, backpropagation causes the network to reduce the error between its actual output and the correct output, resulting in the network "learning" the task it was provided.

The structure of the network is important for determining the complexity of the problems it can solve. Small networks like the one in Fig. 6 can learn to approximate complex mathematical functions, and in fact it has been proven that a neural network with one hidden layer can approximate any continuous function $f(x)$ (Stanford University 2018). However, a network like this is not sufficient for more solving more complex problems, such as analyzing image data, because it simply does not have enough computational power to process all that data and store anything meaningful in its layers. In general, *deeper* networks (networks with more hidden layers) have the capacity to learn more complex problems than shallow ones (Stanford University 2018).

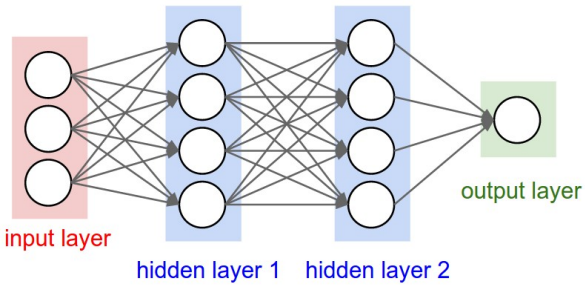


Figure 6: A three-layer neural network comprised of three inputs, two hidden layers, and one output layer. These layers are referred to as being fully-connected, because each unit shares connections with every unit in the next layer. Importantly, units only share connections across layers, not within their own layers. Diagram from (Stanford University 2018).

2.2.2 Convolutional Neural Networks

While deeper networks are generally able to learn more complex problems, fully-connected neural networks such as the one in Fig. 6 do not scale well when applied to full images. For a tiny color image of dimensions 32x32, each unit in the first fully-connected hidden layer would have $32 * 32 * 3 = 3,072$ input connections. If the image size is increased to 300x300, which is still a relatively small image, each individual unit now has 270,000 input weights, which would quickly become an unmanageable number of connections in a deeper network.

This is where the Convolutional Neural Network (CNN) excels. CNN's are a special type of neural network specifically designed to handle image data. They have seen great success in computer vision over the last decade because their structure works so well for analyzing images. A CNN is given hundreds, thousands, or even millions of images that are labeled with the correct names of objects within them, and the network then finds patterns within the image data that allow it to recognize these objects correctly. A modified convolutional neural network is at the center of this tadpole tracking system.

Unlike regular neural networks, CNN's have neurons arranged in three dimensions: width, height, and depth (the number of color channels in an image), as shown in Fig. 7.

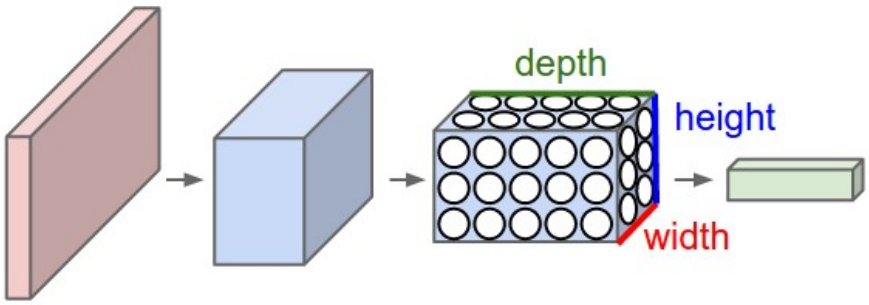


Figure 7: Depiction of 3-D arrangement of layers of neurons in a convolutional network. Every layer transforms the 3D input volume into a 3D output volume of neuron activations. Diagram from (Stanford University 2018).

This architecture is far more efficient at processing image data, as the number of connections between neurons from layer to layer is greatly reduced. In a given hidden layer, the neurons will only have connections to a small region of the previous layer, instead of being connected to all of them as in a fully-connected layer. Fig. 8 shows how the stacking of layers results in increasingly higher-level pattern detection that builds upon previous layer's activations.

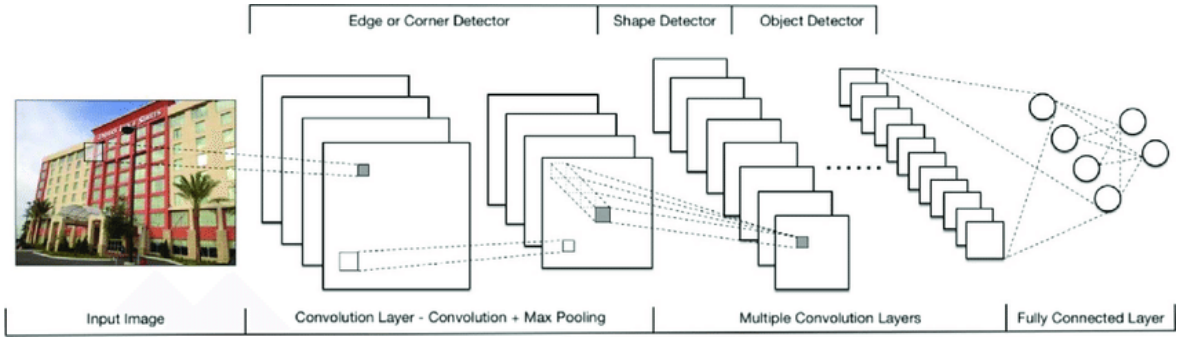


Figure 8: Depiction of how layers of a convolutional neural network build upon the activations of previous levels to form higher-level pattern detectors. Diagram from (Ma et al. 2018).

The layers closest to the input layer often learn to perform basic edge and corner detections, using repeated small feature maps across shared weights. As multiple activations to basic features in the image pass deeper into the network, higher level features in the image are extracted, such as shapes, textures, and then finally full objects. The final layer maps neuron activations to class scores, which is what the network uses to either correct itself during training, or make predictions on new input images. Fig. 9 shows visualizations of the different level feature detectors in a convolutional network trained to recognize faces.

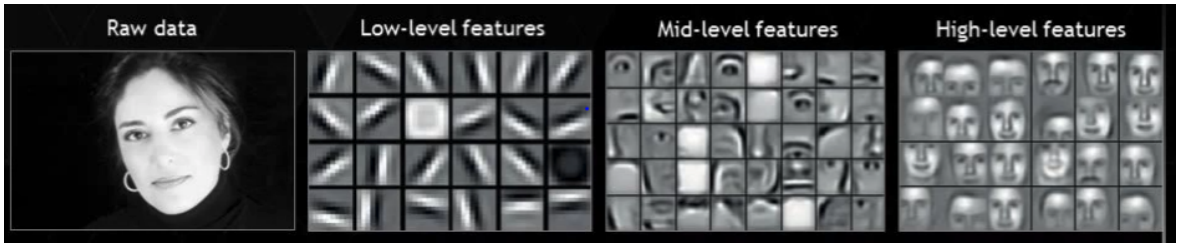


Figure 9: Visualization of feature detectors at different layers in a convolutional network trained to detect faces. The layers closer to the input image react to simple features, such as edges and varying degrees of brightness, the layers in the middle detect familiar patterns such as eyes, noses, and lips, and the final layers detect patterns that match entire faces. Diagram from (Maini 2017).

2.3 Object Detection

Object detection is an area within computer vision that aims to both recognize and localize objects of interest within an image. This is a more difficult task than image classification, which is when the network is only expected to predict a single category for the entire image. Instead, an object detection network must output both a category label and the localized position for every object it detects within the input image, as shown in Fig. 10. Due to the complexity of this problem, it is still an active area of research and improvement.

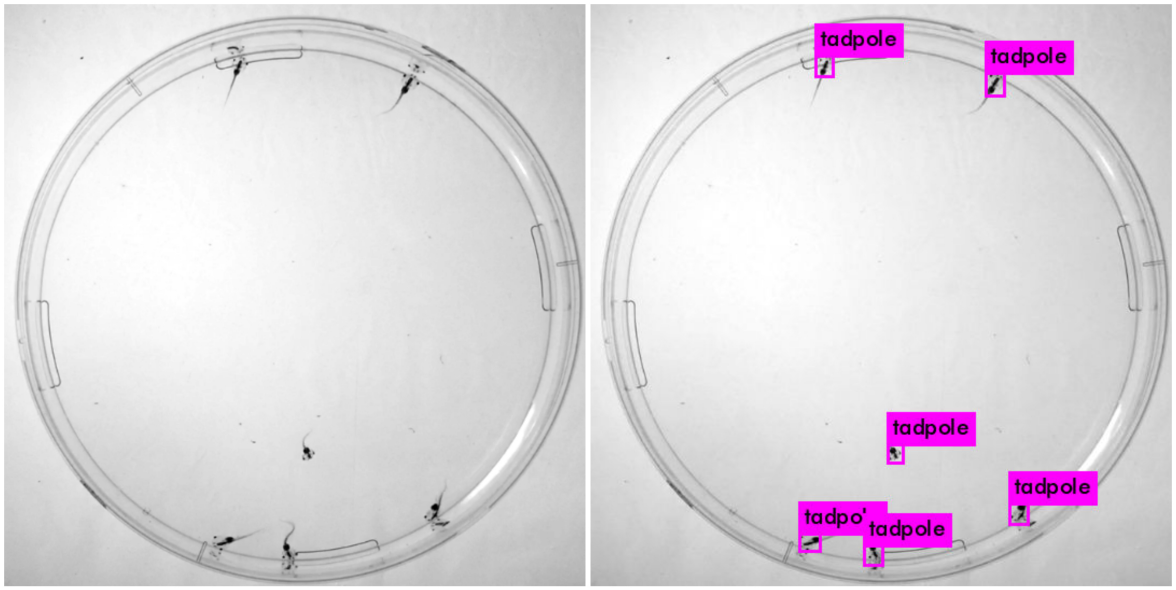


Figure 10: Left: input image to a tadpole detection network, right: desired output with labeled bounding boxes. Image on right generated by the trained model used in this tadpole tracking system.

2.4 Object Tracking

Object tracking is a system built upon object detection. While object detection is statically applied to a single image, object tracking seeks to connect the detected objects produced by the detection model across multiple video frames, in order to store a record of their movement over the course of a video. Algorithms must be designed that can analyze each new set of detected objects and determine how they connect to the detected objects in previous video frames. Fig. 11 demonstrates the desired output of a tadpole tracking system.

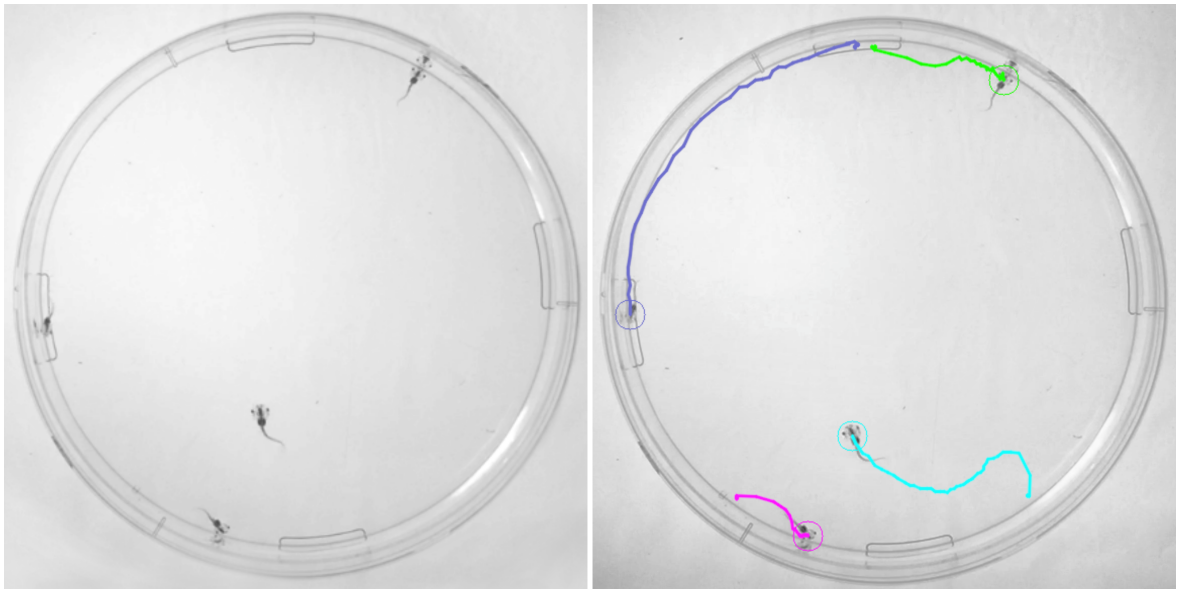


Figure 11: Left: input image to a tadpole tracking network, right: desired output with individual trajectories tracked over time. Image on right generated by tadpole tracking system.

Moreover, the tracking system must have algorithms in place for when the detection model fails to detect all the animals in a given video frame. In situations like these, it is imperative that errors are corrected as swiftly as possible. In this tracking system, such correction and error handling is implemented using individual identity retainment and trajectory prediction.

2.4.1 Individual Identity Retainment

When tracking multiple objects at once, it becomes important to retain individual identities of objects, in order to keep their trajectories distinct, especially in the case when the detection model fails to detect all the tadpoles present in the video. Even if the system loses track of some of the tracked animals, it must try to continue accurately tracking the remaining tadpoles that are still detected. This task of individual identity retainment is a difficult one, and has been approached with many different algorithms and techniques.

The solution used in this tracking system approaches the task of identity retainment as a graph problem. What this signifies is that on each incoming video frame, the set of bounding boxes produced by the detection model is treated as a set of nodes in a graph, and the solution to identity assignment is found by minimizing the total cost of all assignments between object detections and tracked entities. Cost is determined primarily using the distance between objects. The specific implementation used in this system is discussed in the Methods section.

2.4.2 Trajectory Prediction

When the tracking system must handle a missing bounding box, it is often useful to predict the trajectory of the object in question, so that when it is detected again, the tracker object is able to easily reconnect to it again. In this system, this is accomplished by continuously keeping track of both the position and velocity of each tracker object, and when a detection is missed, the object's trajectory is predicted using both its predicted velocity and next predicted location. The specific implementation used is detailed in the Methods Section.

3 Methods

This section discusses the algorithms and frameworks used in the design of this tadpole tracking system. Specifically, it describes the implementation of: the Yolo neural network model using the DeepLearning4J Java library, the Hungarian optimal assignment algorithm for identity retainment, and the Kalman Filter for trajectory prediction; and explains how these components have been integrated into the final system.

3.1 Code Design

From the inception of this project, this system has been designed with the end goal of creating an application to be used by researchers for neuroethological experiments. Researchers will employ this program to track video feeds of *Xenopus laevis* tadpoles' swimming patterns and save the recorded motion data to file. This recorded data can then be statistically analyzed and used in new research studies, such as quantifying the effects of different neurochemical drugs on their swimming behavior.

Moreover, one of the key objectives for this system has been to achieve *real-time* analysis and tracking, so that the tracker could run on a live camera in an experiment and record data for long periods of time, instead of only being able to run on recorded video files. Therefore, throughout the development process of this system, minimizing the computational time requirements of all the algorithms and frameworks to be integrated was always in consideration, and played a large role in the selection of which components were ultimately built into the system.

3.1.1 System Overview

The tracking system consists of a series of end-to-end connected components. Each takes in input information, performs operations with it, and then outputs new information to the next part of the system. Fig. 12 shows the flow of these components, in which each static video frame passes through a series of steps that finally leads to new motion data and a real-time visualization of the tracking. In this way, the detection model, optimal assignment

algorithm, and trajectory estimation algorithm are all integrated into the complete tadpole tracking system. Each of these key components will be discussed later in this section.

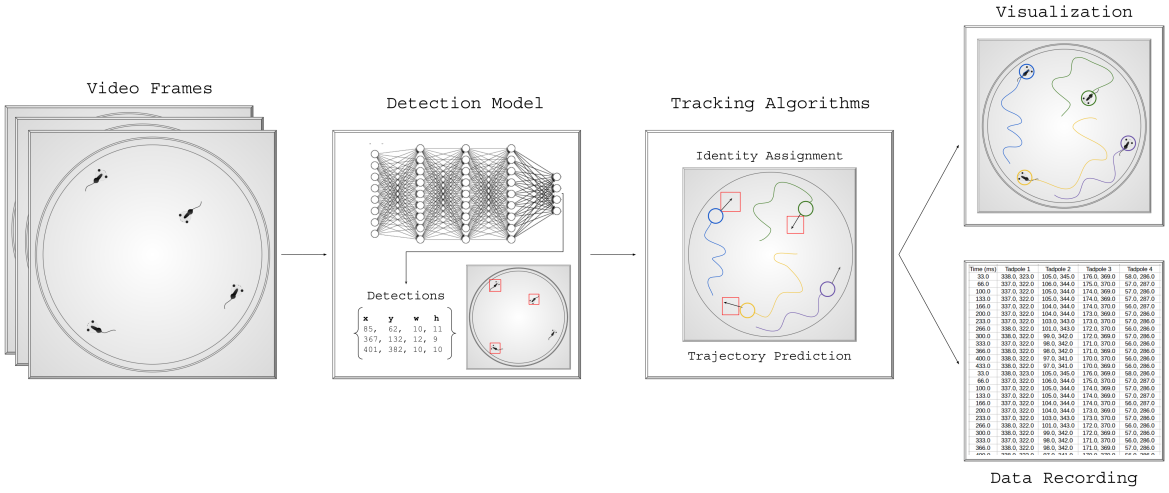


Figure 12: Diagram depicting structure of this tadpole tracking system. On the far left, a video is inputted frame-by-frame into the system, which analyzes it with the components shown and finally displays the visualization and records the tadpole motion data.

3.1.2 Programming Language

The programming language used for this system was Java. Initially, Python was explored for its ease of use and the wide array of deep learning libraries available for it, but ultimately, for usage in a real-time system the overhead of the language proved too time costly to justify its advantages. Java was chosen instead, because as a language it is faster than Python, and it has support for deep learning through the DeepLearning4J library developed by Skymind, a machine intelligence and enterprise software firm based in San Francisco (Eclipse Deeplearning4j Development Team 2018). This language switch was also made with the end goal of creating a standalone application in mind. While creating executable applications using Python is technically possible, it is often very difficult to include all the dependencies used by the code correctly, resulting in the application being unusable. Java handles this dependency inclusion much better, especially when a rich IDE like Eclipse or IntelliJ is used along with a dependency manager like Maven. Both IntelliJ and Maven were used in the design of this system, which makes the code easily exportable to an executable application as a JAR file.

3.1.3 Interface

The client base that this system will be provided to will generally not have experience with computer programming, so it was necessary to design an interface that researchers could use to interact with the tracking program. The initial structure for a visual interface for this system has been implemented using the JavaFx library. JavaFx is a software platform for developing desktop applications that can run across a wide variety of operating systems and devices. Currently, the primary functionality provided by this system's interface is a visual display of the tracking in real time, but future work will focus on developing an interactive GUI with which researchers can schedule timed recording experiments, change key attributes and variables as needed, and even visualize real-time trends in the recorded motion data.

3.2 Tadpole Detection Model

The tadpole detection component of the system is a deep convolutional neural network framework designed for object detection and localization within images. As shown in Fig. 12, during tracking, each individual video-frame is passed in to the detection model, and the model returns a set of localized detections, which are then converted to bounding boxes and used to update the tracked tadpole locations and trajectories.

3.2.1 Object Detection Framework

While simple convolutional neural networks are sufficient for image classification, when it comes to object localization within images, more complex network structures must be employed, which is why object detection is still an active area of research and development. There are three major object detection frameworks that are considered the current state-of-the-art approaches: Faster R-CNN (Ren et al. 2015), Single Shot Multibox Detectors (Liu et al. 2015), and You Only Look Once (YOLO) (Redmon and Farhadi 2016). Typically, the accuracy of systems like these is measured using a metric called mean average precision (mAP). MAP is the average of the maximum precisions at different recall values, where precision is a measurement of the model's prediction accuracy (i.e. the percentage of positive predictions that are correct), and recall is a measurement of how many predictions the model correctly got out of the total number of objects in the data set (calculated using the ratio of

$TP/(TP+FN)$, where TP = true positive predictions and FN = false negative predictions).

All three systems achieve competitive mAP results on popular datasets such as VOC2007 and Coco, but the most recent iterations of Yolo (versions 2 and 3) have demonstrated both high accuracy and significantly reduced inference speed allowing real-time video analysis. Redmond et al. benchmarked their Yolo network against the other state-of-the-art detectors, comparing both inference speed and accuracy on the VOC 2007 dataset. Fig. 13 displays the results of these benchmarkings, and shows that while the original release of Yolo sacrificed accuracy for its speed, the second iteration, Yolov2, performs on par with the most accurate detectors at resolutions of 416x416 and greater, while also achieving an impressively quick inference speed that allows processing videos at 40-60 frames per second.

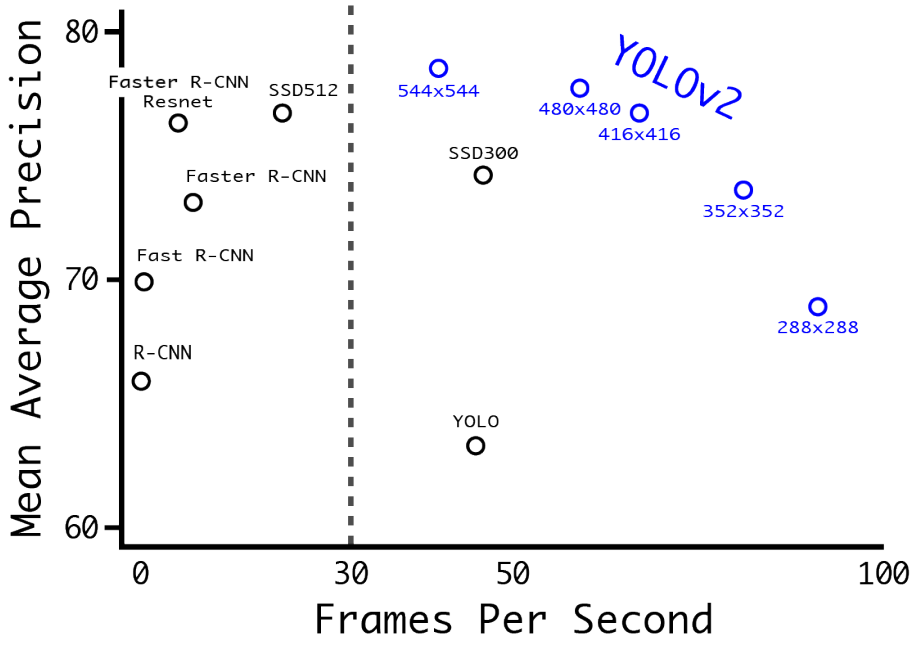


Figure 13: Yolov2 performance benchmarking against other state-of-the-art detectors (Redmon and Farhadi 2016).

When designing a real-time object tracking system, having an object detection network that can run inference on an incoming image as quickly as possible without sacrificing accuracy is of the highest priority. On this basis, Yolo was the best framework for this tadpole tracking system. The second iteration (Yolov2) was chosen because it has an interface already implemented in the DeepLearning4J Java library.

3.2.2 Dataset collection

In order to train a convolutional neural network for tadpole detection, a relatively large dataset of tadpole images was needed. There do not seem to be any publicly available image datasets of *Xenopus laevis* tadpoles, so it was necessary to create one. A set of 737 images of tadpoles were collected using an iPod Touch camera with a resolution of 720x1080 pixels. The images were eventually labeled with bounding boxes using a tool available on GitHub (Shi n.d.). For each image, the program created a corresponding text file with the total number of tadpoles in the image and the pixel coordinates for each bounding box that was manually drawn around each tadpole.

Prior to labeling the training set, however, the images were passed through random crops, filters, and rotations in order to introduce variance into the dataset. Rotations were only performed with an angle of 90, 180, or 270 degrees, because applying angles that were not multiples of 90 caused the empty peripheral parts of the rotated image to be filled with black pixels. Random cropping was also applied to the training images, with each image being randomly cropped to no less than 2/3 of its original size. Finally, various filters from the OpenCV library were applied at random to images, to make the model more robust at detecting tadpole shapes in varied lighting and camera conditions. Fig. 14 shows the effects of filters that were used on the training images.

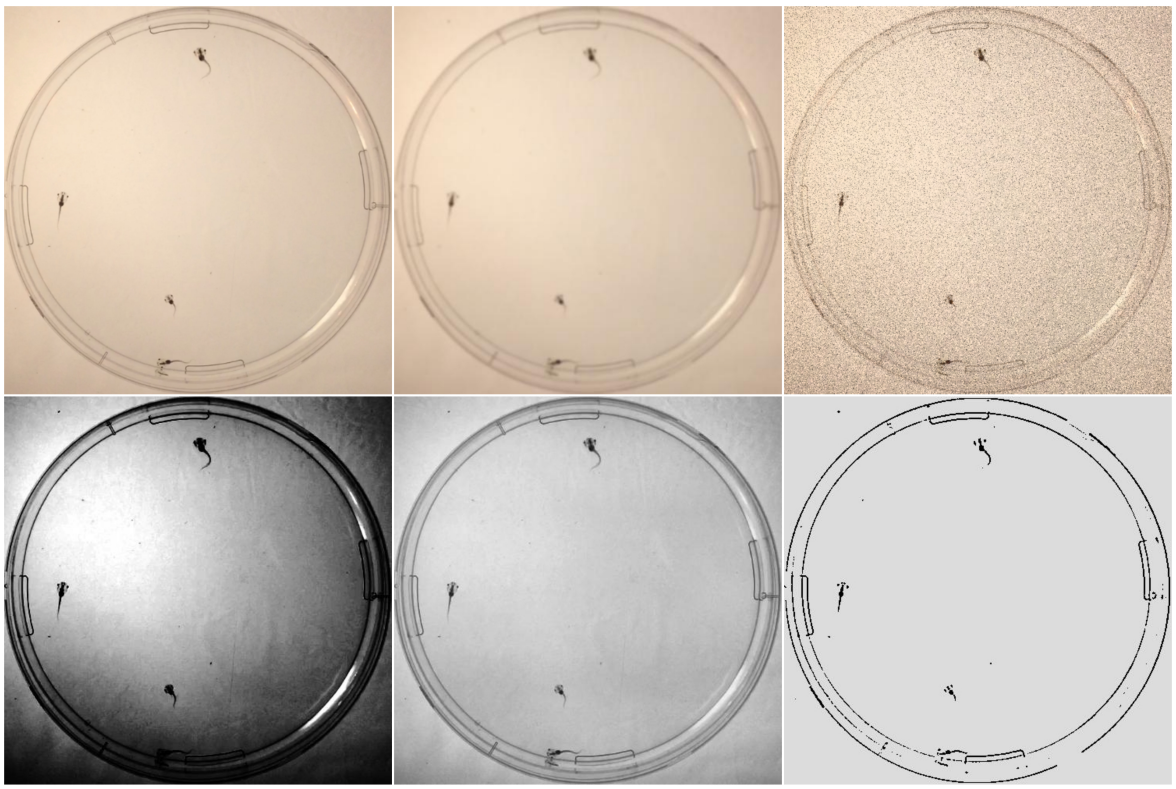


Figure 14: Examples of filters applied to training images. From top left to bottom left, clockwise: raw image, Gaussian blur, HSV noise, adaptive mean thresholding, contrast limited adaptive histogram equalization (CLAHE), and histogram equalization.

3.2.3 Training the Yolo network

The Yolo neural network model was pretrained using the technique of "transfer learning", which is what ultimately made it possible for such a deep network to learn to accurately detect tadpoles from a (relatively) small dataset of tadpole images. Transfer learning entails using a pretrained weights file from a convolutional neural network partially trained on a large classic dataset such as ImageNet. Beginning the training on a dataset like ImageNet initializes the network's weights and allows the units to begin to learn to detect edges and shapes. After a certain number of epochs, the training data is switched to the smaller tadpole dataset and the the network is trained the rest of the way on these images. This final training on the smaller custom dataset fine tunes the final layers of the network to recognize only the object or objects in the custom dataset. Thus by leveraging the power of a large dataset to initialize the weights and units of the network, a relatively small custom dataset is sufficient to fully train a convolutional network.

The model was initialized with the standard Darknet yolov2 weights file provided on Joseph Redmon's website, and then trained on the tadpole dataset using his Darknet framework. In addition to learning object classifications, Yolo also learns to detect and localize objects during training, using a joint training algorithm. This approach allows the network to "learn detection-specific information like bounding box coordinate prediction and objectness as well as how to classify common objects" (Redmon and Farhadi 2016).

The dataset was divided into 48 images for the validation set and 689 images for training. The model was trained through the Darknet framework, which uses the Adam optimizer to minimize cross entropy on the validation set. Darknet does not stop training automatically, but it does output its loss during training and saves model checkpoints every 1,000 iterations. By logging the loss to file, it is therefore possible to visualize the loss over the entire training period, as shown in Fig. 15.

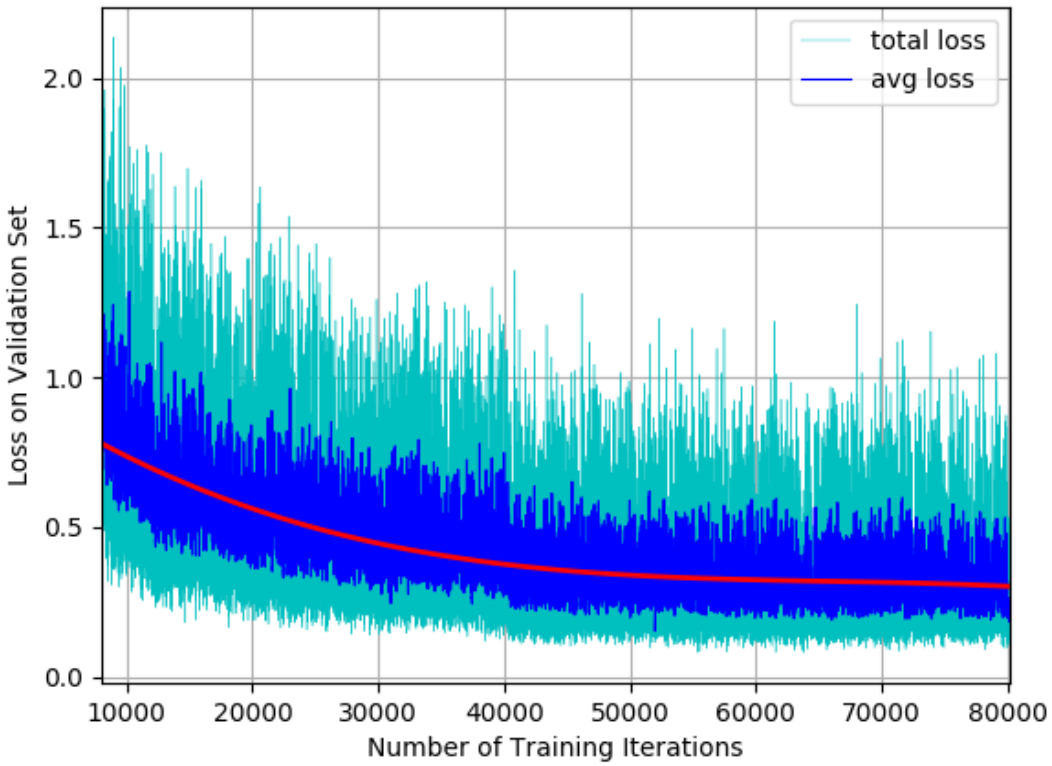


Figure 15: Cross entropy on the validation set.

The training was run using batches of 8 images at a time, meaning that the entire dataset passed through the network approximately every 96 batch presentations. Although Darknet measures training in terms of iterations, instead of the more typical usage of "epochs", note that dividing the number of iterations by 96 gives the number of epochs. By 50,000 iterations, the loss on the validation set all but flat-lined, but the model was allowed to continue training until 80,000 iterations, upon which it stopped automatically after reaching its internal "max_batches" variable. As Fig. 16 shows, the performance on test data of all the model checkpoints after 20,000 iterations is identical.

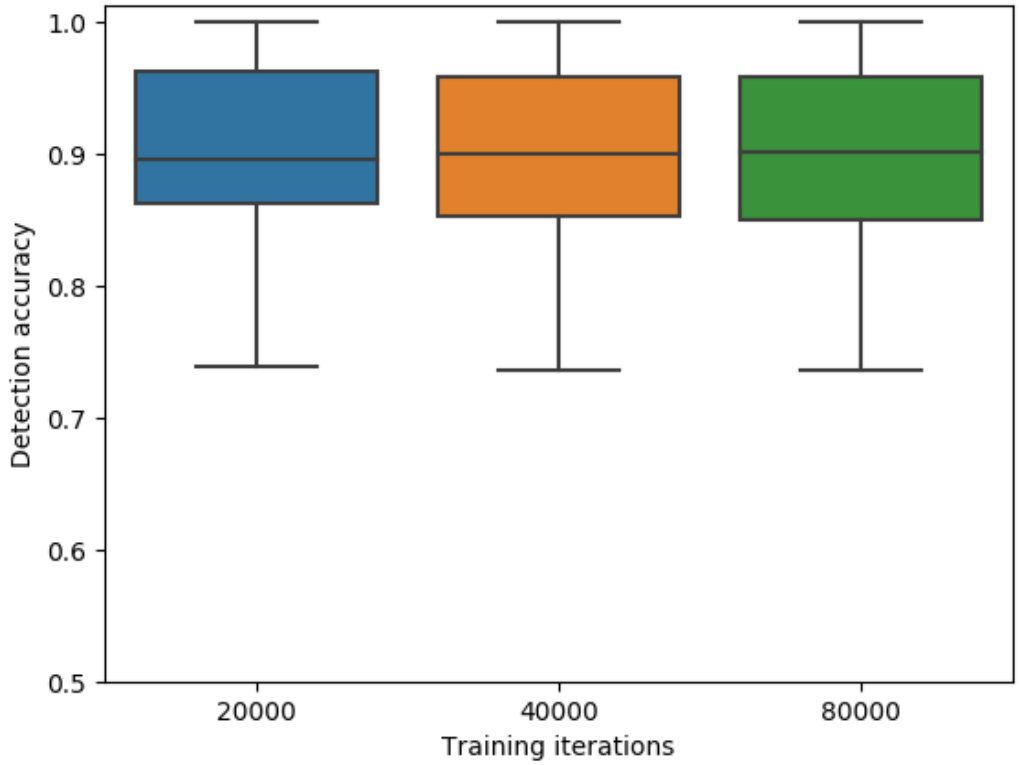


Figure 16: Comparison between detection accuracy of model checkpoints after varied amount of training. Detection accuracy was calculated using ground-truth manually-labeled videos (n=20), and was determined by whether the location of each predicted bounding box corresponds to a true labeled coordinate. Further evaluations of the model are discussed in the Results section.

Finally, in order to use darknet weights from within the Java DeepLearning4J library, two conversions had to take place. First, the weights file was loaded into Keras using YAD2K (cite Yad2k) as a Darknet model, and then serialized to a Hierarchical Data Format 5 (H5) Keras model file. The Keras model file was then loaded in Java using DeepLearning4J's Keras model import functions and re-serialized into the zip file format that DeepLearning4J expects. There were no noticeable changes in either accuracy or speed between the twice converted weights and the original darknet weights.

3.3 Tracking Algorithms

While having accurate detection is certainly the most important component to designing an accurate object tracking system, it must be combined with other algorithms to properly track objects over time. As shown in Fig. 12, each new set of detections from the detection model passes through a series of tracking algorithms before the recorded positions of the animal tracker objects are updated. In this system, there are two such algorithms that are crucial to accurate tracking: identity retainment and trajectory prediction. These have both been integrated with the following Animal class.

3.3.1 Animal Class

For the purpose of retaining the data associated with individual identities, an "Animal" class was created. This class stores important attributes such as the animal's current position and the list of its previous positions. The tracking system relies on knowing the true number of animals that are in the environment to be tracked, as the user is expected to input the number of animals when first starting the program. This number then determines how many Animal object instances the system creates for tracking. The fundamental attributes and functions of the Animal class involved with position coordinates and movement are included in the following code.

```
public class Animal {

    private ArrayList<double[]> dataPoints; // list of previous positions
    private CircularFifoQueue<int[]> linePoints; // trajectory visualization

    public int x, y;
    public double vx, vy;
    private int[] positionBounds;
    public Scalar color;

    public Animal(int _x, int _y, int[] bounds, Scalar clr) {
        this.x = _x; this.y = _y;
        positionBounds = bounds;
        color = clr;
        linePoints = new CircularFifoQueue<>();
        dataPoints = new ArrayList<>();
    }

    public void updateLocation(int _x, int _y, double dt, long timePos) {
        this.x = _x; this.y = _y;
        applyBoundsConstraints();
        dataPoints.add(new double[]{timePos, this.x, this.y});
        linePoints.add(new int[]{this.x, this.y});
        updateVelocity(dt);
    }

    private void applyBoundsConstraints() {
        x = (x>positionBounds[0]) ? x : positionBounds[0];
        x = (x<positionBounds[1]) ? x : positionBounds[1];
        y = (y>positionBounds[2]) ? y : positionBounds[2];
        y = (y<positionBounds[3]) ? y : positionBounds[3];
    }
}
```

3.3.2 Individual Identity Retainment

As discussed in the Background, the task of identity retainment was approached as a graph problem in this system. To solve this problem, the Hungarian optimal assignment algorithm was implemented. The Hungarian algorithm has been successfully applied to the task of identity retainment in computer vision by minimizing the cost of assignments between detections and tracked objects (A. G. A. Perera et al. 2006, Huang, Wu, and Nevatia 2008, Xing, Ai, and Lao 2009). To apply the Hungarian algorithm, an nxn cost matrix is constructed using distances between the predicted bounding boxes and the current locations of the tracker objects, and then this matrix is given to the algorithm, which manipulates the values of the matrix until it arrives at the optimal solution. At the time of the algorithm's publication (1955), very few people had access to computers, so the algorithm was developed to be done by hand. Because of this, the algorithm manipulates the matrix in a very visual manner; by covering and uncovering specific rows and columns in the matrix until the cells that contain zeros represent a complete set of unique assignments.

The actual steps of the algorithm are as follows:

Step 1. Subtract smallest value in each row from all values in that row.

Step 2. Subtract smallest value in each column from all values in that column.

Step 3. Draw the minimum number of lines through rows and columns such that all the cells that contain zeros in the cost matrix are covered

Step 4. If number of lines drawn is $\boldsymbol{n}$, you are done. (The optimal assignments have been found). However if the number is less than $\boldsymbol{n}$, go to Step 5.

Step 5. Find the smallest value not covered by any lines. Subtract this from each uncovered row, and add it to each covered column. Return to Step 3.

This is best illustrated with an example. Consider the following problem, adopted from (Bruff 2005). A sales manager has a salesperson in each of three cities: Austin, Boston, and Chicago. The manager needs to send a salesperson to each of three other cities: Denver, Edmonton, and Fargo, but wants to minimize the cost of airfare. Therefore, the manager must figure out which salesperson to send to each city.

The following table shows the cost of airplane tickets between the cities.

From / To	Denver	Edmonton	Fargo
Austin	250	400	350
Boston	400	600	350
Chicago	200	400	250

Consider one possible assignment:

From / To	Denver	Edmonton	Fargo
Austin	[250]	400	350
Boston	400	[600]	350
Chicago	200	400	[250]

The total cost of this assignment would be $\$250 + \$600 + \$250 = \$1100$.

Another possible assignment gives a smaller cost of $1000:

Austin	[250]	400	350
Boston	400	600	[350]
Chicago	200	[400]	250

Eventually, by checking all possible assignments, the optimal solution can be determined, which gives a total cost of $950:

Austin	250	[400]	350
Boston	400	600	[350]
Chicago	[200]	400	250

Thus the solution to the problem is to send the salespeople from: Austin to Edmonton, Boston to Fargo, and Chicago to Denver. However, this brute force solution is computationally expensive, requiring that we check $n!$ possible solutions. For a small 3x3 matrix, checking all possible solutions is fine, but if the sales manager had 100 salespeople flying to 100 cities, the number of solutions to check would quickly become intractable. The Hungarian method reduces the number of solutions that must be checked drastically.

To solve this with the Hungarian algorithm, we represent the table as a *cost matrix*.

250	400	350
400	600	350
200	400	250

Step 1. Subtract 250 from Row 1, 350 from Row 2, and 200 from Row 3:

0	150	100
50	250	0
0	200	50

Step 2. Subtract 0 from Column 1, 150 from Column 2, and 0 from Column 3:

0	0	100
50	100	0
0	50	50

Step 3. Cover all zeros in the matrix with the minimum number of lines:

~~0~~	~~0~~	~~100~~
~~50~~	~~100~~	~~0~~
~~0~~	50	50

Since the number of lines is 3, which is equal to $\boldsymbol{n}$, we are finished. The optimal assignments are the cells that contain zeros with only one line through them:

0	[0]	100
50	100	[0]
[0]	50	50

These correspond to the values in the original matrix, giving the optimal cost of \$950:

Austin	250	[400]	350
Boston	400	600	[350]
Chicago	[200]	400	250

In order to apply the Hungarian algorithm to the task of identity retainment, the exact same steps are followed, except with an nxn cost matrix between the predicted bounding boxes and the current locations of the tracker objects. The resulting solved matrix is then used to assign each animal to its corresponding assignment.

Importantly, the Hungarian algorithm can also be easily modified to solve optimal assignments in situations where there are missing values. Fig. 17 shows an example situation where there are five tadpoles in the image, but only four of them have been detected.

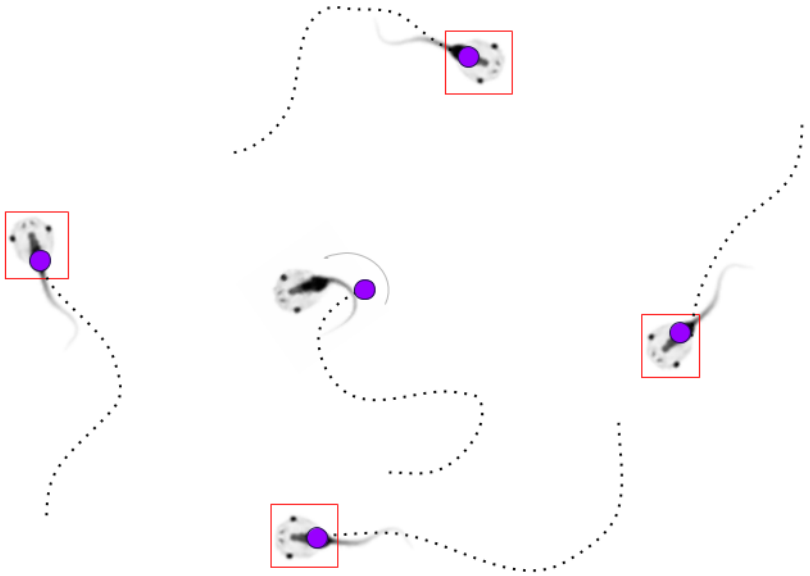


Figure 17: Example scenario with five tadpoles in image, but only four have been detected (red boxes represent detections by the model). The purple dot is a tracker instance, which is currently located at the last known location of the tadpole.

There are five tracker objects in this scenario, but only four bounding boxes to be assigned, so the Hungarian algorithm must correctly assign the four boxes to the correct animals and assign a *null assignment* to the tracker instance in the middle. In order to enable this functionality, when the cost matrix is constructed, an additional null assignment cell is added in between every real value in the matrix, and is given a predetermined "cost_of_no_assignment" value. This is done with the code on the following page. Note that "Animals" are the tracker instances, and boxes are the detections.

```
List<Animal> paddedAnimals = new ArrayList<>(animals.size() * 2);
List<BoundingBox> paddedBoxes = new ArrayList<>(boxes.size() * 2);

for (Animal a : animals) {
    paddedAnimals.add(a);
    paddedAnimals.add(null);
}

for (BoundingBox b : boxes) {
    paddedBoxes.add(b);
    paddedBoxes.add(null);
}

int animalsSize = paddedAnimals.size();
int boxesSize = paddedBoxes.size();

// the cost matrix must have equal dimensions.
// (extra cells are filled with a different cost_of_no_assignment value)
dimension = Math.max(animalsSize, boxesSize);
costMatrix = new double[dimension][dimension];

// calculate costs between tracker instances and boxes
for (int i=0; i<animalsSize; i++) {
    for (int j=0; j<boxesSize; j++) {
        costMatrix[i][j] = costOfAssignment(
            animals.get(i), boundingBoxes.get(j)
        );
    }
}
// fill extra blank cells with COST_OF_NO_ASSIGNMENT value
// note: rows = animalsSize, cols = boxesSize
if (cols < rows) {
    for (int r=0; r<rows; r++) {
        for (int c = cols; c < rows; c++) {
            costMatrix[r][c] = COST_OF_NO_ASSIGNMENT;
        }
    }
} else if (rows < cols) {
    for (int r = rows; r < cols; r++) {
        for (int c = 0; c < cols; c++) {
            costMatrix[r][c] = COST_OF_NO_ASSIGNMENT;
        }
    }
}
```

The COST_OF_NO_ASSIGNMENT value is configured to be higher than the average proximity of a true assignment, so it is never selected as an optimal assignment over a true assignment; but it must also be lower than the cost of an erroneous identity switch. This concept is demonstrated in Fig. 18.

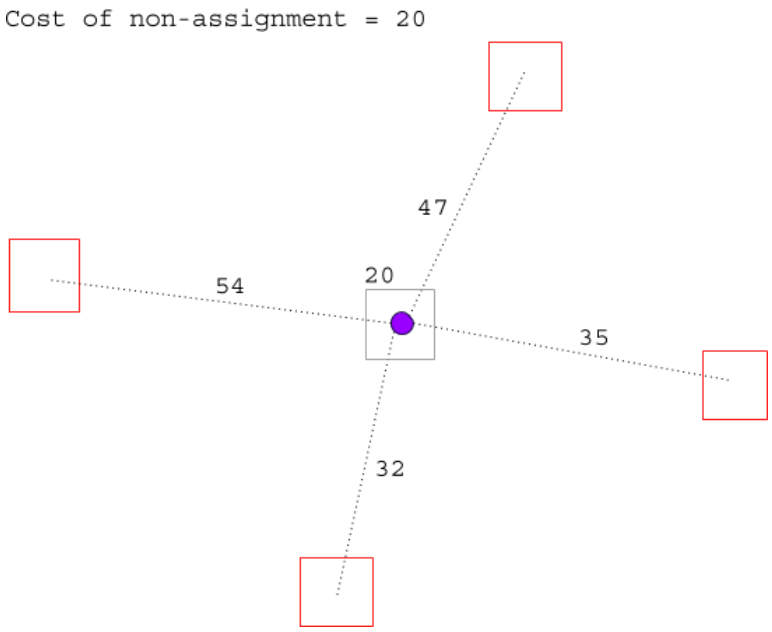


Figure 18: Identity Assignment as Graph Problem. Red boxes are detections, and the purple dot in the center represents the animal tracker instance that should be given a *null assignment* by the Hungarian algorithm. The costs of all possible assignments for this one tracker instance are shown, with 20 being a good "cost_of_no_assignment" value.

In this situation, the Hungarian algorithm would correctly assign each detected box to the respective tracker instance, and the tracker instance in the middle would get no assignment. It is important to note, however, that assignments are not made based on the closest detection (i.e. minimum cost) to each tracker instance. The Hungarian algorithm guarantees optimality by minimizing the *overall* cost of assignments, which sometimes requires assigning a tracker instance a value that was not the minimum possible assignment. This concept is visualized in Fig. 19, which shows how the Hungarian algorithm would determine which of four tracker instances would get a no-assignment, when given only three detections to assign.

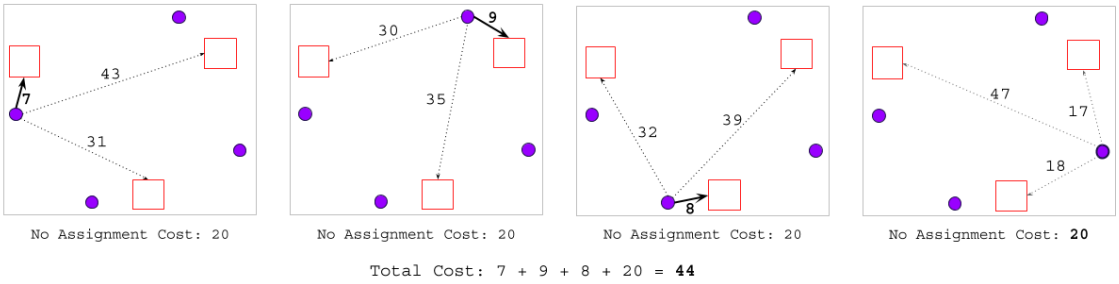


Figure 19: Example assignment scenario with four tracker instances (purple dots) but only three detections (red boxes). Costs between detections and tracker instances are shown using path distances, and the final assignment for each tracker instance that the Hungarian algorithm would give is in bold.

In the first three panels, each tracker instance is given the minimum cost assignment, but note that in the last panel the minimum cost assignment for the tracker would actually be to the detection above it, at a cost of 17. However, remember that the Hungarian algorithm minimizes the *overall* cost of assignments, and since in this situation one tracker instance must be given a no-assignment, the last instance is the one selected because this results in an optimal overall cost of 44.

3.3.3 Trajectory Estimation

Even a well-trained model will invariably miss detections in video frames. When this occurs, it is often useful to use a trajectory prediction algorithm in place to interpolate between the skipped frames and help prevent further error or identity swapping once the detection network is back on track. Kalman filters have been shown to work well for linear trajectory prediction in a wide variety of applications (Erwin, Santillo, and Bernstein 2006, L. P. Perera and Soares 2010, Rosales and Sclaroff 1999). They are ideal for implementation in systems that are continuously changing, and they require very little memory, as they only keep track of a few matrices representing the previous state of the system. Moreover, they are also very fast, making them well suited for implementation in real-time systems. For these reasons, Kalman filters were used for tadpole trajectory estimation in this system.

The Kalman filter is a method of predicting the future state of a system based on its previous states. It was originally developed to reduce noise from data measurements, but it

can also be used for trajectory prediction (L. P. Perera and Soares 2010). When the Kalman filter model makes a prediction, it acts as a linear function of the system's state, taking in the previous state and outputting the new predicted state. The reader is directed to the following resources to learn more about the underlying mathematics of the Kalman filter: (Babb 2015, Kleeman n.d., The MathWorks, Inc. 2018, Welch and Bishop 2001).

The Kalman filter Java code implementation used was the version that is included in the Apache Commons Math standard library. In the current tracking system, each Animal class instance gets an individual Kalman filter instance, which is updated with the Animal's location and velocity at each time step. When a detection is missed in a frame and a given Animal instance does not get an assignment, it calls its predictTrajectory() function, which takes the predicted state from the Kalman Filter to estimate its next location. The following is the code used in the predictTrajectory function:

```
public void predictTrajectory(double dt, long timePos) {

    double[] predictedState = this.trackingFilter.getStateEstimation();

    double predX = predictedState[0];
    double predY = predictedState[1];
    double vx = predictedState[2];
    double vy = predictedState[3];

    if (PREDICT_WITH_VELOCITY) {
        int newx = (int) Math.round(this.x + (vx * dt));
        int newy = (int) Math.round(this.y - (vy * dt));
        updateLocation(newx, newy, dt, timePos);
    } else {
        updateLocation(predX, predY, dt, timePos);
    }
}
```

4 Results

This section presents the key results of testing the tracking system on a dataset of tadpole videos and explores the impact that each of its components make on its final performance. Specifically, the accuracy of the Yolo detection model, as well as the effects of both the Hungarian assignment algorithm and the Kalman Filter on the system's overall accuracy are reported using illustrative materials.

4.1 Yolo Model Detection Accuracy

A well trained model will miss detections far less frequently than a poorly trained one, requiring less error handling and prediction by the tracking system. Effective training results in an overall increase in tracking accuracy. Therefore, making the object detection model as accurate as possible was essential to having a viable tracking system.

4.1.1 Creation of Testing Dataset

A dataset of 80 videos was collected to test the detection accuracy of the trained Yolo model. Each video is 30 seconds long and was recorded at a rate of 30 frames per second, at a resolution of 1280 by 720 pixels. The videos were recorded in four batches of twenty, where each batch contains recordings of a different number of tadpoles. The groupings used were groups of one, two, four, and six tadpoles. For the recordings of one and two tadpoles, new animals were used for each individual video, and for the recordings of four and six, the animals were switched out once halfway through the full recording trial (after ten videos). The tadpole grouping numbers were chosen to represent the spread between the minimum (one) and the ideal maximum (six) number of tadpoles that would be used in an experiment. When the group contains more than six tadpoles confined within a single dish, the accuracy of the Yolo model decreases significantly, due to the higher prevalence of trajectory occlusions and the innate clustering behavior of tadpoles. In addition to this computational limitation, five to six tadpoles in the currently used dish is an ideal number for neuroethological studies as well, because the animals have enough room in their environment to exhibit both individual

swimming behaviors and socially interactive behaviors. Placing higher numbers of tadpoles in the dish results in overcrowding and causes the tadpoles to become overstimulated by interactions with their neighbors.

4.1.2 Detection Accuracy on Testing Dataset

The model was run on all eighty testing videos, and the detection accuracy was recorded for every frame in each of the videos. The accuracy of the model on a given frame was calculated by dividing the number of detections by the known ground truth number of animals in the frame. Extra detections were counted negatively, resulting in the final number always being less than or equal to 1.0 (100%).

This method of calculating accuracy relies on the assumption that all detections that are being counted are accurate detections of tadpoles. In this particular case, this is a fair assumption to make, because the consistent experiment setup and robust training dataset has resulted in the model performing with an extremely low rate of false-positive detections. To demonstrate the validity of this assumption empirically, one twenty-video batch of the testing video set was manually labeled, and the measured accuracy from counting detections was compared against the ground truth. Each detected bounding box was checked against the list of ground truth coordinates for the given video frame, and if a ground truth point fell within the area of the box, it was considered to be a true detection. Each truth point could only be used for verification by one box; if a single point fell within the areas of more than one detected bounding box, only one was marked as being a true detection.

The batch of videos chosen was the group of four tadpoles, because it is the ideal number of animals to use in an experiment with this system. The videos in the batch were hand-labeled with coordinates for the tadpoles using a simple Java interface that allowed the researcher to click on static video frames one-by-one, marking the locations of tadpoles and then saving these coordinates to file. Ground truth coordinates were designated at every 10th frame (1/3 of a second), and then all remaining unlabeled frames were labeled by interpolating between every pair of truth points. The resulting set of coordinates provided the means to quantitatively measure the detection accuracy of the model across all twenty videos, and compare it against the detection counting metric, the results of which are shown in Fig. 20.

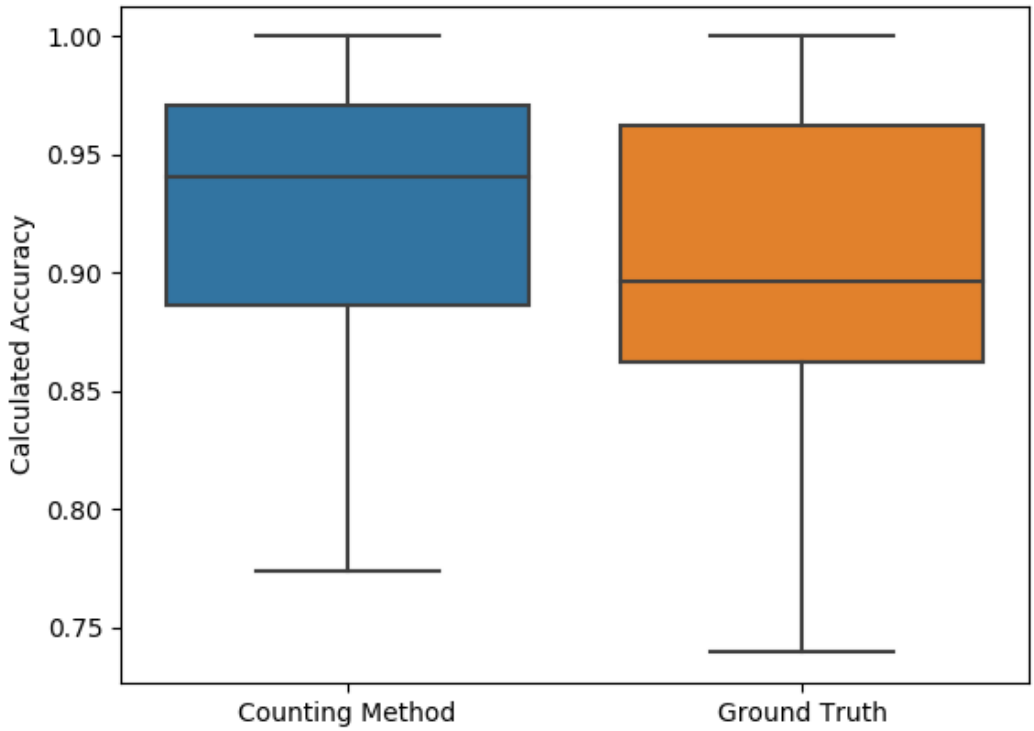


Figure 20: Comparison between using ground truth for evaluating the detection accuracy of the model and counting the number of bounding boxes. As is standard for boxplots, the box encompasses 50% of the data observations, with the middle line showing the mean. The upper and lower bars represent the 75th-percentile and the 25th-percentile, respectively, and the distance between these two values is the "interquartile range" (IQR). (n = 20 videos).

While the difference between the scores calculated by the two methods is significant (Paired Sample T-Test, t = 5.4083, p = 0.00003), the graph suggests that the detection counting method is sufficient for providing an approximation of the system's performance on varying numbers of tadpoles. The error between this method and using ground truth was relatively consistent, giving a score that was on average **2.3%** higher than the true score. This error can be kept in consideration when observing the model's scores on the remaining groupings of tadpoles, shown in Fig. 21.

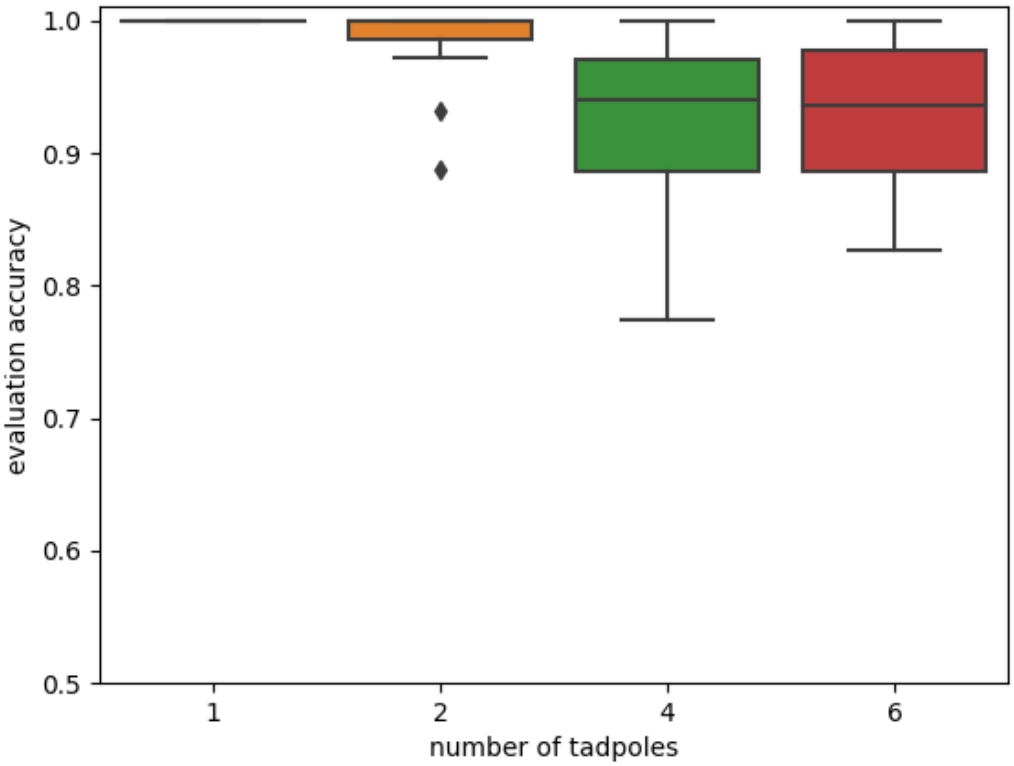


Figure 21: Detection accuracy on all four groupings of tadpoles, measured using the detection counting method. Data observations that fall outside of the IQR are outliers and are represented with dots. (n=20).

4.2 Tracking System Accuracy

The labeled twenty video batch test video set was reused to test the tracking accuracy of the entire system. While detection accuracy is the most crucial part of the tracking system, the overall tracking accuracy is determined by proper integration of the model's detections with the Hungarian optimal assignment algorithm and the Kalman filter for trajectory prediction.

The system achieved an average score of 91.61% tracking accuracy across the entire batch of testing videos. What this signifies is that across all twenty of the thirty-second videos, the system was accurately tracking an average of 91.61% of the tadpoles across the 900 video frames. This metric is the result of the integration of the Hungarian optimal assignment algorithm and the Kalman Filter into the system. Fig. 22 demonstrates the impact that each

of these components individually makes on the overall performance of the tracker system.

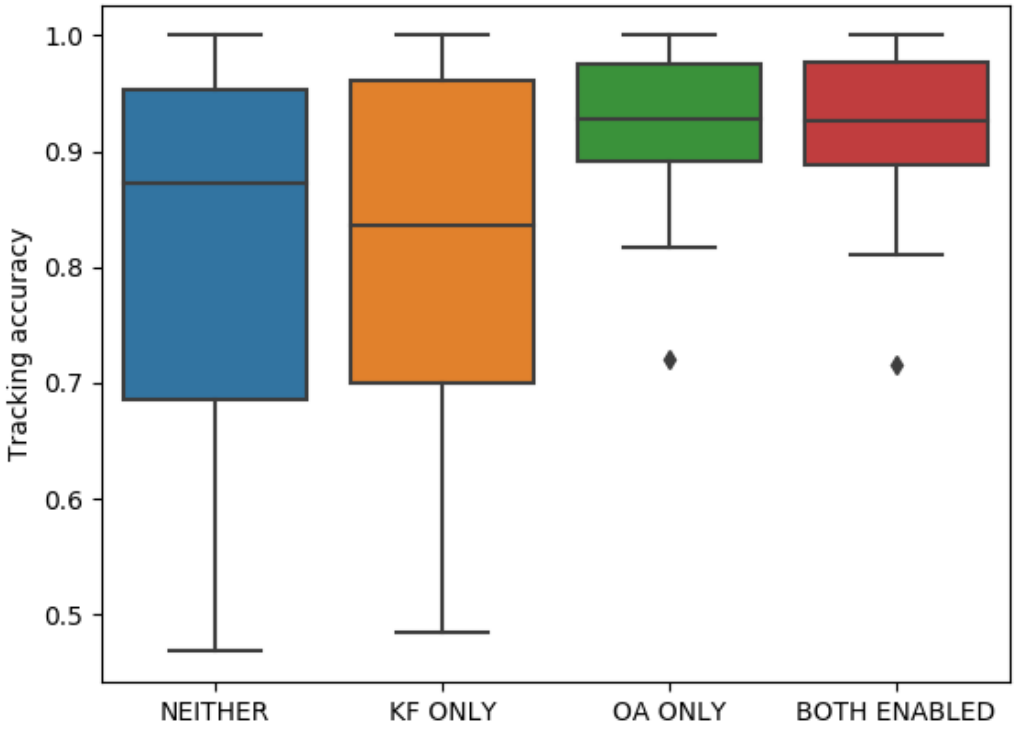


Figure 22: Impact of each individual component on the overall performance of the tracking system, run on the ground-truth-labeled video set. OA = optimal assignment algorithm, KF = Kalman Filter. (n=20).

As the graph suggests, the optimal assignment algorithm has a much larger impact on the score than the Kalman Filter. In order to measure the accuracy without the assignment algorithms, the assignment code was replaced with the following nearest-box assignment code, which does not guarantee optimal assignments:

```
// prevent assignment of multiple animals to one detection box
List<BoundingBox> assignedBoxes = new ArrayList<>();

for (Animal animal : animals) {

    BoundingBox closestBox = null;
    double minProx = Double.MAX_VALUE;

    for (BoundingBox box : boundingBoxes) {
        if (assignedBoxes.contains(box)) {
            continue;   // skip already assigned boxes
        }
        // check if new box is closer to animal than previous box
        double prox = Math.pow(
                    Math.pow(box.centerX - animal.x, 2)
                    + Math.pow(box.centerY - animal.y, 2),
                0.5);
        if (prox < minProx) {
            closestBox = box;
            minProx = prox;
        }
    }
    // if an assigned box exists, update animal location to it
    if (closestBox != null) {
        animal.updateLocation(
                closestBox.centerX, closestBox.centerY, dt, timePos
        );
        assignedBoxes.add(closestBox);

    // otherwise, predict trajectory for current timestep
    } else {
        animal.predictTrajectory(dt, timePos);
    }
```

4.2.1 Perfect Performance Measurement

Tracking accuracy should also be evaluated in terms of its frequency of achieving 100% accuracy. This provides an indication of how trustworthy and reliable the system is for recording meaningful data. Fig. 23 shows the average amount of time that the system performed with perfect performance across the ground-truth-labeled videos.

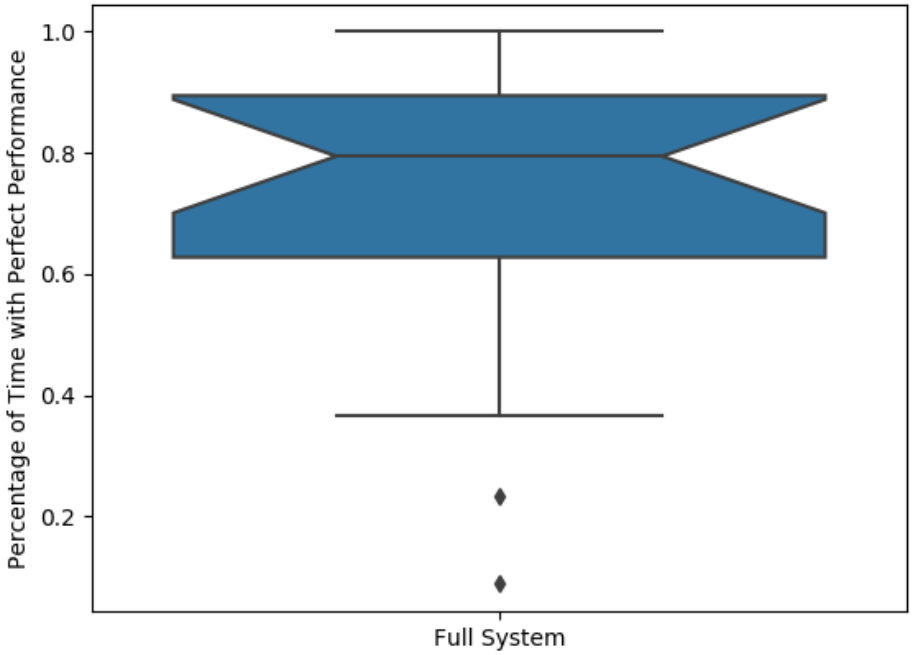


Figure 23: Average percentage of time that the system exhibited perfect tracking performance (n=20). The side notches in the box are the confidence intervals, with a confidence of 0.95.

Perfect performance is determined as the following: for each of the Animal tracker objects, there must be a distinct ground truth point within one half of a tadpole-body-length (approximately 5 pixels in a 416x416 image) of the object's centroid. Across all twenty videos, this system exhibited perfect performance on 79.44% of all the frames. For any given frame in a video, therefore, there was a 79.44% likelihood that the system would be performing at 100% tracking accuracy. This score is much lower than the system's overall accuracy score of 91.61% because the qualifications for perfect performance are much stricter. In a given frame, if even one tracked object is not within the predetermined radius of a ground truth point, this is scored as a lapse in perfect performance.

4.2.2 Sources of Error

The primary source of error in tracking is the tendency for tadpoles to cluster together in the dish, shown in Fig. 24. When several tadpoles group together closely enough that their bodies are touching and even overlapping, the neural network model fails to separate them into distinct detections and instead surrounds them all with a single bounding box. It is very difficult for the system to continue tracking accurately when this occurs, even though it has algorithms in place to try to handle such scenarios. When tadpoles are distributed throughout the space and no clustering is taking place, the system performs at nearly perfect accuracy.

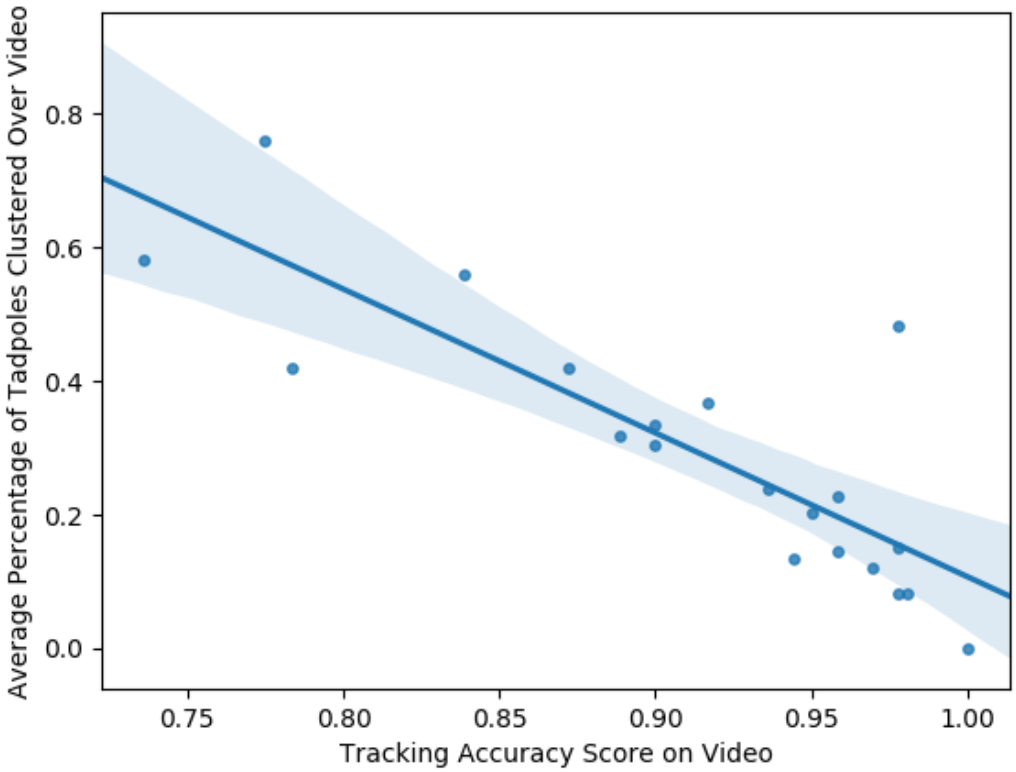


Figure 24: Average percentage of tadpoles clustered over video and corresponding effect on accuracy of tracking system (n=20). The shaded region shows the confidence intervals, at a confidence of 0.95. When the majority of the tadpoles are clustered together, the system performs poorly, with approximately 75% tracking accuracy, but when fewer tadpoles cluster, the system is able to track them with greater accuracy. In one video, no clustering behavior was exhibited, resulting in the system performing with 100% accuracy.

Importantly, however, the average duration of errors made by the tracking system is quite low, with almost all errors lasting no more than 3 seconds, as shown in Fig. 25. This is important, because short-lasting errors are easy to handle and often have no effect on the validity of the tracked tadpole motion data. If the system loses track of a tadpole for 0.5 seconds because the model fails to detect it for 15 frames, this will introduce very little error, because the system will quickly reconnect to the animal when it is detected again by the model. A tadpole cannot swim very far even in a full second, and so it will still be located in close proximity to its last known location.

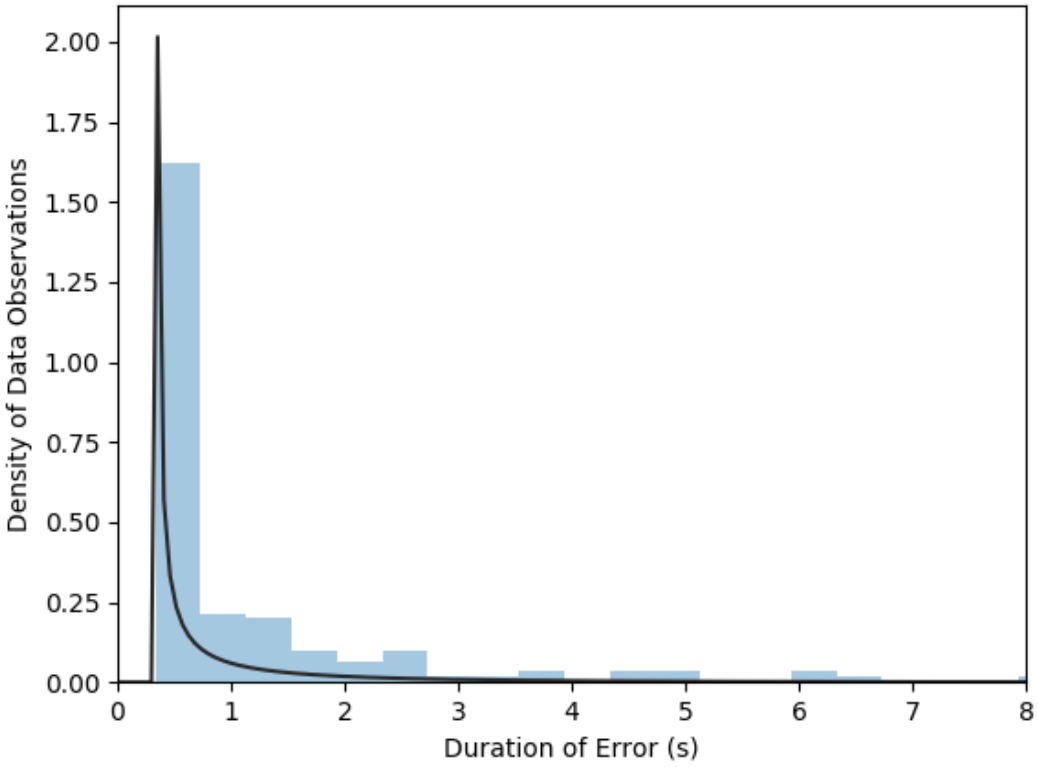


Figure 25: Distribution of recorded error durations, showing that a large majority of data observations are error durations of less than one second. The black curve is a parametric gamma distribution curve fit to the dataset.

If the system were to regularly lose track of animals for longer intervals of time, however, because the model was consistently failing to detect all the animals, this would greatly reduce its ability to produce meaningful recorded motion data. Therefore, the fact that the majority of errors last less than one second is important for verifying that this tracking system is a viable tool for tracking tadpoles in this experimental setup. Fig. 26 shows the correlation between duration of errors and the system's ability to track tadpoles accurately. As can be seen, there is a much higher concentration of error duration data observations in the 0-2 second range than in the longer time interval ranges. When the average duration of errors is shorter, the system's tracking ability is improved.

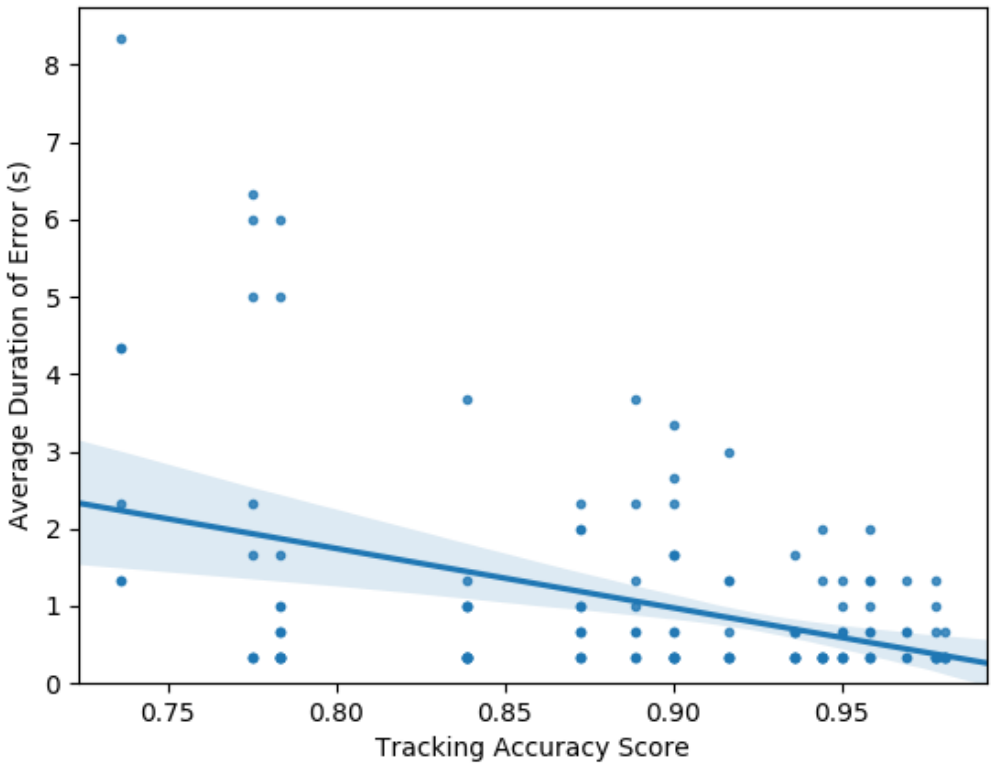


Figure 26: Duration of errors versus tracking score (1.0 = perfect performance). Shaded regions represent confidence intervals at a confidence of 0.95. Short errors are easy for the system to handle, but errors that last for longer periods of time reduce the system's ability to continue tracking animals accurately. (n=20).

4.2.3 Identity Retainment

An important metric of a multi-object tracking system's accuracy is how well it is able to retain individual identities of the objects it is tracking. While this can be difficult to measure without hand labeling individual identities in all the video frames, one method that can be used in this particular case is recording when a tracked object has a sudden displacement to a different position in the screen. If the displacement is more than the animal could have moved naturally in the given increment of time (for example, a tadpole cannot move more than its body length between two video frames), this is an indication that the tracked object jumped to a different animal, thereby failing to retain the identity it was previously tracking.

This system's ability to retain individual identities was improved dramatically by the addition of the Hungarian optimal assignment algorithm. Using the aforementioned metric to count identity swaps, the optimal assignment reduces the average number of identity switches per 30 seconds by a factor of four (from 9.5 to 2.2), as shown in Fig. 27.

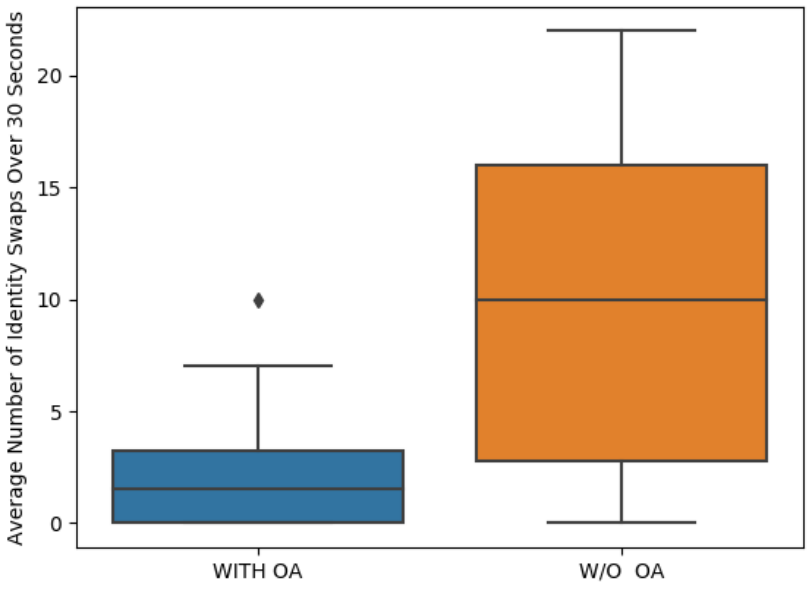


Figure 27: Impact of optimal assignment algorithm on average number of identity switches per 30 seconds (n=20 videos), demonstrating that the algorithm makes a significant difference (paired T-Test, t = -5.362, p = 0.0000356).

5 Conclusion

5.1 Future Work

Future work on this system will focus on improving the system's accuracy when it encounters tadpole clustering behavior, which is currently the primary source of tracking error. The training dataset will be expanded with new training images that contain tadpole clustering behavior. From this, the Yolo model will hopefully learn to better provide separate detections of tadpoles, even when they are in close proximity to each other. In addition to expanding the training set, the Kalman filter will be replaced with an implementation of an "Extended" Kalman filter (EKF), as described in (Erwin, Santillo, and Bernstein 2006), which is a variant of the regular filter that is better equipped for providing non-linear trajectory prediction. The implication of this is that the new EKF would be able to consider multiple previous states of the system and potentially factor in the derivative of the change in velocity to better make its predictions. Along with upgrading the Kalman filter, new algorithms will be included to handle the physics specifically involved in tadpole interactions; this primarily entails implementing new trajectory prediction algorithms that would take into account each tadpole's angle of movement and speed before the collision took place and how this would affect its resulting trajectory after the collision. Finally, the interface for this system will be fully developed for use by researchers. A complete GUI will be designed and integrated into the system to allow users to interact with the program as needed.

5.2 Conclusion

In this project, a multi-object tracking system was developed to record the movement of *Xenopus laevis* tadpoles, thereby making available new data that was previously unavailable to neuroscience researchers at Bard. Measuring the accuracy of this system on a relatively large set of testing videos has shown that this system is successful at tracking groupings of up to six tadpoles in a controlled petri-dish environment. These results signify that this system will be a viable tool for researchers to use in future neurobiology experiments.

6 References

List of Figures

Works Cited

Rosales, R. and S. Sclaroff (1999). "3D trajectory recovery for tracking multiple objects and trajectory guided recognition of actions". In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. Vol. 2, 117–123 Vol. 2. DOI: 10.1109/CVPR.1999.784618.

Welch, Greg and Gary Bishop (2001). *An Introduction to the Kalman Filter*. URL: http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf.

Bruff, Derek (2005). *Math 20 - Introduction to Linear Algebra and Multivariable Calculus Spring 2005: Assignment Problem Notes*. URL: http://www.math.harvard.edu/archive/20_spring_05.

Erwin, R. S., M. A. Santillo, and D. S. Bernstein (2006). "Spacecraft Trajectory Estimation Using a Sampled-Data Extended Kalman Filter with Range-Only Measurements". In: *Proceedings of the 45th IEEE Conference on Decision and Control*, pp. 3144–3149. DOI: 10.1109/CDC.2006.377394.

Perera, A. G. A. et al. (2006). "Multi-Object Tracking Through Simultaneous Long Occlusions and Split-Merge Conditions". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 1, pp. 666–673. DOI: 10.1109/CVPR.2006.195.

Huang, Chang, Bo Wu, and Ramakant Nevatia (2008). "Robust Object Tracking by Hierarchical Association of Detection Responses". In: *Computer Vision – ECCV 2008*. Ed. by David Forsyth, Philip Torr, and Andrew Zisserman. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 788–801. ISBN: 978-3-540-88688-4.

Sillar, Keith T. and R. Meldrum Robertson (2009). "Thermal activation of escape swimming in post-hatching Xenopus laevis frog larvae". In: *Journal of Experimental Biology* 212.15, pp. 2356–2364. ISSN: 0022-0949. DOI: 10.1242/jeb.029892. eprint: http://jeb.biologists.org/content/212/15/2356.full.pdf. URL: http://jeb.biologists.org/content/212/15/2356.

Xing, Junliang, H. Ai, and S. Lao (2009). "Multi-object tracking through occlusions by local tracklets filtering and global tracklets association with detection responses". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1200–1207. DOI: 10.1109/CVPR.2009.5206745.

Perera, Lokukaluge P. and Carlos Guedes Soares (2010). "Ocean Vessel Trajectory Estimation and Prediction Based on Extended Kalman Filter". In:

Pratt, Kara G. and Arseny S. Khakhalin (2013). "Modeling human neurodevelopmental disorders in the Xenopus tadpole: from mechanisms to therapeutic targets". In: *Dis Model Mech* 6.5. 23929939[pmid], pp. 1057–1065. ISSN: 1754-8403. DOI: 10.1242/dmm.012138. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3759326/.

Babb, Tim (2015). *How a Kalman filter works, in pictures.* URL: https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/.

Liu, Wei et al. (2015). "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325. arXiv: 1512.02325. URL: http://arxiv.org/abs/1512.02325.

Ren, Shaoqing et al. (2015). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497. arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

Ellenbroek, Bart and Jiun Youn (2016). "Rodent models in neuroscience research: is it a rat race?" In: *Dis Model Mech* 9.10, pp. 1079–1087. DOI: 10.1242/dmm.026120.

Redmon, Joseph and Ali Farhadi (2016). "YOLO9000: Better, Faster, Stronger". In: *arXiv preprint arXiv:1612.08242*.

Lee-Liu, Dasfne et al. (2017). "The African clawed frog Xenopus laevis: A model organism to study regeneration of the central nervous system". In: *Neuroscience Letters* 652. Plasticity and Regeneration After Spinal Cord Injury, pp. 82–93. ISSN: 0304-3940. DOI: https://doi.org/10.1016/j.neulet.2016.09.054. URL: http://www.sciencedirect.com/science/article/pii/S0304394016307376.

Maini, Vishal (2017). *Machine Learning for Humans, Part 4: Neural Networks and Deep Learning.* URL: https://medium.com/machine-learning-for-humans/neural-networks-deep-learning-cdad8aeae49b.

McQuillan, Molly (2017). "The Effects of Multisensory Integration on the Behavior of Xenopus laevis Tadpoles". In: 101. URL: https://digitalcommons.bard.edu/senproj_s2017/101.

Eclipse Deeplearning4j Development Team (2018). *Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0.* URL: http://deeplearning4j.org.

Ma, Yufeng et al. (2018). "Effects of user-provided photos on hotel review helpfulness: An analytical approach with deep leaning". In: *International Journal of Hospitality Management* 71, pp. 120–131. DOI: 10.1016/j.ijhm.2017.12.008.

Stanford University (2018). *CS231n: Convolutional Neural Networks for Visual Recognition.* URL: http://cs231n.github.io/.

The MathWorks, Inc. (2018). *Understanding Kalman Filters.* URL: https://www.mathworks.com/videos/series/understanding-kalman-filters.html.

Kleeman, Lindsay (n.d.). *Understanding and Applying Kalman Filtering.*

Levin, Golan (n.d.). *Image Processing and Computer Vision.* URL: https://openframeworks.cc/ofBook/chapters/image_processing_computer_vision.html.

Shi, Qiu (n.d.). *BBox-Label-ToolA simple tool for labeling object bounding boxes in images, implemented with Python Tkinter.* URL: https://github.com/puzzledqs/BBox-Label-Tool.