Using Drone Cameras and Computer Vision Toward Improved Stabilization and Landing Algorithms

Gaurav Lahiry

CMU-CS-19-114

May 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Alex Waibel, Chair
Dave Touretzky

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: Drones, Computer Vision, Deep Learning, Neural Networks, Stabilization

Abstract

Drones (or UAVs) equipped with cameras have been fast deployed to a wide range of applications, including agriculture, aerial photography, fast delivery, surveillance, etc. Similarly the growth and effectiveness of Computer Vision algorithms has led to widespread applicability and use in a massive variety of domains. Many commercially available drones make use of their camera systems to perform powerful tasks that are highly appealing to the consumer. Such tasks range from facial recognition, to motion tracking, to in some cases path planning and obstacle avoidance. The aspect of drone flight we wanted to focus on was stabilization over a target area and consequently landing in this chosen area. Stabilization algorithms on less advanced drones typically make use of on board sensors such as gyroscopes and accelerometers to try to keep the drone hovering in place. Additionally commercial drones that do incorporate visual features still struggle to achieve great performance in this realm. Similarly with the landing process, it is often the case that autonomous algorithms for it are done through the use of on board GPS and very little influence from visual cues. In this project, we aim to explore the possibility of incorporating more information into these processes using the downward facing camera on our drone of choice, the Parrot AR 2.0 drone. The bulk of our work makes use of training and evaluating neural networks to predict the position of the drone relative to a particular region of interest and testing our results by allowing the network to fly the drone by sending it controls corresponding to the predicted outputs. We will discuss the approaches we tried and what we found to perform better and worse, through both offline and online evaluation of the networks. We will also look at our implementations that we ported over to a physical drone and performed the aforementioned targeted landing algorithms.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Alex Waibel, for providing me with the guidance necessary to undertake this project. He gave me the tools, expertise and ideas, whenever they were needed, to keep the project moving in the direction we wanted it to go.

I would also like to thank Dr. Dave Touretzky for agreeing to be a part of my thesis committee, attending my presentation, and providing me with valuable feedback that was incorporated into the final version of my thesis.

Huge thanks go out to Peter Steenkiste and Tracy Farbacher for guidance through this Master's program and keeping me well informed.

A final thanks goes out to my family, especially my parents and brother who always continue to support me and believe in me even in the toughest of times.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Direct Methods 2

1.3 Proxy Through Displacement 2

2 Algorithms **5**

2.1 Network Architecture 5

2.2 Direct Control Prediction 5

2.2.1 Network 5

2.2.2 Data Collection 6

2.2.3 Adding Context 7

2.2.4 Evaluation 7

2.3 Pomerleau Inspired Methods 8

2.3.1 Network 9

2.3.2 Data Collection 10

2.3.3 Crosshair Method 11

2.3.4 Evaluation 12

2.4 Vertical Displacement 13

2.4.1 Data Collection 13

2.4.2 Evaluation 14

3 Results **15**
3.1 Direct Prediction 15
3.1.1 Adding Context 18
3.2 Pomerleau Methods 18
3.2.1 Thresholding 18
3.2.2 Crosshair 18
3.2.3 Flight Videos 23
3.3 Vertical Displacement 23
3.3.1 Flight Videos 23

4 Discussion **27**
4.1 Direct Prediction 27
4.2 Pomerleau Methods 28
4.3 Vertical Displacement 29
4.4 Putting It Together 29
4.5 Pattern Complexity 30

5 Future Work **35**
5.1 Smooth Trajectories 35
5.2 Reinforcement Learning 36

6 Conclusion **37**

Bibliography **39**

List of Figures

1.1 Example of training image showing view from the drone 3

2.1 Visualization of neural network for direct prediction of 4 directions (output labeled with left, right, forward, backward) 6

2.2 Visualization of neural network for prediction of displacement vectors . . 9

2.3 Binary mask (right) created by thresholding pixels in image (left) 10

2.4 Example image for crosshair method of size (80x45) with target center labeled as (18,15) 11

2.5 Visualization of neural network for prediction of drone height 14

3.1 X-axis. Top: Green plot showing true labels vs time, with negative for left signals and positive for right signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set. 16

3.2 Y-axis. Top: Green plot showing true labels vs time, with negative for forward signals and positive for backward signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set. 17

3.3 X-axis, adding context. Top: Green plot showing true labels vs time, with negative for left signals and positive for right signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set. 19

3.4 Y-axis, adding context. Top: Green plot showing true labels vs time, with negative for forward signals and positive for backward signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set. 20

3.5 X-axis, center of mass method. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for dataset. 21

3.6 Y-axis, center of mass method. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for back, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for dataset. 22

3.7 X-axis, crosshair method. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame. . . 24

3.8 Y-axis, crosshair method. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for backward, blue plot showing predicted outputs. Bottom: Squared error at each frame. 25

3.9 Top: Red plot showing true labels for scaled vertical displacement vs time, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set. 26

4.1 Example of Amazon smile logo being used as target, showing view from the drone, as received as input to network 30

4.2 X-axis. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame. 32

4.3 Y-axis. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for backward, blue plot showing predicted outputs. Bottom: Squared error at each frame. 33

Chapter 1

Introduction

1.1 Background

Stabilization and landing tasks are fundamentally important aspects of drones for all of the domains we mentioned including agriculture, aerial photography, fast delivery, surveillance, and many more. The importance of being able to hover over a targeted region, whether to transition into landing in that area or to be able to adjust vertical height so as to get clearer shots or get closer to a target or enlarge it in frame, can clearly be seen to be prevalent in all of these disciplines. Traditionally, we see a lot of stabilization tasks be performed simply with on board sensors and devices such as accelerometers, gyroscopes, magnetometers etc, but it is often hard to gather great performance using simple sensors while keeping low costs and minimal overhead. Thus with more simple drones, it would be a reasonable hypothesis to assume a marked benefit of incorporating information that can be gained from the visual systems already fitted on these drones, which are not necessarily being used by them as sensors in performing self-assisting algorithms.

The drone we decided to use throughout this project, which is the Parrot AR 2.0 drone, is largely trivial in nature and does not have very complex methods already built in for the aforementioned tasks of stabilization and landing. In fact this drone does not make use of its visual systems as an input sensor to perform any types of autonomous algorithms, and is simply a tool provided to the user piloting the drone. We hope to extract the most potential out of a setup such as this, in which we have full range of control when dealing with the cameras and have the ability to control any actions the drone takes as a consequence to the scene that it observes.

1.2 Direct Methods

We tried a few different approaches to tackle the problem at hand. As mentioned, we used the downward camera on board the drone, to identify and attempt to center a target region in the frame of the received stream of video. This target region was marked by some arbitrary object (like sheets of paper or a folder etc.) to provide some change in intensity from the rest of the ground around it, such that its position can hopefully be learned by a neural network. Initially, we aimed to directly learn controls for our drone in predicting the direction it should go in, using a supervised learning algorithm involving collecting data by manually flying the drone ourselves. Input data simply consisted of the image frames while our labels were the motor controls in both the x and y directions that we were giving the drone at that particular frame. Throughout this approach, we tried to keep the drone at a constant height and take this factor out of the equation.

Immediate drawbacks of this naive setup were apparent. One large element was that trying to correlate a direct relationship between image and control signals leads to ignoring any temporal aspect of the flight pattern. Thus we were not accounting for potential context around the frame that could have provided more insight into the signal that was being sent at that instant. Thus, a modification that we also tried on top of this method was to augment our input with a concatenation of specific number of frames that came prior to the current one, which we hoped would aid in helping the neural net understand things like ”slowing down” while approaching the target etc.

1.3 Proxy Through Displacement

Due to difficulties we encountered with the previous approaches, especially with learning the motor controls directly due to issues we discuss further below, we chose to try to use Dean Pomerleau's paper (Pomerleau [1994]) as a guideline to the training process. The approach implemented in this paper is designed to guide a space robot to the appropriate position over a target and thus know when it is good to perform its necessary function (applying a screw). The general process dictated here is to attempt to learn the displacement of the target from the robot device and then translate these displacement outputs from the neural network into controls to move the robot. We apply this to our work in a similar fashion, with some modifications to the ranges of displacements used and network architecture, and then having to design our own method of translating displacements into controls suitable for flying and positioning a drone.

In this initial implementation of Pomerleau's paper, we computed the displacement

Figure 1.1: Example of training image showing view from the drone

labels during training by calculating the center of mass of pixels that belonged to our target region. This was an assumption on the abilities of the drone that we preferred not to be making, and we wanted to be agnostic of when the entire target region does not appear in frame of the drone. One method in which we attempted to achieve this was introducing a recognizable and learn-able shape onto our target region. We would then collect training data in a similar way, by flying the drone through different viewpoints of the target, but our generation of labels would be done manually. We used a cross pattern as our shape, and by human observation of the center of the crosses, manually labeled our data, as well as providing best estimates for the center position when it was outside the borders of our frames. As we mentioned earlier, all of these approaches were done while keeping the drone at a constant and stable height from the ground.

As an advancement to the steps mentioned previously, and to attempt to combine 2-dimensional x and y positional adjustment with a vertical adjustment in the z-direction to guide the landing process, we also decided to train a network to predict the height at which the drone was hovering. We normalized a range of possible heights we were considering and using centralized frames at a few selected heights, trained a network to predict this vertical displacement. We then used this in combination with our previous implementation to put together an effective landing algorithm.

Chapter 2

Algorithms

2.1 Network Architecture

We chose to use deep learning and neural networks to perform our algorithms, since they are less fragile and more reliable than classical vision algorithms. For all of the algorithms we will discuss, we used a fully connected neural network with Sigmoid activation functions in the hidden layers, and a ReLU activation in the output layer to scale our output values appropriately between $[0, 1]$. We minimized a mean squared error loss function using SGD, using a learning rate of $\alpha = 0.01$ and a momentum of 0.1. The differences in architecture between the algorithms are limited to aspects outside of the ones above, specifically varying in number of hidden layers, number of hidden units per layer and dimensions of the input and outputs in our network.

2.2 Direct Control Prediction

Our first idea was to train a neural network to go from an input image to outputting the motor control needed to be given to the drone to direct it to the center of our target.

2.2.1 Network

Our network for regular direct prediction without use of context, consisted of an input layer, 3 hidden layers and an output layer. Input images were of the size 64x36 and the number of units in each layer following the input was $36, 16, 8$ respectively, with 4 units

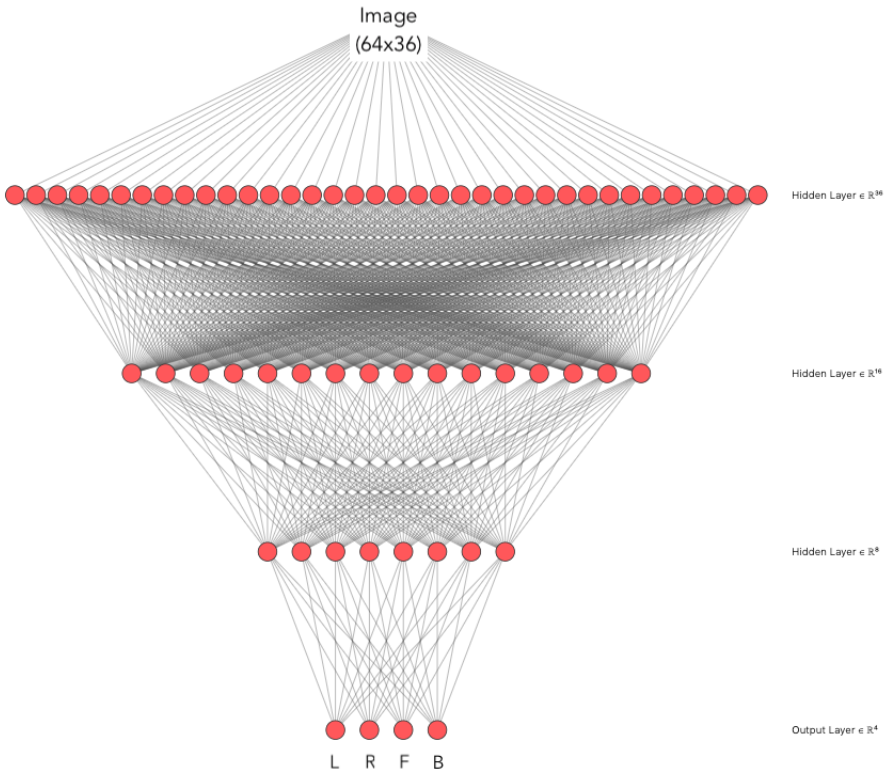

Figure 2.1: Visualization of neural network for direct prediction of 4 directions (output labeled with left, right, forward, backward)

in the output layer (for each possible direction). The network can be seen to be visualized in figure 2.1.

2.2.2 Data Collection

Our setup included a blank sheet of paper on the floor and manually flying our drone to collect the training data. We controlled the drone to keep the target (paper) inside the frame of its downward facing camera, and kept the drone at a constant height throughout the process. We then saved each captured frame during flight, to disk, along with the control signal that we were giving the drone at that frame's instant. We were sending these controls through joystick control and their values fell in a continuous range of $[-1, 1]$ for each of the x and the y directions (values from $[-1, 0]$ represent going left or forward for x and y respectively, while values from $[0, 1]$ represent right or backward, with higher

values depicting a stronger force in the respective direction). The signal values were then translated into 4 separate outputs, corresponding to left, right, forward and backward. Thus each output value was scaled to be in the range $[0, 1]$, where the opposite direction to the true label for each axis, at each frame, was set to 0 (i.e. if the signal in the x direction was < 0, the output corresponding to "right" would be set to 0 and vice versa, while doing the same in the y direction as well). During evaluation, the max between the 2 pairs of (left, right) and (forward, backward) was taken to be the output in each axis. We went through 4 iterations of collecting datasets, trying not to incorporate too much noise into the labels by being very deliberate and careful with the signals we sent to the drone.

2.2.3 Adding Context

As we mentioned above, a major aspect that made this learning task difficult was ignoring any temporal aspect of the flight pattern. We were not accounting for potential context around the frame that could have provided more insight into the signal that was being sent at that instant. This caused issues with in some cases, image frames being labeled with 0 signals in all directions because when flying the drone, the signal is typically given only for a short duration to let the drone's momentum, after providing a burst in a particular direction, carry it to the desired location. We hoped providing context of previous frames would then help alleviate this issue since the network could use the fact that it is getting closer to the target and thus not need to continuously apply the signal, making more sense out of these "0" labels. We hoped to smooth out the drone's signal outputs in general using this tactic, and hoped it would learn a more gradual control mechanism, especially as it was approaching the target (i.e. the target was approaching the center of the drone frame). We therefore augmented our input data to use not just the current frame for each input example, but also 4 of the frames that came immediately before it. In this way, the input dimensions were scaled by 5x and we scaled the number of hidden units for each hidden layer by the same factor, leaving the output layer to contain the same 4 output units. We hoped this would be a more efficient and quicker way to test if the network could be modified to work better in the style of a recurrent net, and perhaps give us the confidence to implement something in a fully recurrent fashion, if so.

2.2.4 Evaluation

We plotted graphs to look at the true labels and compare them against the predicted outputs. We also looked at the mean squared error between the two, and we discuss the graphs in the Results section. We also tried to test this implementation by using the network's

predicted outputs in real time while flying the drone, and giving the drone signals during flight. However, we did not produce anything of value that would be worth discussing further.

2.3 Pomerleau Inspired Methods

Due to shortcomings of our attempt at direct prediction of controls using the supervised flight learning method, which we discuss in more detail in the Discussion section, we decided to take another approach to tackling this problem. The motivation, logic and results of applying this method in the original context of a space robot is explored by Pomerleau in his paper (Pomerleau [1994]) but we will provide an overview of the adaptation of the algorithm for the purposes of our problem. Essentially, in this method, we wished to predict the displacement of a particular target from the center of the image frame of our drone's downward camera. Once we could extract such a displacement, we needed to figure out the best way to translate this into control signals that could be sent to the drone, when evaluating the implementation online through drone flight. This was in contrast to our previous method where we were trying to directly predict the control signals themselves. This alleviated the difficulties in data collection, as we discuss below.

The generation of our network labels were two-fold. First, we devise a method to get the x and y coordinates of the center of the target, and then use these coordinates to compute the displacement and generate a label using methods described in detail, later in this section. In our initial version, we were computing centers of our target using vision algorithms to threshold the image to isolate the target region and calculating the center of mass of the thresholded pixels. However, this makes assumptions about the neural net's ability to learn this directly from the image, and this becomes difficult in situations where the target isn't completely in frame. Moreover, this isn't a particularly interesting task by itself since classical vision algorithms would also be able to just compute a thresholded image, taken from the drone's camera, and directly calculate the displacement of the target. Thus we would prefer to learn something more tangible such as recognizing the presence of a pattern in the landing region (which we represented with a cross), as well as be able to represent target centers that are not bounded by the field of view of our input frame. This involved manual labeling of the centers as we discuss below. We evaluated both the above implementations through plots comparing true and predicted displacements as well as qualitatively through drone flight.

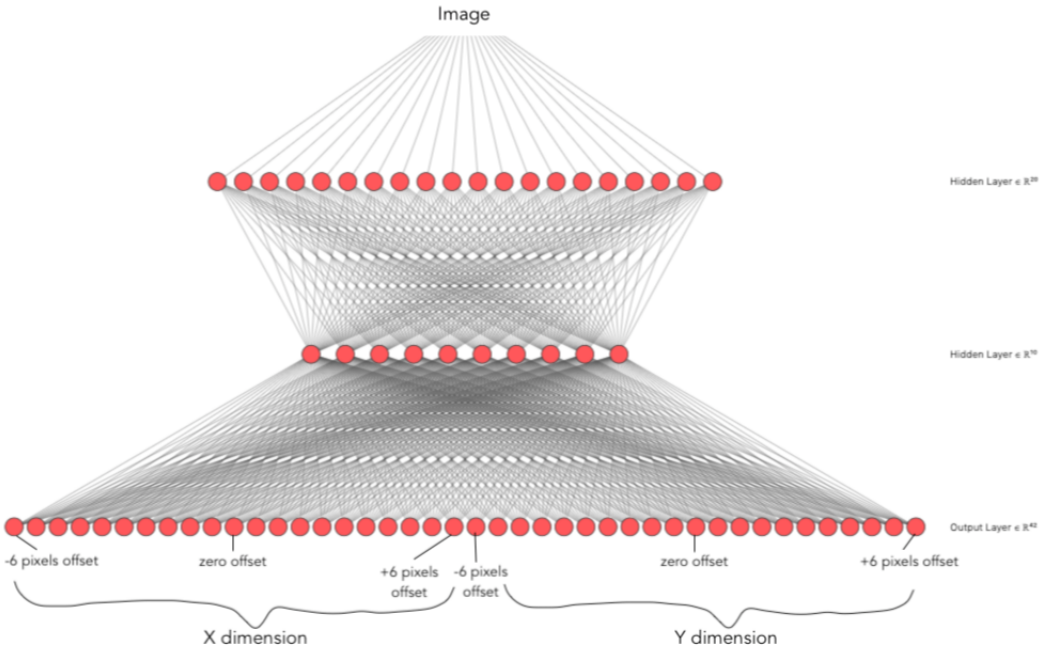

Figure 2.2: Visualization of neural network for prediction of displacement vectors

2.3.1 Network

As well as following the general architecture highlighted at the start of this section, we used Pomerleau's layout for the network layers as a guideline, while augmenting it somewhat with our own observations of results. Our network for prediction of displacements consisted of an input layer, 2 hidden layers and an output layer. Our input images were scaled to be of size 24x20, and the number of hidden units in each following layer was 20 and then 10, with 42 units in the output layer. This number of units was to model the representation described by Pomerleau. We had the first 21 units as a linear representation of the displacement of the drone relative to the target center in the X dimension, ranging from -6 pixels for the leftmost output unit to $+6$ pixels for the rightmost output unit. The second set of 21 units is identical to the first except that it represents displacement in the Y dimension. This was essentially Pomerleau's proposed setup with some slight modifications in values (to better fit our range of displacements etc.) and an additional hidden layer which we found to give much better results with our data. We see the network described here, visualized in figure 2.2.

After completing a forward pass through the network, the networks estimate of the

Figure 2.3: Binary mask (right) created by thresholding pixels in image (left)

drone's X and Y displacements is extracted from the output vector. The displacement indicated by the network in each dimension is taken to be the center of mass of the "hill" of activation surrounding the output unit with the highest activation level. Using the center of mass of activation instead of the most active output unit when determining the estimated displacement permits more precise estimates, and therefore improves the networks performance, as highlighted by Pomerleau (Pomerleau [1992]).

2.3.2 Data Collection

Similarly to our previous implementation, we collect data by simply flying the drone around a target. This time however, the first method required us to ensure that the center of the target was always visible in frame of the drone's camera. Additionally, one aspect that we did not need to worry about in this case was recording the control signals we were sending to the drone during flight. In fact, it was even possible to collect the data by simply holding up the drone and moving it in a fixed 2-dimensional plane (trying to keep it at a constant vertical height), and adhering to keeping the target in frame.

After getting the input images we need, we generate their labels by first thresholding their gray-scale image to expose the pixels above a specific intensity (shown in figure 2.3), and computing the center of mass of the pixels found in this computed mask. We then compute displacements in both axis by taking the difference between the computed center of mass and the center of the image frame. This gives us x and y values that we then translate into our labels by using a Gaussian distribution approximation which depicts a peak at the cell corresponding to the closest displacement to each value. For this we use 42-length vectors, the creation of which is described in the evaluation section below.

Figure 2.4: Example image for crosshair method of size (80x45) with target center labeled as (18,15)

2.3.3 Crosshair Method

As we mentioned, a particular drawback of this naive computation of center of mass, is the target center appearing in the limited bounds of an image frame. Another shortcoming of the implementation was the seemingly simple learning task that this presented, in that it could potentially be achieved by a classical vision algorithm on the fly, and hence we wanted to make it more interesting by having the network try to learn or recognize some kind of pattern or shape. We chose to mark our target with a large cross pattern, mostly for our own benefit of generating labels by marking the target centers, but also potentially providing the network with another source or indicator of target position.

The method we used for data collection was identical to our initial implementation, except that we wanted to allow the target center to not always be in full view of the drone camera. Hence, the extraction of the coordinates of the center was not done by thresholding the image, but instead by manually labeling the images by using the mouse pointer to select the center of our cross pattern for each example. In this way we also accounted for centers appearing outside the bounds of the image, by using negative coordinates or coordinates exceeding the size of our dimensions. An example of such an image with the new target design is shown in figure 2.4. Once we had the centers, the method for generating the label followed exactly as in our original implementation, by computing the displacement and creating an appropriate vector.

One thing we did need to factor in, in this case, was using slightly larger input images

so that we could see the cross pattern more clearly, and thus our inputs were of dimension 80x45. We then also adjusted the number of hidden units in our first hidden layer to be 36, but kept the number of units in the last two layers the same. While we enlarged the images slightly to add this aspect of clarity, we did not want to increase it too much, to avoid the risk of multiple unwanted possibilities. These included not increasing the training times of our models, not increasing the space requirement for our models, and most importantly, not increasing the inference time for our model, since our goal was still to run our implementation on a physical drone during flight, as close to real time as possible.

2.3.4 Evaluation

The network is trained to produce a Gaussian distribution of activation centered around the actual displacement in each dimension. As in the decode stage, the actual displacement may fall between the displacements represented by two output units. The following approximation to a Gaussian distribution is used to precisely interpolate the target output activation levels:

$$x_i = e^{-\frac{d_i^2}{10}} \tag{2.1}$$

where x, represents the desired activation level for unit i, and d_i is the i^{th} units distance from the correct steering direction point along the output vector. The constant 10 in the above equation is an empirically determined scale factor that controls the width of the Gaussian curve. The above equation corresponds to a normal distribution with a standard deviation $\sigma = \sqrt{10}$.

As an example, consider the situation in which the actual drone displacement along one dimension falls halfway between the displacements represented by output units j and $j+1$. Using the above equation, the desired output activation levels for the units successively farther to the left and the right of the correct displacement value fall off rapidly, with the values $0.98, 0.80, 0.54, 0.29, 0.13, 0.04, 0.01$, etc.

By requiring the network to produce a probability distribution as output, instead of a "one of N" classification, the learning task is made easier, since slight changes in the drone's position relative to the target require the network to respond with only slightly different output vectors. This is in contrast to the highly non-linear output requirement of the "one of N" representation, in which the network must significantly alter its output vectors (from having one unit active in each vector and the rest off to having a different

unit active in each vector and the rest off) on the basis of fine distinctions between slightly shifted images of the target.

Thus we used this Gaussian approximation to generate our labels and then plotted graphs to compare the true displacements against the predicted displacements for each frame, extracted using the center of mass of each set of 21 output units. This was done for both implementations of the thresholding and crosshair methods. We then also performed similar online evaluation as with the previous implementation, through the use of drone flight to see how well the new methods were able to guide the drone. This was an experimental, qualitative test, and we recorded a video of the flight, which we also discuss in the Results section.

2.4 Vertical Displacement

A final task we tackled was allowing the drone to learn the height at which it is hovering at by observing the scale at which the target appears in the frame. We did this in a quick and naive fashion by simply training a neural network to predict a value between $[0, 1]$ based on the image frame it received as input. We used this range of values to represent a set min and max height specific to our task. We can see the network visualized in figure 2.5.

2.4.1 Data Collection

We collected data for this task by first measuring the height that we wanted the drone to be at and then simply holding up the drone such that the target was centered in the frame of its camera, and shifting it ever so slightly in this centered position. Our aim with this approach was that the drone would first use the network trained in the above implementations to center the target by moving in the x and y dimensions, and then use this network to lower itself in its centered position, until it reaches a height that it can deem itself "safe" to land. We defined the range of heights that correspond with the range of labels $[0, 1]$ by the range $[85cm, 135cm]$, which we just devised from experimentation and observation. Thus the goal was to centrally position the drone with the target, at a height of 85cm, before giving it a signal to perform landing. To collect our data, we held the drone at three different heights, specifically at 85cm, 115cm and 135cm.

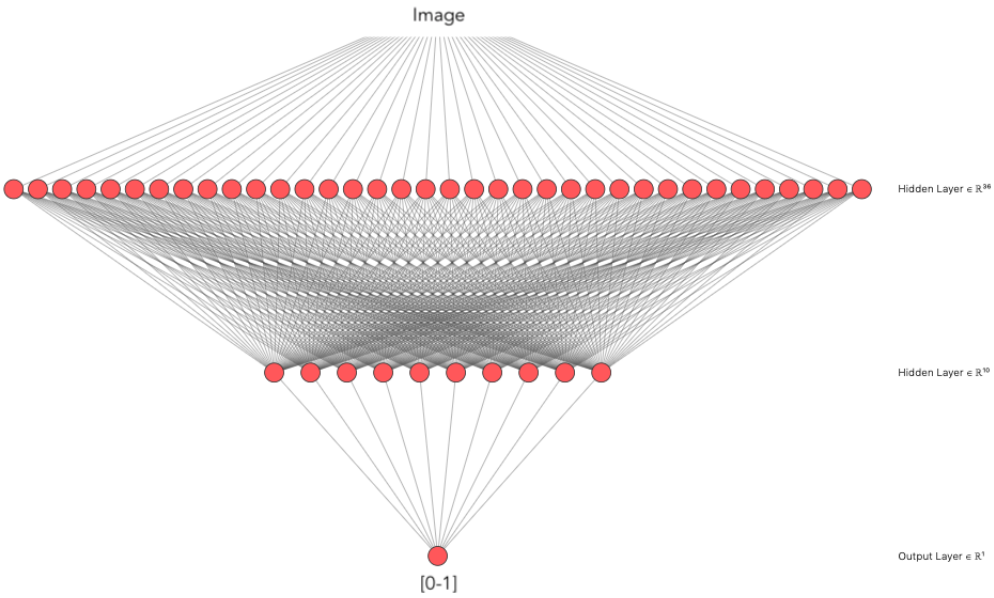

Figure 2.5: Visualization of neural network for prediction of drone height

2.4.2 Evaluation

We plotted graphs to compare the true labels of the (scaled) height of the drone against the predicted height and we also flew the drone, starting from a centered position, to try to get it to lower itself to the appropriate height before performing landing. Additionally, we also recorded a video of independently running this network while the drone was flying with the target centered, and lowering itself to an appropriate height before landing.

Chapter 3

Results

For all of the graphs plotted in this section, the x-axis represents a particular data point which is the image frame being evaluated by the network, while the y-axis is the output specific to the model, i.e. being either the control signal or the displacement in a particular direction/axis, for either the direct methods or the Pomerleau methods respectively.

3.1 Direct Prediction

We split our dataset in a $65/35$ ratio, meaning out of 690 examples, 450 were used for training and 240 for testing. Figures 3.1 and 3.2 compare the performance of our network for the base direct prediction method. The green plots are the true labels for the control signals, where the respective pair of signals (left/right) or (forward/backward) has been represented in graph format through a continuous value between $[-1, 1]$ to better reflect how these controls are actually received by the drone. The negative values then reflect either left (Figure 3.1) or forward (Figure 3.2) directions to the signals while the positive values reflect right or backward respectively. The yellow plots are then the predicted outputs by our network, in which the cumulative output signal has been calculated by taking the maximum between the values of the signals at each frame, and adding a negative sign if necessary, depending on which signal the maximum comes from.

The bottom plots in red then represent the corresponding squared error between the true labels and the predicted labels at every frame (time-step), and are titled with the mean squared error for this entire test set.

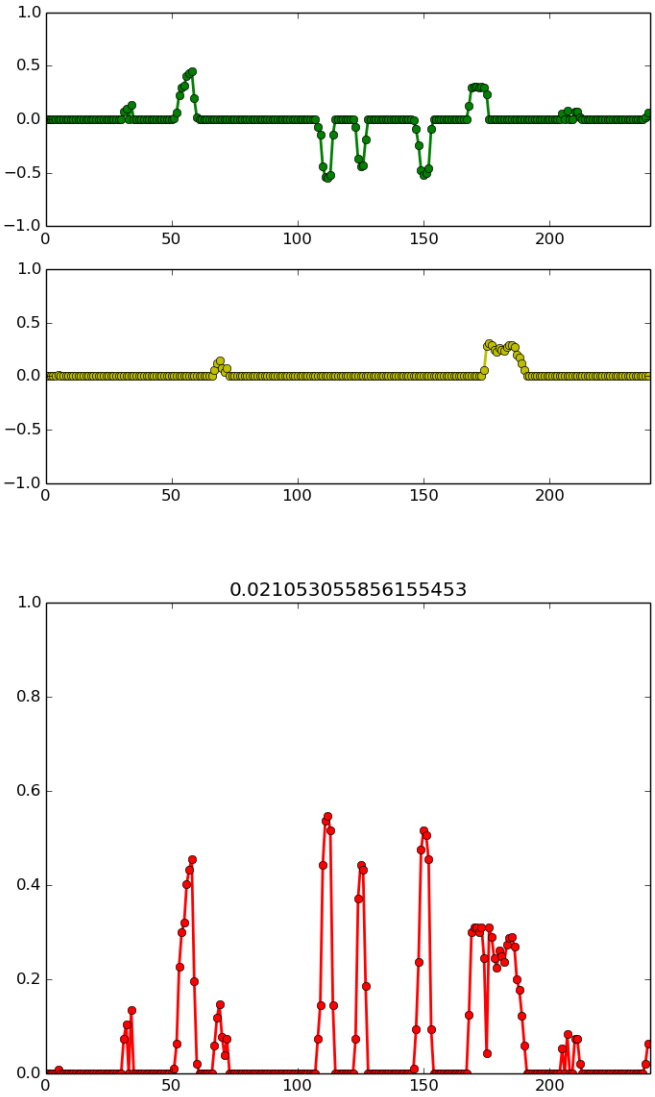

Figure 3.1: X-axis. Top: Green plot showing true labels vs time, with negative for left signals and positive for right signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set.

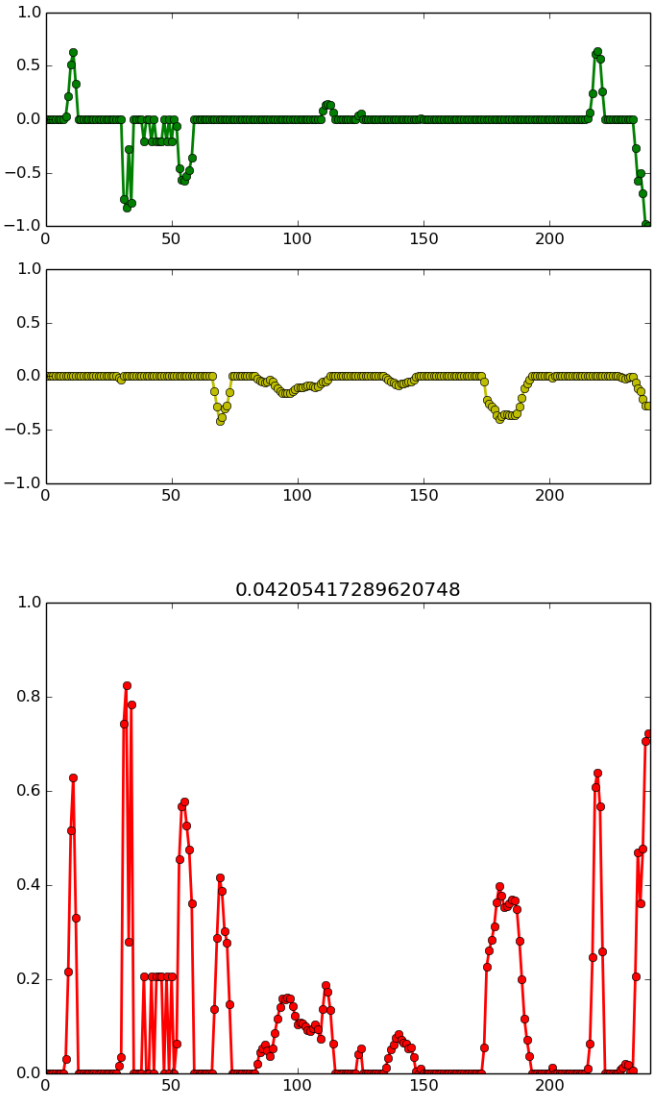

Figure 3.2: Y-axis. Top: Green plot showing true labels vs time, with negative for forward signals and positive for backward signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set.

3.1.1 Adding Context

After implementing the modified "recurrent" type algorithm, by adding context using immediately previous frames, we plotted the graphs as above, and the x-axis results can be seen in figure 3.3, and the y-axis results in figure 3.4.

3.2 Pomerleau Methods

3.2.1 Thresholding

We plotted the graphs for the true displacements, which we computed by thresholding our image and isolating the white pixels, and then calculating the center of mass, and subtracting this from the center of the image frame. In this method, our images were of size 24x20, and given our dataset, the range of displacements was devised to be between $[-6, 6]$. Negative displacements represent the target being to the left or forwards (depending on axis) of the drone's current position, while positive displacements represent the target being to the right or backwards. The predicted labels were computed with the method described earlier, in that we took the center of mass of each half of the output vector - the first half representing the x-axis and the second half being the y-axis. In figure 3.5, in the top graph, we see the true displacements in the red plot and the predicted displacements in the blue plot, for the displacements in the X-axis. The same is plotted for the Y-axis, as can be seen in figure 3.6. Both figures also include the corresponding squared error at every frame (time-step), in the bottom graphs, which are titled with the mean squared error of the entire set.

3.2.2 Crosshair

Similarly to above, we plotted the graphs for the true displacements, which we labeled by manually using the mouse pointer to click in the center of the cross pattern for every example (described earlier), against the predicted displacements which were obtained by the same extraction method using the centers of mass of the output vector. In this method, our images were of size 80x45, and given our dataset, the range of displacements was devised to be between $[-35, 35]$. In figure 3.7, in the top graph, we see the true displacements in the red plot and the predicted displacements in the blue plot, for the displacements in the X-axis. The same is plotted for the Y-axis, as can be seen in figure 3.8. Again, both figures include the corresponding squared error at every frame (time-step), in the bottom graphs.

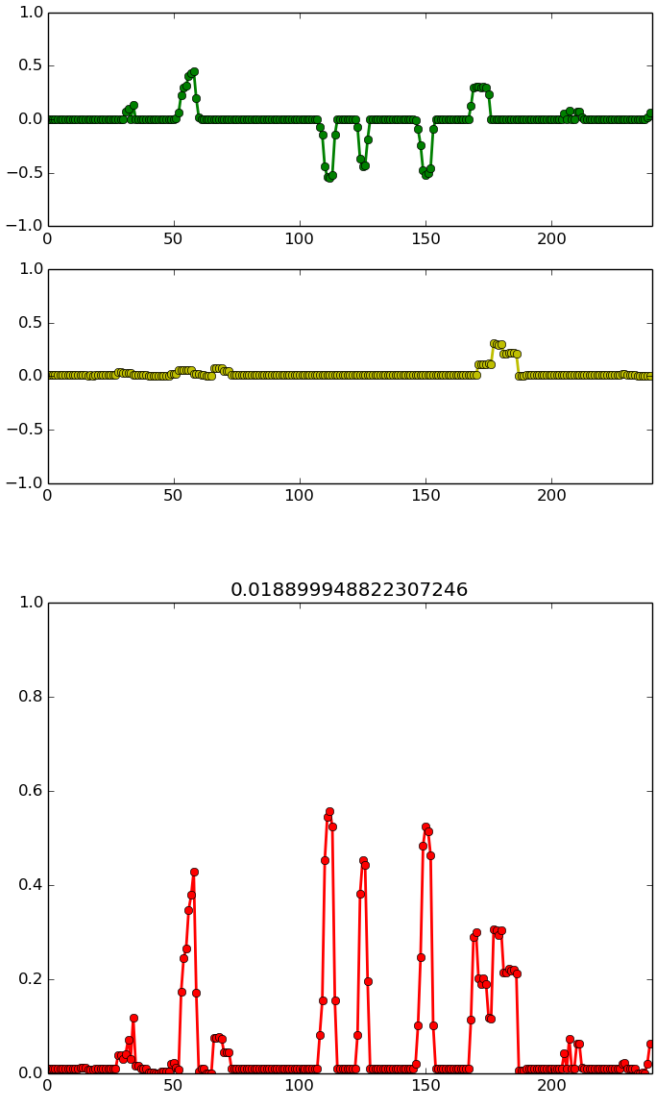

Figure 3.3: X-axis, adding context. Top: Green plot showing true labels vs time, with negative for left signals and positive for right signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set.

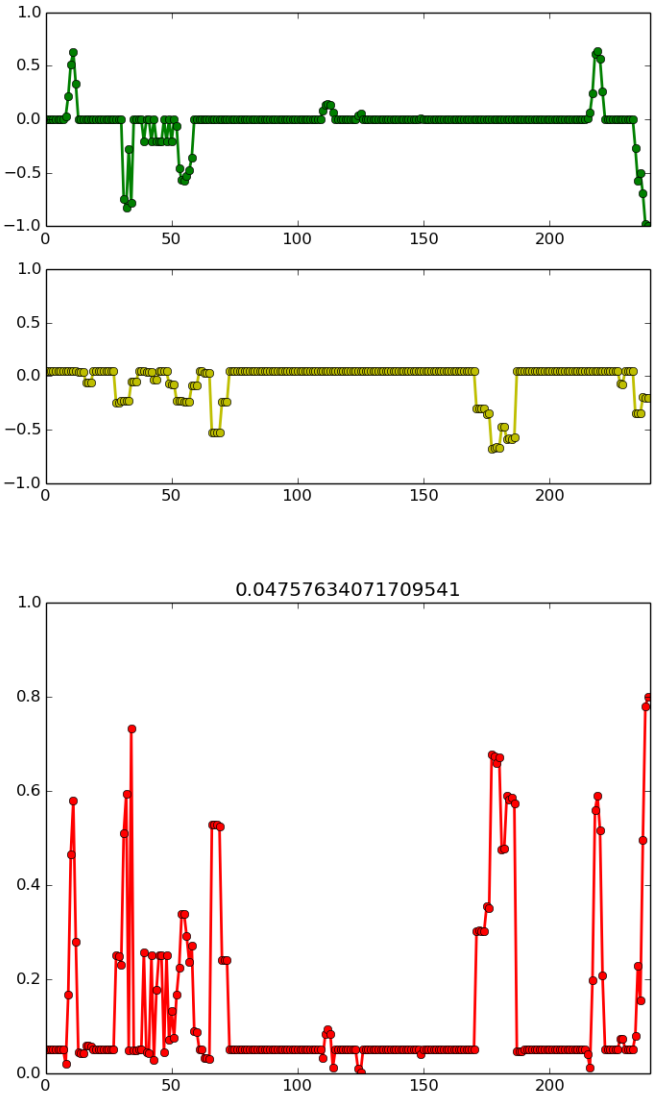

Figure 3.4: Y-axis, adding context. Top: Green plot showing true labels vs time, with negative for forward signals and positive for backward signals, yellow plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set.

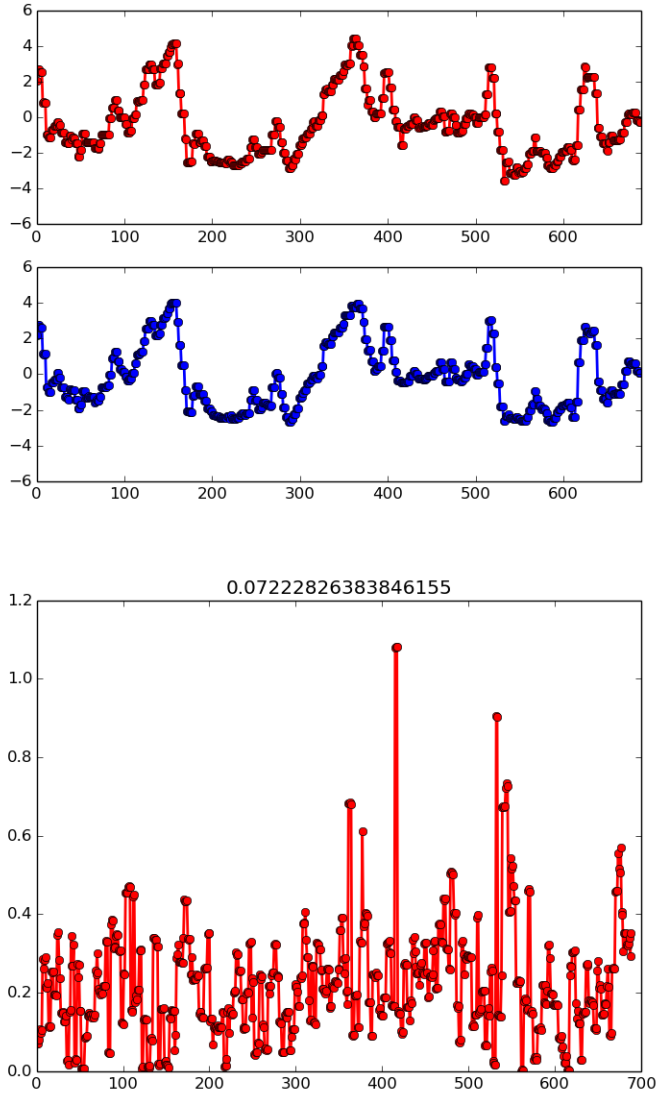

Figure 3.5: X-axis, center of mass method. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for dataset.

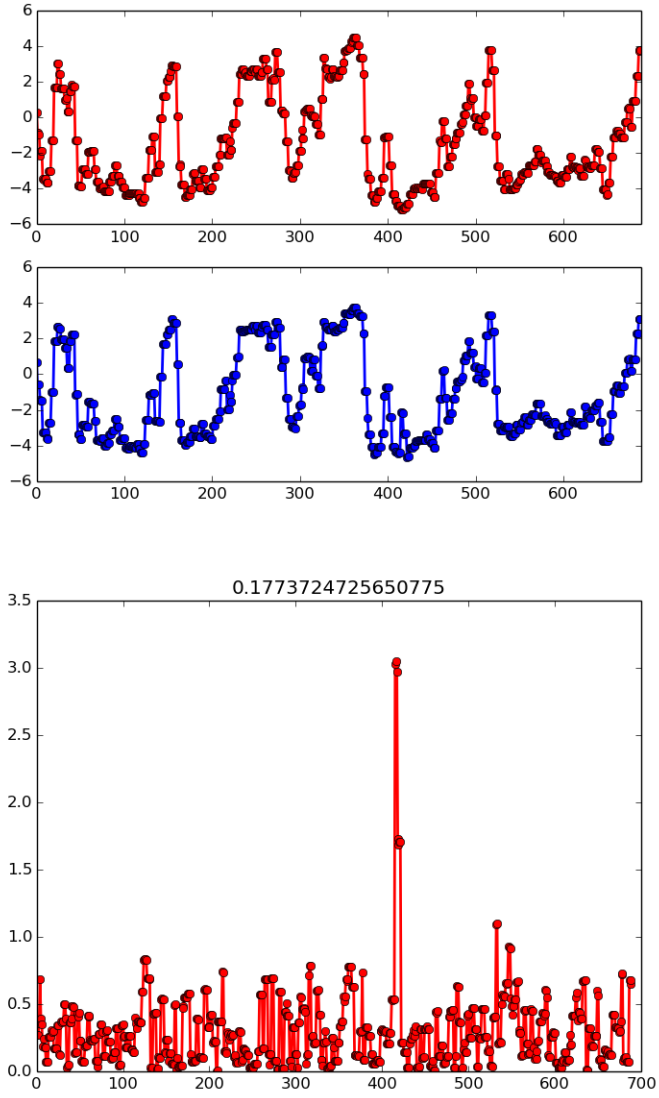

Figure 3.6: Y-axis, center of mass method. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for back, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for dataset.

3.2.3 Flight Videos

We additionally tested our implementations by porting the model over to our physical drone, and qualitatively observed the effectiveness of the model on drone flight. The video for the thresholding method can be found here (Lahiry [2019a]) and the video for the cross hair method can be found here (Lahiry [2019b]).

3.3 Vertical Displacement

We plotted graphs for our test set (where our data was split 75/25 and thus we have 150 test images out of 600), in figure 3.9, to show, in the top graph, the true height labels in the red plot and the predicted heights in the blue plot. The data was collected at different heights while the target was centered in the drone's frame. The heights were scaled from a range of $[85cm, 135cm]$ to values between $[0, 1]$. The bottom graph in the figure also shows the squared error at every frame and is titled with the mean squared error of this test set.

3.3.1 Flight Videos

As before, we additionally tested our implementations by porting the model over to our physical drone, and qualitatively observed the effectiveness of the model on drone flight. The video can be found here (Lahiry [2019c]).

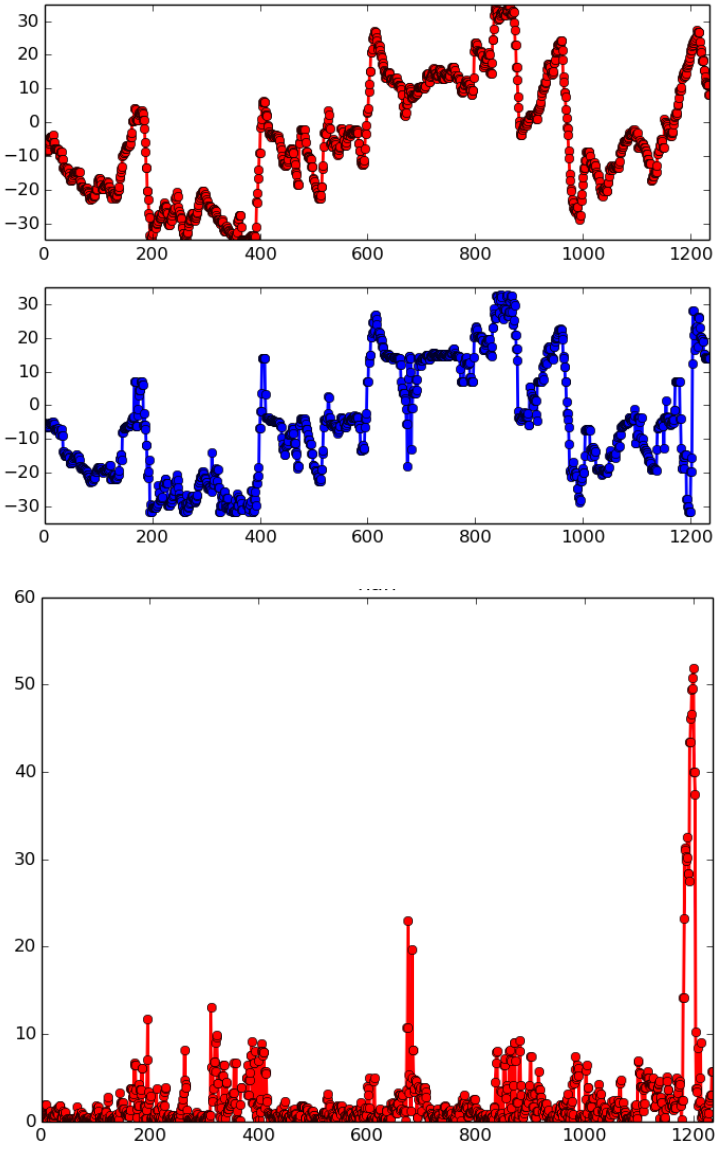

Figure 3.7: X-axis, crosshair method. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame.

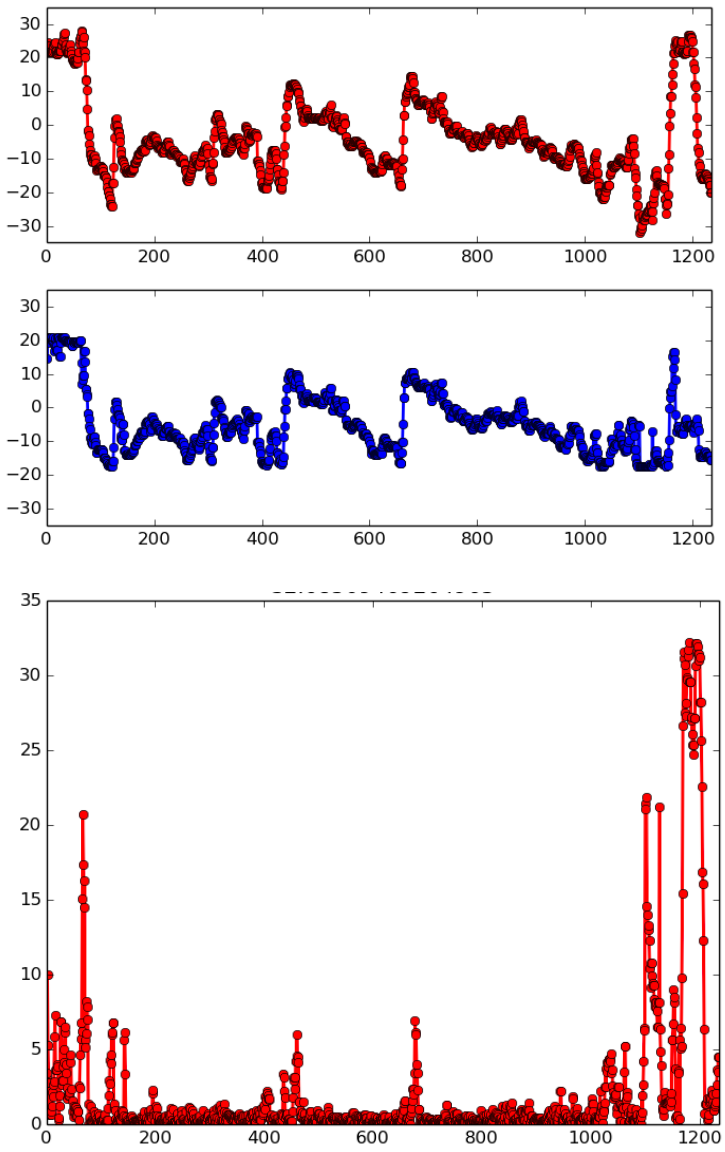

Figure 3.8: Y-axis, crosshair method. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for backward, blue plot showing predicted outputs. Bottom: Squared error at each frame.

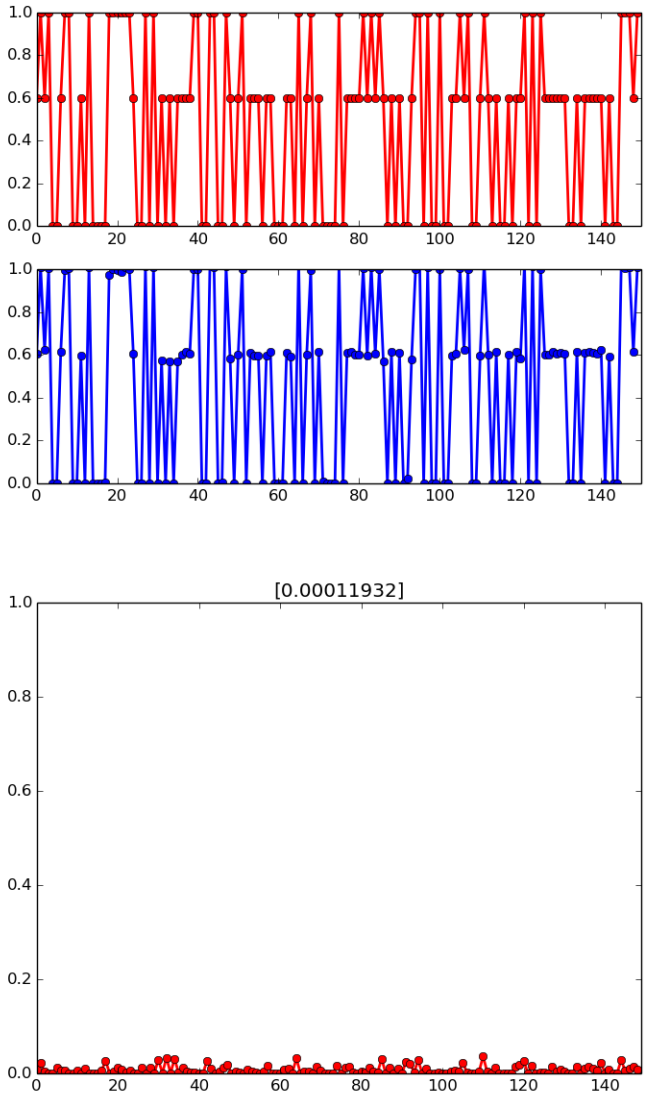

Figure 3.9: Top: Red plot showing true labels for scaled vertical displacement vs time, blue plot showing predicted outputs. Bottom: Squared error at each frame, titled with mean squared error for test set.

Chapter 4

Discussion

4.1 Direct Prediction

We found a little bit of success in our initial implementation of predicting control signals directly (figures 3.1-3.4). Looking first at the results for the implementation without context, we first see that in the plot for the x-axis (figure 3.1), the network only predicts outputs in the positive values, or ”right” direction, and never in the ”left” direction. We do see some semblance of the trend of the signals in the direction that it does predict, since we see peaks in areas approximately around the peaks for the true labels, but the predictions seem misaligned and somewhat delayed. Looking at the error curve, of course we see that the error peaks are at the regions of providing the strongest signals, and especially high in the regions when the negative or ”left” signals should be predicted. Moving to the plot for the y-axis (figure 3.2), we again see that the network does not predict signals in one direction, this time in the positive or ”backward” direction. The control signals are also highly misaligned and there are many unnecessary bumps in the predictions in the middle region, which would need to be smoothed out.

Next we observe the plots for when we augment our input data with previous frames, and see the plots for each axis in Figure 3.3 and Figure 3.4. We are using the same test data as with the non-contextual implementation so we can see the changes in the network's behavior more easily. Figure 3.3 shows us very little change in predicted behavior as far as X-axis signals, as the peaks remain in similar regions, and the negative or ”left” signals are still not being predicted. Figure 3.4 is slightly more interesting in that firstly, the predictions for the negative or ”forward” direction surrounding the 50 frame mark are definitely more representative of what we expect given the true labels. Additionally, the

middle region (between frame 75-150), which contained a lot of over-predicted bumps in the model without context, has been smoothed out to reflect the true labels better. However, we of course do see that there are still areas of incorrect predictions, as well as the fact that the positive or "backward" signals are still not being predicted. This was something we kept in mind as we thought of how to change our approach following this implementation.

Hence, we see that we encounter many issues with learning drone signals directly. Although we were slightly encouraged by the trends of the networks at times, there were too many drawbacks and obstacles in this process. A particular point of difficulty was the labor intensiveness and inaccuracy involved in data collection. On top of having to collect our own data through manual flight of the drone, we needed to make sure that the controls we were sending the drone corresponded to the direction it needed to go, at basically all times during the flight. This was important since the control signals needed to match up as the labels of the images that were being received as input frames, and thus any lapses in the correct signal being sent would be harmful to the learning ability of the neural net. Therefore, this led to both inaccuracies in labeling, as well as many "0" labels since the controls we were sending only needed to be short bursts, which carried the drone using the momentum from its acceleration, rather than being continuously sent until the drone reached the desired position. We found that this was something that even adding contextual information could not rectify due to the large amount of noise introduced in the data collection process. This inclined us to look to Dean Pomerleau's paper (Pomerleau [1994]) for inspiration.

4.2 Pomerleau Methods

We experienced a lot more success when using Pomerleau's ideology as a basis for our implementation. Figures 3.5 and 3.6 show the resulting plots for the thresholding algorithm, showing the comparisons in the top graphs, of true displacements in red and predicted displacements in blue, for the x-axis and y-axis respectively. We see that the predicted outputs mirror the true labels almost perfectly in both axis. The bottom graphs show the squared error at every frame (time-step) and are titled with the mean squared error for the whole set, in each respective axis. As we can see, the mean squared error was reduced to very small quantities for both axis, and thus the network learned this task pretty well. This is to be expected however, since as we mentioned earlier, this task could also potentially be achieved by classical vision algorithms, but it was encouraging to see positive results to incline us to continue along this path.

We then attempted the more difficult and interesting task of trying to learn some sort of

pattern or shape more directly, generating labels through manual labeling using the mouse pointer. We again plotted the true displacements against the predicted outputs, using a larger range of displacements due to our slightly larger input images. These results also looked fairly promising, and for the most part also mirrored the true labels at any given frame. We can see the largest areas of our error by examining the squared error curves and observing the peaks. For the x-axis results (figure 3.7), there was a slight error for the frames slightly after the 600th frame mark, where the predictions seem to be much lower values than the true displacements, and even flipping directions for a handful of cases. A similar case happens when the predictions dip drastically around the 1200th frame mark. Examining these cases, they seem to be caused by either tricky lighting situations that confuse the network, or by cases where very little of the target is currently visible. The y-axis results (figure 3.8) fare pretty well too, the only real issues coming around the 1100th frame where the predicted displacements aren't quite low enough (or large enough in the forwards direction) followed by a peak in the true labels, where the predictions aren't quite high enough. These frames were around the same ones which caused problems with the x-axis and so the reasoning here is similar to the previous.

4.3 Vertical Displacement

This task turned out to be easily learn-able for our network, once we decided to only worry about learning the height of the drone from a centered state. Our goal in this method was to be able to predict the height once the target was centered using an implementation from above, and then the drone to lower itself in a stable state, to the lowest height in our range of consideration, and then perform the landing algorithm.

As we see in Figure 3.9, our network's predictions mirror the true labels very well with only slight perturbations at each example frame, but never being particularly wrong. The performance of this method is comparable to the results we got from the thresholding version of displacement learning. As we see from the squared error curve, the errors are almost negligible and the MSE has been minimized drastically. Thus this gave us encouragement in the possibility of performing these steps that lead to a smooth landing.

4.4 Putting It Together

As a final step, we combined the results from the crosshair method and the vertical displacement method to experimentally achieve a complete pipeline for the stabilization and

Figure 4.1: Example of Amazon smile logo being used as target, showing view from the drone, as received as input to network

landing of a drone given a target. We did this by first computing a control signal by linearly scaling our displacement by our max displacement ($\textbf{signal} = \frac{\textbf{predicted disp}}{\textbf{max disp}}$). This gave us a continuous value between $[-1, 1]$, and then an experimental procedure that worked well was to send these controls to the drone for 5 frames, followed by pausing for 5 frames. The purpose of this was essentially to counteract the possibility of the drone over-shooting our target, which was a recurring problem in our direct prediction approaches. Consequently, we would then continuously check for the event that both our X and Y displacements were below a certain threshold, and then use our vertical displacement network to reduce the height of the drone if so. Once we reached a height that was small enough, we could then choose to perform our landing algorithm as we deemed it "safe" to do so. The outlined procedure worked pretty well and can be found conducted in these videos: (Lahiry [2019d]) (Lahiry [2019e]) (Lahiry [2019f]). As we can see from this, it is effective to take the approach of predicting displacements instead of control signals directly, since it gives us more flexibility with aspects like the frequency with which we send controls to the drone.

4.5 Pattern Complexity

One concern we had with the previous methods was the potential lack of complexity for the pattern being learnt by the network. We were wary of a possibility for the network to

be learning the general rectangular shape of the target rather than focusing on the cross pattern that we augmented the target with (which it could potentially just be treating as noise). To attempt to build a more robust model that could learn a more complex shape we decided to replicate our previous experiments with the Amazon smile logo, which can be seen in Figure 4.1.

We then plot the true and predicted displacements in Figures 4.2 and 4.3, for the X and Y axis respectively. In these preliminary experiments, we see similar levels of success with this new target pattern as with the previous results of our crosshair method. There are definitely more regions of high error with the more complex pattern but the main source of error (found in the peaks of the MSE curves just after the frame 800 mark) are caused due to similar reasons as the errors in our original crosshair method itself. This is largely because of the target appearing occluded by being partially outside the bounds of the frame, which is an even trickier problem with the new pattern since it is much more indistinguishable in partial views of the target. Therefore it is even more important for the drone to be able to see the entirety of the target in this case, which is an aspect of potential improvement in this method.

However, these results do give us confidence and encouragement in the fact that we are able to learn complex patterns or shapes with regards to our landing target (and are not dependent on the regularity of the rectangular shape of our previous target), and hence this is a rich area for exploration of these kinds of algorithms.

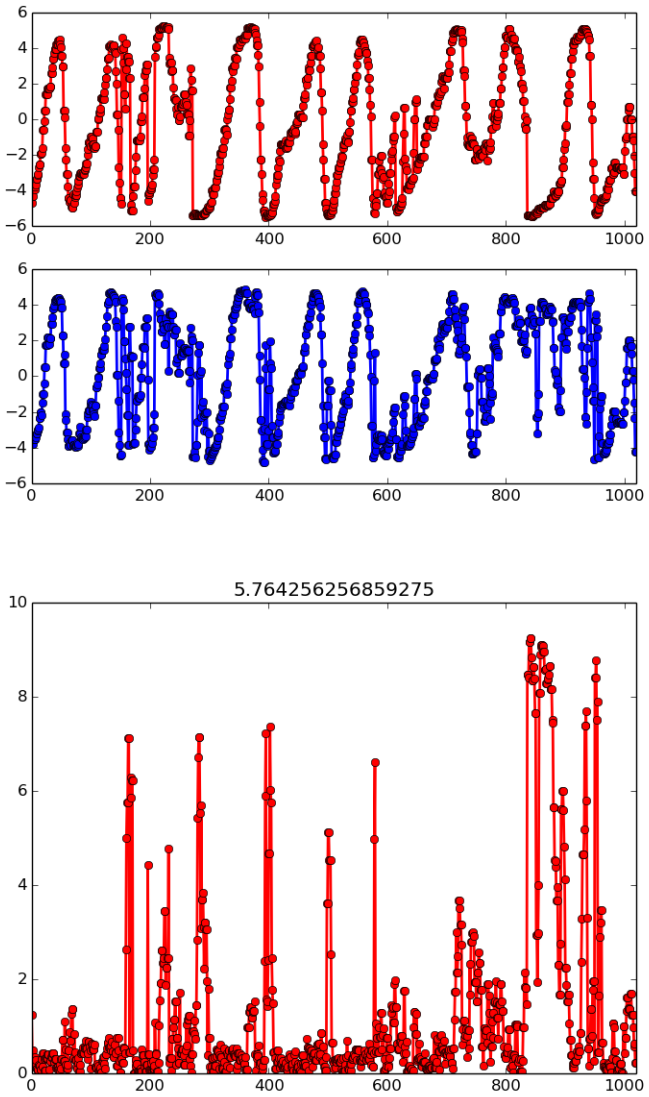

Figure 4.2: X-axis. Top: Red plot showing true labels for target displacement vs time, with negatives for left and positives for right, blue plot showing predicted outputs. Bottom: Squared error at each frame.

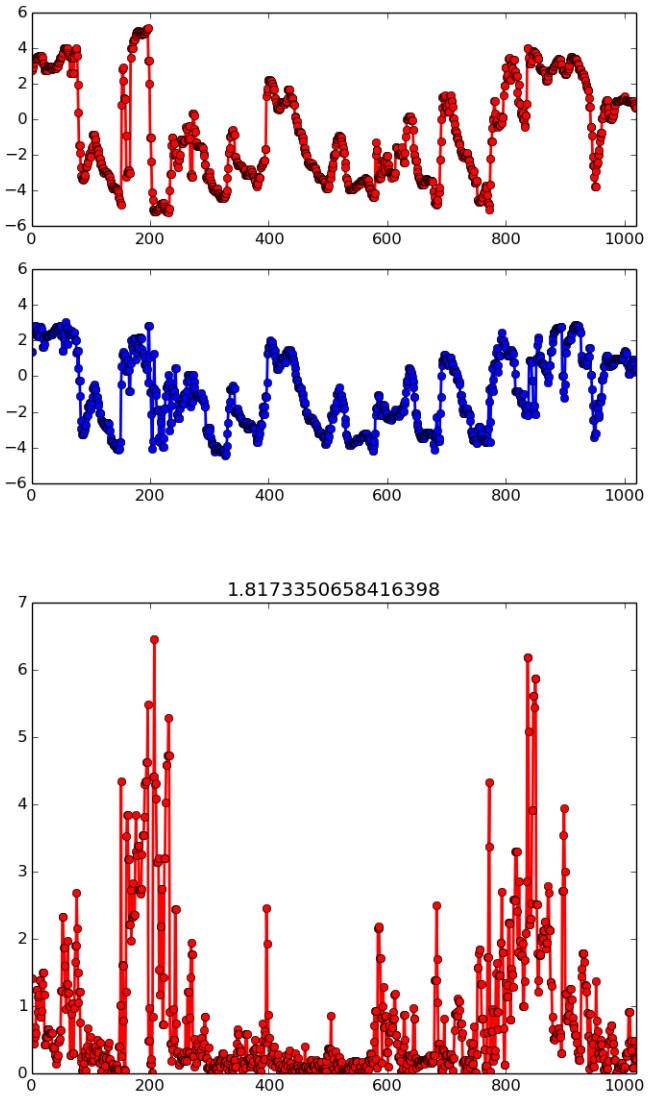

Figure 4.3: Y-axis. Top: Red plot showing true labels for target displacement vs time, with negatives for forward and positives for backward, blue plot showing predicted outputs. Bottom: Squared error at each frame.

Chapter 5

Future Work

5.1 Smooth Trajectories

Another possible direction to improve upon would be towards a more robust landing algorithm. This would direct the drone towards the target in a more seamless and efficient manner, rather than dealing with the x and y axis positioning first, followed by the z component, in two very disjoint steps. This makes for a stable but clunky algorithm. One approach toward this would be training the drone to learn both aspects simultaneously, using something like a convolutional neural network, to deal with shift invariance (which also could be done through time-delay neural networks (et al [1989])), to account for changes in scale as the drone approaches the target. Another possibility would be looking to Nguyen and Widrow's work (Nguyen and Widrow [1990]) on a two-part system that trains an emulator and controller for backing up a truck to a loading dock, training its kinematics first by trying to predict the next state given a control signal, and then training its path to the target by using the emulator as a guide. This could be similarly applied to the drone's kinematics and tracing a path to the target, but would be a much harder task in reality, since we would not have the same access to angles, coordinates and other attributes that they had in their simulations.

5.2 Reinforcement Learning

Finally, another aspect that we think would be a powerful tool in this type of task would be the use of reinforcement learning algorithms to guide the drone to learn to center itself. We see applications of RL in current work with drones, in the space of attitude control (W. Koch and Bestavros [2018]) or in very relevant work with navigation in a discretized state space (H. X. Pham and Nguyen [2018]). We can imagine a formulation in which we have a high reward associated with the target centered in frame, and decreasing the reward as the target shifts away from the center, where the decrease could be based on displacement. This would then allow the drone to explore the state space by itself until it figures out the actions to take to direct it towards higher rewards, removing the very granular level correspondence of neural networks going from image frames to displacements/control signals.

Chapter 6

Conclusion

Our drone was trained to perform a stabilization algorithm based on visual cues, using its downward camera to keep a given target centered in its frame. This turned out to be much more effective using a method that tries to predict the displacements rather than trying to learn control signals directly, largely due to the difficulty and noise involved in data collection, but also due to a lack of direct correspondence between the image frame and an immediate signal. We used Pomerleau's method of learning displacements through the center of mass of vectors and translated these displacements into the best way of controlling the drone to guide it in the appropriate manner. We then also trained a network to predict the height of the drone, after being positioned centrally above the target, such that it could be guided to lower itself to a height that was deemed appropriate before it performed a landing. We then combined these aspects to complete a pipeline for the stabilization and landing process for the drone, which we found to have an encouraging amount of success. We additionally increased the complexity of our target pattern to that of Amazon's smile logo, and found a comparable amount of success with this as well, giving us confidence in being able to deal with irregularities in the target shape, and not be dependent on the relatively simple rectangular pattern we worked with previously.

Overall, with our final method, we achieve an average displacement error of 2.4 pixels in the X-dimension and 1.4 pixels in the Y-dimension, extracted from the titles (squared errors) of the MSE curves in Figures 4.2 and 4.3. Predominantly, we found that it is effective to use the drone's camera systems to help it with these tasks, especially due to its less advanced hardware and lackluster IMU stabilization, making the camera an affordable and useful tool to utilize as an additional sensor.

Bibliography

A. Waibel et al. Phoneme recognition using time-delay neural network. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339, 1989. 5.1

D. Feil-Seifer H. X. Pham, H. M. La and L. V. Nguyen. Autonomous uav navigation using reinforcement learning. *arXiv:1801.05086*, 2018. 5.2

G. Lahiry. Drone flight sequence with pixel thresholding neural network. *https://youtu.be/SbDcNTualvg*, 2019a. 3.2.3

G. Lahiry. Drone flight sequence with crosshair neural network. *https://youtu.be/tCM_Drvib8Q*, 2019b. 3.2.3

G. Lahiry. Drone flight sequence lowering height and landing. *https://youtu.be/Wi_-kpsA7zw*, 2019c. 3.3.1

G. Lahiry. Drone flight sequence combining stabilization and landing (attempt 1). *https://youtu.be/egu0zaK2NAs*, 2019d. 4.4

G. Lahiry. Drone flight sequence combining stabilization and landing (attempt 2). *https://youtu.be/e-cbGUkLC3Y*, 2019e. 4.4

G. Lahiry. Drone flight sequence combining stabilization and landing (best attempt). *https://youtu.be/AgDMRz_105Q*, 2019f. 4.4

D. H. Nguyen and B. Widrow. Neural networks for self-learning control systems. *IEEE Control Systems Magazine*, 10(3):18–23, 1990. 5.1

D.A. Pomerleau. Neural network perception for mobile robot guidance. *Ph.D. dissertation Technical Report CMU-CS-92-115*, 1992. 2.3.1

D.A. Pomerleau. Neural network-based vision for precise control of a walking robot. *Machine Learning*, 15:125–135, 1994. 1.3, 2.3, 4.1

R. West W. Koch, R. Mancuso and A. Bestavros. Reinforcement learning for uav attitude control. *arXiv:1804.04154*, 2018. 5.2