ECE 473: Embedded Systems - Final Project Report

CatNet - A Smart Cat Door System

Desmond Carter & Roderick Renwick

December 10, 2019
Fall 2019

Honor Code:

I have neither given nor received unauthorized assistance on this graded report.

X_____Desmond_Carter_&_Roderick_Renwick_____

Table of Contents

Concept

This project aims to solve a problem with household animals that do not get along. Specifically, the goal of this project is to create a system that will operate a cat door smart enough to let one type of cat pass through, while being able to actuate a latch that will lock the cat door to prevent another type of cat from access to a given room.

Imagine that there are two cats: one orange and the other black. These cats do not get along as the orange cat likes to attack the black cat. This system will effectively recognize which cat is allowed to enter a room that is safe from another cat which is aggressive and hostile to it. In this case, the black cat will be allowed to enter a room, while the orange cat will be locked out of the room via the smart cat door.

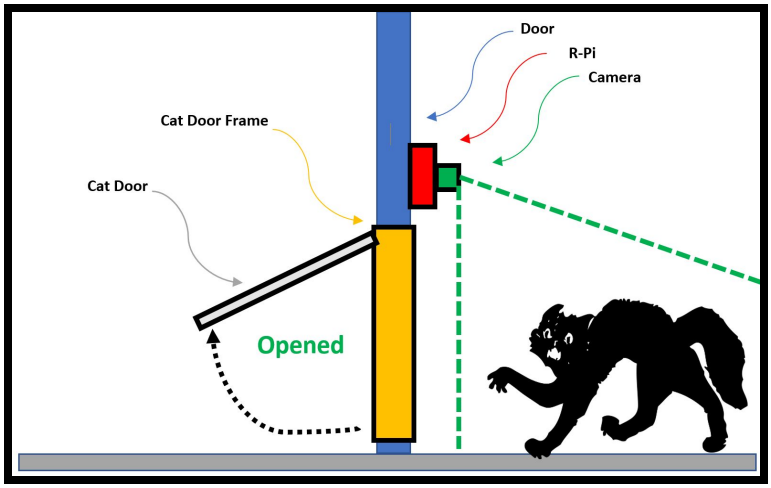

Figure-1: Cat Door Opened and Unlatched

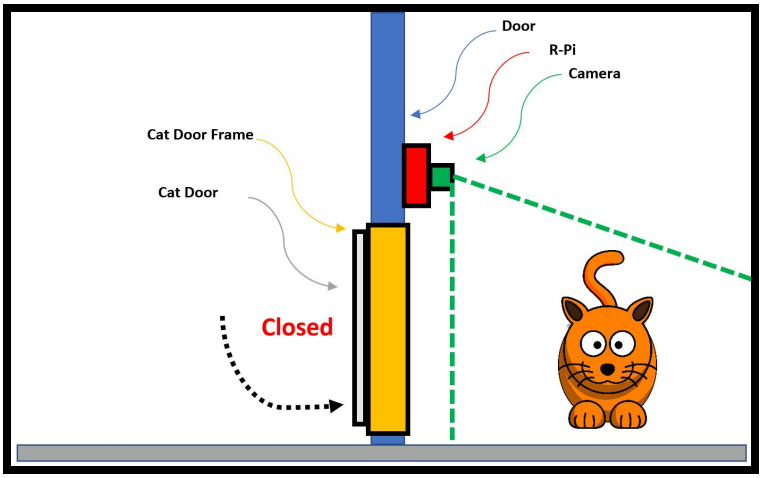

Figure-2: Cat Door Closed and Latched

Requirements

As for the functional requirements of this project, the aim is to have an embedded system make real-time predictions as to whether or not there is a cat present that is not allowed to pass through the cat-door, and then accentuate the latch accordingly to lock the cat out. While it is unrealistic to use real animals for the demoing of this project, stuffed animals will do just fine.

On the hardware side, the necessary tools will be the Jetson Nano microcontroller that will interface with a camera module and a servo-motor which will serve as the latch for the cat-door. For image classification, we'll be using the NVIDIA Jetson Nano Developer kit. For the purpose of consistency, a testing environment will be constructed out of a wooden base and wooden wall to hold the cat-door frame as well as mount the electronic equipment onto.

The necessary tools for this on the software side will consist of a convolutional neural network which will be trained to recognize and predict the color of the cat. Upon real-time detection of a cat that is deemed to not have access through the cat-door, the system will relay a command to the servo-motor to accentuate the latch. The core libraries to facilitate this project will be the Open Computer Vision (OpenCV) library and the TensorFlow library.

Timing constraints to consider for this project are the speed of a cat, the debouncing period of the cat door, the image processing time of the network, and the actuation time of the latch. Some of these constraints must be determined as the project develops. However, it we were to look at the edge cases for the lower time bound (or minimum time period) in which the latch must be repositioned, there are situations where one kat may be infinitely close to the other cat and be able to get in while the door is opened, or where the actuation of the latch may happen before the cat door is in a stable state. On the other hand, the edge cases for the upper time bound (or maximum time period) in which the latch must be repositioned may ideally be the time period it takes for a cat to enter the field of view of the camera until it reaches the door destination. However, the cat can most likely afford a waiting period. For these reasons, we have determined the classification of this system to be a firm real-time system.

Design

This section covers the overall design needed to move the initial conceptualization of the project to the implementation phase. The initial diagram gives a *bird's eye view* of the project as it lays out how different abstraction layers interact with one another.

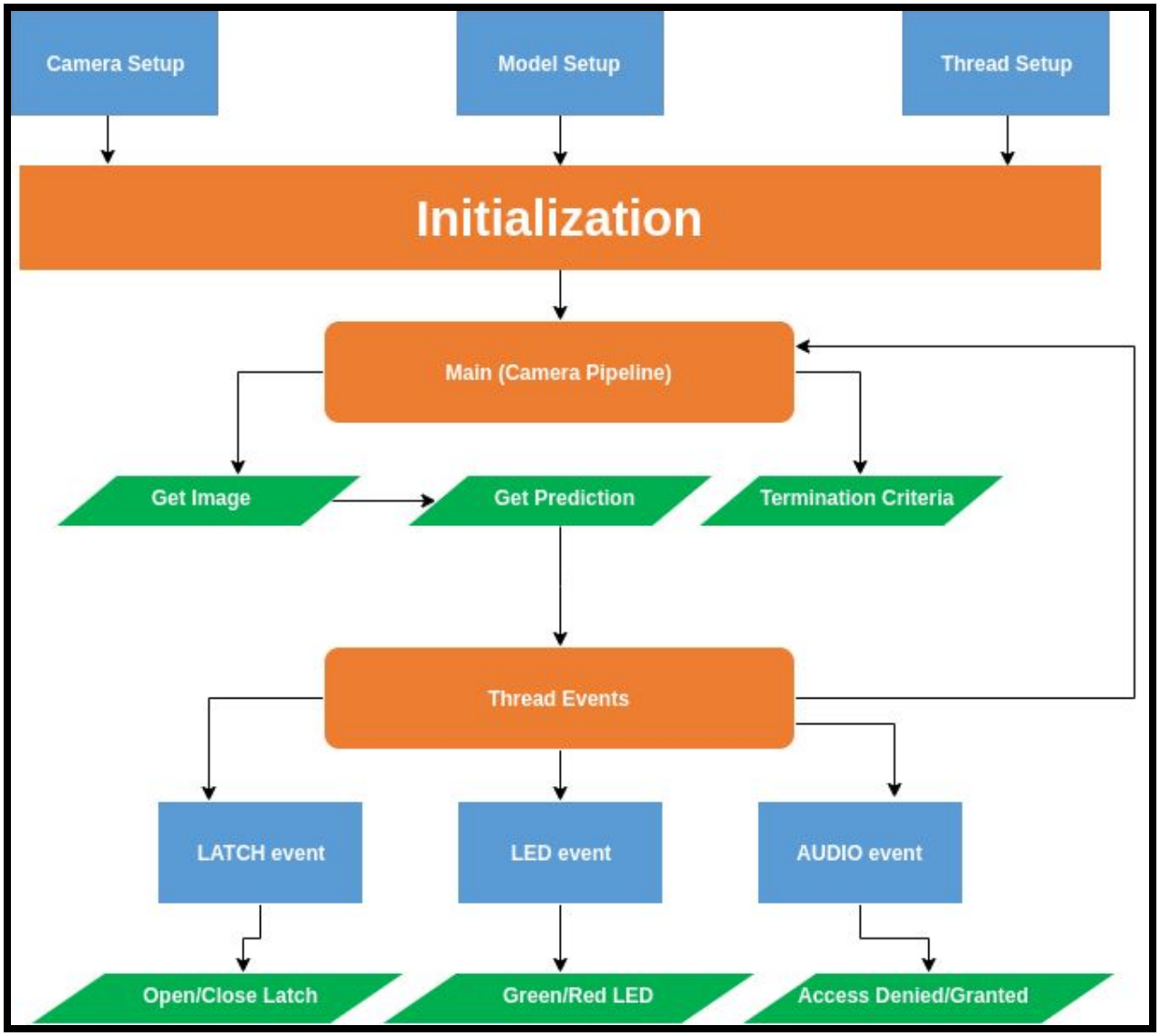

Figure-3: A bird's eye view of the project

The initial setup section obscures some of the complexity necessary to design the project. An excursion into the requisite details will help shine a light on the setup procedure.

Design: Camera Setup

In order to capture an image and make it easy to classify by the neural network during the training and model building phase, we need to simultaneously simplify it enough so as to not capture an unnecessary amount of detail, remove any distortion of the image created during the capturing process, and capture enough detail so that the neural network can distinguish the differences between objects being observed. The first issue issue is resolved by perspective transformation, while the other two concerns are handled by camera calibration. Given its ease of use, ubiquitous usage in industry, and the amount of documentation available, we've settled on using solutions for these issues provided by OpenCV. One of the constraints of OpenCV is that it can only calibrate a camera using three sorts of objects: black and white chessboards, symmetrical circle patterns, and asymmetrical circle patterns. Through our calibration and our transformation procedure, we've employed the chessboard to configure our images.

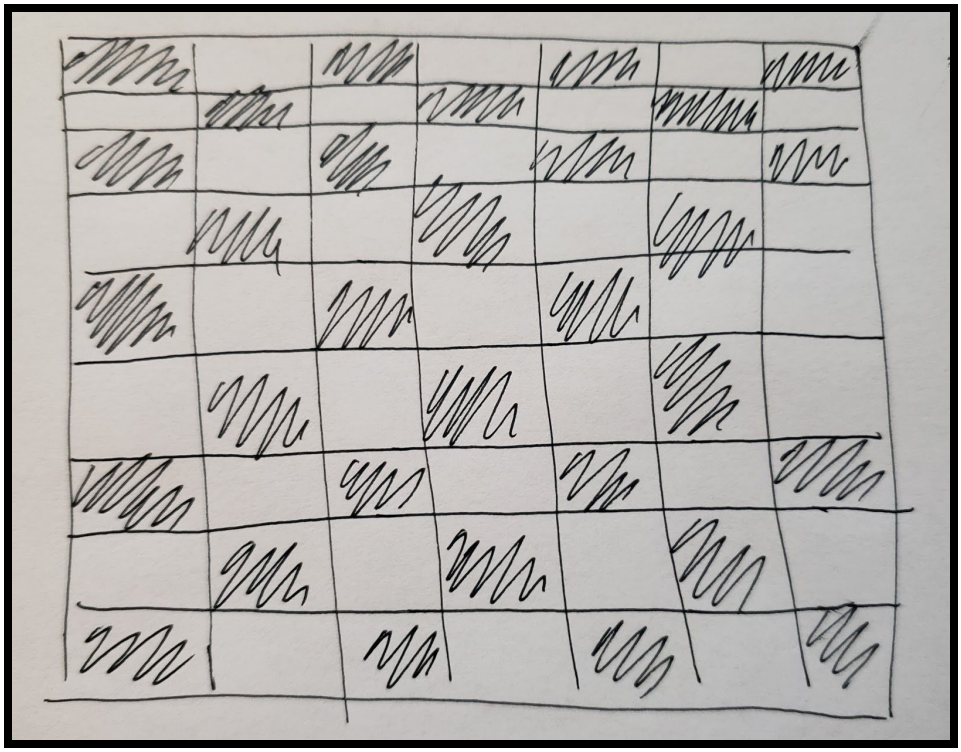

Figure-4: A sketch of a chessboard that will be used

Camera calibration deals with correcting any bias stemming from the camera's lens. It is important to remember that the camera is being used to capture light being reflected from an object through a lens that has a particular shape. This step is used to remap the dimensions of the world to pixels. The two types of distortion that stem from the curvature of a camera's lens are radial and tangential distortion. Radial distortion occurs because light extends (bends) at the points in which the lens curves. This can lead to two different types of radial distortion, depending on the focal length achieved: the barrel effect, which stems from a small focal length, and the pincushion effect, which arises when a large focal length is present.[3]

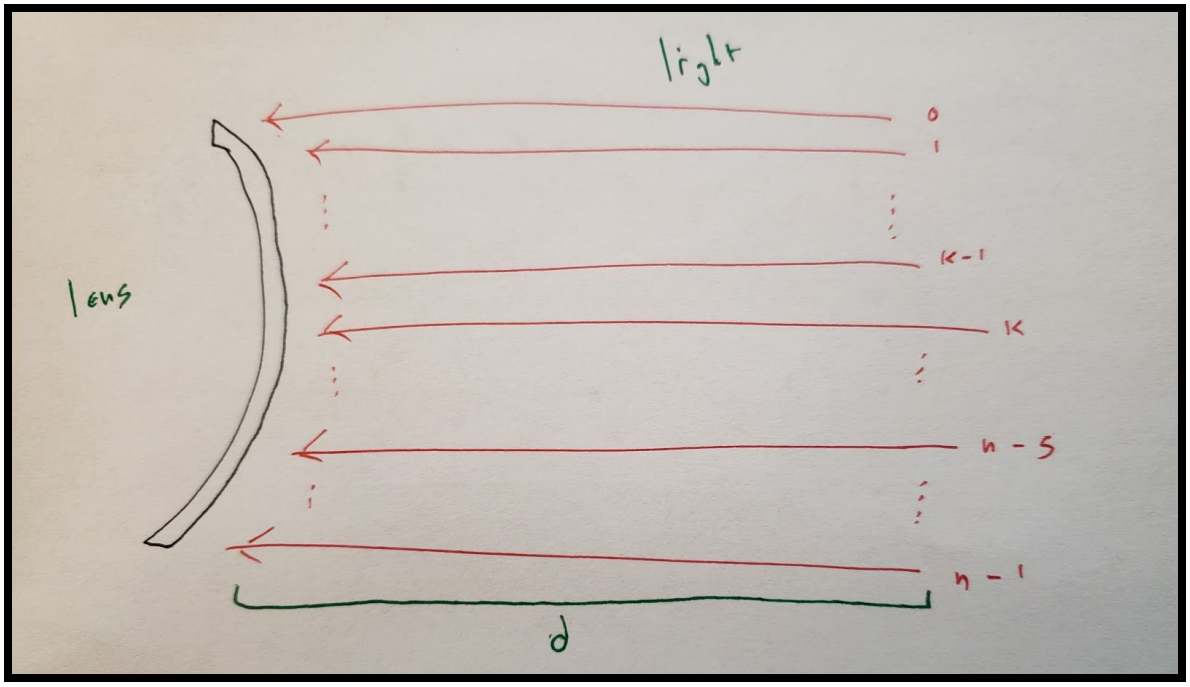

Figure-5: Radial Distortion Example of a Camera Lens

Unlike the camera calibration step, perspective transformation seeks to correct things being captured from the world, rather than correcting for inherent aspects of the camera being used. That is, it seeks to correct for the different distances that an observed objects' sides are from the camera lens.[5] An ideal situation is one where this isn't a problem, but this ideal situation would require the camera to be directly above the observed object. In most real world scenarios, this is not possible and so this sort of transformation is necessary to ensure that an undistorted image is captured. Another way one could describe this transformation is as a "flattening" of the world in order to achieve the idealized "from above" perspective.[6]

The structure of the camera initialization process is laid out in the following module (figure-6). This module will be responsible for performing two functions on a given input image (camera frame): a perspective transformation, and distortion correction. To perform those calibration functions, there are three input parameters that are required. One of these parameters is the homography matrix (mapping 3D-object points to a 2D-image plane) required for the perspective transformation. The other two parameters are the distortion matrix, and the intrinsic camera values matrix, which are required for the distortion correction in order to calibrate the camera correctly.

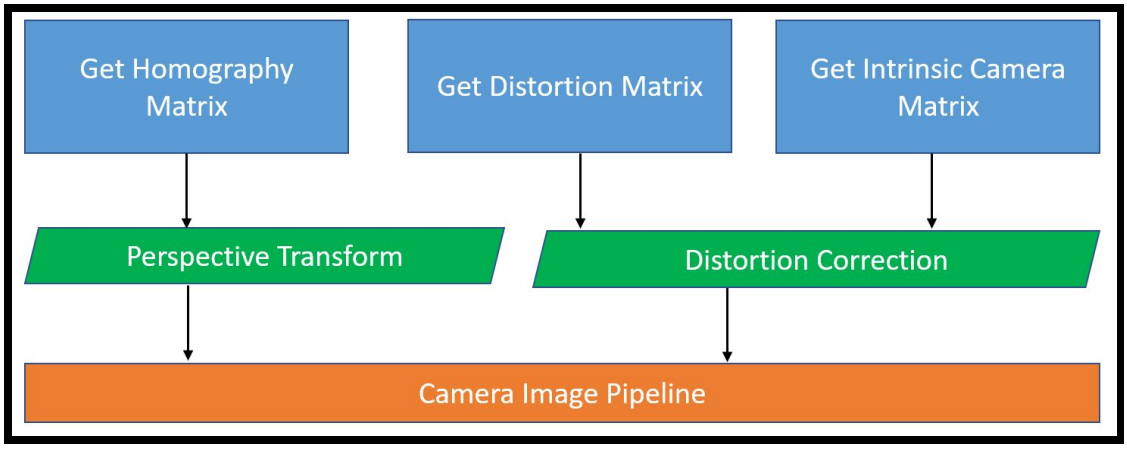

Figure-6: Camera Initialization Process

The next figure (figure-7) shows the design for the data collection submodule. The first step is to initialize the camera via the homography, distortion, and camera intrinsic values matrices. After that, we must pass in the region of interest (ROI) parameters to filter out irrelevant pixels in a given camera frame. We may then stream a live camera feed to collect data from. To do so, we will use each frame as an input that undergoes the perspective transform, distortion correction, and ROI filtering to get the apropos output image that we wish to store.

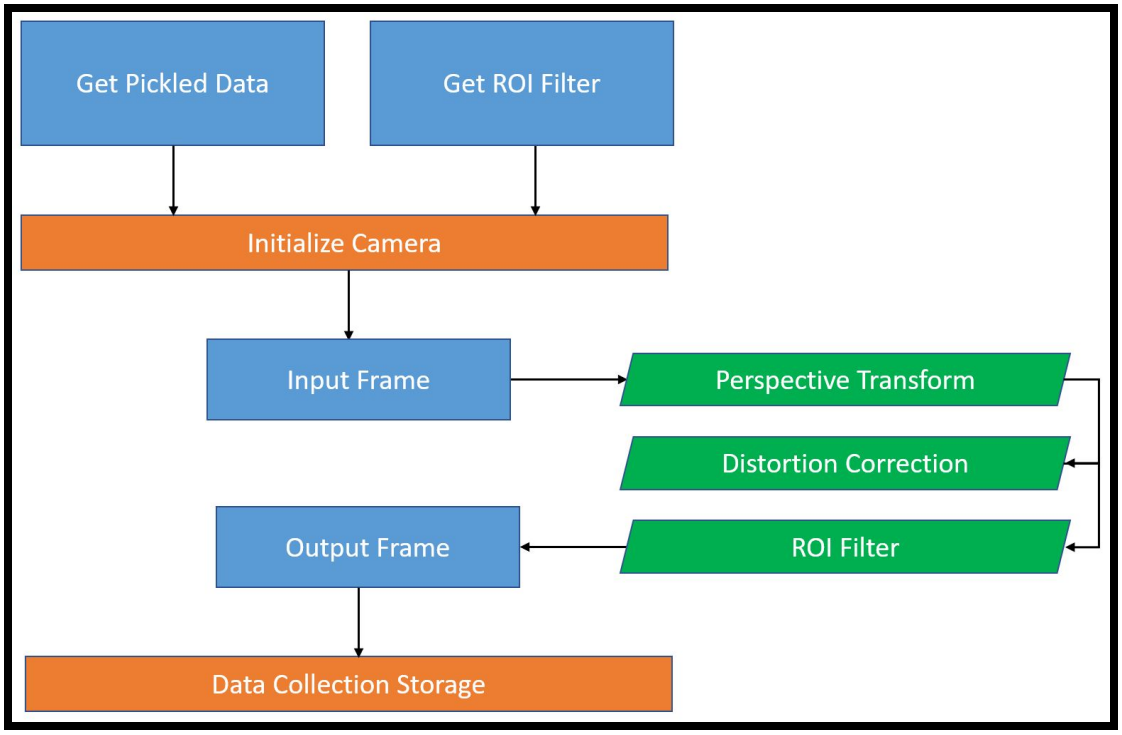

Figure-7: Data Collection Process

Design: Convolutional Neural Network Setup

For image object classification purposes, a convolutional neural network (CNN) makes for a very powerful tool in terms of accuracy and performance [10]. The value of a CNN is in the ability to learn features of an object (e.g., curvature) without respect to the object's position, or rotation within the image. By applying a filter (or kernal) to an image, and convolving the filter with the image, these key features are able to be extracted and preserved. These filters are also a means of dimensionality reduction depending on the stride of a given filter, unless "zero-padding" is used to maintain the original size. After multiple layers of convolution and pooling are applied, a fully connected layer is the final vectorized feature map, with every element uniquely corresponding to an object classification [10]. For these examples, the fully connected layer is a 3x1 vector where the final classifications are a white cat, brown cat, and no cat. The following figures go over the construction of the CNN model.

The following figure shows the flowchart process of building the CNN [8]. Specifically, there are two datasets: training, and testing. Both datasets are composed of images that may be classified into one of three categories: white cat, brown cat, or no cat. The training images need to go through "one-hot" encoding which is just to label the training images with their respective classification (done by a human). After the datasets are ready, they are converted into NumPy arrays which may then be pickled to store for later usage. From there, the pickled datasets are pulled in from the file that is ready to construct the network, and it then slits and reshapes the data (in the form of NumPy arrays) to the specified values wanted for training purposes. Lastly, the hidden layers of the network (composed of convolutional layers, dropout layers, and pooling layers) must be determined as well as the network's parameters (i.e., the batch size, epoch count, learning rate, etc.). Once all of this has been determined, the network is ready to train. Finally, when the training is complete, the model is saved, as well as the parameters used to structure the CNN so that we may reconstruct it later when using the model to make inferences.

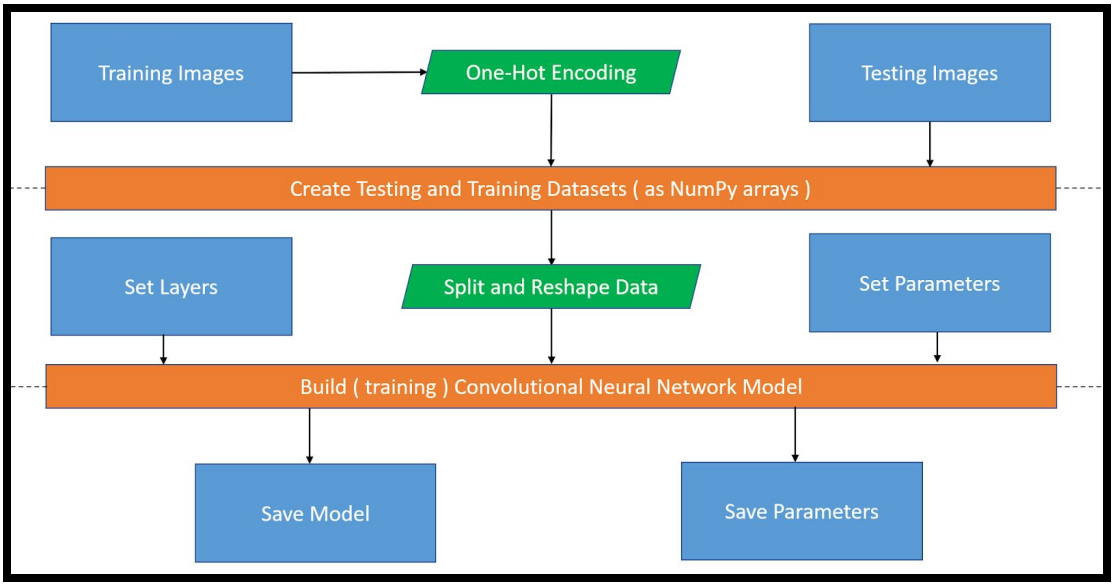

Figure-8: Building the CNN Model

The next figure is a visualization of the CNN structure [9]. It starts with the input layer, which has a depth of three -- one for each color channel from an RGB input image. Then there are a multiple of convolution and pooling layers that take place inside the hidden layers section of the diagram. Finally, the features are then mapped to the fully connected layer which has three classifications for image objects: a brown cat, white cat, and no cat.

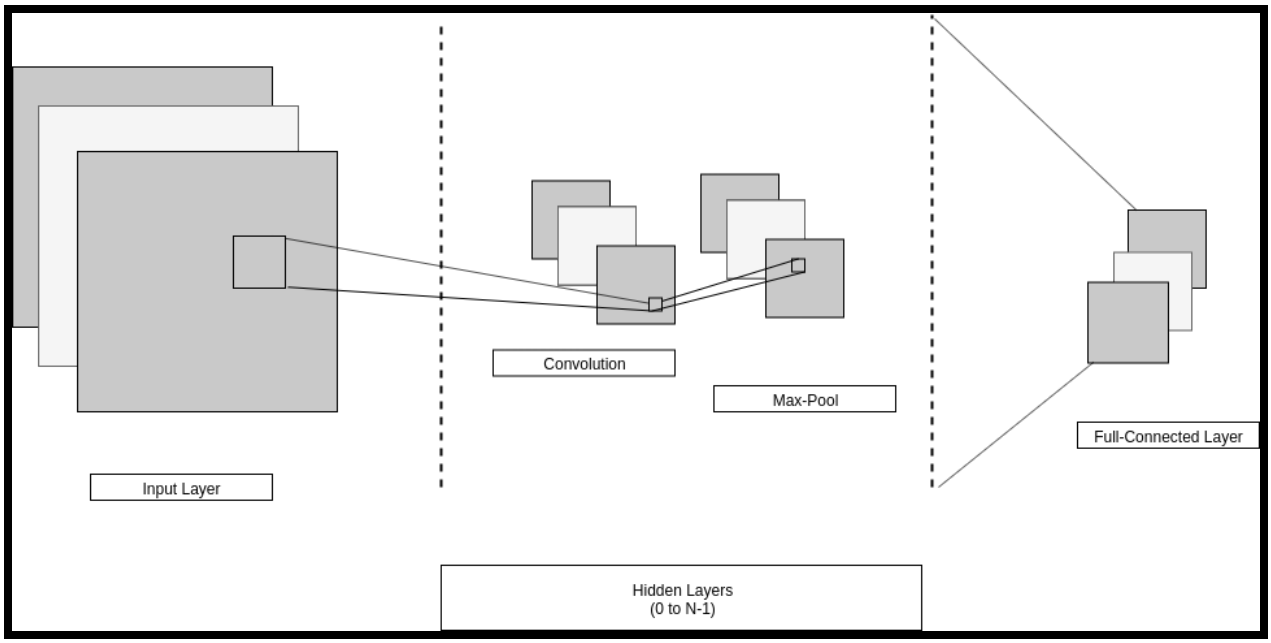

Figure-9: CNN Model Visualization

Design: Peripheral setup

For the peripherals, there are three separate functionalities: the servo motor (i.e., actuating latch), the audio speaker, and the green and red LED lights. Each of these functional areas have their own threads. To sleep / wake-up individual threads, there are two conditional boolean arguments passed into each thread: a global "termination_criteria" boolean that will tell all threads to terminate when set, and then a local boolean unique to each thread that will wake it up as needed once it is set.

Every thread uses a third boolean argument as an "event" setter. This event boolean will determine which one of the two functions should be called once the thread is ready to run. The event boolean for the latch thread will call the "actuate open" function if set high, or else it will call the "actuate close" function when set low. Similarly, the event boolean for the audio thread will call the "access granted" function is set high, or else it will call the "access denied" function when set low. Likewise, the event boolean for the LED thread will call the "green light" function if set high, or else it will call the "red light" function when set low.

The first figure below is a flowchart for calling the threads, along with their event booleans, and respective functions.

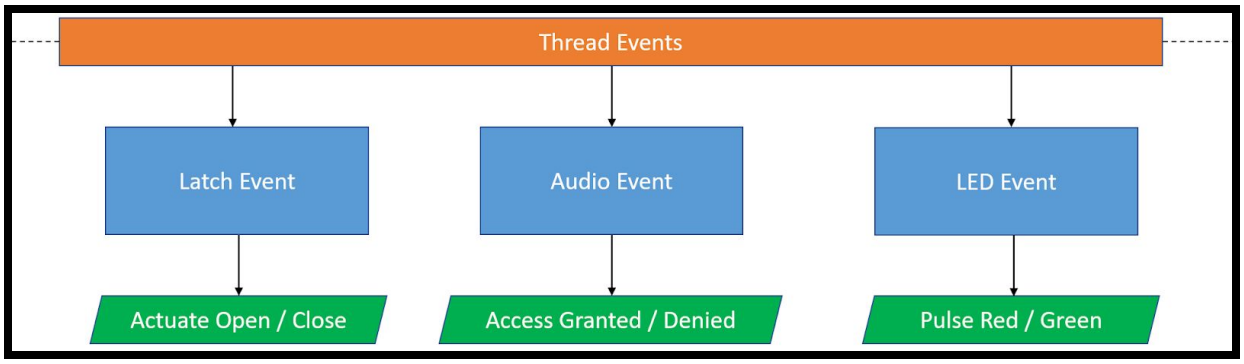

Figure-10: Threading Event Functions Flowchart

The next figure (figure-11) is a moore state diagram to show the appropriate times to wake-up each thread and call functions from it. There are three states to be considered: the "no cat" state (default), the "brown cat" state (where the brown cat is detected), and the "white cat" state (where the white cat is detected).

As shown in the legend, three bits (b0, b1, and b2) are used to tell whether there is an event required for a state change. The first bit (b0) indicates a latch event, the second bit (b1) indicates an audio event, and the third bit (b2) indicates an LED event. If any bit is zero above the arrows traveling between state changes, it means that there is no event trigger required when switching states and thus, there is no need to wake-up that given thread. Conversely, if any bit is set to one above the arrows traveling between state changes, it means that the thread's local boolean is set, and requires that thread to wake-up and run a given function, specified by the thread's event boolean.

In short, a “1” indicates that the thread needs to toggle function calls. For example, when switching between the “brown cat” state to the “white cat” state, the “111” indicates that all three threads require waking up to toggle the current states of the peripherals they control. So this switch requires that the latch state changes from opened to closed, the audio state changes from the “access granted” clip to the “access denied” clip, and the green LED state changes to off while the red LED state changes to on.

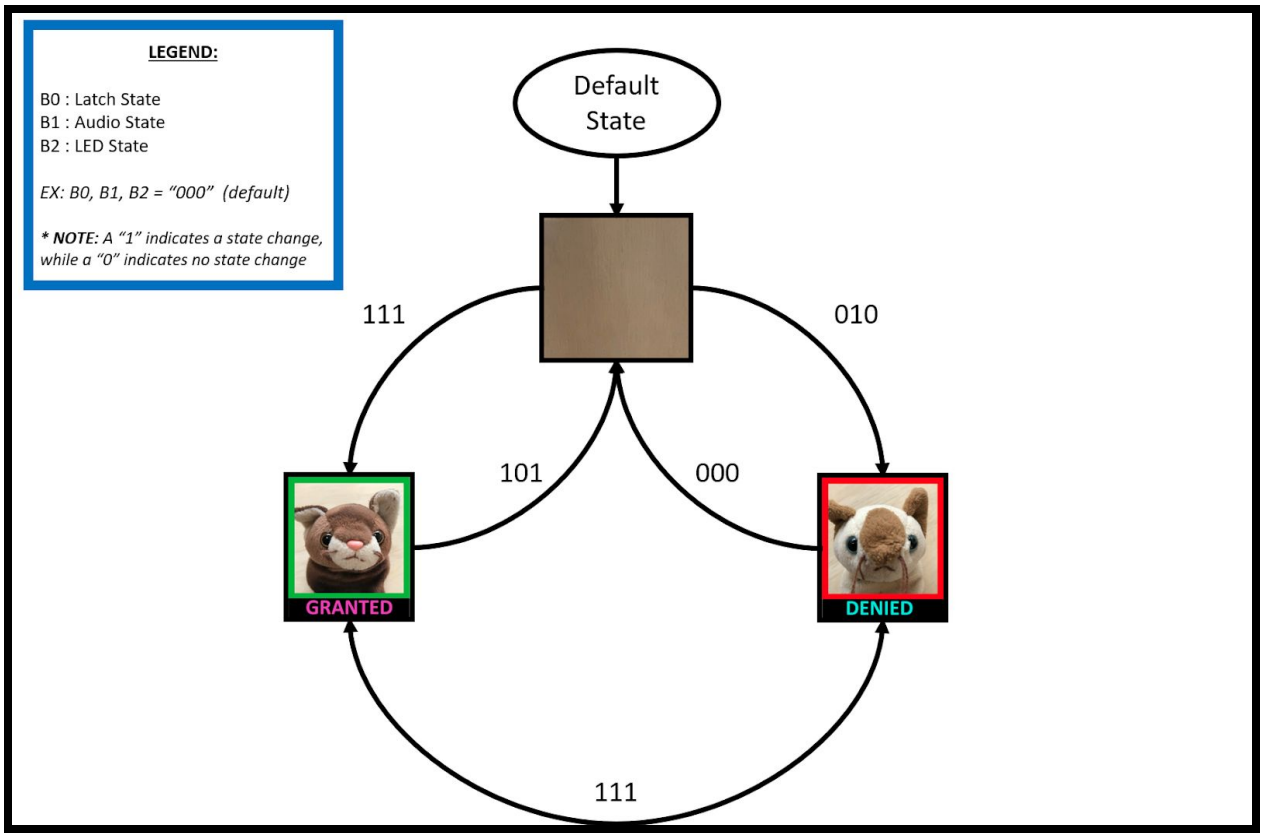

Figure-11: Moore State Change Diagram

Design: Physical Structure

The following figure shows a basic design for the physical structure that will be used to implement the cat door simulation. The drawing is proportionally scaled, and shows the key required modules for the cat door: the door and hinge, the latch actuated by a servo motor, the camera and display, and the dashed line indicates the placement of the processing hardware on the backside.

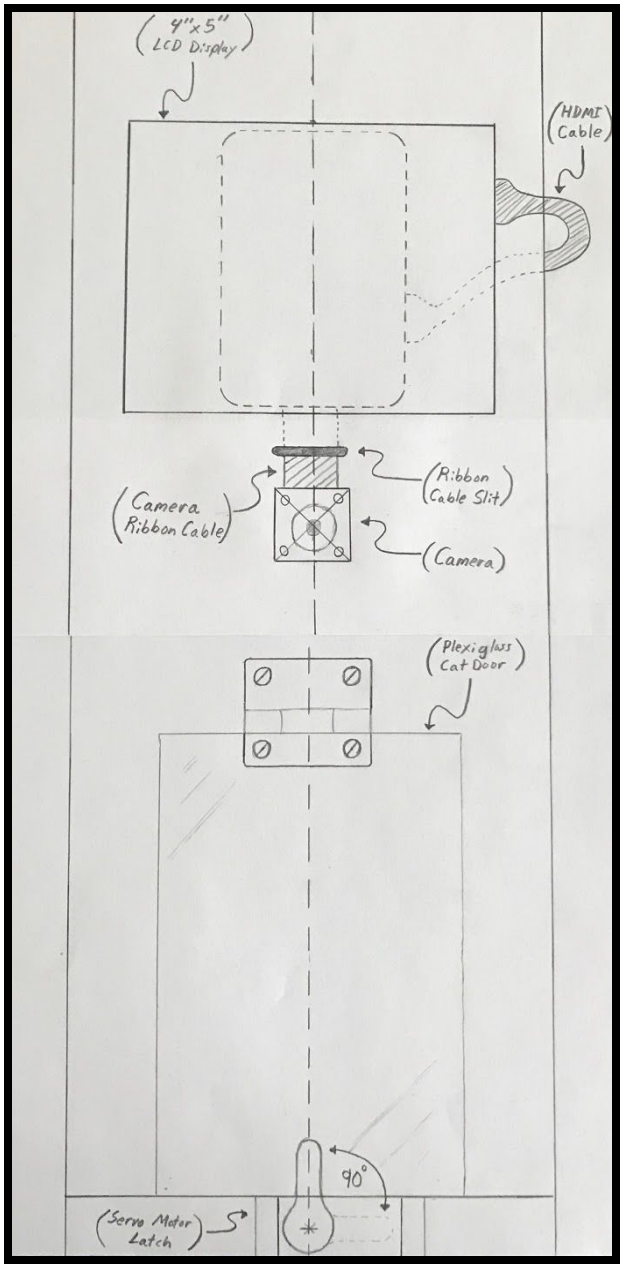

Figure-12: Physical Structure

Implementation

Implementation: Camera Setup

In order to implement solutions to the two problems addressed in the Camera Setup subsection of the Design section, we used functions from the OpenCV library. [2, 5, and 6] Below is the actual chessboard that we used to correct the issues mentioned in the camera calibration subsection above.

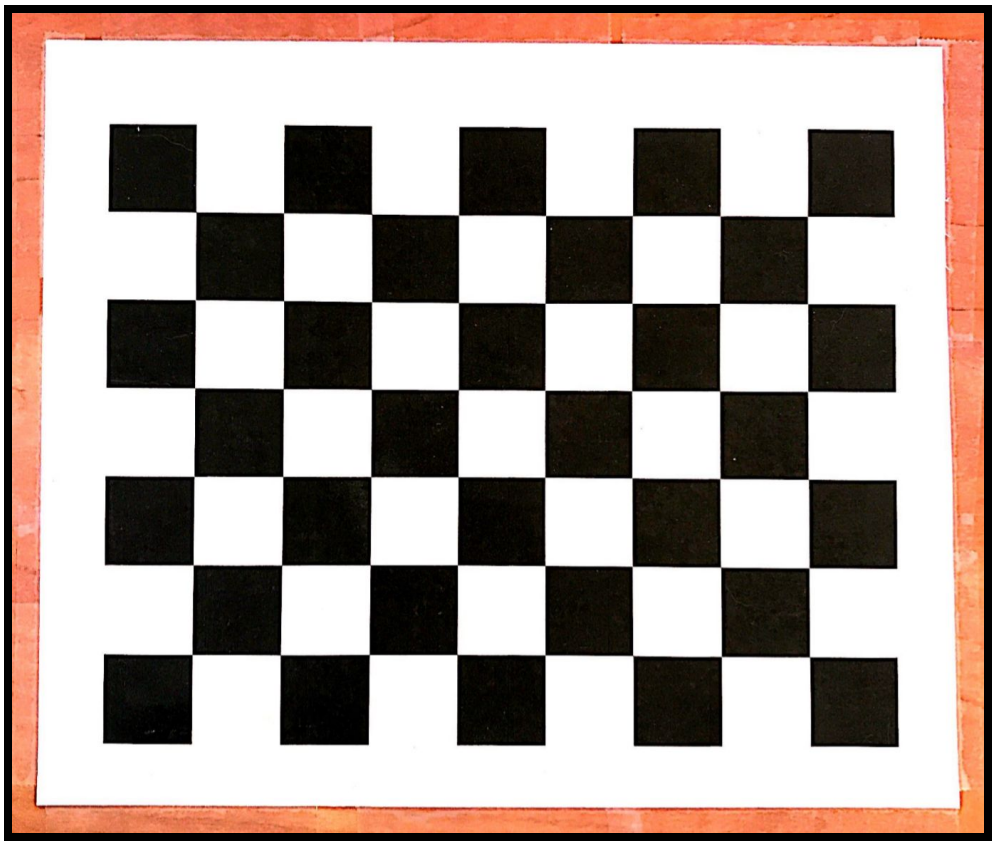

Figure-13: Physical Calibration Chessboard

The next flowchart (figure-14) describes the actual program that we used to resolve the calibration of the camera. We first collected each calibration image into an array and then reclassified them using the glob function from the glob library. We then used a for loop to iterate through the glob array of images until the last image was reached. Within the for loop, we converted each individual image to the required OpenCV format, converted that OpenCV reformatted image to grayscale, used the findChessboardCorners function from the OpenCV library to find the corners of each image, and then stored those corner points (2 dimensional points) in both the camera matrix and distortion matrix. If no corners could be found, the program would then remove the image from the array and not send it through the process.

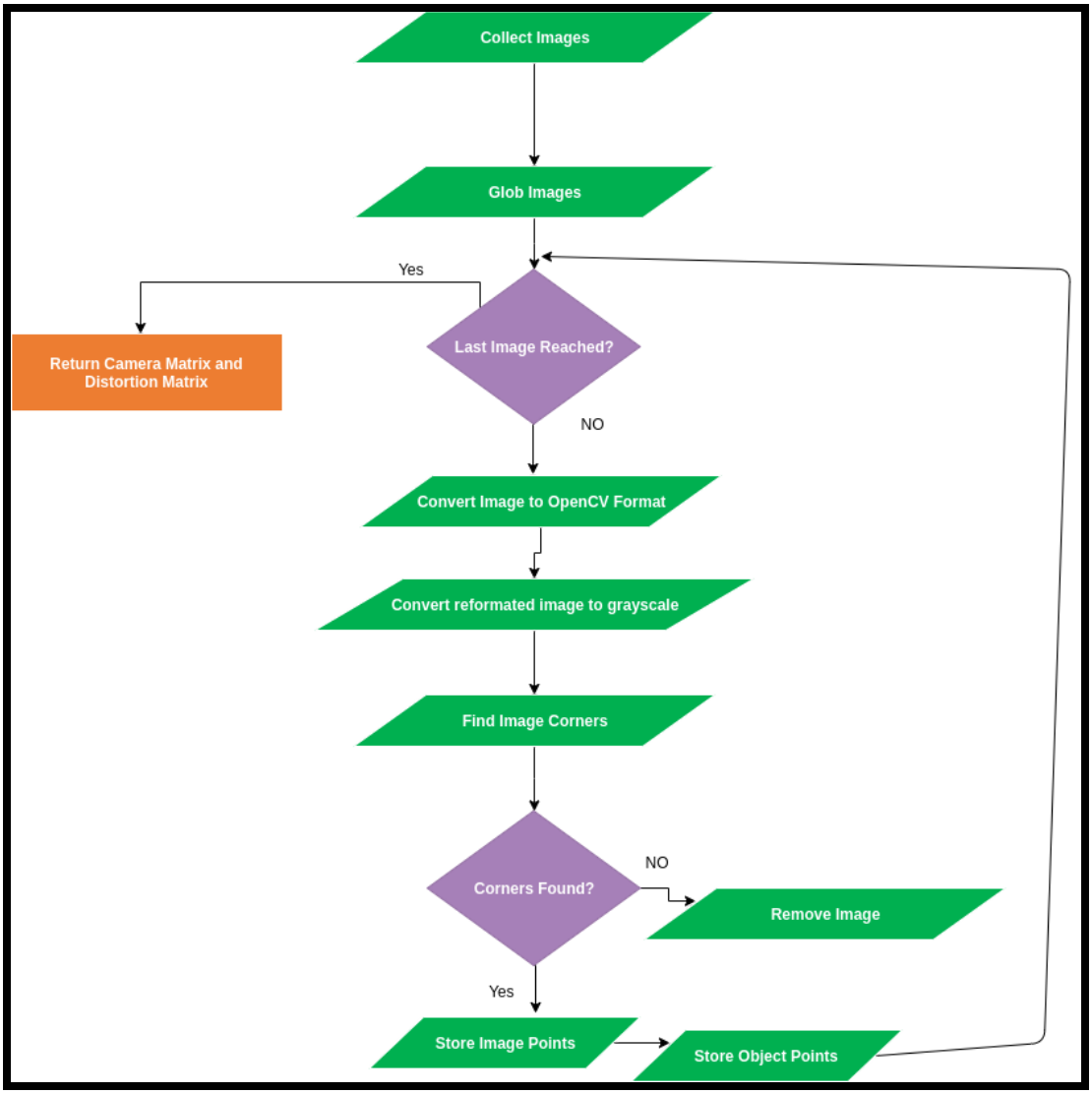

Figure-14: Camera Calibration Flowchart

The following code excerpts showcase the implementation of the flowchart shown above. The output images listed in the 3 x 3 grid show the sorts of images required in order to calibrate the camera.

```
# CHESSBOARD_VERTICES
points_per_row = 8 # horizontal_vertices
points_per_col = 6 # vertical_vertices

# OBJECT_POINTS -- E.g.: (0,0,0), (1,0,0), (2,0,0), ..., (6,5,0)
object_points = np.zeros( ( points_per_col * points_per_row, 3 ), np.float32 )
object_points[ : , : 2 ] = np.mgrid[ 0 : 8, 0 : 6 ].T.reshape( -1, 2 )

# OBJECT_POINTS_&_IMAGE_POINTS_ARRAYS
obj_points_arr = [] # 3D points in the real world space
img_points_arr = [] # 2D points in the image plane

# CHESSBOARD FINDER TERMINATION CRITERIA, CHESSBOARD SIZE, & IMAGE SIZE
termination_criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.01)
pattern_size = ( points_per_row, points_per_col ) # 8,6
```

Figure-15: Variable Declarations for Camera Calibration Implementation

```
# READ_IMAGE_FRAMES -- get object & image point arrays via chessboard corners finder
for fname in images:

  # convert image to an opencv readable format
  original = cv.imread( fname )
  gray_img = cv.cvtColor( original, cv.COLOR_BGR2GRAY )

  # FindCheassboardCorners -- ret bool is true iff board is found
  ret, corners = cv.findChessboardCorners( gray_img, (8,6), None )

  if ret == True:
    obj_points_arr.append( object_points ) # store object points
    corners2 = cv.cornerSubPix( gray_img, corners, ( 11, 11 ), ( -1, -1 ), termination_criteria )
    img_points_arr.append( corners2 )      # store image points
```

Figure-16: Implementation for Finding 3D-Object and 2D-Image Points (chessboard corners)

```
# CALIBRATION -- returns the camera matrix, distortion coefficents, rotation & translation vectors
ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera( obj_points_arr, img_points_arr,
        gray_img.shape[ : : -1 ], None, None )
```

Figure-17: Function-Call to Get Distortion (dist) and Intrinsic Values (mtx) Matrices

Here is a sampling of the images compiled for the camera calibration module (figure-18).

Figures-18: Data Collection of Images Used for Camera Calibration

With respect to solving the issues in the perspective transformation section, OpenCV offers a built-in system for “flattening” the image observed by the camera. This “flattening” is achieved via the homography matrix, which is derived from the relationship between a warped image of the Chessboard and the target image of the Chessboard.[6] The homography matrix maps the corners of the warped image to the corners of the target image. As stated earlier, this is to reduce the amount of details that the neural network will have to observe while it works to pick out identifying features of an observed object.

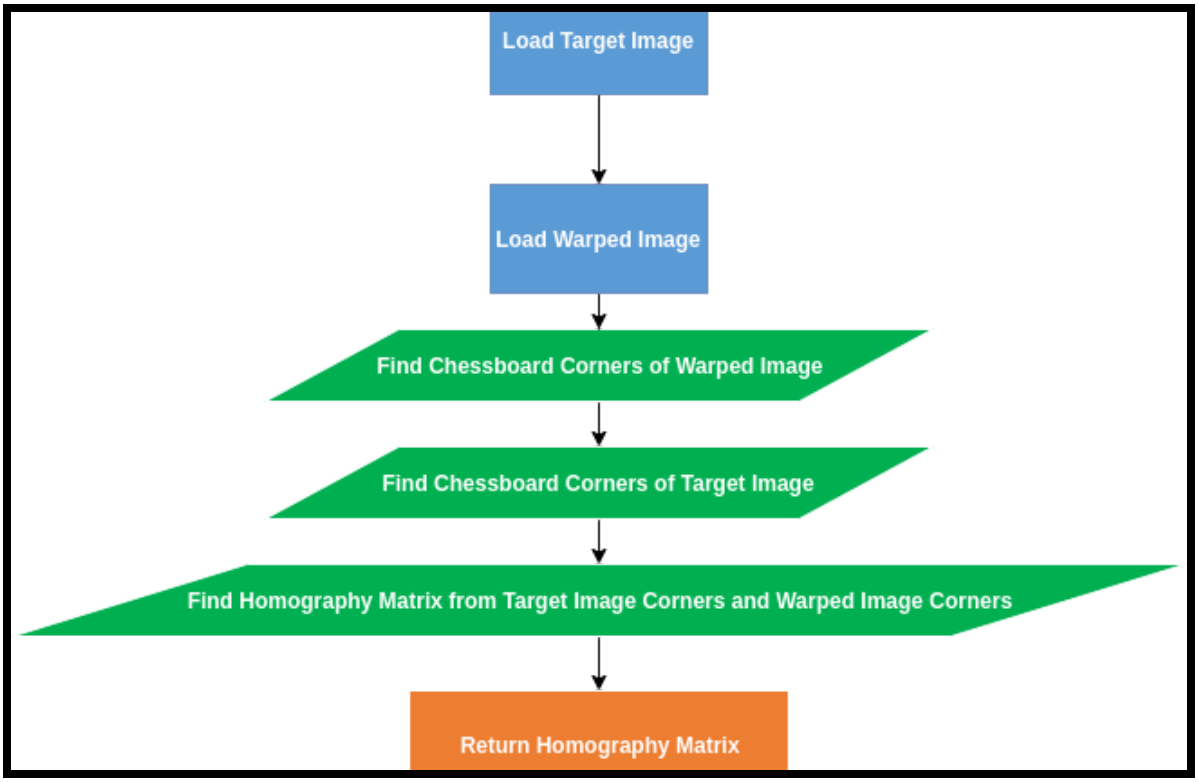

Figure-20: Flowchart Implementation for Homography Matrix

The next code (figure-21) displays the usage of the OpenCV method “findChessboardCorners”. This finds the chessboard vertices from both the warped and unwarped images. The findHomography method is then used to find the homography matrix. With this, we’re finally ready to collect the necessary data to build the convolutional neural network.

```
# LOAD_IMAGES
src_img = cv.imread( img_warped )
dst_img = cv.imread( img_target )

ret1, corners1 = cv.findChessboardCorners( src_img, ( points_per_row, points_per_col ), None )
ret2, corners2 = cv.findChessboardCorners( dst_img, ( points_per_row, points_per_col ), None )

"""# Perspective Transform via homography matrix"""

# HOMOGRAPHY_ESTIMATION
M, homography_matrix = cv.findHomography( corners1, corners2 )
```

Figure-21: Homography Estimation Matrix from 3D-Object and 2D-Image points

Here is a sampling of the images compiled from the perspective transform module (figure-22).

Figure-22: Perspective Transformation Implementation (warped images are on the right, unwarped images are on the left)

Implementation: Convolutional Neural Network Setup

As the design stage for the CNN setup explained, the image datasets are slip into two divisions: training images and testing images; and there are three classification sets within each division: no cat, brown cat, and white cat. The figure below (figure-23) provides example images of those three classification sets, with the “white cat” classification image on the left, the “brown cat” classification image is on the right, and the “no cat” classification image is in the middle. Note that all three images have a blacked out region -- this is due to the region of interest (ROI) filter, as we constrained the ROI to be within the field of view (FOV) of the wooden running board (i.e., cropped the images fed into the camera pipeline). This is to ensure consistency across images and when validating the network, as the image backgrounds are homogeneous throughout the training and testing images, regardless of the testing environment location.

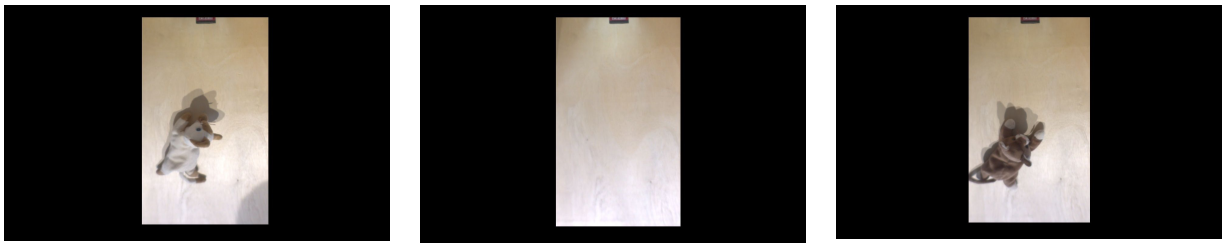

Figure-23: Image Classification Examples -- White Cat, No Cat, Brown Cat (Respectively)

The next figure below (figure-24) shows the program implementation for the global variables and CNN parameters, such as the batch size, number of epochs, and the learning rate of the model [10]. There are also specified paths for saving the model and it’s parameters after it has been constructed and trained.

```
LR = 0.5e-4
batch_size = 4
n_epoch = 100

IMG_SIZE_H = 306                    # Resize all training imgs
IMG_SIZE_W = 189                    # Resize all training imgs
CWD = "/home/sviatoslav/Desktop/CatNet"
log_path = CWD + '/data/model/log'
model_path = CWD + "/data/model/tfl"
pickle_path = CWD + '/data/pickle/dnn_network.p'
```

Figure-24: Global Variable Declarations for CNN Model Construction

After the datasets of the training and testing images have been pickled and stored (in the form of NumPy arrays), we need to load these datasets as arrays (see implementation shown below in figure-25).

```
train_data = np.load( CWD + '/data/numpy/train_data.npy', allow_pickle=True )
```

Figure-25: Loading Training and Testing Datasets (via NumPy Arrays)

Before we may train the network we need to split the "train_data" dataset (composed of 2400 images) up into train and test sets for the network to perform validation checks during the training. Generally speaking, it is safe to assume that you should set aside 20% of your training data for testing purposes when training the network -- thus, in figure-26 you can see that we split the datasets and set aside 480 images for testing purposes during training. This means that while training, the CNN model has 1920 images to test on, while using 480 images to test it's accuracy during each epoch. This also will produce two different accuracy measurements while training: one "accuracy" score for the training images (images that the network trains on and has "seen" before), and one "validation" score for the testing images (using images which the network has not yet "seen", to test itself during each epoch).

```
train = train_data[ : -480 ] # Save some for testing
test  = train_data[ -480 : ]

X_train = np.array( [ i[ 0 ] for i in train ] ).reshape( -1, IMG_SIZE_H, IMG_SIZE_W, 3 )
X_test = np.array( [ i[ 0 ] for i in test ] ).reshape( -1, IMG_SIZE_H, IMG_SIZE_W, 3 )
Y_train = [ i[ 1 ] for i in train ]
Y_test = [ i[ 1 ] for i in test ]
```

Figure-26: Splitting-Up Testing and Training Datasets

The final figure for this section (figure-27) shows the skeletal structure of the CNN model, as it has the input layer, five convolutional layers, five pooling layers, a dropout layer, and two fully connected layers that provide the final image classification [7, and 10]. These input parameters determine the number of weights and how each layer interacts with each other. Each of the three object classifications has a set of features within the fully connected layers that are scored by an activation function (in the first case, the "ReLu" activation function), and then a final logistic regression activation function (in this case, the "softmax" activation function) is used to make an inference about an input image. This "inference" is a weighted score in the form of a percentage that describes the "fit" of an image with respect to one of the object classifications, and this percentage is effectively what produces a "prediction" that classifies the image.

```
def init_model( model_path ):
    convnet = input_data( shape = [ None, IMG_SIZE_H, IMG_SIZE_W, 3 ], name = 'input' )

    convnet = conv_2d( convnet,  32, 3, activation = 'relu' )
    convnet = max_pool_2d(  convnet, 3 )

    convnet = conv_2d( convnet,  64, 3, activation = 'relu' )
    convnet = max_pool_2d(  convnet, 3 )

    convnet = conv_2d( convnet, 128, 5, activation = 'relu' )
    convnet = max_pool_2d(  convnet, 5 )

    convnet = conv_2d( convnet,  64, 5, activation = 'relu' )
    convnet = max_pool_2d(  convnet, 5 )

    convnet = conv_2d( convnet,  32, 5, activation = 'relu' )
    convnet = max_pool_2d(  convnet, 5 )

    convnet = fully_connected( convnet, 1024, activation = 'relu' )
    convnet = dropout( convnet, 0.8 )

    convnet = fully_connected( convnet, 3, activation = 'softmax' )
    convnet = regression( convnet, optimizer = 'adam', learning_rate = LR,
                         loss = 'categorical_crossentropy', name = 'targets' )

    model = DNN( convnet )
    model.load( model_path, weights_only = True )

    return( model )
```

Figure-27: CNN Model Structure

Implementation: Peripheral setup

The first peripheral to setup was audio. The following figures below (figures 28, & 29) show a speaker synthesizer circuit proposed from a *circuitbasics* website that takes an audio jack input, a potentiometer for volume control, and an operational amplifier to output the sound to a physical speaker.

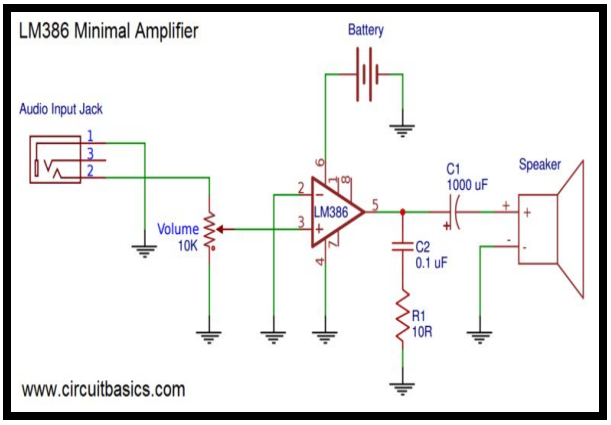

Figure-28: Audio Speaker Schematic
(from circuitbasics.com [11])

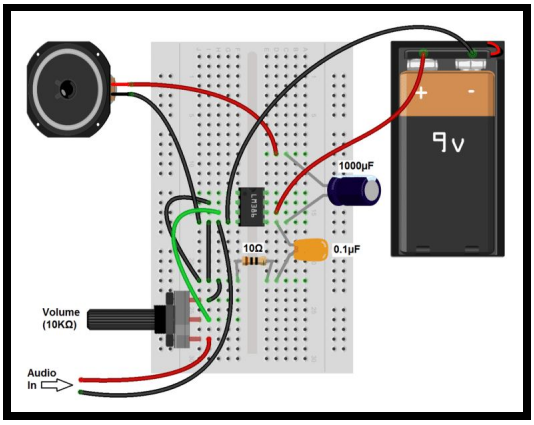

Figure-29: Audio Speaker Implementation
(from circuitbasics.com [11])

We replicated this circuit schematic for this project (shown in figures 30, & 31) in order to play our “access denied” and “access granted” sound clips in real-time from the cat door structure [12].

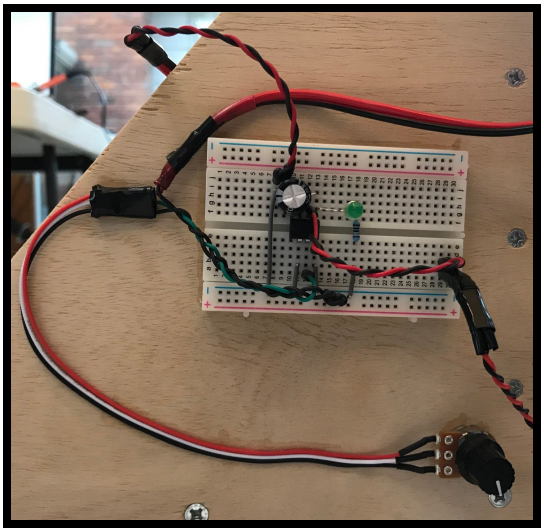

Figure-30: Audio Circuit Implementation

Figure-31: Speaker Implementation

The next peripheral to setup was the pwm driver to control the servo latch and the blinking LEDs. The first figure below on the left (figure-32) is the adafruit pwm driver that we used to power the servo motor and the LEDs. The figure to the right of that (figure-33) is the Jetson Nano pinout table that we used to find the I2C bus pins (power, ground, data, and clock) [14, 15, and 16]. After the pwm driver was connected to the Jetson Nano, the next steps were to configure the servo motor and LEDs to attach to the pwm driver.

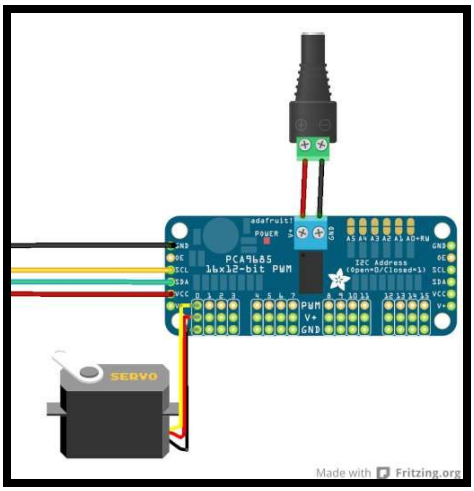

Figure-32: PWM Driver Layout
(from adafruit.com [13])

Alt Function	Linux(BCM)	Board Label			Board Label	Linux(BCM)	Alt Function
DAP4_DOUT	78(21)	D21	40	39	GND		
DAP4_DIN	77(20)	D20	38	37	D26	12(26)	SPI2_MOSI
UART2_CTS	51(16)	D16	36	35	D19	76(19)	DAP4_FS
		GND	34	33	D13	38(13)	GPIO_PE6
LCD_BL_PWM	168(12)	D12	32	31	D6	200(6)	GPIO_PZ0
		GND	30	29	D5	149(5)	CAM_AF_EN
		D1/ID_SC	28	27	D0/ID_SD		
SPI1_CS1	20(7)	D7	26	25	GND		
SPI1_CS0	19(8)	D8	24	23	D11	18(11)	SPI1_SCK
SPI2_MISO	13(25)	D25	22	21	D9	17(9)	SPI1_MISO
		GND	20	19	D10	16(10)	SPI1_MOSI
SPI2_CS0	15(24)	D24	18	17	3.3V		
SPI2_CS1	232(23)	D23	16	15	D22	194(22)	LCD_TE
		GND	14	13	D27	14(27)	SPI2_SCK
DAP4_SCLK	79(18)	D18	12	11	D17	50(17)	UART2_RTS
		RXD/D15	10	9	GND		
		TXD/D14	8	7	D4	216(4)	AUDIO_MCLK
		GND	6	5	SCL/D3		
		5V	4	3	SDA/D2		
		5V	2	1	3.3V		

Figure-33: Jetson Nano Pinout
(from element14.com [16])

The following figures below (figures 34, & 35) show the physical implementation of the pwm driver and servo motor latch on the cat door structure itself.

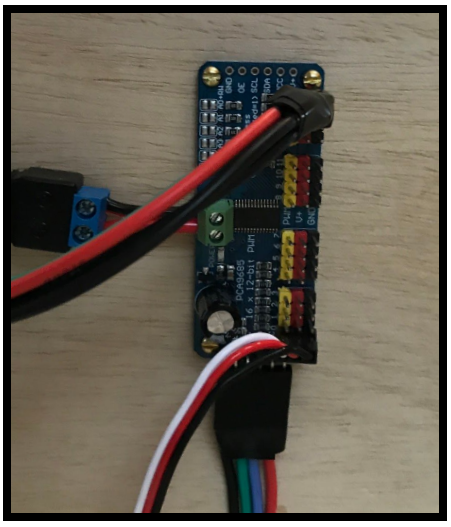

Figure-34: PWM Driver Implementation

Figure-35: Servo Latch Implementation

Similar to the servo motor needing power, ground, and pwm input signals controlled via I2C, we setup the green and red LEDs from the pwm as well, however they only needed ground and pwm input signals as the pwm duty cycle determines the illumination of them and whether they appear on or off. The implementation of these LEDs are visible in the next two figures (figures 36, & 37) [14].

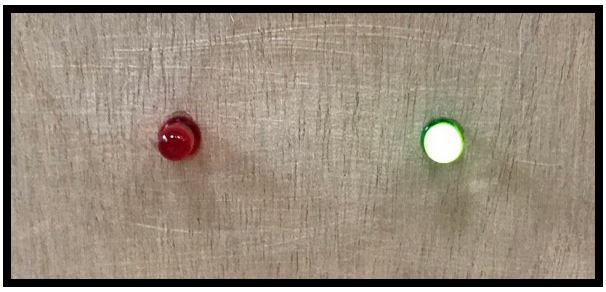

Figure-36: Green LED Implementation

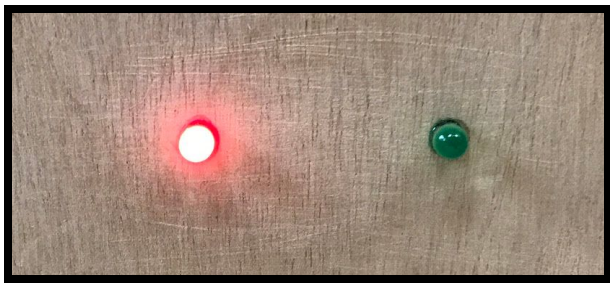

Figure-37: Red LED Implementation

The code for all three of these peripheral devices is laid out below. The first figure of code (figure-38) shows the I2C bus configuration for the pwm driver. Here we setup three independent channels: one for the motor latch, and two for the red and green LEDs [15].

```
access_granted_audio_path = "/home/sviatoslav/Desktop/CatNet/data/audio/access_granted.wav"
access_denied_audio_path = "/home/sviatoslav/Desktop/CatNet/data/audio/access_denied.wav"
#-------------------------------------------------------------------------------
i2c_bus = ( I2C( SCL_1, SDA_1 ) )                    # Create i2c bus obj
pca_driver = PCA9685( i2c_bus )                      # Set i2c bus and addr
pca_driver.frequency = 50                            # Set PWM width to 50 Hz
#-------------------------------------------------------------------------------
r_led_channel = PWMChannel( pca_driver, 15 )         # Set channel 15 as red
g_led_channel = PWMChannel( pca_driver, 14 )         # Set channel 14 as grn
latch_channel = PWMChannel( pca_driver,  0 )         # Set channel  0 as pwm
#===============================================================================
```

Figure-38: I2C Bus Configuration for PWM Driver

The next section of code (figure-39) on the following page lays out the audio implementation via threading. You can see that the function takes in three thread events - one of which is the global termination event passed along to all of the running threads [17, 18, and 19]. The sound event is the trigger to wake-up and run the thread, upon which it will play either the “access granted” or “access denied” audio clip [12]. To determine which audio clip is wanted, the audio boolean event is passed along, where it will play “access granted” if it is set high, or else it will play “access denied” if the boolean is false.

```
def init_audio( sound_event, audio_boolean, terminate_event ):

    while( not terminate_event.isSet() ):

        sound_event.wait()                  # wait for sound event (blocking)

        if( audio_boolean.isSet() ): playsound( access_granted_audio_path )
        else: playsound( access_denied_audio_path )

        sound_event.clear()                   # Clear flag and wait for next trigger
```

Figure-39: Audio Thread Function-Call

Similarly, the latch thread (figure-40) is setup to trigger on the latch event being set, determine the function-call from the latch boolean event, and terminate when the global termination event is set. However, since the latch requires the pwm, the latch channel must have it's duty cycle set accordingly. The pulse width of the pulse period must gradually increase from the starting value to its ending value to mirror actuation as the latch rotates accordingly. For example, an increase from 5 milliseconds to 9 milliseconds corresponds to the physical latch rotating from the horizontal (open/unlocked) position to the vertical (closed/locked) position and vice-versa.

```
def init_latch( latch_event, latch_boolean, terminate_event):

    while( not terminate_event.isSet() ):

        latch_event.wait()                    # wait for latch event (blocking)

        if( latch_boolean.isSet() ):          # close
            for duty_cycle_ratio in range( 5000, 9000, 5 ):
                latch_channel.duty_cycle = duty_cycle_ratio

        else:                                 # open
            for duty_cycle_ratio in range( 9000, 5000, -5 ):
                latch_channel.duty_cycle = duty_cycle_ratio

        latch_event.clear()
```

Figure-40: Latch Thread Function-Call

The final section of code for the peripheral implementations (figure-41) shows the same structure as the threading function for the latch, but is in charge of the LEDs functionality. As the pwm driver controls these LED channels, a duty cycle of 100% (or 0xffff) gives the channel a continuous pulse which effectively leaves the LED in the “on” state. Conversely, having the duty cycle of 0% (or 0x0000) gives the channel zero pulse which effectively leaves the LED in the “off” state.

```
def init_led( led_event, led_boolean, terminate_event):

    while( not terminate_event.isSet() ):

        led_event.wait()                    # wait for led event (blocking)

        if( led_boolean.isSet() ):
            r_led_channel.duty_cycle = 0x0000
            g_led_channel.duty_cycle = 0xffff

        else:
            r_led_channel.duty_cycle = 0xffff
            g_led_channel.duty_cycle = 0x0000

        led_event.clear()
```

Figure-41: LED Thread Function-Call

Implementation: Physical Structure

Following the physical setup of the peripherals shown above, the physical structure implementation is shown below for the car door (figure 42, & 43). The mounted hinge on top of the door frame that holds up the plexiglass plates (used as the door itself) has a tension screw in the middle to help determine the friction to resist rotational movement. However, there was initial concern with trying to position the door perfectly vertical when it was at rest. Below the center of the door frame is the latch mounted to actuate it's arm in place of the opening door to put the system in a locked state.

Figure-42: Cat Door - Frontside

Figure-43: Cat Door - Backside

It should be noted that the debounce period of the door to rest in a stable position after being swung open was rather significant. In order to combat this issue, magnets were cemented into the plexiglass plates as well as the sides of the wooden structure. These magnets effectively held the door closed in the intended position and helped dampen the debounce time needed for the door to rest in a stable state. This implementation worked out very well.

Implementation: Timing-Constraints

Upon implementation of the physical structure, it became apparent that the main factor that would dictate the timing constraints was the debouncing period of the door. We determined that the latch actuation time needed to be slower than the door debouncing time period but quicker than the time period it kats for the cat to enter the image frame until the cat reaches the door frame. This time constraint is detailed in the figure below (figure-44).

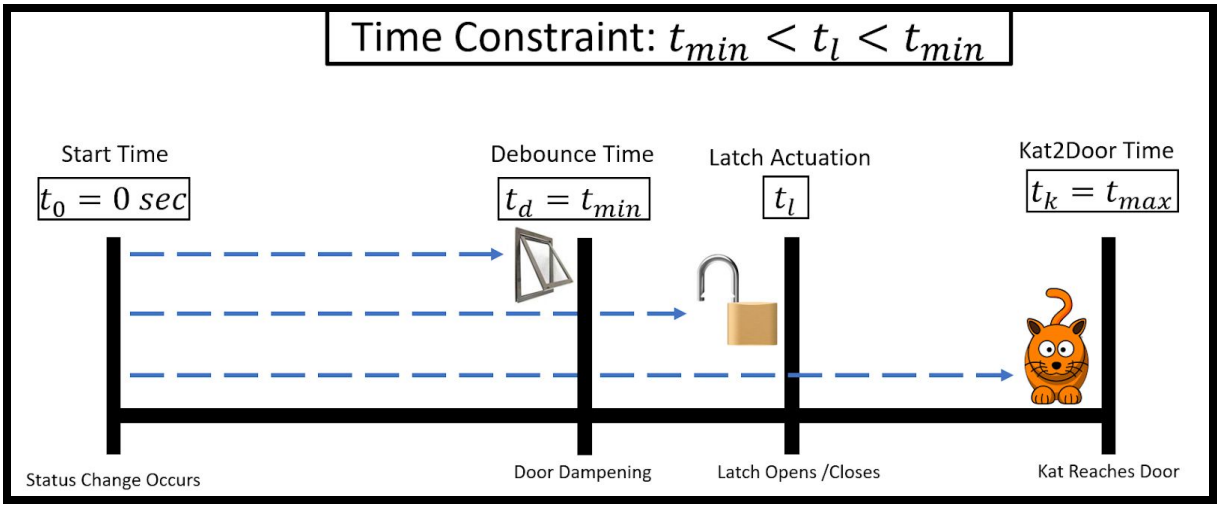

Figure-44: Theoretical Time Constraint Diagram

This theoretical time constraint is illustrated in the following example (figure-45).

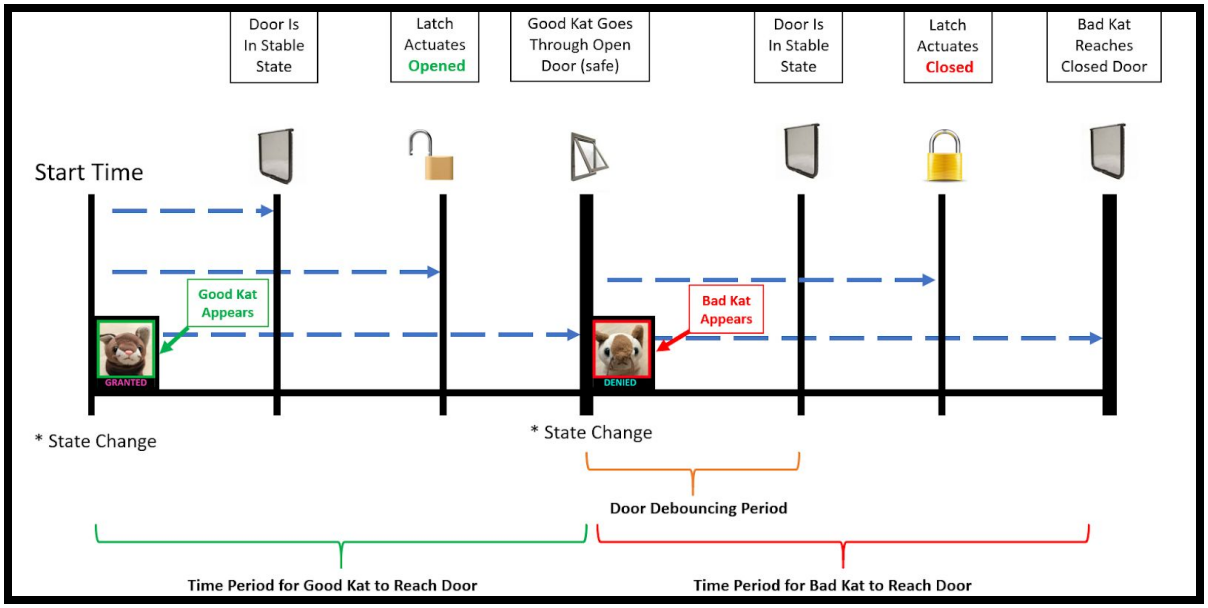

Figure-45: Time Constraint Example

Implementation: Main

In order to execute this main program, we first need to specify a few global variables (listed in figure-46) which are key to the application. Firstly, there are the data paths to the pickled data files. Then there is the LCD screen width and height so that we may properly configure the display image window. Also, we must locate and load in the profiles of each of the three states that this system will classify itself as being in throughout the program, and then we store these profiles in a vector called “kat_profile_arr”. After all of that is done, we are ready to declare our threading events to control the peripherals [18].

```
CWD = "/home/sviatoslav/Desktop/CatNet"
pickle_data_path_global = CWD + '/data/pickle/dnn_network.p'
camera_data_path_global = CWD + '/data/pickle/camera_calibration_pickle.p'
homography_data_path_global = CWD + '/data/pickle/perspective_transform_pickle.p'
gstreamer_str = "nvarguscamerasrc ! video/x-raw(memory:NVMM), width=(int)%d, height=(int)%d

LCD_HEIGHT = 480 - 20
LCD_WIDTH  = 800

kat0_profile = cv.imread( CWD + '/data/kat_profiles/kat_0.JPG' )
kat1_profile = cv.imread( CWD + '/data/kat_profiles/kat_1.JPG' )
kat2_profile = cv.imread( CWD + '/data/kat_profiles/kat_2.JPG' )

kat0_profile = cv.resize( kat0_profile, (LCD_HEIGHT, LCD_HEIGHT) )
kat1_profile = cv.resize( kat1_profile, (LCD_HEIGHT, LCD_HEIGHT) )
kat2_profile = cv.resize( kat2_profile, (LCD_HEIGHT, LCD_HEIGHT) )

kat_profile_arr = [ kat0_profile, kat1_profile, kat2_profile ]

# Set events that will trigger threads ( all events are initially false )
led_boolean = Event()
audio_boolean = Event()
latch_boolean = Event()

led_event = Event()
aux_event = Event()
latch_event = Event()
terminate_event = Event()

threads = []
```

Figure-46: Global Variable Declarations in Main Program

Opening the pickled files from the data paths is done in the following figure (figure-47). There you can see how we loaded in the CNN model parameters, and camera the initialization matrices (i.e., the distortion, intrinsic values, and homography matrices).

```
with open( pickle_data_path_global, mode = 'rb' ) as f: file = pickle.load( f )
MODEL_NAME = file[ 'MODEL_NAME' ]
IMG_SIZE_H = file[ 'IMG_SIZE_H' ]
IMG_SIZE_W = file[ 'IMG_SIZE_W' ]
model_path = file[ 'model_path' ]
log_path   = file[ 'log_path' ]
X_train    = file[ 'X_train' ]
Y_train    = file[ 'Y_train' ]
X_test     = file[ 'X_test' ]
Y_test     = file[ 'Y_test' ]
LR         = file[ 'LR' ]

with open( camera_data_path_global, mode = 'rb' ) as f: file = pickle.load( f )
roi = file[ 'roi' ]
mtx = file[ 'mtx' ]
dist = file[ 'dist' ]
new_cam_mat = file[ 'newCamMat' ]

with open( homography_data_path_global, mode = 'rb' ) as f: file = pickle.load( f )
m = file[ 'M' ]
```

Figure-47: Global Variable Declarations from Pickled Data in Main Program

Another key function for our program is defined on the next page (figure-48). This is where we implement the switching functionality between our peripheral threads. As you can see, not every state change requires waking up every thread and looking for new function calls. By implementing the moore state diagram to determine which threads need awakening after different predictions, we can minimize the time spent context-switching between threads, and ensure that our program runs at a high efficiency rate.

```
def set_status( new_status, old_status ):

    if( new_status == 1 ):      # STATE: X_2_WHITE

        if( old_status == 0 ):  # STATE: NONE_2_WHITE (010)

            audio_boolean.clear()   # access denied sound clip
            aux_event.set()
            #time.sleep(0.01)        # GOOD -- LEAVE ALONE FOR NOW

        else:                   # STATE: BROWN_2_WHITE (111)

            audio_boolean.clear()   # access denied sound clip
            latch_boolean.set()     # close latch
            led_boolean.clear()     # red led

            latch_event.set()
            time.sleep(0.3)
            aux_event.set()
            led_event.set()

    elif( new_status == 2 ):    # STATE: X_2_BROWN (111)

        latch_boolean.clear()   # open latch
        audio_boolean.set()     # access granted sound clip
        led_boolean.set()       # grn led

        latch_event.set()
        time.sleep(0.3)
        aux_event.set()
        led_event.set()

    elif( old_status == 2 ):    # STATE: BROWN_2_NONE (101)

        latch_boolean.set()     # close latch
        led_boolean.clear()     # red led

        latch_event.set()
        time.sleep(0.3)
        led_event.set()

    #else: STATE: WHITE_2_NONE (000) -- change nothing
```

Figure-48: Thread Switching and Interfacing in Main Program

The next key function of our program is actually the main function (starting at figure-49). Here is where we begin the camera pipeline initialization. After we read in the CNN model and its parameters, we start spinning off the peripheral threads to be ready when called upon. After that, we are able to construct the video feed which displays a running loop of the camera frames captured via OpenCV.

```
def show_camera():

    # Setup local variables
    kat_profile = kat_profile_arr[ 0 ]                    # Local var to
    guess_arr = [ '0', '0' ]
    guess_idx = 0
    status = -1

    # Setup convolutional neural network model
    print( "Initializing CNN model..." )
    model = init_model( model_path )

    # Setup running threads to perform actions
    print( "Initializing threads..." )
    run_thread( init_led, led_event, led_boolean )
    run_thread( init_audio, aux_event, audio_boolean )
    run_thread( init_latch, latch_event, latch_boolean )

    # Init peripherals
    default_state()
    camera = cv.VideoCapture( gstreamer_pipeline(), cv.CAP_GSTREAMER )
```

Figure-49: Camera and Thread Initializations in Main Program

The following figure (figure-50) shows the display window configuration for the LCD screen.

```
    if camera.isOpened():

        cv.namedWindow( "CSI Camera", cv.WINDOW_NORMAL )
        cv.resizeWindow( "CSI Camera", ( LCD_WIDTH, LCD_HEIGHT ) )
```

Figure-50: Display Window Initialization in Main Program

Finally for the real “work-horse” of the program, we can jump to the while loop (as shown in figure-51). Here we continuously read in a new image frame from the live camera. Once that frame is formatted correctly, we can then display it as well as feed it into the CNN model to make a prediction from. From speed and image processing concerns, we only act when two consecutive predictions are the same as this gives a more robust and stable output predictor while running the detection in real-time.

```
        while( True ):

            ret, frame = camera.read()

            # Capture camera frame and display
            output_frame = camera_pipeline( frame )
            output_frame = cv.resize( output_frame, ( 347, LCD_HEIGHT ) )
            concat_img_window( output_frame, kat_profile )

            # Resize image frame and feed into CNN to make prediction
            img_data = cv.resize( output_frame, ( IMG_SIZE_W, IMG_SIZE_H ) )
            guess_arr[ guess_idx ] = np.argmax( model.predict( [ img_data ] )[ 0 ] )

            if( guess_idx >= 1 ):
                status = compare_guesses( guess_arr, status, output_frame )
                kat_profile = kat_profile_arr[ status ]
                guess_idx = 0

            else: guess_idx += 1

            # Stop the program on the ESC key
            key_code = cv.waitKey( 30 ) & 0xFF
            if( key_code == 27 ): break
```

Figure-51: Camera Pipeline Loop in Main Program

The last figure in the main program shows the termination functions. For OpenCV we must first close the current window display, release the camera structure, and then we can call a termination program to clean up the running threads in order to exit the program in a controlled manner.

```
    cv.destroyAllWindows()
    camera.release()
    end_program()

if __name__ == "__main__": show_camera()
```

Figure-52: Termination Function Calls in Main Program

Testing

Upon final testing, we were able to verify that the network was trained as intended and produced the results we were looking for. Thus, the application was successfully implemented and we were pleased with the final product. The first figure below (figure-53) shows a screenshot of the CNN as it finished training.

As you can see, the “accuracy” score (from datasets during training that the network had previously seen before) towards the end was excellent as it approached 100%. Then there is the “val_ac” score (validation accuracy), which hovered around a 98.5% success rate for the network training on data that was previously “unseen” or new to it. This shows a successfully trained CNN model.

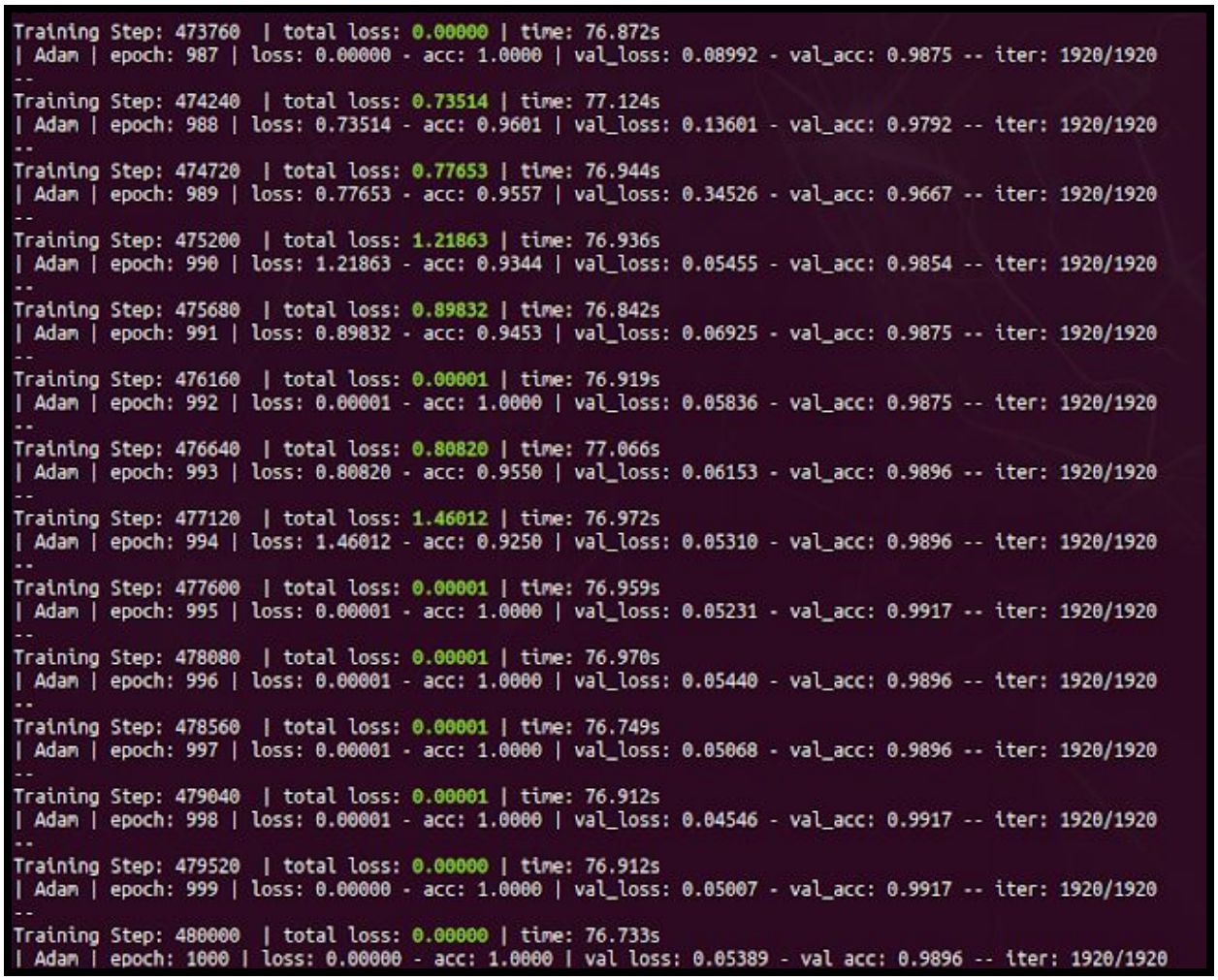

Figure-53: CNN Training Results

Following the trained CNN, the physical testing was done by running the model, and showing different objects in the camera’s field of view.

As you can see in the figures on the next page (figures 54, 55, & 56), the CNN model was able to precisely predict the proper state of the system with respect to the input frame of the camera. On the left hand side, the brown cat was correctly identified on the LCD screen, the green light LED has been triggered, and the latch is down (in the open state). On the right hand side you can see the white cat has been correctly identified on the LCD screen, the red light LED has been triggered, and the latch passed the door is vertical (in the closed state). As well, in the middle you can see that when neither cat is in on the runway, it correctly displays no cat as it's prediction.

Figure-54: Brown Cat Prediction

Figure-55: No Cat Prediction

Figure-56: White Cat Prediction

The last two figures (figure 57, & 58) are close up images of the prediction profiles for both cats.

Figure-57: Brown Cat Profile

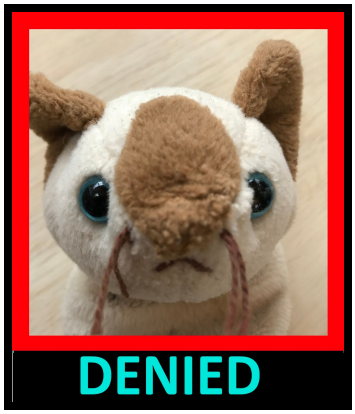

Figure-58: White Cat Profile

Documentation

Each of the sections below are organized in the order in which they are to be executed in the terminal. The documentation presupposes that the reader has already set up the peripherals mentioned in the implementation section.

Documentation: Data Collection

After obtaining the Camera matrix and distortion matrix, one can then go about collecting training data that will be used to build the convolutional neural network model. In order to simplify the process and create a clear training set, the possible scenarios that were trained on are no cats present (kat 0), white cat present (kat 1), and brown cat present (kat 2). The first step is to run the following code in order to capture images of the no cat (kat 0) scenario:

```
python collect_training_data_kat0.py
```

This application will start the camera and provide the user with a display that shows what the camera is observing on the runway to the cat door. The user will then be able to use the spacebar on the keyboard in order to collect images for the training set. It is important for the user to collect to not place any images of the cats in this training set as it will lead to issues with training the neural network. The user will also have to collect images using this procedure under a multitude of different lighting scenarios. This will ensure that the neural network does not misidentify a shadow on a blank scenario as a “cat present” scenario.

After collecting a large number of images of the blank runway under different lighting scenarios, the user can then press “ESC” on the keyboard to leave the first program and run the next command:

```
python collect_training_data_kat1.py
```

This will start up the next application that will be used to collect images of the white cat. Similar to the process above, the user will again use the spacebar to collect images as well as have the ability to see what the camera is collecting via a display. The user will collect the same number of images of the white cat as she did in the blank runway scenario. The one difference between this setup and the one above is that the user in this setup must place the cat on the runway in many different positions in order to ensure that the neural network will have “observed” different parts of the cat in question. The user will again press “ESC” in order to exit the program.

In order to collect images of the brown cat, the user will nearly follow the exact same procedure that was used to collect data on the white cat. The only difference will come in the command that the user will use to do this:

```
python collect_training_data_kat2.py
```

The rationale behind this duplication is that it ensures that the data collected on the three scenarios will remain clean as it reduces the possibilities of human error during the collection procedure.

Another program that may be necessary is the following:

```
python rename_kat_images.py
```

This program will only really be necessary if the user decides later on that she did not initially collect enough images to properly build the neural network. This program simply renames every image that was collected during the collection procedure mentioned above. There is one proviso that one must be aware of when using this program. If the user does not modify the zfill argument shown below before running the renaming program, some of the data that was collected may be overwritten and lost. The suggested workflow for the user to implement would be one where the user changes the zfill before running any of the data collection programs listed above, running the data collection programs, and then rerunning the naming program after changing the zfill argument to another number entirely.

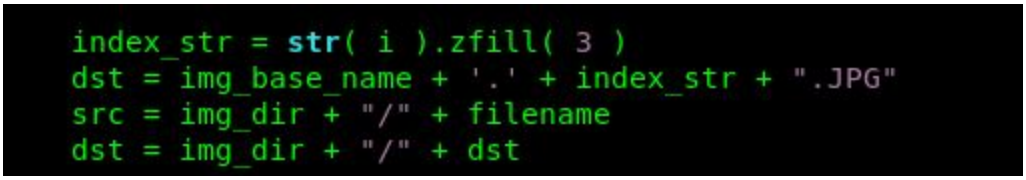

Figure-59: Renaming image files warning

Documentation: Building Convolutional Neural Network

After collecting all of the requisite image data in the preceding section, the user can now move on to the process of building the neural network model. In order to successfully complete this portion of the project, the user must make the data easier to process for the neural network, separate the data into training and testing sets, train a network based around the aforementioned sets, and validate that the neural network is able to make predictions accurately. All of these things will be completed by three commands. The first of these commands tackles the step before the data processing and data set splitting:

```
python copy_kat_images.py
```

This program copies all of the images and places one copy of the data into a testing directory and the other in a training directory. The two sets of data are then processed by the following command:

```
python create_datasets.py
```

This program does two things: it relabels the training data and resizes both sets of data. The data in the training directory is labeled with one-hot encoding in order to ensure that the neural network does not use the original image labels during its training phase.

The last step of this section is to actually create the convolutional neural network. The command that will do this for the user is:

```
python create_cnn_model.py
```

This program takes the training dataset created with the last two commands and runs it the images through a collection of processing layers that attempt to identify an image based on features that are present in it. This process is implemented by using the TensorFlow library mentioned in the earlier sections of this document. Based on the features of an image it observes, it tries to assign the correct label to the image. It does this by assigning weights to the connections between features and an image's label. During this process, the neural network's correct connections are identified and the weights that led to this correct choice is then saved as a part of the neural network model. These weights are what the neural network in the following section will use in order to identify the cats.

Documentation: Running Convolutional Neural Network

After the last section's final command, the user is now in possession of a convolutional neural network model, which is just the weights that provided the neural network with the correct predictions. The user will then run the following command to access those weights and use them in realtime to identify the cat when it appears in front of the camera:

```
python cnn_camera_pipeline.py
```

This command loads the model mentioned above, starts the camera, runs a display that shows what the camera is observing, and builds a neural network using only the weights from the model. This command also starts the threads needed in order for the peripherals mentioned in the implementation section. The user can now observe the neural network correctly identifying the cat and either allowing or disallowing access to the other side of the cat door, depending on what cat has presented itself in front of the camera.

References

Camera Setup

Calibration Configuration

1. Beauchemin, S. S., & Bajcsy, R. (2001). Modelling and Removing Radial and Tangential Distortions in Spherical Lenses. *Multi-Image Analysis Lecture Notes in Computer Science*, 1–21. doi: 10.1007/3-540-45134-x_1
2. Camera calibration With OpenCV. (n.d.). Retrieved from https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.
3. Drap, P., & Lefèvre, J. (2016). An Exact Formula for Calculating Inverse Radial Lens Distortions. *Sensors*, *16*(6), 807. doi: 10.3390/s16060807
4. Heikkila, J., & Silven, O. (1997). A four-step camera calibration procedure with implicit image correction. *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. doi: 10.1109/cvpr.1997.609468

Perspective Transformation

5. Canu, S. (2019, September 13). Perspective transformation – OpenCV 3.4 with python 3 Tutorial 13. Retrieved from https://pysource.com/2018/02/14/perspective-transformation-opencv-3-4-with-python-3-tutorial-13/.
6. Rosebrock, A. (2018, August 2). 4 Point OpenCV getPerspective Transform Example. Retrieved from https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/.

Convolutional Neural Network Setup

7. Tflearn. (2018, July 27). tflearn/tflearn. Retrieved from https://github.com/tflearn/tflearn/tree/master/examples.
8. Damien, A. (n.d.). Deep Neural Network Model. Retrieved from http://tflearn.org/models/dnn/.
9. Lenail, A. (n.d.). NN-SVG: Publication-ready NN-architecture schematics. Retrieved from http://alexlenail.me/NN-SVG/LeNet.html.
10. Mishra, P. (2019, July 20). Why are Convolutional Neural Networks good for image classification? Retrieved from https://medium.com/datadriveninvestor/why-are-convolutional-neural-networks-good-for-image-classification-146ec6e865e8.

Peripheral setup

Audio Circuit

11. Build a Great Sounding Audio Amplifier (with Bass Boost) from the LM386. (2018, June 21). Retrieved from http://www.circuitbasics.com/build-a-great-sounding-audio-amplifier-with-bass-boost-from-the-lm386/.
12. Use tone() with Arduino for an Easy Way to Make Noise. (2018, October 31). Retrieved from https://www.programmingelectronics.com/an-easy-way-to-make-noise-with-arduino-using-tone/.

Servo Motor Latch

13. Earl, B. (n.d.). Adafruit PCA9685 16-Channel Servo Driver. Retrieved from https://learn.adafruit.com/16-channel-pwm-servo-driver/hooking-it-up.
14. Warren, S., & Tan, A. (2019, October 30). NVIDIA/jetson-gpio. Retrieved from https://github.com/NVIDIA/jetson-gpio.
15. Jetson Nano - Using I2C. (2019, September 29). Retrieved from https://www.jetsonhacks.com/2019/07/22/jetson-nano-using-i2c/.
16. Element14: NVIDIA Jetson Nano Developer Kit Pinout and Diagrams. (2019, May 21). Retrieved from https://www.element14.com/community/community/designcenter/single-board-computers/blog/2019/05/21/nvidia-jetson-nano-developer-kit-pinout-and-diagrams.

Threading

17. Chaturvedi, S. (n.d.). Let's Synchronize Threads in Python. Retrieved from https://hackernoon.com/synchronization-primitives-in-python-564f89fee732.
18. Python Library Reference: Event Objects. (n.d.). Retrieved from https://docs.python.org/2.0/lib/event-objects.html.
19. threading - Thread-based parallelism. (n.d.). Retrieved from https://docs.python.org/3/library/threading.html.

Python language

20. Van Rossum, G., & Drake Jr, F. L. (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Appendix

Python Libraries Installation Process

In the main directory of the project, there is a file named “setup_nano.py”. This file will install all the non-standard libraries required to run the project.

```
#!/bin/sh
if ! [ -x "$(command -v python3)" ]; then
        sudo apt-add-repository ppa:jonathonf/python-3.6
        sudo apt-get install python-3.6
        sudo apt-get install libatlas-base-dev gfortran
        sudo apt-get install git cmake
        sudo apt-get install libhdf5-serial-dev hdf5-tools
        sudo apt-get install python3-dev

fi

if  ! [ -x "$(command -v pip)" ]; then
        wget "https://bootstrap.pypa.io/get-pip.py"
        sudo python3 get-pip.py
        if [ -x  "$(command -v pip)" ]; then
                rm get-pip.py
                pip install --user numpy
                pip install --user --extra-index-url /
                        https://developer.download.nvidia.com/compute/redist/jp/v42 /
                        tensorflow-gpu==1.13.1+nv19.3
                pip install --user pickle5
                pip install --user glob2
                pip install --user matplotlib
                pip install --user adafruit_pca9685
                pip install --user playsound
                pip install --user board
                pip install --user cv
                pip install --user RPi.GPIO
                pip install --user tflearn

        fi
else
        pip install --user numpy
        pip install --user --extra-index-url /
                https://developer.download.nvidia.com/compute/redist/jp/v42 /
                tensorflow-gpu==1.13.1+nv19.3
        pip install --user pickle5
        pip install --user glob2
        pip install --user matplotlib
        pip install --user adafruit_pca9685
        pip install --user playsound
        pip install --user board
        pip install --user cv
        pip install --user RPi.GPIO
        pip install --user tflearn
fi
```

Figure-60: Bash Script to Install Required Libraries

In order to run this file, you’ll have to use the following commands:

```
Chmod +x setup_nano.sh
./setup_nano.sh
```