Machine Learning Camera

University of Evansville

Christian Olson

Advisor: Mark Randall

4/29/2019 Evansville, Indiana

Table of Contents

1. Introduction

The goal of this project was to write a program that uses a camera to recognize patterns and images based on machine learning. Most cameras simply take a picture and store it in memory, however with modern advances in computing technology it is possible to teach a computer to understand the images that it receives from a camera input. More specifically, this camera is connected to a small Raspberry Pi computer which reads the images in real time and give feedback via a connected monitor. In addition to recognizing its target pattern, this camera is a base platform and proof of concept which can be modified to assist future University of Evansville competition teams going to the Trinity College Fire-Fighting Home Robot Competition by acting as the visual input device for their fire-fighting robot in order to identify specific target objectives of the competition.

2. Problem Definition

When a human or an animal looks at an object they essentially capture an image of the object, which is processed through their brains to provide them with the knowledge and understanding of what they are looking at. For instance, if you looked at a cat the rods and cones in your eyes would capture the image and send it to your brain, which would process the image and tell you that what you are looking at matches your preconceived notion of what a cat ought to look like. Computers do not inherently have this ability which we take for granted, their understanding of a picture of a cat stops at the level of image and does not naturally continue on to recognizing a cat. The primary aim of this project is to provide a machine learning algorithm which will be able to look at several examples of an abstract concept and learn how to understand what is or is not an example of this object. This is done by forming a reference vector library of processed image vectors and the values of their abstract representations. This project serves as a prototype and proof of concept for other applications of machine learning computer vision systems such as facial recognition systems, self-driving cars, medical imagers, industrial monitors, and agricultural monitors.

2.1. Trinity Fire-Fighting Home Robot Competition

Future iterations of this camera will be used in conjunction with our Trinity Fire-Fighting Home Robot team in order to provide it with the visual inputs required for the competition. The Trinity Fire-Fighting Robot competition is broken up into three levels, in order to advance to the next level a robot must complete the tasks required by the previous level. In the first level there is a single arena area with predetermined dimensions and layout which represents a simulated house, the goal of level one is to find and put out a single candle which has been set up somewhere in the simulated house. In level two the robot must complete the same task but the layout of the simulated house is randomized. In level three the robot is also required to rescue a simulated baby which is located in a simulated cradle somewhere in the simulated house, additionally level three consists of two arenas connected by a bridge. The Computer Vision Camera comes in to play in the third and final round of the competition, where one of the options is for the robot to identify the simulated cradle which contains the simulated baby by identifying the patterns on its front and sides. Additionally, it will be used to identify the

Figure 1: Previous University of Evansville Robots

safe area where the simulated baby should be taken in order to score. [1] (see Appendix A for target images).

2.2. Client Requirements

- The Machine Learning Camera recognizes the cradle base patterns and the safe area symbol.
- The Machine Learning Camera recognizes a chess pawn on a standard chess board.
    - Note: there was a miscommunication, this second requirement was never expected, but was left in the original proposal by mistake.

2.3. Requirements Completed

- The Machine Learning Camera recognizes the central cradle target accurately.

3. Design Solution

The solution to the problem of developing a camera, which would recognize what it is looking at, is to use image processing to accentuate the relevant features of the image and then to format the image into a vector of features which can be fed into a machine learning algorithm. For the implementation used in this project a processed image vector is given to a program which implements the K-Nearest Neighbors algorithm on a large sample set of processed image vectors which have already been identified. This produces a simple yes or no answer to whether the image contains the target object.

3.1. Hardware

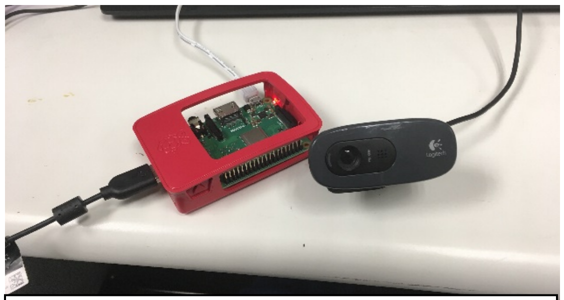


Figure 2: The Machine Learning Camera

Though the Machine Learning Camera is primarily a software project; no software exists in a vacuum. The Machine Learning Camera is composed of two parts: A Raspberry Pi 3B+ single board computer [3] and a Logitech C270 HD Webcam [5]. The exact camera does not make much of a difference as the software will act on the image files captured by it rather than directly on its input, though the program expects all images to be 640x480 pixels. Additionally, as it is, the device requires a user interface system (monitor, keyboard, mouse) to operate, though future changes can be easily made to reconfigure the project for integration with a robot, these changes would include setting the device such that the program starts on boot and make the output on an IO pin rather than to a monitor.

3.2. Software Design

The Machine Learning Camera's software is the primary component of the project. There are three primary programs and a function library which were written for the project. The first program is the image capture program imcap.cpp which captures images into the desired file. The second program is the vectorization program vectorize2.cpp, so named because it is the second iteration of the program, which can read in the images from a file and print the resulting vector representation along with image values into an output file which serves as the sample data set for the K-Nearest Neighbors algorithm. The last, and arguably most important program is the actual runtime program KNN.cpp which uses the K-Nearest Neighbors algorithm to perform the image recognition and machine learning. The KNN program has four separate run modes, each of which call the vectorize function to turn their captured input images into formatted vectors and the compare_knn function to perform the actual K-Nearest Neighbors analysis. The four run modes are: single capture, single capture with user confirmation, single capture with image process display, and continuous capture and compare. (see appendix B for code, see appendix C for function and program usage guide).

3.2.1. Vectorize Function

The vectorize function is the backbone of the Machine Learning Camera. The function takes an image matrix and returns a 12-intager vector which represents the image mathematically. The image matrix is first put through an image processing stage which performs the following transforms and operations:

1. Input image is converted to grayscale to reduce complexity.
2. Image intensity is decreased by 75% to help remove small differential edges from the later Sobel filters.
3. Image is blurred to remove extraneous details; this is done because the patterns the device is looking for are composed of hard edges which will still be preserved through the blurring process.
4. Edges are detected by using a Sobel filter in both the X and Y dimensions and adding the resulting images together.
5. The image is resized to one quarter the original size in each dimension, this is done to make the image more manageable in the vectorization algorithm.

6. The image has its pixel intensities multiplied by four.

These processes are performed using the OpenCV library [4], which is a free open source library designed for image processing and computer vision. The images produced in the image processing step are converted to vectors by first iterating over every pixel to produce a 160-intager vector containing the sums of the pixel intensities of every column as well as a single sum of all the pixels. The column intensity vector is then divided by the total intensity and multiplied by 10,000 in order to cause every image to have a total value of 10,000 at this stage. The standardized vectors are then used to find the base 2 logarithmic histogram of the column intensities represented as a 12-intager vector. This vector is then returned to whatever function called the vectorize function. There is also a variant vectorize_for_demo function which saves images partially through the processing function for the purpose of demonstration and debugging (see appendix B for code, see appendix C for function and program usage guide).

3.3. K-Nearest Neighbors

The K-Nearest Neighbors algorithm, or KNN, is one of the simplest and most versatile machine learning algorithms. It takes arguments K, S, and V, where S is the sample set of labeled image vectors, K is the number of vectors which will be used for the comparison, and V is the input image vector which is supposed to be identified. The KNN algorithm treats the vectors as points in higher dimensional space and compares the distances between V and each vector in the set S, keeping the K vectors from set S

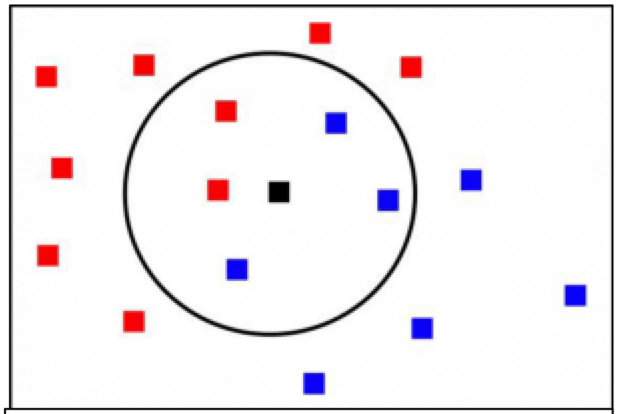


Figure 4: K-Nearest Neighbors illustration with a K-value of 5 [6]

with the shortest distance from V. These K image vectors have their labels compared, whatever the majority of the K labels are is what V is labeled. When running in learning mode, a human operator determines whether the answer is correct or not, and V is added into S with the appropriate label, increasing the size of S and thus increasing the accuracy of the KNN algorithm. Though the KNN algorithm is technically not a learning algorithm, as it does not self-instruct, it is, however, colloquially considered a machine learning algorithm. [2]

3.4. Considerations

Any project undertaken should be considered from all angles before it's realization. Below are the considerations of safety, environmental protection, social and political ramifications, and the manufacturability of the Machine Learning Camera.

3.4.1. Safety

With any electronic device there are certain safety concerns to be considered. Any interaction between the Machine Learning Camera and water is inadvisable, users are also advised to keep the device out of reach of small children. No part of the Machine Learning Camera is edible.

3.4.2. Environmental Protection

All components of the Machine Learning Camera are, in part or in whole, recyclable, and should be accepted at any electronics recycling location.

3.4.3. Social and Political Ramifications

In the modern social and political landscape of the Western World there are many concerns as to the potential societal dangers of a camera which can identify items by sight. These concerns run the gamut from very real concerns about invasion of personal privacy to the most outlandish theories regarding robot uprisings. While it is possible that the Machine Learning Camera could be used in these ways, it is considered to be a very unlikely risk that this project will contribute to any such problems.

3.4.4. Manufacturability

The hardware for the Machine Learning Camera is relatively inexpensive, and would thus be quite simple to mass produce. There is, however, the issue that the project will be using devices, such as the Raspberry Pi [3], who's manufacturing rights belong to neither the creator nor to the client.

4. Results

The final result of the Machine Learning Camera is that it is a resounding success at detecting one of the Trinity Fire-fighting Robot Competition patterns (the center target, see appendix A). The maximum reliable detection range is 18 inches, not ideal for real-world applications, but serviceable within the confines of the competition testing environment. This project provides both a framework for future work along these lines and a proof of concept for the idea of using machine learning for this sort of small scale problem.

Figure 4: detection image

References

[1] Trinity College. (2018, October). ‘Trinity College Fire-Fighting Home Robot Contest 2019 Rules V1.0.’ Hartford, Connecticut. [Online]. Available: https://www.dropbox.com/sh/m0i9448atx5fa3l/AABnYrj5m2lRhOtFFz7JsYeba?dl=0&preview=TCFFHRC2019RulesV1.0.pdf. [accessed: 3- Dec- 2019].

[2] Brownlee, J. (2019). *K-Nearest Neighbors for Machine Learning*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/k-nearest-neighbors-for-machine-learning/ [Accessed 22 Mar. 2019].

[3] Raspberry Foundation, ‘Raspberry Pi — Teach, Learn, and Make with Raspberry Pi’, Raspberry Pi, 2018. [Online]. Available: https://www.raspberrypi.org/. [Accessed: 04-Dec- 2018]

[4] OpenCV Team, ‘OpenCV library,’ About - OpenCV library. 2018 [Online]. Available: https://opencv.org/. [Accessed: 05-Dec-2018].

[5] Webcam, C. and Webcam, C. (2019). *Logitech C270 HD Webcam, 720p Video with Built-in Mic & Lighting Correction*. [online] Logitech.com. Available at: https://www.logitech.com/en-us/product/hd-webcam-c270 [Accessed 22 Mar. 2019].

[6] Chattopadhyay, *researchgate.net*, 2015. [Online]. Available: https://www.researchgate.net/figure/In-this-k-Nearest-Neighbor-illustration-with-k-5-the-central-black-square-more_fig1_281289672. [Accessed: 29- Apr- 2019]

Appendix A: Trinity Fire-Fighting Home Robot Competition Images

Figure A: front side of cradle

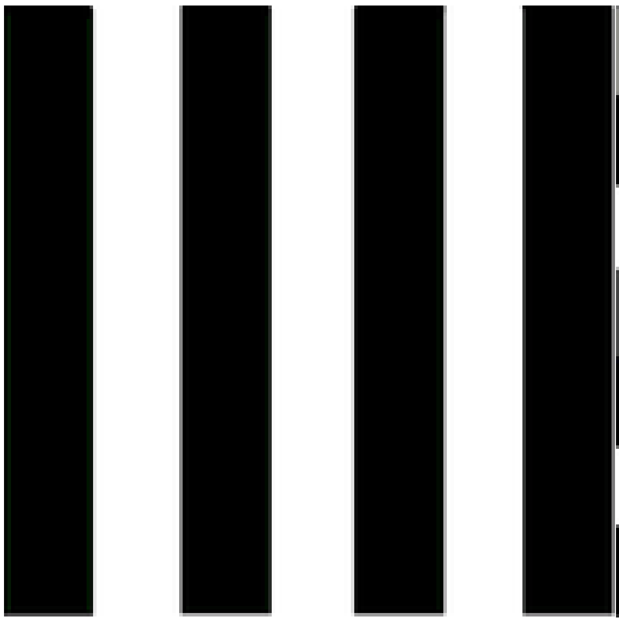


Figure B: right hand side of cradle

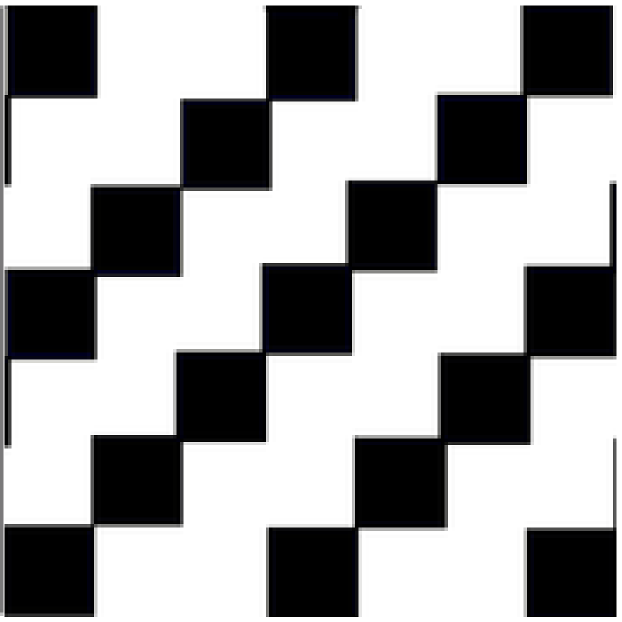


Figure C: left hand side of cradle

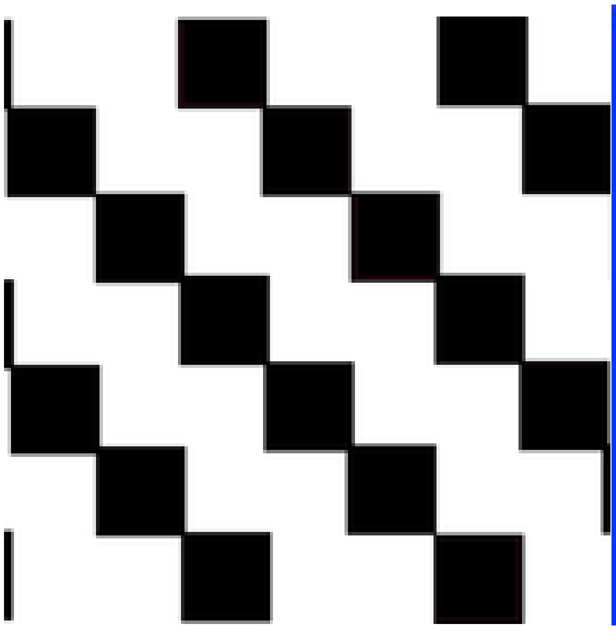
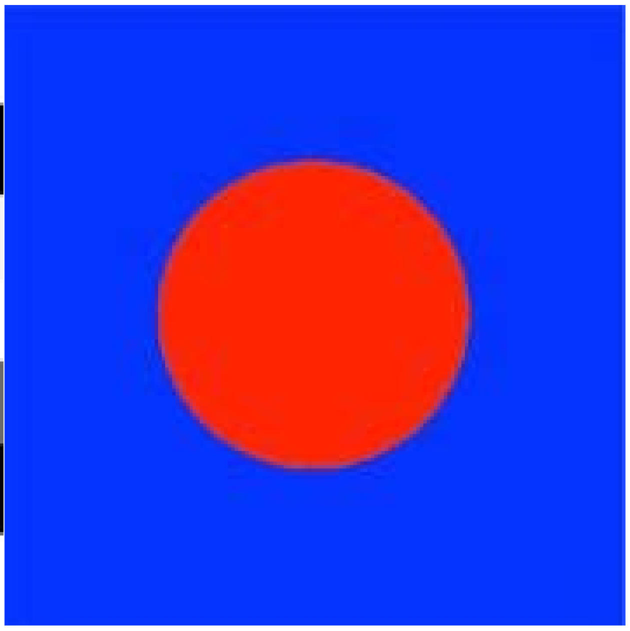


Figure D: safe area symbol



Appendix B: Source Code

Code for KNN.cpp

```
//KNN.cpp
#include <opencv2/opencv.hpp> //image processing
#include <fstream> //filestreams
#include <iostream> //iostreams
#include <vector> //for the output vectors
#include "vectorizer.hpp" //for vectorizing the captured images

using namespace std;

int compare_knn(int k, string vecfile, vector<int> input);
//return 1 if match, else 0
double vecdist(vector<int> A, vector<int> B);
//returns distance between A&B
void continuous_knn(int k, string infile, cv::VideoCapture cap);
//continuously call compare_knn and display the result

int main(int argc, char ** argv)
{
	if(argc < 4)
	{
		cout << "not enough arguments, expect "
		 << "\"./KNN input_filename K mode\"" << endl;
		return 1;
	}
	string vecfile = argv[1];
	int k = atoi(argv[2]);
	int mode = atoi(argv[3]);
	//1 for learning mode, 0 for regular, 2 for data_capture,
	//	3 for continuous runtime

	cv::VideoCapture cap(0);

	if(!cap.isOpened())
		return -1;

	if(mode == 3)
	{
		continuous_knn(k, vecfile, cap);
		return 0;
	}
	while(1)
	{
		cv::Mat frame;
		cap >> frame;
		cv::imshow("img", frame);
		if(cv::waitKey(10) >= 0)
		{
			break;
		}
	}
	cv::destroyWindow("img");
	cv::Mat image;
```

```
        cap >> image;
        vector<int> base2hist(12);
        if(mode == 2)
                base2hist = vectorize_for_demo(image);
        else
                base2hist = vectorize(image);

        int test_result;
        test_result = compare_knn(k, vecfile, base2hist);

        string imstr = "/home/pi/MLC/" + to_string(test_result) + ".png";
        while(1)
        {
                cv::imshow("result", cv::imread(imstr));
                if(cv::waitKey(10) >= 0)
                {
                        cv::destroyWindow("result");
                        break;
                }
        }
        if(mode == 1)
        {
                cout << "is this correct? (0 for no, 1 for yes)"<< endl;
                int val;
                cin >> val;

                ofstream printer;
                printer.open(vecfile, std::ofstream::out | std::ofstream::app);
                printer << endl;

                if(val == 0)
                        test_result = 1 - test_result;
                printer << test_result << " ";
                for(int i = 0; i<12; i++)
                {
                                printer << base2hist[i] << " ";
                }
        }
        return 0;
}
int compare_knn(int k, string vecfile, vector<int> input)
{
        vector<int> candidate(12);
        double candidate_dist;
        vector<double> neighbor_dists(k);
        vector<int> values(k);
        for(int i = 0; i < k; i++)
        {
                neighbor_dists[i] = 3000;//impossibly big number
        }
        ifstream neighborhood;
        neighborhood.open(vecfile);

        int value;
        while(neighborhood >> value)
        {
                for(int i = 0; i < 12; i++)
                {
                        neighborhood >> candidate[i];
                }
                candidate_dist = vecdist(candidate, input);
                if(candidate_dist <= neighbor_dists[0])
                {
                        //replace neighbor_dists[0]
                        neighbor_dists[0] = candidate_dist;
                        values[0] = value;
                        //sort neighbor_dists so that the highest is at index 0
                        for(int i = 1; i < k; i++)
                        {
                                if(neighbor_dists[i] >= neighbor_dists[0])
                                {
                                        double temp = neighbor_dists[0];
                                        neighbor_dists[0] = neighbor_dists[i];
                                        neighbor_dists[i] = temp;
                                        temp = values[0];
                                        values[0] = values[i];
                                        values[i] = temp;
                                }
                        }
                }
        }

        int result = 0;
        for(int i = 0; i < k; i++)
        {
                result += values[i];
        }
        if(result >= k/2)
                return 1;
        return 0;
}

double vecdist(vector<int> A, vector<int> B)
{
        double difsquaresum = 0;
        for(int i = 0; i < 12; i++)
        {
                        difsquaresum += pow((A[i]-B[i]), 2);
        }
        double dif = sqrt(difsquaresum);
        return dif;
}

void continuous_knn(int k, string infile, cv::VideoCapture cap)
{
        int test_result;
        vector<int> base2hist(12);
        cv::Mat frame;
        string imstr;
        while(1)
        {
                cap >> frame;
                cv::imshow("img", frame);
                base2hist = vectorize(frame);
                test_result = compare_knn(k, infile, base2hist);
                imstr = "/home/pi/MLC/" + to_string(test_result) + ".png";
```

```
            cv::imshow("result", cv::imread(imstr));
            if(cv::waitKey(10) >= 0)
            {
                  break;
            }
      }
}
```

Code for imcap.cpp

```
//imcap.cpp
#include "opencv2/opencv.hpp"
#include <string.h>
#include <stdlib.h>
#include <string>

int main(int argc, char** argv)
{
      if(argc < 4)
      {
            std::cout << "not enough arguments, expect \"./imcap"
            << " output_filename start_num stop_num\"" << std::endl;
            return 1;
      }
      cv::VideoCapture cap(0);
      if(!cap.isOpened())
            return -1;
      std::string fpath = "/home/pi/MLC/images/";
      fpath += argv[1];
      fpath += "/";
      while(1)
      {
            cv::Mat frame;
            cap >> frame;
            cv::imshow(argv[1], frame);
            if(cv::waitKey(10) >= 0)
            {
                  break;
            }
      }
      for(int i = atoi(argv[2]);i < atoi(argv[3]);i++)
      {
            std::string fname = fpath;
            fname += std::to_string(i);
            fname += ".png";
            cv::Mat frame;
            cap >> frame;
            cv::imshow(argv[1], frame);
            cv::imwrite(fname, frame);
            if(cv::waitKey(10) >= 0)
            {
                  break;
            }
      }
      return 0;
}
```

Code for vectorize2.cpp

```
//vectorize2.cpp
#include <opencv2/opencv.hpp> //image processing
#include <fstream> //filestreams
#include <iostream> //iostreams
#include <string> //will make filenames easier to mess with
#include <cmath> //log2
#include "vectorizer.hpp" //for the actual vectorization
#include <vector> //vectors

int main(int argc, char ** argv)
{
	if(argc < 6)
	{
		std::cout << "not enough arguments, expect \"./vectorize2 "
		 << "input_filename output_filename start_num stop_num value\""
		 << std::endl;
		return 1;
	}
	std::ofstream outfile;
	string infname = argv[1];
	string outfname = argv[2];
	outfile.open(outfname, std::ofstream::out | std::ofstream::app);
	std::string infolder = "/home/pi/MLC/images/";//path to all image folders
	infolder += infname + "/"; //specific folder
	for(int input_item = atoi(argv[3]); input_item < atoi(argv[4]); input_item++)
	{
		std::string infile = infolder + std::to_string(input_item) + ".png";
//specific image file to be vectorized
		cv::Mat image_alpha = cv::imread(infile);
		vector<int> base2hist(12);
		base2hist = vectorize(image_alpha);
		//WRITE TO FILE HERE
		outfile << atoi(argv[5]) << " ";
		for(int i = 0; i < 12; i++)
		{
			outfile << base2hist[i] << " ";
		}
		outfile << std::endl;
	}
	return 0;
}
```

Code for vectorizer.hpp

```
//vectorizer.hpp
#include <opencv2/opencv.hpp> //image processing
#include <iostream> //iostreams
#include <vector>

using namespace std;

vector<int> vectorize(cv::Mat image)
{
	cv::cvtColor(image, image, CV_BGR2GRAY);
	image.convertTo(image, -1, .25, 0);
```

```
        cv::GaussianBlur(image, image, cv::Size(11,11), 0, 0, cv::BORDER_DEFAULT);
        cv::Size dsize;
        cv::Mat imgradx, imgrady, grad;
        cv::Sobel(image, imgradx, -3, 1, 0, 3, 1, 0, cv::BORDER_DEFAULT);
        cv::Sobel(image, imgrady, -3, 0, 1, 3, 1, 0, cv::BORDER_DEFAULT);
        cv::convertScaleAbs(imgradx, imgradx);
        cv::convertScaleAbs(imgrady, imgrady);
        cv::addWeighted(imgradx, 0.5, imgrady, 0.5, 0, grad);
        cv::resize(grad, grad, dsize, 0.25, 0.25);
        grad.convertTo(grad, -1, 4, 0);
        vector<int> vertical(160);
        double intensity = 0;
        for(int i = 0; i < 160; i++)
        {
                vertical[i] = 0;
                {
                        for(int j = 0; j < grad.rows; j++)
                        {
                                vertical[i] += (int)grad.at<uchar>(cv::Point(i,j));
                                intensity += (int)grad.at<uchar>(cv::Point(i,j));
                        }
                }
        }
        intensity = intensity/10000 + 1;
        vector<int> base2hist(12);
        for(int i = 0; i < 160; i++)
        {
                int histval = log2((vertical[i]/intensity) + 1);
                if(histval > 11)
                        histval = 11;
                else if(histval < 0)
                        histval = 0;
                base2hist[histval]++;
        }
        return base2hist;
}

vector<int> vectorize_for_demo(cv::Mat image)
{
        cv::imwrite("/home/pi/MLC/tests/stage_1_input.png", image);
        cv::cvtColor(image, image, CV_BGR2GRAY);
        cv::Mat A = image;
        cv::imwrite("/home/pi/MLC/tests/stage_2_grayscale.png", A);
        image.convertTo(image, -1, .25, 0);
        cv::imwrite("/home/pi/MLC/tests/stage_3_darkened.png", image);
        cv::GaussianBlur(image, image, cv::Size(11,11), 0, 0, cv::BORDER_DEFAULT);
        cv::imwrite("/home/pi/MLC/tests/stage_4_blurred.png", image);
        cv::Size dsize;
        cv::Mat imgradx, imgrady, grad;
        cv::Sobel(image, imgradx, -3, 1, 0, 3, 1, 0, cv::BORDER_DEFAULT);
        cv::imwrite("/home/pi/MLC/tests/stage_5A_x-gradient.png", imgradx);
        cv::Sobel(image, imgrady, -3, 0, 1, 3, 1, 0, cv::BORDER_DEFAULT);
        cv::imwrite("/home/pi/MLC/tests/stage_5B_y-gradient.png", imgrady);
        cv::convertScaleAbs(imgradx, imgradx);
        cv::convertScaleAbs(imgrady, imgrady);
        cv::addWeighted(imgradx, 0.5, imgrady, 0.5, 0, grad);
        cv::imwrite("/home/pi/MLC/tests/stage_6_added gradients.png", grad);
        cv::resize(grad, grad, dsize, 0.25, 0.25);
        cv::imwrite("/home/pi/MLC/tests/stage_7_resized.png", grad);
        grad.convertTo(grad, -1, 4, 0);
        cv::imwrite("/home/pi/MLC/tests/stage_8_sharpened.png", grad);

        vector<int> vertical(160);//intensity values of entire columns
        double intensity = 0;//total intensity of all pixels

        for(int i = 0; i < 160; i++)
        {
                vertical[i] = 0;
                {
                        for(int j = 0; j < grad.rows; j++)
                        {
                                vertical[i] += (int)grad.at<uchar>(cv::Point(i,j));
                                intensity += (int)grad.at<uchar>(cv::Point(i,j));
                        }
                }
        }
        intensity = intensity/10000 + 1;
        vector<int> base2hist(12);
        for(int i = 0; i < 160; i++)
        {
                //convert the 160 column intensities into a 12 intager long base
                //      2 logarithmic histogram
                int histval = log2((vertical[i]/intensity) + 1);
                if(histval > 11)
                        histval = 11;
                else if(histval < 0)
                        histval = 0;
                base2hist[histval]++;
        }
        return base2hist;
}
```

Appendix C: Program and Function Usage Guide

imcap.cpp

expected command line arguments: output_filename, start_num, stop_num

imcap.cpp displays the current live camera feed until the user hits a key indicating that it should start capturing images. N images will be captured where N is stop_num – start_num. These images will be saved as i.png, where i is a number between start_num and stop_num, in the output file designated output_filename. Images are captured this way every 10 microseconds. Capture may be interrupted at any time by pressing any key on the keyboard.

vectorize2.cpp

expected command line arguments: input_filename, output_filename, start_num, stop_num, value

vectorize2.cpp converts N images, where N is stop_num – start_num, from the input file input_filename into vector entries in the output file output_filename. Each image whose name is I.png, where i is a number between start_num and stop_num, is read in as an image matrix and given to the vectorize function, the result of the vectorize function is written to the output file along with the given value.

KNN.cpp

expected command line arguments: input_filename, K, mode

KNN.cpp is the main program of the Machine Learning Camera. It has four separate run modes, selected by the command line argument mode.

Mode 0: standard single capture. Allows the user to take a single capture from the camera, then sends that image to the vectorizer function, that result is then sent to the compare_knn function. The result is then displayed to the user, either that the target was identified, else it was not.

Mode 1: teaching single capture mode. Same as mode 0 except that the user tells the program whether it was correct or incorrect after its result is displayed. The correct value and the image vector are then added to the data set in the input file input_filename.

Mode 2: image processing display mode. Same as mode 0 except that the vectorize_for_demo function is called in place of vectorize.

Mode 3: continuous run mode. This mode causes the program to call the continuous_knn function which indefinitely performs vectorization and KNN on the camera input. This mode would be used during any practical application.

int compare_knn(int k, string vecfile, vector<int> input)

Arguments:

k: the number of samples to be kept as the nearest for comparison.

vecfile: the filename of the dataset for the KNN algorithm.

input: the vector which the dataset is being compared with to find the closest entries.

Returns: 1 if input is a match with the target image, 0 otherwise.

The compare_knn function iterates over the dataset, running each through the vecdist function with the input to determine which are the closest to it. These K-Nearest Neighbors are then compared to see what the majority of their values are. If the majority are a match, then a 1 is returned, else a 0 is returned.

double vecdist(vector<int> A, vector<int> B)

Arguments:

A, B: two vectors of length 12 which will have their distances compared.

Returns: the Pythagorean distance between A & B.

The vecdist function compares the two input vectors A & B by using a 12 dimensional version of the Pythagorean theorem.

void continuous_knn(int k, string infile, cv::VideoCapture cap)

Arguments:

k: the number of neighbors compared in the KNN algorithm.

infile: the filename of the dataset for the KNN algorithm.

cap: the image capture connection to the camera.

Returns: void

The continuous_knn captures images and has them vectorized by the vectorize function then sends the vectors to the compare_knn function, the result is then displayed to the monitor.

vector<int> vectorize(cv::Mat image)

Arguments:

image: the image which will be vectorized

Returns: a 12 integer vector which represents the image for the KNN algorithm.

See section 3.2.1 for specifics on the transforms performed.

vectorize_for_demo

Arguments:

image: the image which will be vectorized

Returns: a 12 integer vector which represents the image for the KNN algorithm.

Same as vectorize except that images are periodically saved for future observation by the user.