Android based Object Detection and Classification: Modeling a Child's learning of what's Hot/Cold

Matthew L Weber

***Abstract*—As part of a young child's learning process, there are specific items that are discovered through the embodied experience. It's been shown that an infant is going to using their hands and feet to touch things and characterize how they experienced the objects. The goal of this project is to model that learning, by organizing the characteristics of learned objects, allowing a prediction of what new similar objects might be like. Success would be the software/robot determining if a object is going to be hot or code based on previous experience.**

***Index Terms*—Self Organizing Map, SOM, OpenCV, Android, Developmental Robotics, AI**

I. Introduction

THE concept is to attempt to model a child's learning process, when related to predicting what new objects are like based on previously learned objects. A object is defined for this project as something like a ice cream cone or a pizza. It's something that consists of color and a specific set of environmental traits (hot/cold, luminous, etc). This experiment takes in those traits via imagery and environment sensor data to define what an object is like. That data is then used to form a model of relationships of similar objects that can be later compared to a new object. When a new object is discovered, the prediction process would try to fit it to an already learned object. Even if the new object isn't predicted correctly, there is still a valid path in it's process to learn that new objects characteristics. If that path is taken, the algorithm performs a confirmation step where a correction could be made to then correctly learn that new object's characteristics. This concept is relying on using multiple sensor data sources to decrease error in the learning/predicting algorithm, but may still run into cases where the noise is to high to get an understanding of what is being analyzed. Understanding this noise is going to be part of the experimentation process and may limit the initial project inputs to controlled background colors and images that appeal to the available image detection algorithms.

This concept also strives to provide the user, without an embodied robot, a method for doing sudo embodiment through the use of hand sensors and eye camera. Even though only a subset of sensors are being used, the design allows for further enabling of more sensors as needed to enhance/supplement the existing sensory data. Sensors like a camera could be adjusted to also enable video input instead of the initial still shots currently planned. The flexibility in this concept is rooted in the open standards based communication protocols and libraries used to leverage well abstracted hardware.

This project is targeted to students who have simple technology like their cell phone and micro-controller kits available to run experiments on. With the end goal of providing a framework that allows object and sensory input learning to be organized with relationships. This framework could be then used to take a Android based device and make it a brain for a robot. There have been some attempts to use a Android device in a cellbot[6] as shown in Figure 1, but the cellbots seem to rely on preprogrammed functionality. By using the concepts in this project, the cellbot would become a powerful platform for prototyping autonomous movement.

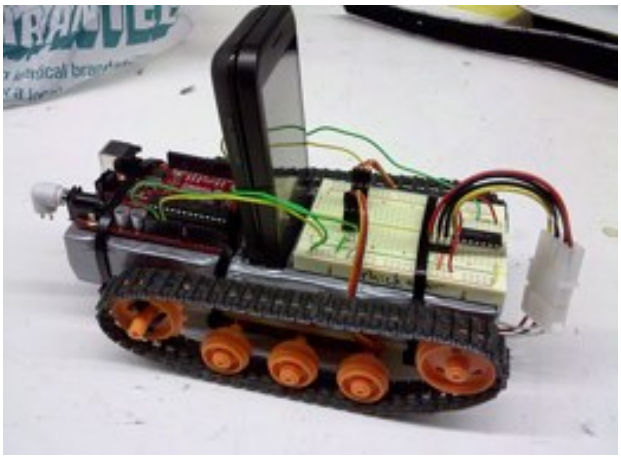

Fig. 1. An example of a Android based robot.[6]

Based on initial research this looks like the first attempt at utilizing the computer vision and neural networks on a Android platform. The new application would processes inbound sensory data and create the necessary predictions for learning of new objects. Since this is an resource intense process it should have some interesting results and opportunity for further study (discussed below).

II. Changes from Proposal

One of the first complications while creating the data processing algorithms was the embodiment principle and the assumptions required to "work around" not having a robot that is self aware. For example, if the sensors take in input by using a simulated arm that is manually moved to touch an object, the visual image is going to have the arm in the picture. Without actively removing that arm from the picture it's going to bias the image processing results at varying levels based on the amount and color of the arm in the picture. So to resolve this for this experiment, the sensor data was gathered separately from the image to allow a clean image to be fed into the processing algorithm.

Anther issue was related to the amount of test data and the size of the SOM. When a large SOM lattice(matrix) is allocated,

the amount of time required to calculate the weighted distances and find the best matching unit (BMU) grows exponentially. The BMU is calculated by looking at surrounding nodes and figuring out the best match for the input vector being located (The concept is discussed in more detail later on). This wasn't an issue until loading the application onto a phone and not having any optimization for the floating point matrix math. To work around this issue small data sets were used and calculated with shorter training intervals. This provides a reasonable set of match results and shows that the algorithms work correctly. However the PC based test applications (developed to verify the algorithms) can run at full speed and have been demonstrated with long training intervals and large data sets. Those results will be discussed below and some concepts are discussed describing the optimizations that could be performed on the mobile embedded platform by relocating some of the math code to gain a performance improvement.

The last item relates to some really good feedback in the proposal review noting something overlooked. It related to objects that were pictured in the proposal as the test cases. Most of those objects are containers with the contents having specific properties. So there was a unintended assumption that all those objects would inherit all the properties. So if a picture is taken of a bowl of soup, the learned concept is that a bowl of soup can be hot or cold, not that soup could be hot or cold. To work around this issue a new set of objects (mostly non-food based) and a process to test them is explained in the experiment section. It looks at individual items with limited noise in the image background and combines that with other sensor inputs. Since this experiment is centered around the processing and identification of similar objects, it's assuming a simple case of image inputs. Obviously there are much more complicated scenarios where the end goal would be to extract specific objects out of a noisy picture and then relate those to similar or exact learned property that's recalled based on inputting the object into the learning algorithm.

III. Existing Research

A few papers/projects have approached learning using Self-Organizing Maps (SOM). This project is modeled partly after "A System for Learning Basic Object Affordances using a Self-Organizing Map" [4], "Bringing up robot: Fundamental mechanisms for creating a self-motivated, self-organizing architecture"[8], and "Sense of Touch in Robots With Self-Organizing Maps"[13]. Those papers looks at the effectiveness of the SOM and how it performs learning. It also evaluated the effectiveness of the different training sets. To deviate slightly from what that paper was investigating, this project eliminates the predefined training set and focuses on the fact that over time the robot would develop it's own training set that would evolve to be effective at making object affordances. This paper's experiments are partially verification that a larger scale learning using the suggested approach is practical.

The best examples for understanding the SOM concepts were found on a SOM resource website[1] and the original paper[18] [11]. The SOM sudo code was also leveraged from the SOM resource website [12]. One of the key limitations of the SOM design was the matrix size and the processing power required as more items are added to the input vectors of the SOM. Since it seemed that a system that is constantly learning and adding more input vectors would need to dynamically grow the matrix that holds each of the relationship nodes. New concepts for optimizing this and segmenting parts of the SOM into processable pieces, quickly becomes a major issue. Another relevant experiment, found that SOMs were efficient in forming relationships in images when presented with the applicable image properties [5]. The research followed the Kohonen engineering applications paper[16] that suggests the application of a SOM to the Computer Vision area. That Kohonen paper specifically covered the ways to approach distilling the correct data out of imagery in order to organize and learn using a SOM. The example implementations provided an overview into the usage of a SOM based on the concepts presented in the original Kohonen paper [18] and applied them to a couple different test scenarios with lots of data. This project leveraged the image processing concepts in that research and extracted the relevant texture and uniqueness that could be distilled into a format that the SOM could process. Since that research only provided a review of how the concept was verified, this project had to take a example SOM and implement the concepts needed to do the image processing it discussed. So the first step was to extend the concept of SOM input vectors beyond a single variable (just an image or just sensor, etc), and instead generalize a data set that consisted of normalized image and sensor data.

A similar experiment, to this paper, was done with a robot using sound to recognize objects and a SOM to organize and predict the current object based on the previously learned sound representations when the object is dropped[19]. Extending on this concept, an attempt is made in this paper show the same concept, but take in multiple different sources of sensory input and utilize a single SOM to produce a best fit for object similarities.

IV. Approach

As mentioned above in the "Changes from Proposal" section, some challenges were found with having multiple objects in a image. It's fine if the goal is to learn a scene, but it complicates the concept of having a 2-year-old child grasping the concept of separating multiple objects within a picture and being able to specifically select the objects. So the concept for this experiment has been adjusted to interact with one object at a time, in order to have a chance at recognizing new objects based on previous experience. However during some initial testing, it was found that the multi-object scenes should work with this algorithm, but the relationships they form are much more complex because of the large amount of detail being captured.

This software application is designed to not contain any relational knowledge when initially invoked. If the user chooses, they can load a "memory" from a previous run that would allow the robot to accelerated relearn what it had previously experienced. This keeps the learning process closely mimicking that of a child. Allowing the robot to interact with

each individual object and acquire knowledge about that object through a set of a sensors and imagery input.

Equipment

Because of logistics related to taking the class via distances education, this project is using simulation combined with some sensing hardware. The main component is an Android phone that collects sensor data and images. The sensor data comes from an external Arduino sensor package that would normally be attached to a robot's hand. The sensors being used are a one-wire temperature sensor and a resistive photo sensor. For the experiment, the sensors would be manually placed close to the object under study. In Figure 2, the high-level connectivity is shown between components.

1) Design: For a project with new concepts and complexity, there either needs to be large quantities of time or a good basis of existing resources (source code, conceptual dialog, etc). This project is designed around a few concepts that leverage existing work and combine a few proposed approaches to the problem area.

The first resource is the OpenCV library. It was ported last year to the Android platform and provides a set of image/video manipulation libraries that aid in doing the calculations required. A couple core demos using a good modular approach, allowed fast prototyping of a new concept. Since this library is an active Open Source project and used across multiple processing architectures, the resources for understanding how it works were plentiful. The concepts this project implemented were a camera preview engine to plug into OpenCV that provided greyscale histogram and a pixel texture processing. The end result is a general classification of the picture using a histogram and a texturization that captures more locational pixel density. The texture area calculation attempts to remove light level issues and also provides more content detail by breaking the image into nine sections and performing the calculation in each. A histogram is a graph of the amplitude of the number of pixels (read left(black) to right (white)). Shown in Figure 3 is a example image w/ it's respective histogram. For this project the histogram data would be stored as part of each input vector that represents an objects set of traits. The pixel texture calculation is performed on an image as shown in Figure 4. Each section of the image has it's 3 channels of color broken out and each channel has a average density calculated. This results in 27 values that represent the color intensity/density of each section.

The second is the Self Organizing Map (SOM). A SOM is an unsupervised self organizing multidimensional structure of data that can be used to organize vectors of input data.

Fig. 3. Example of a image and it's histogram[17]

A1	A2	A3
A4	A5	A6
A7	A8	A9

Fig. 4. When calculating the average pixel area, the image is broken into 9 sections.

During learning, a vector is presented to the SOM and has it's data points (weights) compared to another target node's weights. The calculation performed is the Euclidean distance between a SOM node and the input data points. Depending on the delta/error of the weights, the nodes and input vector readjust to better alight and reduce the delta. This results in a topology of nodes that are organized with every input vector connection optimized for minimal distance. Shown in Figure 5 the SOM lattice (matrix) is made up of the red nodes. Each of the green nodes are input vectors that have a relationship to every lattice node.

There were a number of good websites and papers that described the concept for using SOMs and as an added benefit, there were also base utility classes written in Java that could be adapted/ported to this application. So it seemed like a good fit for this project. Another feature SOMs presented was the capability to self organize as data points are added. This seemed like the best approach for "on the fly" decision making. Although there still is a challenge in coming up with large data sets to aid in that initial training, especially if accelerated learning is a requirement. Those data sets would present a fine line between a robot's experience creating the data set or if it's man made. For this project it

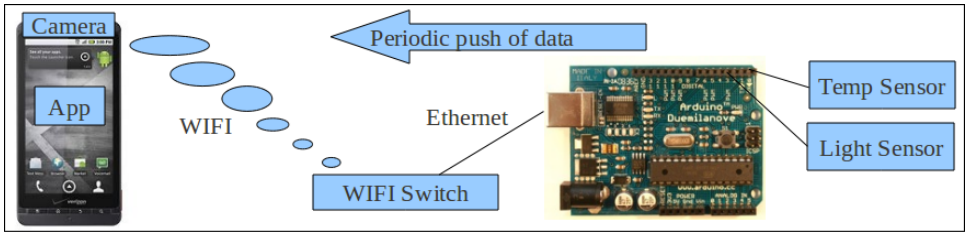

Fig. 2. Highlevel connectivity between hardware components.

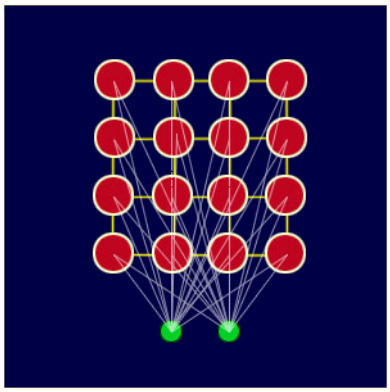

Fig. 5. An example of a SOM showing the lattice nodes and input vectors[1]

is possible to start with nothing and train as items are added, but it will take much longer. Obviously that learning process would closer model the human learning process, but instead of taking that path, this project tries to accelerate that learning by using some prelearned data sets. It takes in each item from those data sets and organizing the data to form relationships. So it definitely isn't doing a data search and lookup process, it is still learning from this data, just at an accelerated rate. Long term, having a repository for learned data could also be valuable to deployed to new robots. Just like how this project is accelerating the experiences of the robot. So building on that prelearned data, the actively acquired new data can be organized into the SOM and resulting relationships can be shown. These relationships are the key to allowing the robot to understand what something is similar to.

The third were concepts for gathering normalized data that could be fed in as part of a SOM input vector organization. These broke down into a couple areas (sensors and imagery). These areas were selected to create a minimal/controlled example of how the approach could work. Long term many sources of data input should be used, since further input options create more permutations of relationships that could form. For this project the interpretation of the sensor data was done by decomposing it into categories. i.e. for a temperature sensor it didn't really matter to get the exact temperature. So as part of the classification of the data, it's separated into categories of what was cold, hot, warm, really cold and really hot. This is then normalized and fed as an input to the SOM when organizing around a new input vector. For the image data it was done slightly different, there were sets of values that were normalized based on the maximum value in the data set.

Use Cases

The following UseCases help define the top level functionality of the application and how the user would expect to interact with the system.

UseCase	User button press on GUI (Figure 6)
Description	User finds new object to analyze, to predict if we can find a match.
Preconditions	Sensor data is valid and camera is focused on new object.
Postconditions	New object is either stored in the SOM or a object was displayed that matched what was analyzed.

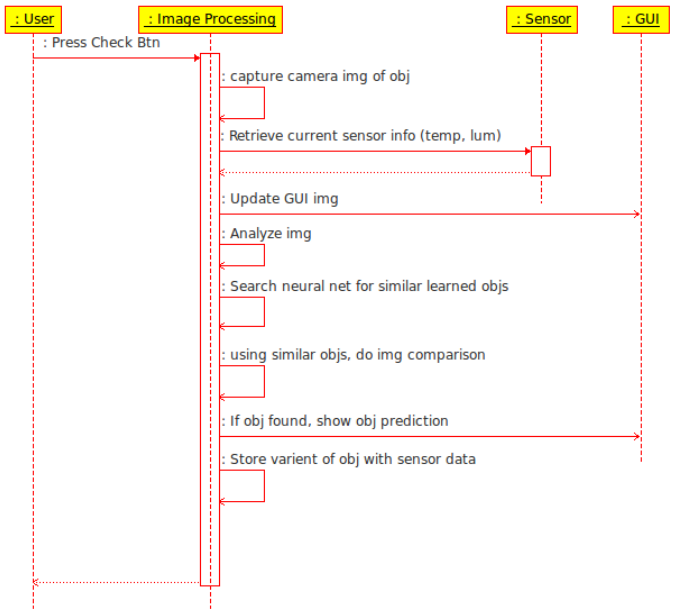

Fig. 6. UseCase for User button press on GUI.

UseCase	Sensor data provider (Figure 7)
Description	Micro-controller pushes data out via Ethernet for application use
Preconditions	Ethernet connectivity. Sensors initialized.
Postconditions	Repeat in endless loop pushing data.

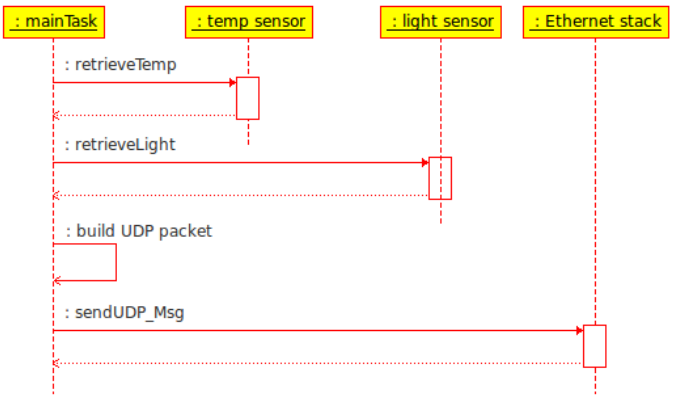

Fig. 7. UseCase for Sensor data provider.

UseCase	Image processing (Figure 8)
Description	Convert images into a hash or simplified value that can be compared.
Preconditions	Image is png formatted
Postconditions	Image has been converted into a mathematical representation.

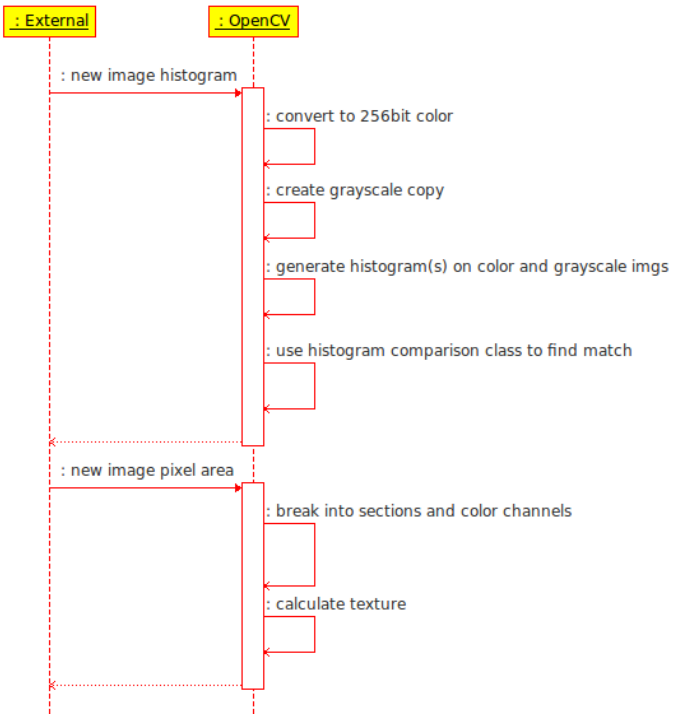

Fig. 8. UseCase for Image processing.

UseCase	SOM processing (Figure 9)
Description	Organizes data by normalized values unique to each node
Preconditions	Data is normalized for insertion to map. New node was added to map and needs to be placed.
Postconditions	Map is organized with correct relationships formed.

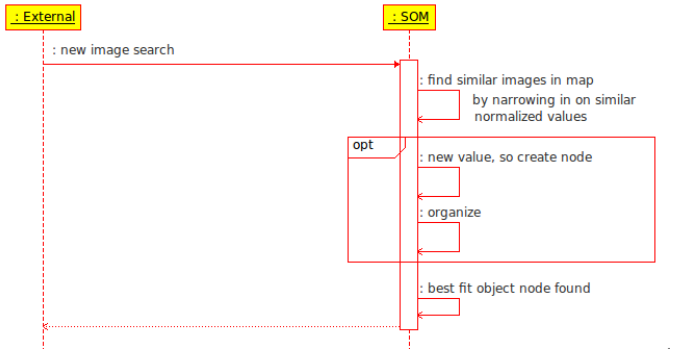

Fig. 9. UseCase for SOM processing.

Data Structures

The Arduino micro-controller communication packets were defined as a UDP packet of data size 10bytes to port 20001 with the format shown in Table I.

2 byte(s)	1 byte(s)	2 byte(s)	2 byte(s)	3 byte(s)
NA	Temperature	NA	Light level	NA

TABLE I
ARDUINO SENSOR DATA FORMAT.

Object data storage is based on the neural net configuration, and ends up as a vector of float values. In the early data processing, the object data is formatted as a comma separated string that contains normalized values for the following.

- 59 values for Image features (Histogram/Edge Point data) (Array of normalized floats)
- 1 value for Temperature (normalized float)
- 1 value for Light level (normalized float)

A nice feature of this SOM algorithm is how the input vectors are stored. It's easy to export them and reload at a later point and reuse the previous input data. In the future research section there is some discussion on the issue of how these input vector values are weighted. For example how the image values in this case completely out weigh the individual sensor values.

Libraries

The baseline software for doing the image analysis is the OpenCV port to the Android platform[15]. It's libraries seem to support all the operations needed to distilling down a image into comparable characteristics. It currently directly interfaces with the Android application allowing direct control of the camera feed into OpenCV for analysis.

The Arduino micro-controller software[3] is provided with the kit. A few examples for the sensor implementation were provided by the manufacture. It has built in sensor and UDP Ethernet libraries that can be directly reused to relay all the sensor information.

The Android application software is created using the freely available Android Software Development Kit(SDK) and Native Development Kit (NDK)[9]. The NDK is required to build the OpenCV framework for Android use. There ends up being a special version of the NDK required for the current version of OpenCV[7] because the current Google release does not support c++ concepts fully like Run-Time Type Information (RTTI), exceptions, and most of the Standard Template Library (STL).

2) Integration: Just like in an embedded software project, doing application development on a smart phone is going to require cross-compiling the code. Things may go a step further and also require language wrappers between the native code and run time code. It really pays as a first step to figure out the complete algorithm design using simple test applications that run natively on a development PC. And as a positive side effect the test applications turn into a nice suite of regression tests to verify data when the phone cant give the same visibility or the data is getting mangled somewhere in the language wrappers.

This project takes the development effort and breaks it into three distinct phases. The first, development of PC based test applications that model the algorithms. The second, porting the test applications into a real smart phone application. The third, using the test applications to verify the resulting sets of data generated by the algorithm(s) running on the smart phone.

Phase 1

To verify the concepts required to do the image processing and SOM learning, two PC based test applications were developed. The first was a C based application that integrated with the OpenCV library to perform a grey scale histogram

and produce a 32 bin representation of the object image. Also as part of that application, it calculates the average pixel area (equivalently the texture) of the image. This was represented by 27 values. The resulting values for both were normalized into a string of floats that could be used as input to the SOM. The basic process within this application is as follows.

- Images are gathered from either from the web or taken with a camera and added into the processing folder.
- The application is invoked and indexes the folder.
- Records are created for each of the images for later bookkeeping.
- The application opens the image and converts the 3 (RGB) color streams to a single greyscale.
- The histogram is calculated for a range of {0-255} over 32 bins.
- The histogram is searched for min and max values.
- The histogram is normalize relative to the max values.
- The resulting 32 normalized histogram values are written with the images filename to a CSV formatted file.
- The image is reopened for the pixel area calculation.
- The image is broken into the 9 sections
- Each section is broken into 3 (RGB) channels.
- Each channel is averaged for the pixels within that section.
- Each sections values are normalized between 1 & -1.
- The resulting 27 normalized histogram values are appended to the CSV formatted file.

The second is a Java application that first implemented a basic algorithm to show an example of how it could organize a set of color inputs vectors[1]. This was modified to take in a set of picture inputs using the input vectors produced by the C application above. In addition the application was retrofitted to produce a visualization of the organization and show the resulting set after organizing. This provided a good simulation environment for tweaking of the amount of learning iterations required and allowed easy adjusting of SOM matrix size and number of test images. The one limit with this application is that it is a manual process that interprets a preset data set of values produced by the C based image processing app. To add a new picture, it had to be ran through the image processing app and then the input vector added to the list to be loaded in the java application for analysis.

To verify the algorithms were working correctly, it needed a dissimilar data set. The resulting image set contained a combination of random art, football, and outdoor pictures. Having a visualization of the data lead so a few conclusions. The art pictures provided some vivid color checks on what color and image details things organized around. Each of the football images contained a set of colors that were similar and in each image were slightly moved around. For example a bunch of images of players from the same team (same jersey) are very similar in the grand scheme of all the images. Lastly the outdoors pictures were all over the place for texture and usually had the upper half as blue sky, but they still organized differently relative to other texture/features in the image. Each picture was processed in a way to work around known issues with specific hashing or other methods for capturing image uniqueness. For example, if only a histogram was performed, the visualization would show different light levels affecting images to be treated as containing different colors. This leads to images being disorganized because of incorrect color similarity. Adding pixel area texture helped to fix the light level issues and optimize the results to group better. This application helps to visualize the effectiveness of this implementation's algorithms that focus on capturing the busyness and colors of each image. The basic internal process of this application is as follows.

- Sets number of iterations and dimensions of SOM lattice
- Loads in the set of input vectors from the CSV file output by the image processing application.
- Begins the training process that iterates through learning cycles
- Each learning cycle takes each of the input vectors and calculates the Best Matching Unit (BMU), organizing around that best fit node within the SOM.
- Application redraws the screen and gives a visual representation of the movement of the images as they organize.
- Once learning is complete the resulting SOM can be visually viewed or exported to CSV file for later analysis.

Here's an example (Figure 10), of a high-level view of a SOM that has been loaded with 300 images input vectors and ran through one iteration of learning. It can be seen that the images have organized a little bit, but for the most part are fairly spread out and mixed up. However in Figure 11, which has iterated through 100 learning cycles, there is a definite trend of organization. The images with a lot of green are mostly football players and in the 100 learning cycle image there is even a separation between players of different teams (red and white jersey vs those with yellow and green). These resulting

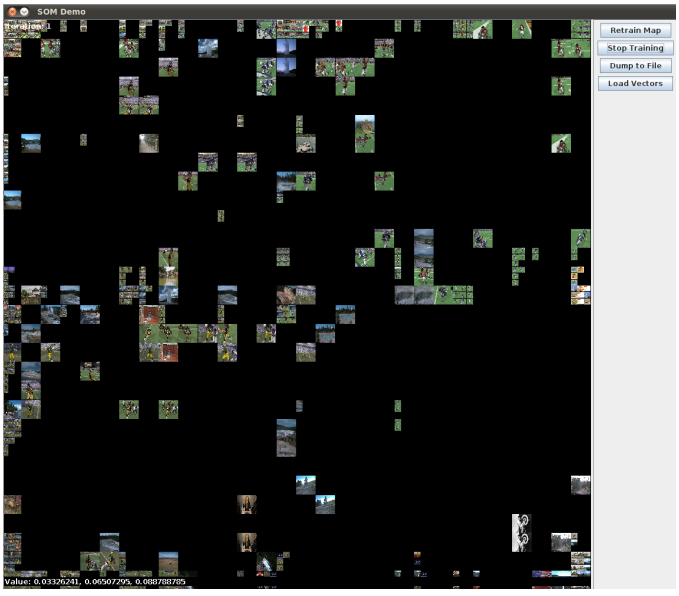

Fig. 10. SOM java application before learning.

findings lead to some confidence to continue to the next phase and port the software over to the Android platform. It's also been a invaluable tool in verifying concepts with more debug visibility then the phone.

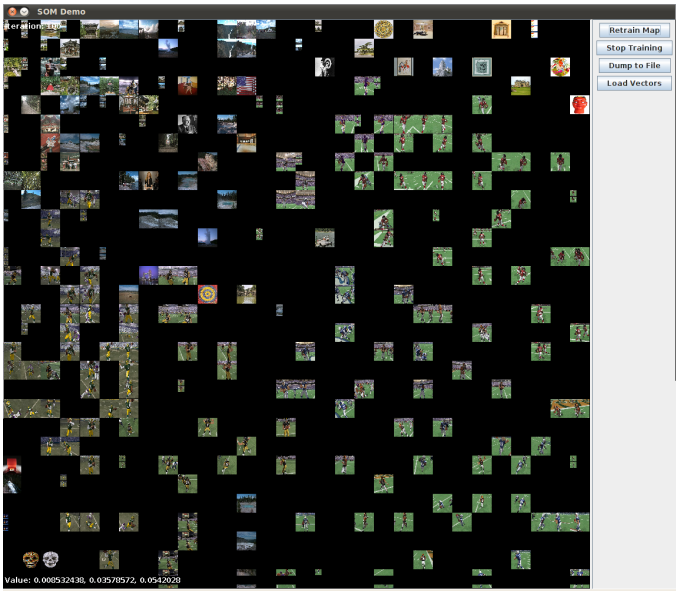

Fig. 11. SOM java application after 100 iterations of learning.

The last part of Phase 1 was configuring the Arduino platform. It was integrated with a basic set of sensors and an adaptable UDP socket push protocol. The one huge benefit of the Arduino IDE and it's built in development libraries, is their ease of use. Specifically for the temperature sensor, it was developed using the Dallas Temperature Control Library which made using a 1-Wire interface[10] transparent. For the light sensor, the design used an analog interface [2] instead of digital like the temperature sensor. This resulted in a simple interface and the data was gather by reading an ADC value from a basic analog photo resistor circuit. Each the results from these sensors was relayed periodically over Ethernet/Wifi to the Android phone. The Android application implemented a UDP server and listened for updates to refresh it's local status for the current temperature and light level at the location of the sensors. This is where the embodiment became tricky. As shown in Figure 12, the sensors are not attached to anything but a breadboard (simulated arm) that could be manually relocated to the object being investigated.

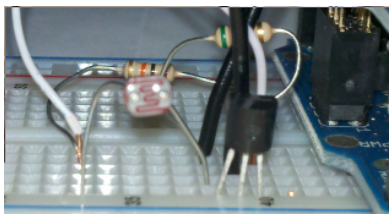

Fig. 12. Light and Temperature sensors attached to the Arduino platform.

The hardware configuration as shown in Figure 13 provides a USB connection for power, Ethernet for communicating with the Android phone over Wifi, and connectivity to the sensors breadboarded on the left side of the figure. This micro controller configuration was verified using Wireshark and associated serial test output. After verifying the data stream was valid and the all boundary signed/unsigned data issues

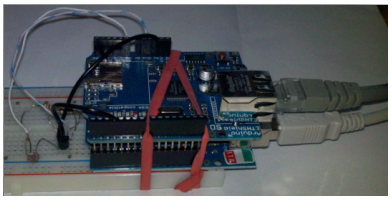

Fig. 13. Ardunio sensor and connectivity configuration.

were resolved this component was completed and ready for integration with the Android application.

Phase 2

The Android user interface is based off of the OpenCV CameraCV project. As shown in Figure 14 & 15, it implements a constant camera preview mode on the left that's coupled with sensor data overlaid on a bar at the bottom of the screen. The sensor data does some conditional color changes, like for temperature it switches between red and blue for hot and cold. All depending on whatever value the sensor has just reported. Located on the top is a camera button used capture images for processing. After an image is captured, it appears just to the right of the "Focus" button. On the upper right side of the screen there's a space to show the neighbors of the image that was just taken. These neighbors are the images that are the "most" similar to the new object just learned. To get those neighbor results, the application does a process just like in the test application where it calculates the input vector data from the new object, adds that to the input vector set that the SOM is processing and starts the learning process. Since the display is relatively small and the processing power is limited, the same visual output wasn't implemented for the phone, but it can be monitored via debug output over the Android Debug Bus (ADB). Once the SOM has completed learning, a node is identified that is linked to the new input vector. Using that node's position all images local to that node or surrounding it are presented as neighbor images that should have a best fit and should be closely similar to the new image. Another part of the UI is the menu

Fig. 14. Basic Android Application User Interface(UI) with sensor registering as cold and dark.

used to transition the different application states. As shown in Figure 16 the application has the following states:

- Init - Initializes/resets the SOM lattice and allocates objects for the training process.
- Learn - Opens an existing dump file (if exists) from SD Card and loads in the previously learned input vectors.

Fig. 15. Basic Android Application User Interface(UI) with sensor registering as hot and bright.

- Capture - Enables the camera to take pictures, the algorithm to do the image processing and the server to collect sensor data.
- Run/Train - Forces a manual rerun of the SOM learning/training algorithm. This is the same process that automatically follows a capture of a new camera image.
- Dump - Dumps a file to the SD Card that contains all the input vectors for either post processing or to be loaded during the learning state when the application is restarted. Also on the SD Card are all the images that had been captured.

The expected sequence of execution would be running init, learn, capture, and lastly dump before closing the application. After enabling capture, the sensor status bar will light up and the camera button will activate. Any images that are captured are automatically ran against the SOM and the neighbor results shown to the right. The process of capturing images and analyzing them isn't limited, so many images can be captured and analyzed. If the data needs to be saved though, the dump menu option should be selected before exiting. The data that's dumped, contains a textual matrix of the image names organized with how the SOM last arranged them and a second file that has the raw export of the input vectors that could be fed back into the Android application or into the PC SOM application for simulation and analysis of any behaviors that maybe didn't seem correct.

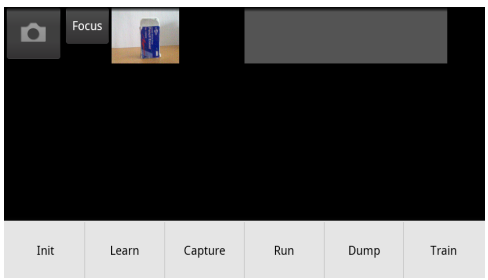

Fig. 16. Basic UI with menu for select mode of operation.

To enable the Android application to have the capabilities it needed, there where a couple modules that had to be integrated into the application. The first was the C implementation of the OpenCV image processing code from Phase 1. That code had to be ported into the Java native library that contained modules allowing active processing of the camera preview stream. The new code basically ended up like a decoder for the image stream. With the algorithm capturing the camera button push, grabbing a single frame from the preview stream, and doing the histogram/pixel algorithms. On completion it would pass the resulting input vector data for that image up to the Java layer of the application. The second part was porting the SOM code over to the Android Java layer. It required some adapting because of classes that leveraged inheritance to pieces of the Java UI framework. Once the SOM code was cleaned up, a middleware was created to bridge the native C algorithms and the SOM. It took the values generated from the image processing and added them to the input vector set; in addition to providing utilities to export and import input vector sets for dumping and loading respectively. Next the control points for this code were hooked into the menu options to activate the specific processing and learning processes.

Phase 3

To verify the new Android application, the software was again split into two halves. The image processing and the SOM. The verification of the image processing consisted of taking a known image and running it through the PC test application to generate the input vector. Followed by rerunning the same image manually through the Android image processing code. The resulting image input vectors were compared and it was found that small rounding errors in the 3&4 decimal places had occurred. That seemed acceptable being the code was ported between architectures (PC to ARM). Next a set of example image input vectors were loaded into both the PC test application and the Android application. Both were configured to use the same lattice matrix size and number of learning iteration. The resulting SOMs were compared by dumping the Android applications to a file and manually comparing to the PC test application's visualization of the organization. It was found that both applications organized the images differently, which is what was expected, but they correctly grouped the images. So for the most part this verifies the algorithms were correctly ported over. Although one issue did come up related to processing resource issues with how Java does floating point (mentioned in the future research section). The processing time for the SOMs learning was reasonable as long as the data sets were small. However over time the duration would grow when the number of input vectors increase. So far this issue has only come up while attempting to use the debugger, which on it's own is slow, but amplifies this issue.

V. Experiments

The original statement for the experiment was as follows. "A setup consisting of a set of images to use in the test cases. The set of images would contain a variety of black and white and also color images. Each set would be independently tested to see the impacts of color on determining if a object is going to be cold/hot. There were example images shown that would be used to initially train the SOM. Once the algorithms were worked out, some test would be performed on new pictures that would be taken with a camera. Those

pictures would include sensor data as an extra trait to be analyzed. Success would be to have these samples grouped by common specific shapes and colors.“ The resulting outcome, from studying how the learning algorithms work, has lead to the experiment being more focused on reproducing the image learning algorithms done by other papers. It uses an image set to initial verify the learning algorithm. Followed by manually taking some pictures and documenting the learning process is within bounds for how those pictures are related to other images. The sensor data is included as extra traits for each of the objects, but here is a weighting problem that affects how that extra data is being used in the decision process. (Many input vector values for image representation, but only two for the sensors)

The first part of the experiment tests how adjustments of the SOM parameters affect it's performance when learning. The parameters being adjusted are the number of learning iterations ranging from 10 to 100 and the number of input vectors 10 to 300 (directly affects lattice size). This test will utilize a test app on a PC because of the processing resources required to produce the result sets.

The second part of the experiment tests how the Android phone application behaves with taking pictures and learning how they fit into things it's previously learned. A couple different scenarios will be tested, ranging from having no history of objects, a history of similar objects, a history of dissimilar objects, and a history of a mix of objects. Each test case would take place in a controlled environment where most of the image would be white and the details of the object being learned would be obvious from it's basic shape, size, color, and texture. The objects would be identified, have their pictures taken, and temperature/light eminence measured. Post analysis would look at specific points of the SOM organization to make sure the learning process is organizing correctly.

The evaluation of the results will look at the sets of inputs and output data to verify that a reasonable match has been made that fits the expected results (Similar in color and sensor hot/cold properties). If the result isn't what is expected, the post analysis of the SOM data and whatever the results did end up as, should suggest future research or ideas for the deviation.

VI. Results

The test results breakdown into two sections, the first studying the outputs of the PC based SOM test application and the second looking at some actual test data using example objects and recording the resulting SOM neighbors.

Test Application

The SOM test application was setup with each of the configurations show in Table II. Each of the test configurations was executed by randomly assigning input vectors to nodes and organizing around them. When complete, an image was captured of the results. After all tests completed the resulting distribution of images was analyzed for level of organization. Using the best and worst test images as the bounds for calculating the percent organized.

The first test (Figure 17) is configured with a lattice that's matching the number of input vectors. It however only gets 5 learning iterations. During the learning process there are radiuses defined, so that only a small area is organized around a node at a time. This causes some limits on the amount of area an input vector (object/image) could move in a single learning iteration. For example, if many similar objects started out in opposite corners, (depending on the lattice size) they'd need multiple iterations to migrate together. This is shown in the figure, as the outdoor scenes are still scattered around the SOM and need more iterations to come together. A smaller learning count did reduce the objects ability to completely move towards similar objects.

Iterations	Input Vectors	Lattice Size	Resulting % Organization
5	100	10x10	50
10	100	10x10	60
50	100	10x10	80
100	100	10x10	100
5	100	20x20	20
100	100	20x20	80

TABLE II
Level of approximate organization based on SOM inputs.

Fig. 17. 5 learning iterations, Input Vector of 100, and 10x10 lattice.

Fig. 18. 10 learning iterations, Input Vector of 100, and 10x10 lattice.

Between Figure 18 and Figure 19, there is some improvement shown with the objects organizing into the corners, but they still don't come together in a single grouping. Figure 20 shows the organizing case that's as close to 100 percent as can be achieve with it's configuration. There is a definite separate between the colors and many of the outdoors pictures are

Fig. 19. 50 learning iterations, Input Vector of 100, and 10x10 lattice.

Fig. 20. 100 learning iterations, Input Vector of 100, and 10x10 lattice.

stacked up with multiple to a node. This is the configuration that the Android application has been setup to use.

The next two test cases enlarge the lattice size to cause more empty nodes. This should negatively impact the learning convergence rate because of the learning radius used to find similar objects. Figure 21 is showing a pretty random organization of images, so learning hasn't really had a chance to do any reorganization. Figure 22 however does have some better color separation, but exhibits the issues found in Figure 19 where there are groupings of similar objects in opposite corners. Additional learning or a smaller lattice should bring the clusters of similar objects together.

Fig. 21. 5 learning iterations, Input Vector of 100, and 20x20 lattice.

Based on testing results with lattice sizing, one key aspect sticks out. The lattice and the input vectors should roughly

Fig. 22. 100 learning iterations, Input Vector of 100, and 20x20 lattice.

match in size. With no more then 5 percent more lattice nodes then input vectors. Otherwise the required number of learning iterations has to increase to give the input vectors more attempts to organize with similar items. This seeks to prevent one aspect of the localization(random separated grouping) issue, by preventing excess unconnected nodes. It doesn't however solve the issue of localization when the lattice grows much larger in size. Sometimes in that case, increasing the number of learning iterations or resizing the lattice doesn't necessarily solve the problem. There were some interesting papers on multilevel SOMs and how it might approach fixing that issue[14].

Android Application

Provided in the Appendix are each of the objects used in in the trial runs for this test. With each test result below it notes the previous learning conditions, since in some cases it was the first object learned and others a few valid neighbor(s) already existed. A neighbor(s) are the nodes in the lattice that surround and include the node that contains the input vector understudy (aka the new image/object). The result images shown in the figures below, consist of 4 images, the first is the new image that was just taken. The second/third/fourth are the neighbor images that the SOM found as surrounding the first image after learning. If the first image was completely new (with no previous similar images), neighbors would still be suggested, but may not be of any match. If none of the objects are even close to the color or texture of the first image, then there's a change that the image is orphaned to a part of the SOM that doesn't yet have any other images. Like in Figure 23, it is a new object and the SOM, during the learning process doesn't find any similar images around it. This is ok, because the new object has now had it's properties learned and next time it will be recognized. However, there still is a slight chance that you could also have a similar problem when there are some neighbors, but not enough similar items have been learned so they don't really match. This occurs when having a small data set of similar objects. Figure 24 is an example of this issue, with the new object being a battery and the SOM locates it next to another battery, but also an eraser. As more objects are learned this problem occurs less because there is a larger selection to be organized around, resulting in a better fit for new images.

Fig. 23. Learning a new object: A whistle.

Fig. 24. Learning a new object: A battery.

Here's an example of a progression. It's assumed that at least one similar image was learned before processing the first image in Figure 25. Figure 25 shows a new image captured of an eraser with a slightly different orientation then the one the SOM retrieved as it's new neighbor.

Fig. 25. Android App presenting one option for the eraser.

Figure 26 shows another new image captured of a slightly larger eraser. The SOM manages to find two neighbors for this image, one the previous image and the other the original eraser. If the same new image was captured again, it would more then likely be added as the third image in the resulting neighbors.

Fig. 26. Android App building on the last run and presenting two options for the eraser.

Figure 27 shows another new image captured of a slightly larger eraser at a different angle. The SOM manages to find three neighbors for this image. Each one being one of the previous learned similar images.

Fig. 27. Android App building on the last two runs for more eraser neighbors.

This test case has shown the grouping that occurs when objects are learned by the SOM and organized so the neighbors can be selected when a similar new image is captured.

This next test case looks at the ability for the algorithm to pick out similar objects and shapes. It also looks at the possible issue with a slight color blindness that might occur depending on the percent of the object that changes color. Figure 28 shows a picture taken of a grey and blue capped pen. The new image was captured as the 28th image to be stored in the SOM (See appendix for table of images). Prior to it there were a handful of images that should be similar. The SOM picked out two and they happened to be image 21

Fig. 28. Finding a similar pen.

& 26. These turned out to be a solid set of neighbors. The image processing also was able to mask the issue of light level and looks to have successfully organized around more of the texture or object content of the image. However if the pen was to drastically move toward a corner there might be a chance it would not organize as well, since the position may come into more play if the light level has a larger difference between images. (Histogram is doing color matching & Pixel texture provides segmented texture matching)

Fig. 29. Finding a sort of similar pen.

The color blindness comes into play with objects that are very similar(few color differences) and have the same backgrounds. Figure 29 is a good example. One resolution to this issue is to learn more objects that clarify the differences. The other resolution would be to enhance the image analysis to generate additional details that could be compared to eliminate the issue. Depending on the situation this might not be a bad issue. For example when you're trying to find a medicine bottle or possibly the floss (Figure 30), the only requirement could be to find something similar in nature, not necessarily color.

Fig. 30. Finding the floss.

A test was performed to understand how the sensor weighting values affect the decision made during the SOM learning. The test consisted of an inspection of the algorithm utilizing the data and the assumptions surrounding how the data could come into play. The algorithm that determines placement of objects within the SOM looks at the raw input vector data set and performs a Euclidean distance calculation between that set and every node in the lattice (trying to find the optimal placement). So the two sensor values end up in a summation of the differences between all 61 input vector data points between two input vectors. This results in the images determining the positioning in the SOM unless the image is more neutral between the two input vectors. If that happens then the senor values data points would come into play, but primarily the image would steer the organization. This might actually be OK depending on the assumptions. Specifically if it's assumed that the image carries the weight in the input vector values. Making the visual image of the object more important to the decision process when later recalling the object. This wouldn't rule out the sensors, but a combination of sensory data should be represented in the input vector since it's a characteristic

of the object. Possibly though a multi-dimension input vector feeding multiple SOMs there could be a better solution, so as to organize around each characteristic/sensory input and tie the results together with the individual object. So for this test, it was concluded that the current algorithm works, it just doesn't favor the sensor input, although still incorporating it when learning.

A test was attempted with embodiment, but issues arose with the sensor breadboard being part of the image (Figure 31. It weighed in to heavily in the learning process and caused skew in the resulting data. The solution for this would be to incorporate the intelligence for the robot to remove the parts of it's body present in the images before the images are processed. This is an area for future research and improvement.

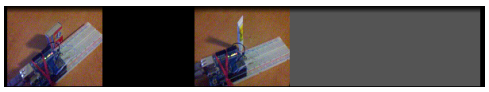

Fig. 31. Attempting to place sensors by object.

Here's the last test case. It analyzes the organization of the SOM based on provided new inputs of captured images. In Table III there's an example text representation of the SOM used to generate the results in Figure 32. The table shows the lattice nodes of the SOM and represents each node as an entry in the table. Inside each node is the image number(s) of the particular input vectors that are mapped to the node. A node can have multiple images stacked up (making the matrix 3D). In this table, only the last image number (input vector) to be added to the node is shown. The new image for Figure 32 is image number 62. The images displayed as neighbors are 21 and 28. It just happens that those two neighbor images were located in the same node as 62 so they're not visible in the table. Since the GUI only draws out the first three neighbors (This number was arbitrary picked) there might be other close matches. So looking around number 62, there seems another close fit with number 27. Since 27 didn't make the cut for being added to the node that 62 is part of, there might be some slight differences in the image. There's a chance it could have been displayed if the local node (for 62) didn't contain any input vectors that would be closer matches to 62.

62	27	30		13	05			01
61				11				
44			14					4
41	35		39	2	6	20		
40	45	57	42					
3			58	17				
60	59				16	15	10	18
		55		50				
53		56		48				
54			32	38	34		49	

TABLE III
EXAMPLE TEXT DUMP OF SOM. (SEE APPENDIX A FOR MATCHING SOM IMAGES)

Fig. 32. Learning of a new image, prior to dump of SOM into the table above.

The earlier eraser example is centered around number 3/35/40/41/57 which all correlate to various images of erasers. It'd be interesting to redo this test with a hexagonal or other dimension node layout in the lattice. It was mentioned in the existing research that it may lead to better relationships through the use of more edges connecting to other nodes.

To conclude the test results, a few tests ran into issues that will require further study to understand how they could be resolved. Overall though, the application has successfully been able to organize and suggest predictions based on weighted organization of a new object with previously learned objects. Also if objects are cold/hot and are always one or the other, then this set of algorithms will find the object based on it's image and sensor weighting. However if the object is both recorded as being hot and cold, no real resolution can be made unless the object is touched to find out. The SOM would more likely then not give suggest both objects as results.

VII. FUTURE RESEARCH / LESSONS LEARNED

A couple areas of SOM neural networks that were not issues for this small scale experiment were the limitations with scalability and storage capabilities. Specifically with this implementation of a SOM it's designed as a 2D matrix of nodes that are interconnected, each with a connection to each input node. So as more input vectors are added to the SOM the input vector permutations that have to be checked grows. Eventually the matrix could get sized to large creating a scaling issue where either there is not enough processing resources or the time involved in doing that processing isn't acceptable.

To further mature this application, algorithms could be added to aid in creating knowledge of embodiment and handling of complex images to extracting relevant objects for recognition. It would be an interesting application of parallel computing using a GPU. If the SOM was optimized to use a GPU engine for all learning activities the size of the input vectors could grow larger allowing for more details to be gathered during image or other sensor analysis. That combined with adding the robot's ability to control the motion of arm movement would be an interesting addition to studying how it effectively processes and stores the learned data. Since by adding the ability for a robot to interact with the object being learned, it would effectively simplify the existing manual process that was pretty much scripted and required human interaction. This isn't assuming that it would simplify the algorithms required, since new processing would have to provide the ability for the robot to first learn that it has an arm that it can use to interact with objects.

A possible improvement to the SOM localization learning issue could be to implement the bias discussed in this paper.

It was an area that had only been briefly looked at during this project[20]. A single SOM input vector is the set of data points that represent the characteristics of an object. The one issue with the SOM input vector data is how each vector element is incorporated into the calculations. For example in this project the image characteristics consisted of around 59 data points that were normalized and added to the vector. Then additional temperature and light sensor data was added. So you have a overwhelming bias (59 verse 2 data points) towards the imagery data steering the direction of where that input vector is organized into the SOM lattice. To be more fair, each sensor should possibly have an equal number of data points in the vector to make sure a single sensor has some weight steering towards the correct node(s).

One opportunity for future research could be processing optimizations. Specifically to optimize the SOM math calculations through using Android native code. Or it could be possible to use OpenCL APIs or another graphics engine APIs to get some matrix math speed ups. Another optimization area is streamlining the memory allocations to prevent excessive garbage collection. Either of these ideas could definitely improve the performance of the learning activities that require larger amounts of processing as each new input vector is added.

One area of improvement and not necessarily research would be to streamline the graphical user interface for easier use. It could possibly be in conjunction with creating a less interactive version of this concept with the robot actively engaged in finding objects. Leaving the display as more of a feedback interface.

An item to add to the list of lessons learned would be adequate debug visibility between all stages of the application development, including on the Android phone. Out of necessity during this integration a UDP based stdio was implemented with a client/server concept. Allowing clients written both in C and Java to put out debug within the Android Application Java and native C layer. This aided to get some time tagged breadcrumbs out of the system to tell how long processing was taking and the different stages the processing was at. It definitely didn't replace the GNU Debugger (GDB) interface provided by the Android SDK, but instead supplemented it's functionality.

VIII. Conclusion

The original goal of this project was to model a child's learning process to associate images and sensory feedback to previous experience. At completion, this project has created an application that has the ability to find similar objects based on new objects presented. Within some limitations this project has met the original goal. The areas where there are some complications were the processing resources needed for doing the neural network on a Android phone, bias adjustments to give the sensor data more play in the learning process, and the changes needed to the original set of test objects to better define success.

Secondarily there was another goal of create a open source framework that would allow further experiments using self-organizing maps (SOMs) within Android. (Possibly with computer vision applications for other self learning through sensory input.) i.e. Using this project as a basis, many other applications came to mind. From simple things like selecting the perfect bottle of wine by pointing your phone at the store shelf and letting it pick-out the correct bottle based on previously learned characteristics. That example could be taken even more extreme with it fine tuning the bottle selected based on food pairing and atmosphere/event. This goal was successfully met, the source code has been made available on http://code.google.com/p/mlw-proj/source/browse/. (Parts of the code have different licensing so that's still being sorted out.)

A positive outcome from this effort was the creation of a framework that resulted in proving (at least to me) that some sort of embodiment is a definite requirement for artificial intelligence. Without that it's extremely difficult to manually anticipate different scenarios where the robot might be interacting with something and really needs to understand that what it's observing is partially its own arm/leg/etc and an object that the appendage is interacting with.

References

[1] AI Junkie, *Kohonen's Self Organizing Feature Maps*. http://www.ai-junkie.com/ann/som/som1.html, 2005.

[2] Arduino, *Arduino Photo Resistor Tutorial*. http://www.arduino.cc/playground/Learning/PhotoResistor.

[3] Arduino, *Arduino IDE*. http://arduino.cc/playground/Linux/Debian.

[4] Barry Ridge, Danijel Skocaj, Ales Leonardis, *A System for Learning Basic Object Affordances using a Self-Organizing Map*. http://www.cognitivesystems.org/publications/ridgeCogSys08.pdf, 2008.

[5] Casey Chesnut, *Self Organizing Map AI for Pictures*. http://www.generation5.org/content/2004/aisompic.asp, 2004.

[6] Cell bot, *Cellbot: A Androd based Robot*. http://mashable.com/2010/03/06/cellbot/.

[7] Dmitry Moskalchuk, *CrystaX: Android NDK r4*. http://www.crystax.net/android/ndk-r4.php.

[8] Douglas Blank, Deepak Kumar, Lisa Meeden, James B. Marshall, *Bringing up robot: Fundamental mechanisms for creating a self-motivated, self-organizing architecture*. http://science.slc.edu/ jmarshall/papers/bringing-up-robot-preprint.pdf.

[9] Google, *Android SDK*. http://developer.android.com/.

[10] Hacktronics, *Arduino 1-Wire Tutorial*. http://www.hacktronics.com/Tutorials/arduino-1-wire-tutorial.html.

[11] Jefferson Provost, Benjamin J. Kuipers, Risto Miikkulainen, *Developing navigation behavior through self-organizing distinctive state abstraction*. Artificial Intelligence Lab. The University of Texas at Austin.

[12] Kintar, *SOM Example Java demo*. http://www.ai-junkie.com/files/SOMDemo_java.zip.

[13] M. Johnsson, C. Balkenius, *Sense of Touch in Robots With Self-Organizing Maps*. IEEE Transactions on Robotics, Volume: PP Issue:99, p1-10, March 2011.

[14] Norfadzila Mohd. Yusof, *Multilevel learning in Kohonen SOM network for classification problems*. Masters thesis, Universiti Teknologi Malaysia, Faculty of Computer Science and Information System. June 2006.

[15] OpenCV, *OpenCV library for Android*. http://opencv.willowgarage.com/wiki/Android.

[16] T. Kohonen, Erkki Oja, Olli Simula, Ari Visa, Jari Kangas *Engineering Applications of the Self-Organizing Map*. Proceedings of the IEEE, Vol. 84, No. 10, October 1996.

[17] Shotaddict, *Shotaddict:Reading A Histogram Correctly*. http://www.shotaddict.com/tips/article_Reading+A+Histogram+Correctly.html.

[18] T. Kohonen, *Self-organizing maps*. Springer Series In Information Sciences; Vol. 30, p. 426, 1997.

[19] Taylor Bergquist, Ugonna Ohiri, Conner Schenck, *Interactive Multimodal Recognition Of Household Objects by a Robot*. Developmental Robotics Laboratory, Iowa State University. 2009.

[20] Vinod K. Valsalam, James A. Bednar, *Establishing an Appropriate Learning Bias Through Development*. Proceedings of the Fifth International Conference on Development and Learning, 2006.

APPENDIX A
ANDROID APP TEST OBJECTS

These images are referenced in the test results section above. For most of the Android Application examples where a new image is captured, it uses this set of images as a base set to generate the SOM learning results.

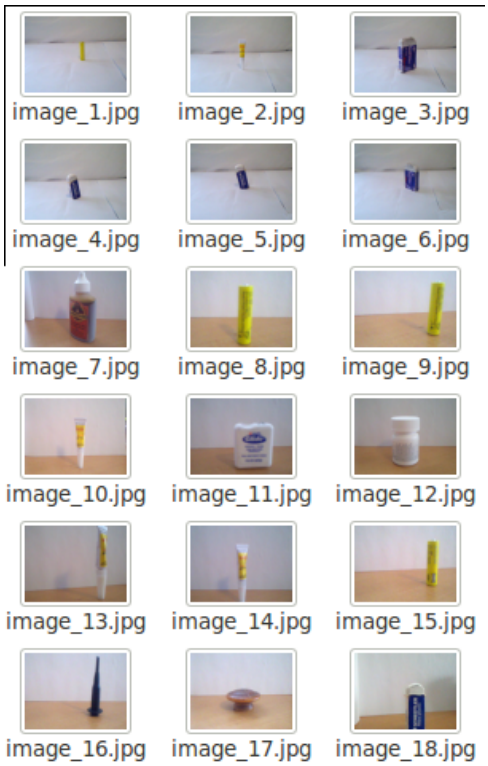

Fig. 33. Android App test objects (1of4).

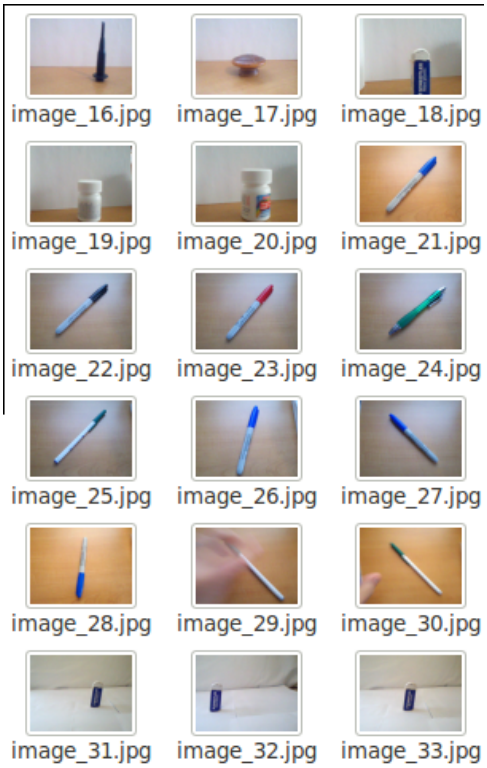

Fig. 34. Android App test objects (2of4).

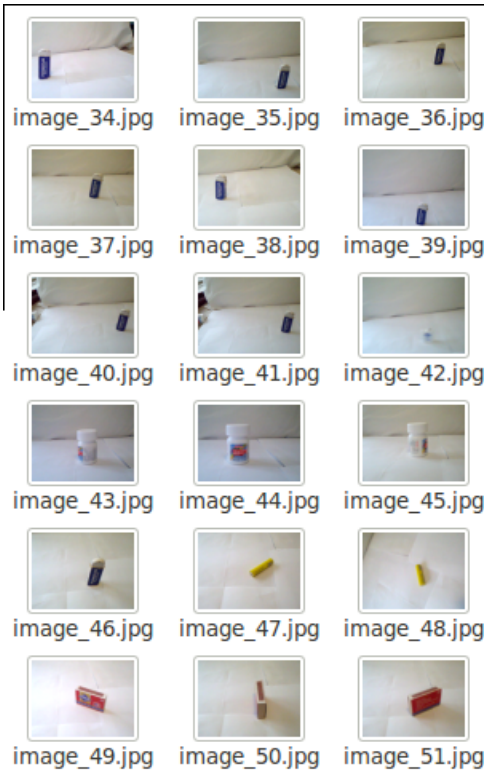

Fig. 35. Android App test objects (3of4).

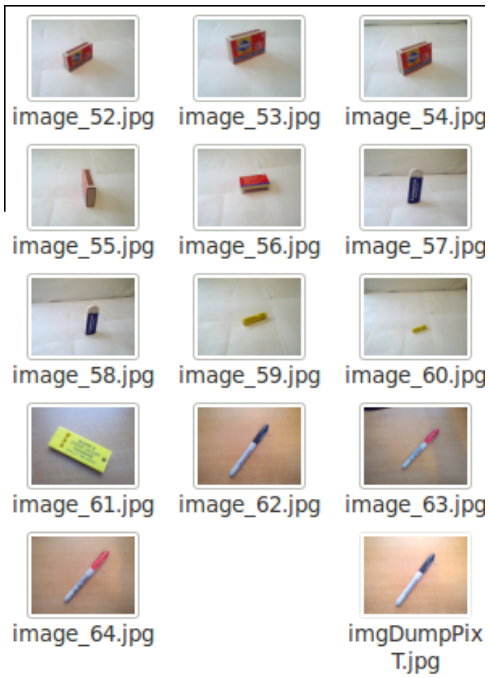

Fig. 36. Android App test objects (4of4).