A Machine for Automated Lego Identification Using the YOLOv5 Network

Team Members

Chung, Derek
University of Minnesota
Undergrad, Comp.Eng.
chung595@umn.edu

Essner, Nicole
University of Minnesota
Undergrad, Elec.Eng.
essne003@umn.edu

Flickinger, Ian
University of Minnesota
Undergrad, Comp.Eng.
flick077@umn.edu

Martin, Jacob
University of Minnesota
Undergrad, Comp.Eng.
mart4322@umn.edu

Wang, Rick
University of Minnesota
Undergrad, Comp.Eng.
wang7304@umn.edu

Xie, Ruide
University of Minnesota
Undergrad, Comp.Eng.
xie00056@umn.edu

Team Advisor

Bazargan, Kia
University of Minnesota
Associate Professor,
Electrical/Computer Engineering
kia@umn.edu

05/05/2022

Abstract

How is a smart Lego identifier machine built up? The project requires efficient algorithms of data processing as well as insightful construction of hardware. In this paper, we explore the use of machine learning in Lego pieces detection by applying a CNN based model algorithm. We argue that after training a load of data, the accuracy of the correct Lego piece identification will be higher than 70%. We empirically evaluate the efficiency of the proposed model against other related object detection models on a huge load of image datasets we will generate. Furthermore, we also built our physical machine with calculated dimensions of 3D printed hardware pieces. Our final demonstration of the machine shows the success of hardware implementations, the high accuracy of Lego identification, and the combination of the two.

Introduction

The team worked with Professor Bazargan to build a machine that can automatically find a piece of Lego out of a pile of Legos using machine learning. This final report on the project will first go over the basic concept of artificial neural networks, and how the object detection neural network has evolved in the last few years to the point whereby far one of the most advanced neural networks, the YOLO model was born. Then, the report goes into detail on how the YOLO model is structured under the surface, while also mentioning how the team incorporated YOLO's unique structure to create synthetic pictures of Lego for training purposes.

The next section of the report shows the complete design process of the physical part of the machine including how the flow rate of the Legos is controlled, how Lego pieces got separated from each other, and how the Lego pieces are presented to the camera in a controlled manner. After this section, the report talks about the remote operation of the machine where the team used a powerful remote computer for image inference, and how the Raspberry Pi controller is connected to the remote computer. In the user experience section, the report describes the website the team built for interacting with the Raspberry Pi and the remote machine. Lastly, the report shows the result of five months of development with details of the performance of the machine and the custom-trained YOLO v5 model.

Background

It can be very tedious to find a specific Lego piece in a large pile of Legos. This can lead to lots of wasted time building with Legos and cause seemingly endless frustration when a certain piece needs to be located. This human shortcoming of labor-intensive sorting and searching could be solved with the Lego identification machine. This machine would be supplied with a pile of Legos and then determine if a select Lego piece was present in a pile or if the machine needed a new pile to search through.

To make the machine recognize a Lego piece automatically, machine learning and computer vision are used in combination with a custom-designed Lego piece dispenser to recognize the desired Lego piece based on the user input. The general approach used under the hood is the neural network in Artificial Intelligence, which simulates how humans' brain functions when people look at things. How successfully the computer learns the shapes depends on how many convolutions and layers are under the process. It has been proven useful globally, as many of today's applications rely on it for object recognition. The Lego identification machine used an existing object detection model called YOLO v5 to get a head start, and upon that, the team edited and put in extra effort to fill the project's specific needs.

The machine is able to pick out a user-selected Lego piece with a minimum of 70% accuracy using a Convolutional Neural Network (CNN). The network will be trained to differentiate between 100 different types of Lego pieces and will be capable of identifying at least two pieces per second (equivalent to 120 pieces per minute).

The front end of the device will run on a Raspberry Pi platform. This will handle the operation of the mechanical components such as the conveyor belt, camera, and separator paddle. Additionally, it will present the web interface to the user that allows part selection and notifies when a piece has been identified. The compactness of the Raspberry Pi allows the machine to remain relatively simple and compact, but a Raspberry Pi alone will not be computationally powerful enough to handle the neural network's inference process. Thus, the Raspberry Pi will be remotely connected to a more powerful machine that will handle the inference of the Lego pieces using the custom-trained YOLO v5 model. The product will be built to cost less than $300 and be powered from a standard outlet.

Artificial Neural Networks

An Artificial Neural Network (ANN) is a computer science system which is intended to mimic the functionality of the human brain. The idea behind the process (illustrated in Figure 1) is to model a complex function which maps data from an obscure format to more informative structures. For example, this project sought to transform the obscure red, green, and blue values of each individual pixel in an image to the locations and part numbers of the Lego bricks in the camera frame. To model such a complex function, layers between the input and output (called hidden layers) are inserted into the mapping process.

Layers consist of a set of nodes, each of which reduces all the values in the previous layer to a single quantity. Each node can then be considered to represent a useful feature (e.g., a circle, or the edge of a brick) in the input data, and its corresponding quantity can be interpreted as the likelihood that the node's feature is present. Then, the features extracted from each layer are mapped to a new set of higher-level features (e.g., a bullseye target, or a brick shape) in the next layer using the exact same process. Because the neural network maps low-level features to higher-level features in each step, it should be unsurprising that increasing the number of hidden layers also increases the expressiveness of the model. However, this expressiveness can still be vastly improved.

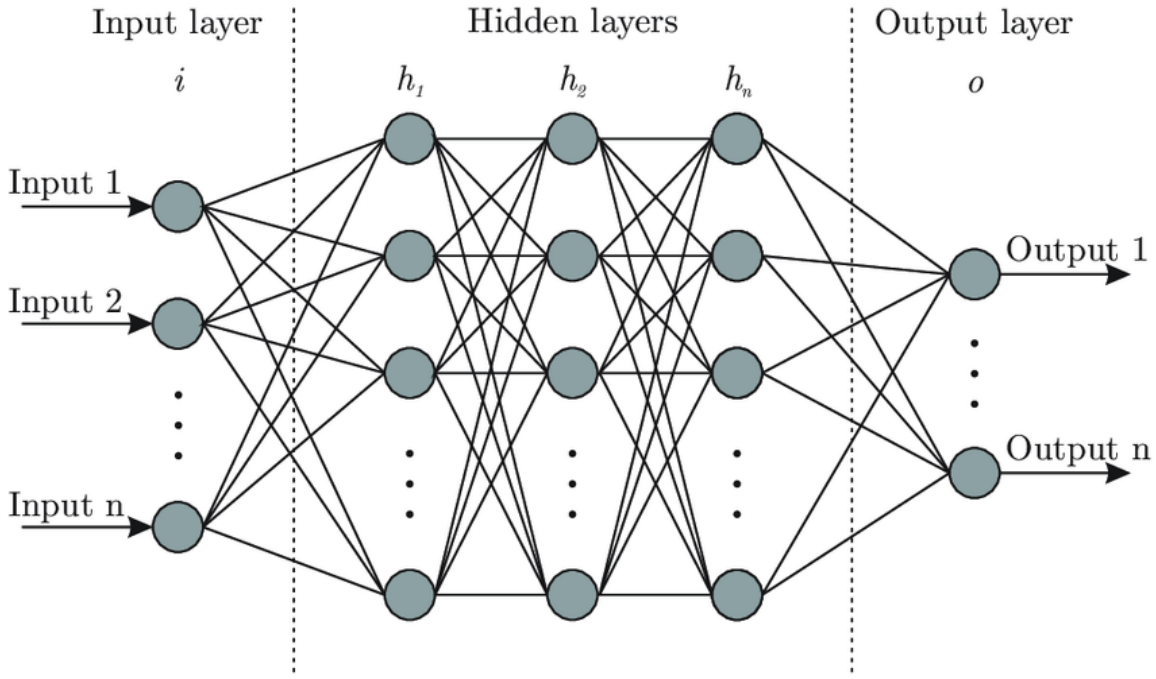

Figure 1: Classical structure of an Artificial Neural Network

The Locality Problem

The basic model of the ANN trains a node to quantify how important each of its individual inputs are to the feature that it represents. These measures of importance are called *weights*. In an image, this means that each node in the first hidden layer will decide how important each pixel is to the feature mapping. A problem arises when the object is not in the same spot that the ANN trained for; the pixels which are most important to each feature have moved.

When searching for objects in an image, it is never guaranteed where - in the picture frame - the features of import will be located. To find a brick in the input image, an ideal neural network would produce the same results if the brick were in the center of the image as it would if the brick were in the lower left corner. If a network is trying to learn to identify a panda - for instance - in every possible location in the picture frame, then a different node for each location has to learn identical weights. The network would need an enormous amount of data to train each of these nodes, a job that is completely redundant since it's a copy of every other node learning the exact same feature in different locations. With the average smartphone taking pictures at 12MP resolution, that’s twelve million pandas.

Convolutional Neural Networks

To avoid such an absurd amount of nodes in the network, the convolutional layer was born. In a convolutional layer, a kernel is applied to a small patch of the overall image (Figure 2). This kernel is reapplied to every potential location in the image frame, and the value from each location is remembered. Because of this, a node in a convolutional layer doesn’t have just one value, it has the values of the kernel applied at every location in the image. Now the network doesn’t have to train for twelve million pandas, but the value of the node can’t just be interpreted as a single magnitude anymore.

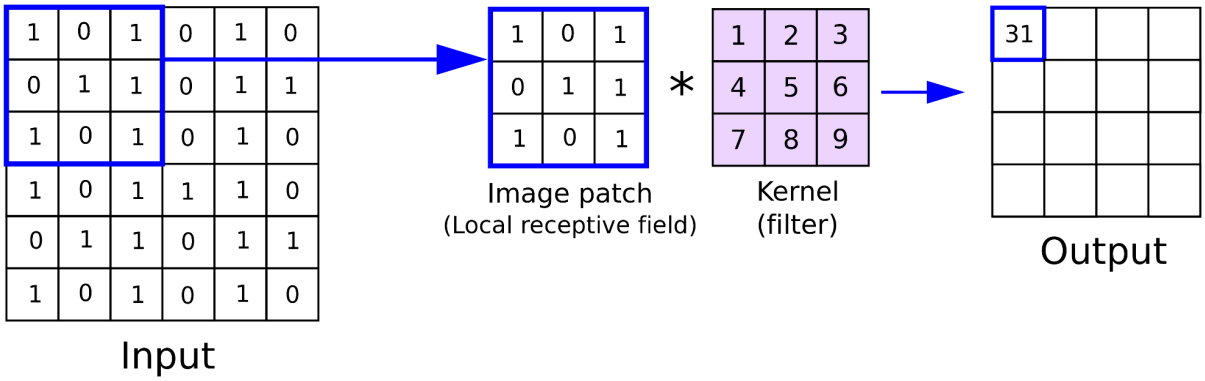

Figure 2: application of a kernel in a convolutional layer

To reduce the number of values that each node passes on to the next layer, another technique is applied, known as *pooling*. In pooling, sets of node values that are close in proximity to each other are *pooled* together into one value. A common example of pooling is the *max pool* (Figure 3). The max pool method takes adjacent cell values in a node, and reduces it to only the maximum value. In terms of the feature that the node is meant to represent, max pooling says that the magnitude of the feature in a given area is equivalent to the highest estimate from the kernel in the surrounding locations.

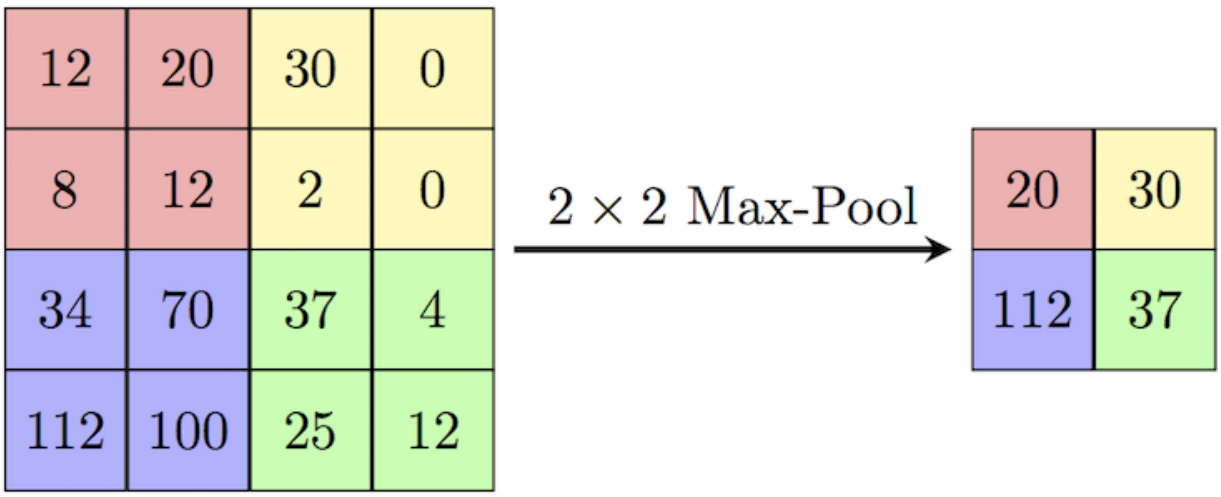

Figure 3: application of Max-Pooling following a convolution

A neural network which uses these convolutional layers is termed - somewhat unoriginally - a Convolutional Neural Network (CNN). After significant processing is performed on the output of the convolutional layers, the data is fed through the more familiar design of the fully-connected ANN introduced earlier (Figure 1).

While these CNNs will now recognize the same feature across every location, it still depends on the object remaining in the same size and orientation. One solution to this problem was to develop a *region-proposal network*, which predicted areas in the image where there was a high probability of finding an object. In these regions, the objects would be of a predictable scale. The regions were then fed individually to the CNN for object recognition. This process provided improved results at the cost of speed.

YOLO

Searching for a lightweight and dependable approach to object recognition in computer vision, a team of four researchers from the University of Washington set out to design a neural network architecture that capitalized on the speed of a purely convolutional architecture. Doing away with the inefficiency of region-proposals, Joseph Redman et al. [x] developed the first iteration of a soon-to-be famous architecture they duly termed *You Only Look Once* (YOLO).

After three initial versions (YOLOv1, YOLOv2, and YOLOv3) were released from the same team, official development was halted. Since then, several improvements and derivatives have been produced by individual contributors. The project presented here was built around one of the most successful versions released so far - YOLOv5 - designed by Glenn Jocher, founder and CEO of Ultralytics.

YOLOv5

The full architecture of the network (Figure 4) is separated into three sections: the *backbone*, the *neck*, and the *head* (sometimes *detect,* as in Figure 4). The backbone of YOLOv5 is the CSPDarknet architecture, called so because it is a combination of the Cross-Stage Partial Network (CSPNet) and Darknet53. The neck is called the Path Aggregation Network (PANet). Finally, the head (detect) is affectionately referred to as the YOLO layer.

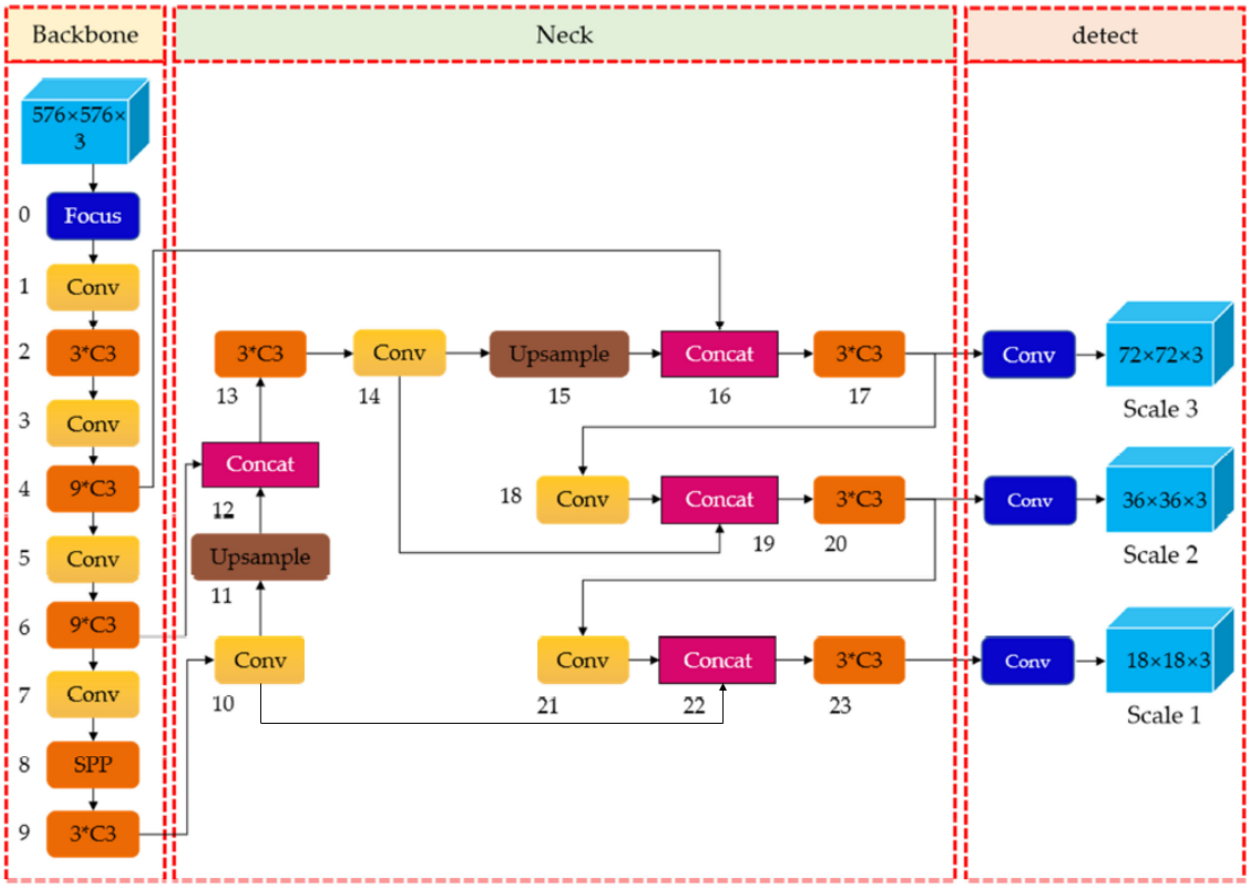

Figure 4: architecture of the YOLOv5 network

The primary duty of the backbone is to extract the initial pyramid of features from low-level to high-level. The CSPDarknet architecture was designed to do so with as few computations as possible, due to the increasing demand for accurate, fast, and mobile applications. One significant alteration that reduced the necessary memory and computations was to increase the amount of pooling at the very beginning of the process. This reduced the number of outputs at each convolution layer significantly, thus requiring less storage space and fewer computational resources, and opening the door for mobile implementation.

The Neck layer is largely responsible for aggregating the features detected in the backbone into a comprehensive map of the input image. By connecting several convolution layers from the backbone into different parts of the neck, the PANet architecture ensures information from low-level features and high-level features are combined. Because low-level features like lines are better indicators of location, while higher-level features like object curvature are better indicators of the abstract figures (the type of Lego brick), combining this data produces the maximal amount of information.

Finally, the YOLO layer handles the problem of differently-sized objects in the image by using three separately scaled convolution layers. Each layer searches for the objects in different sizes, so that the proximity to the camera does not significantly reduce the accuracy of the network.

Synthetic Data

Training a neural network requires a large amount of data. The lead designer of the YOLOv4 network - Alexey Bochkovskiy - recommends at least 2000 instances of each class. Additionally, images should have random numbers of objects in them, distractors that are intended to confuse the network, and many images should be empty. For each image, the network also needs a set of bounding boxes which tells it what objects are in the frame and where those objects are (Figure 5). Gathering the necessary images for just one of the bricks would be overly time-consuming, not to mention for the full dataset of all Lego pieces.

To reduce the amount of human input required to train the YOLO network, a python script was developed to automatically create the necessary images with perfect bounding boxes. Using the free Blender animation software, and the LDraw parts library, three-dimensional models of every Lego brick were used to prepare the neural network for the real world. After creating and importing the three-dimensional models into a Blender environment, real physics were simulated to better represent the environment inside the machine that the camera would be operating in. In the end, over ten thousand images were created for the training of the prototype.

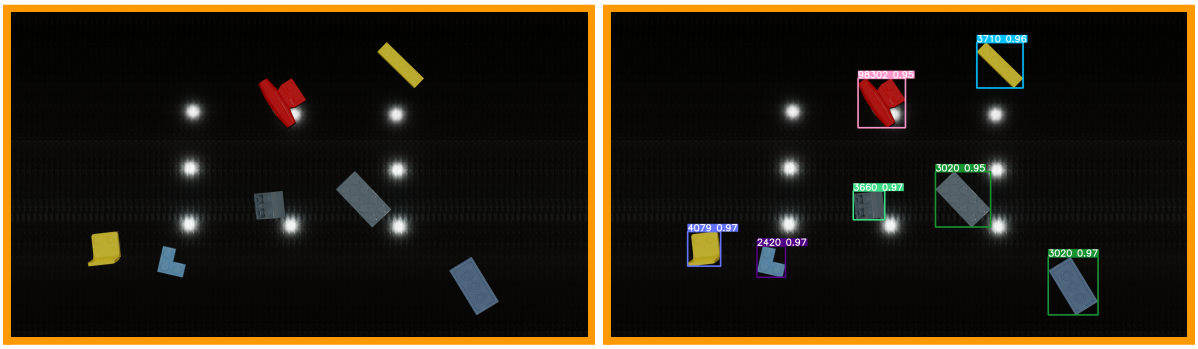

Figure 5: synthetic images without and with bounding boxes for generated for training

Even after training the model on thousands upon thousands of synthetic images, the network was very sensitive to the differences in the real images being taken by the camera. This adjustment to varying lighting conditions, angle adjustments, and other minute disparities is known as the sim-to-real problem. To train the YOLO network to ignore the extraneous factors and focus solely on the bricks, it was necessary to take real pictures and annotate them with bounding boxes manually. This was the final step in preparing the neural network for operation.

Mechanical Design

The major parts of the mechanical design were based on a Lego sorter YouTube video from Daniel West [7]. The Lego sorter made by West was created from nearly all Lego parts. This was not feasible for the machine in this project because of the price of Legos and the 300 dollar budget constraint. But the basic subsystems of the machine that West created were what the machine in this project was modeled after. The first subsystems was a central holding unit for all the unsorted Legos. The next subsystem was a shaker table that Legos from the holding unit were emptied onto to separate the pieces so they did not overlap. If an image of overlapping parts was fed into the neural network, it would be much less accurate at identifying a given Lego piece. The last subsystems was a conveyor belt that the Legos exited onto after the shaker table. These subsystems of the machine were called the hopper, the shaker table, and the conveyor belt respectively. A camera was mounted above the end of the conveyor belt to take images of the Lego pieces to be identified. Since the machine was not going to be made out of Legos, it was decided that it would be made from 3D printed and laser cut parts. Prototyping was done and CAD models were then created to make these 3 subsystems that have more detailed descriptions below.

The Hopper

The hopper was prototyped with cardboard and tape as shown in the figure below.

Figure 6: The cardboard prototype of the hopper.

This prototyping was done in order to get rough dimensions for the number of Legos that the hopper would hold and how big the bottom door of the hopper needed to be to allow the flow of Legos. It was decided that a servo motor would be used to actuate the bottom door of the hopper. Once this prototype had been created, a CAD model was made in SolidWorks as shown below.

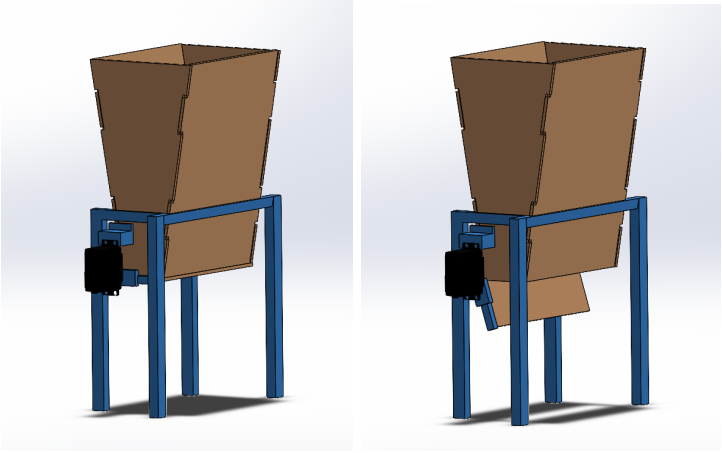

Figure 7: The CAD model of the hopper.

The blue parts correspond to 3D printed parts and the brown parts are those laser cut out of wood. The figure shows the door at the bottom of the hopper all the way closed (left) and open (right). The actuation of the door was done with a HiTEC HS-311 servo motor (link) that is controlled via PWM.

The Shaker Table

The shaker table was designed to vibrate the Legos so they were not overlapping once they exited onto the conveyor belt. There was not a prototype created for this part of the machine because it just needed to a table attached to a motor. The SolidWorks model is shown below.

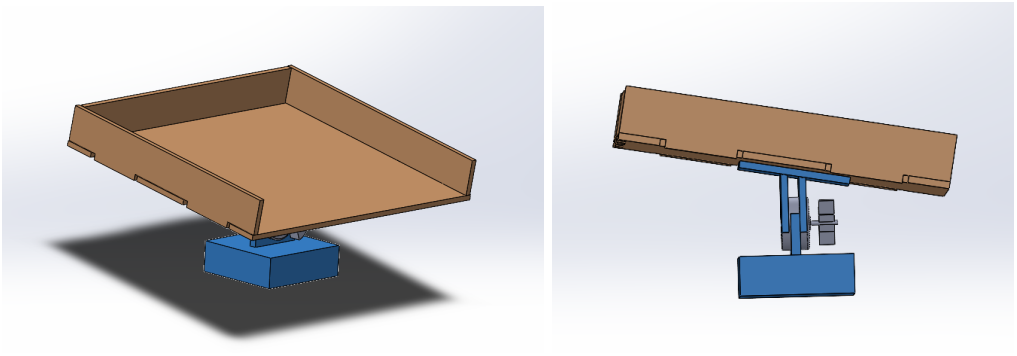

Figure 8: The CAD model of the shaker table.

As shown in the figure above the shaker table works by using a motor with an offset weight attached to its axel. When the motor spins it then causes the motor to shake. The motor used for this shaker table was a Vybronics VJQ24-35F580C (link). The table was also made at a slight slant to allow Legos to slide down it at a slow enough rate that they could still be separated by the vibration of the motor.

The Conveyor Belt

The conveyor belt was prototyped as shown below with cardboard, reusable straws for its axles, and paper for the belt as shown below.

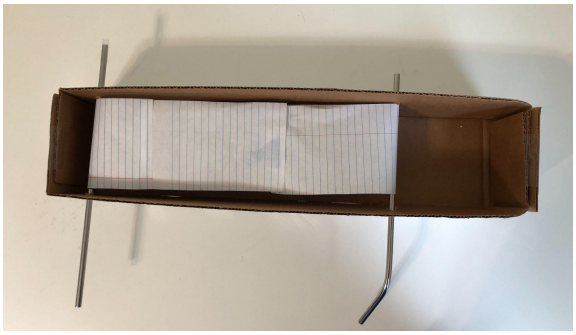

Figure 9: The cardboard prototype of the conveyor belt.

A couple of things were learned from the creation of this prototype. First, there was a lot of friction between the axles and the cardboard, so bearings would need to be used in the final design. Second, the paper was very difficult to get to turn on the axles, so some sort of fabric would need to be used in the final product. From the information learned by creating this prototype, a final CAD model was made as shown below.

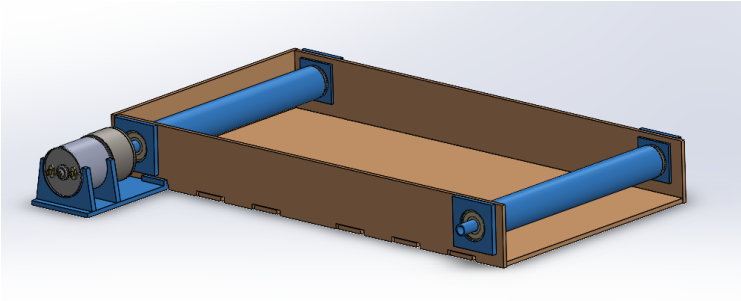

Figure 10: The CAD model of the conveyor belt.

As shown in the figure above, the conveyor belt is powered by a Greartisan 12V 100RPM DC motor (link) with attached gearbox. There are 4 VOLV Precision 608ZZ bearings (link) for each point where the axles are mounted to the supporting frame structure in order to reduce friction. A belt was added between the axles that was made out of cotton fabric.

The Full Machine and Assembly

An image of how all the subsystems of the machine connect is shown below.

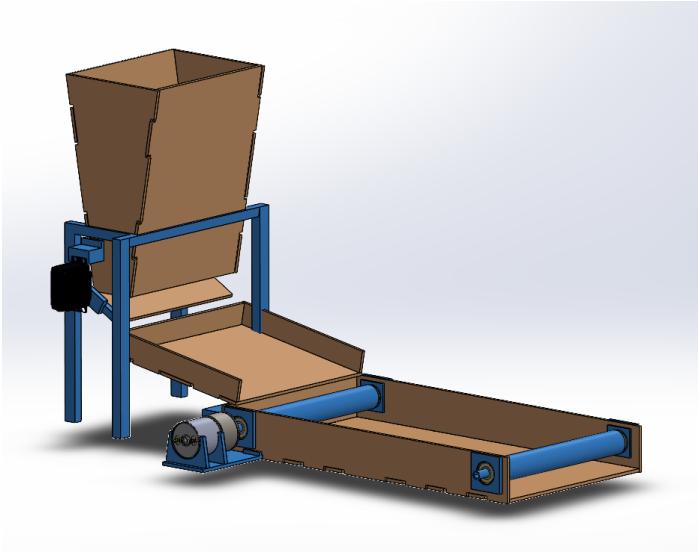

Figure 11: The full CAD model.

The CAD model in the figure above was used to manufacture the final parts for the machine. The parts that are blue in the figure were 3D printed on an Anycubic Mega Zero 2.0 FDM 3D printer (link) in Hatchbox PLA filament (link). The parts in brown were laser cut on the University of Minnesota Anderson Labs's PLS6.150D Laser Cutter (link) in ⅛ inch MDF wood. The machine was assembled and its completed version is shown below.

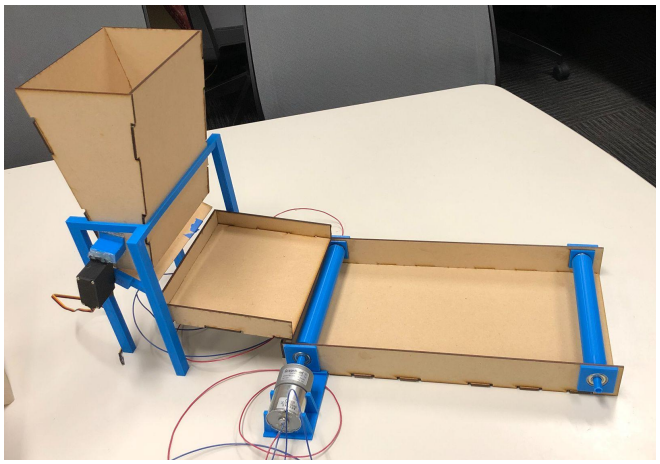

Figure 12: The full assembly of the machine's parts.

Most of the parts were designed to press fit together. This means that, for example, the blue squares that hold the bearings in place were made just big enough to hold the beading without it slipping and to fit into the square cut into the wood frame without the need for glue. The dimensions on these parts needed to be exact to just be able to fit together like this which required reprinting parts several times to get the correct fit. Instead of reprinting the full part several times, which would have wasted a lot of filament, a partial version of the part was printed to just test the necessary part of it that needed to be a certain tolerance. All these test prints are shown in the figure below.

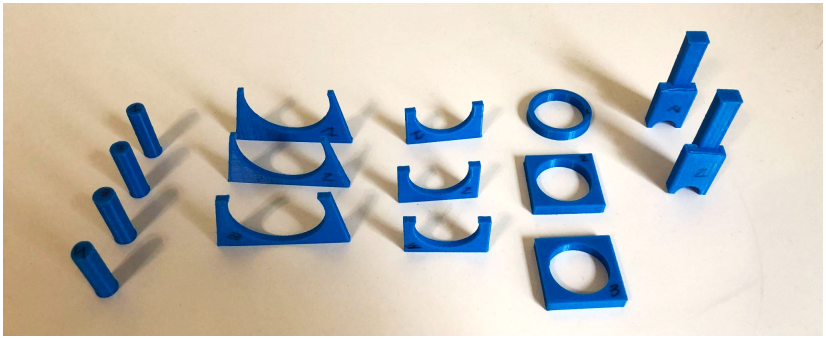

Figure 13: The tolerance test prints required to get the machine to fit together without glue.

The different parts that went through this tolerance testing were (listed from left to right corresponding to the image above) the axel, the conveyor belt motor mount, the shaker table motor mount, the conveyor belt bearing holder, and the servo adapter for the hopper door. Once these partial parts had a good enough fit around the parts they were interfacing with, the dimensions in the CAD model were adjusted and the whole part was 3D printed. The wooden parts were assembled with wood glue. The servo motor mount on the hopper and the shaker table motor mount were glued onto the wooden frame with hot glue.

Remote Operation

We use a Raspberry Pi 3B as the controller of our machine. It runs the web server for our machine's UI; captures and compresses images from our USB camera; and controls the conveyor belt, shaker motor, and hopper flow-control servo. The BCM2837 processor that the Raspberry Pi 3B is equipped with is not computationally powerful enough to handle inferencing at the rate we require, which is at least one image processed per second. To reconcile this issue, a remote machine equipped with a general purpose GPU is used for the inference part of the pipeline.

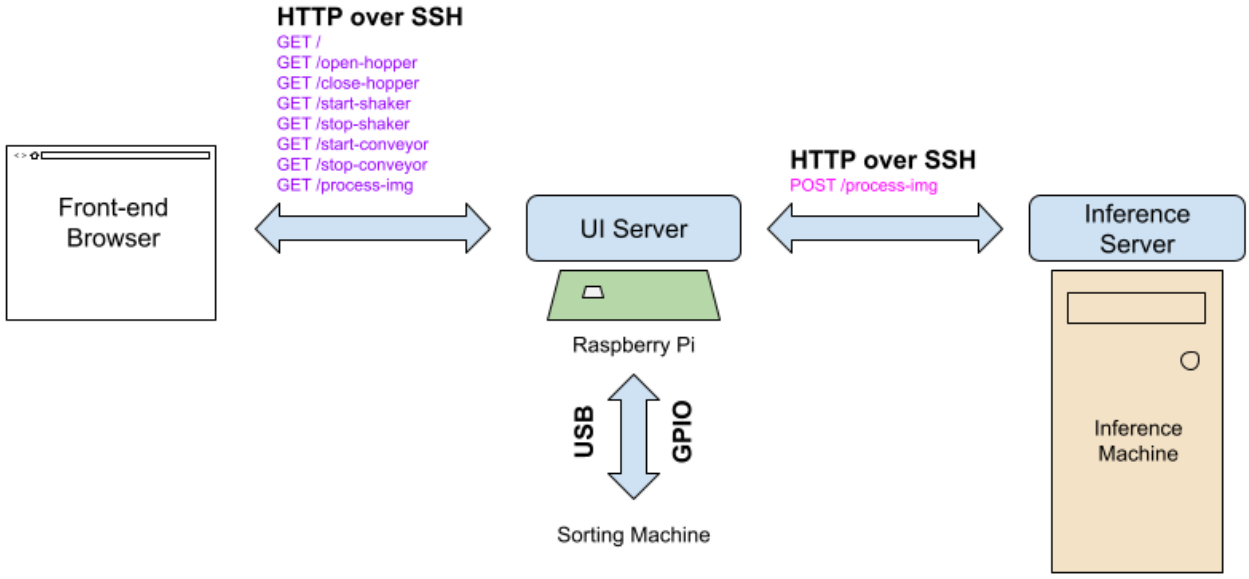

Figure 14: A diagram depicting the overall architecture of our remote communication infrastructure, along with the available endpoints on each web server, and the communication protocol(s) in use by each interconnect.

The front-end of the machine takes the form of a web page. While this web page is described in detail in the User Experience section of this report, the technology behind it is relevant here. The client-side JavaScript on the page makes the web requests necessary for the machine to work as intended. It issues GET requests to command the Raspberry Pi to open and close the hopper, as well as to start and stop the conveyor and shaker motors. No additional information is transferred beyond the request itself. This is contrasted by the `/process-img` endpoint. A GET request to this endpoint every second causes the Raspberry Pi to capture an image, make a POST request to the inference server with that captured image, and forward the response from the inference server back to the web browser. The actual behavioral logic of the machine is located within the web page's client-side JavaScript, which uses these single-purpose endpoints to control the machine as necessary.

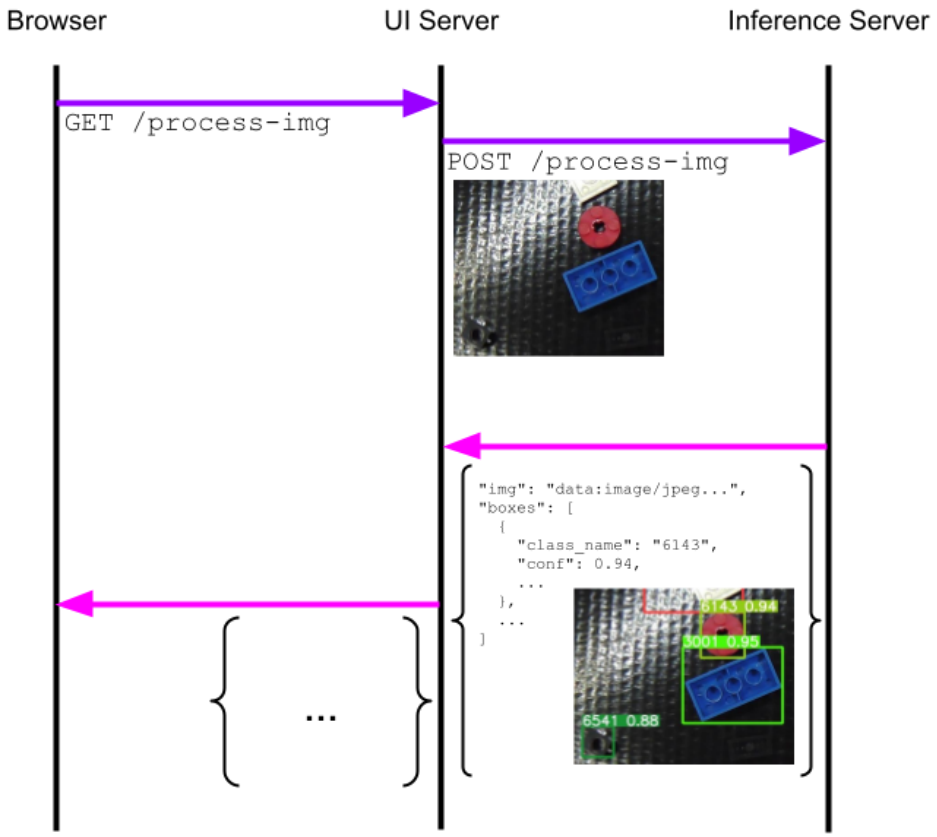

Figure 15: A time-sequence diagram for a browser's GET request to the /process-img endpoint on the UI server. This prompts the Raspberry Pi to send the last image it captured to the inference server, have the inference server annotate that image, then forward the returned results back to the browser.

The web server running on the Raspberry Pi, dubbed the UI server, is the only server the front-end website directly interacts with. This web server provides endpoints to control the two DC motors and single servo of the machine: `/open-hopper`, `/close-hopper`, `/start-shaker`, `/stop-shaker`, `/start-conveyor`, and `/stop-conveyor`. Each action is activated by a simple get request to the appropriate endpoint, and no data need be transferred beyond the HTTP GET request and 200 OK response. The implementation of these endpoints utilize the RPi.GPIO and gpiozero Python libraries to manage the state of the Raspberry Pi's GPIO pins. The RPi.GPIO library is used to set the state of the GPIO pins to toggle the transistors that manage the shaker motor and conveyor belt, while the gpiozero library is used to generate a PWM signal to set and maintain the position of the hopper door's servo motor.

Additionally, the UI server maintains a secondary thread that captures an image from the camera every 300 milliseconds and stores the pixels of that image in a shared memory buffer. Access to this shared memory buffer is protected by a mutex lock, and the last image stored in this shared buffer is read from whenever a request is made to the `/process-img` endpoint. This architecture is used to minimize the time required to service an HTTP request to this endpoint, as the last image taken will almost always be ready to send to the inference server as soon as a request comes in, rather than requiring the added delay of reading a frame from the camera. The only time this buffer will not be ready to send is when the mutex lock is held by the secondary thread, in which case the request must at most only wait the 20 milliseconds required for the secondary thread to copy its already-captured image into the shared buffer and release the lock. This secondary thread also handles edge-cases where the USB connection to the camera may be severed, in which case it continues attempting to reinitialize the camera until it succeeds. In the mean-time, the last captured image stored in the shared memory buffer continues to be served.

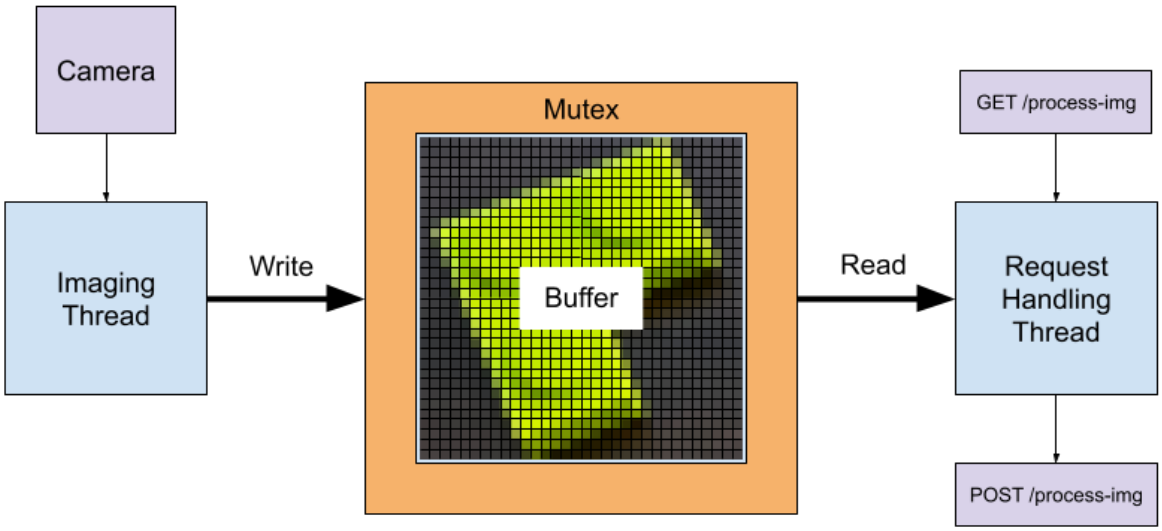

Figure 16: A diagram illustrating the interaction between the two threads of the UI server. The imaging thread captures an image and places it in the shared memory buffer every 300 milliseconds, and the request handling thread reads from that shared buffer whenever a GET request to the /process-img endpoint is received. Access to the shared memory buffer is arbitrated by a mutex lock.

The web server running on the inference machine, also called the inference server, handles POST requests to its `/process-img` endpoint. While this endpoint has the same name as the UI server's `/process-img` endpoint, it is actually used by the UI server in servicing the inference step of its own `/process-img` endpoint, as illustrated in the time-sequence diagram of figure 15. To service this request, the inference server reads the image file sent by the UI server, resizes the image to match the size the neural network was trained with, runs the inference process on that resized image, Base64-encodes an annotated version of the input image, and finally writes the Base64 string and a list of detected Lego pieces to a JSON object to be sent back to the UI server, then forwarded back to the client browser by the UI server.

The inference server has a comparatively simpler architecture, as another thread does not need to explicitly be created for its operation. Rather, on launch of the inference server, its inference capability is initialized, then afterward the web server begins listening for requests. When a request comes in, the same thread handling the request also runs the inference code on the image that arrived with the request.

To implement the inference process, one obvious solution would be to simply call the detect.py script provided by the authors of YOLOv5 within our program. However, this detect.py

script requires at minimum 5 seconds to execute, and 95% of this execution time consists of the initialization process where the trained model is loaded onto the GPU, and that model is briefly "warmed up" to improve its initial accuracy. Rather than deal with this penalty for every incoming request, another solution was used instead. The required Python code was extracted from the detect.py script and added directly to the inference server. Upon starting the inference server, the initialization process required for inference is done immediately. This involves selecting the appropriate compute device, loading our trained model onto that device, and then running the warmup process to improve initial accuracy. After those steps are complete, the server begins listening for incoming requests. When handling requests, the already-initialized model is simply reused, completely alleviating the aforementioned 5 second penalty.

When a POST request to the `/process-img` endpoint is received, the image is decoded, resized to be compatible with the trained model, ran through the model, and finally annotated and sent back, along with a list of the detected pieces, as a JSON string. When the results are returned, they are filtered to remove predictions below a certain threshold, which in this implementation the confidence threshold is 0.35. A single region of the photo is permitted to be labeled with multiple parts numbers. This was deemed permissible as false negatives are more detrimental to the user experience than false positives. If the machine detects a piece that is not there, the piece can be removed and the machine easily resumed. However, if the machine fails to detect the piece desired by the user, that user will have to feed all the Legos back through the machine another time. Allowing a piece to be identified as multiple possible part numbers makes these false negative situations less likely. The results are used to then annotate the original image to show the user what has been detected, and to create a computer-friendly list of detected pieces for the client-side JavaScript to search for the desired part number within. A scaled version of the annotated image is Base64-encoded, and both the Base64 string and the list of part numbers are serialized into a single JSON string, an example of which is shown in figure 17.

```
{
  "boxes": [
    {
      "class_name": "3001",
      "conf": 0.3962140381336212,
      "x": 108.5,
      "y": 94,
      "w": 35,
      "h": 110
    },
    {
      "class_name": "3031",
      "conf": 0.5624125003814697,
      "x": 102.5,
      "y": 25.5,
      "w": 39.66666793823242,
      "h": 51
    }
  ],
  "img": "..."
}
```


Figure 17: An example of the JSON string returned by the /process-img endpoint of the inference server, containing both a list of seen objects and a smaller, annotated version of the input image. The Base64-encoded image is truncated for brevity.

User Experience

A website is used as a user interface for builders to interact with the machine. A user can enter a Lego part number to be searched for into the text box. The text box also has the ability for users to select which piece they would like to search for via a drop-down menu. When the start button is clicked, the website will issue a request to the Raspberry Pi that will start the machine. This includes opening the hopper, starting the shaker plate, rolling the conveyor belt, and beginning to take and process images. At this time, the user can start pouring Lego pieces into the hopper, where they will flow out and be separated with the shaker plate and deposited onto the conveyor belt. An image of the conveyor belt will be displayed on the website below the start button that will be updated with the results of the YOLO algorithm as fast as possible.

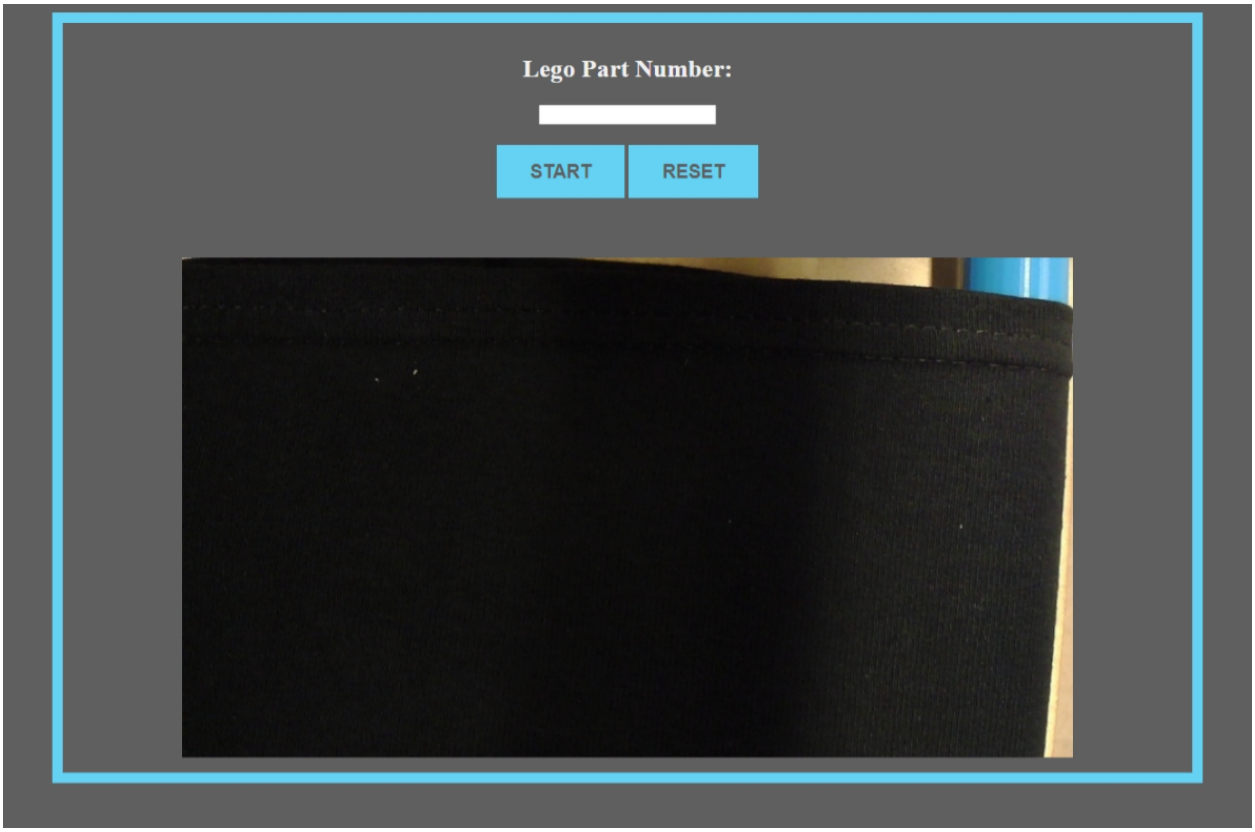

Figure 18: The user interface website

The machine will continue to run until the piece is found, or the reset button on the user interface is pressed.

Results

In analyzing the performance of our implementation, we will consider the rate at which it can process Lego pieces, the accuracy of our neural network as tested by our validation dataset, and how that translates to real-world accuracy. The most significant bottleneck for processing Lego pieces comes from the rate at which images can be taken and annotated. More specifically, the bottleneck originates from how fast images can be transferred between the browser, Raspberry Pi, and inference machine. In our primary setup, the client machine and Raspberry Pi were on the same local area network (LAN), giving them a high-speed and reliable connection to each other. However, the Raspberry Pi and inference machine were connected over the Internet via a mobile cellular hotspot, which is a relatively unreliable and low-speed connection. Thus, depending on the connection quality, transferring the 1920x1080-pixel JPEG image captured by the Raspberry

Pi to the inference machine could take anywhere from tens of milliseconds to tens of seconds. In contrast, the rest of the image processing execution time consistently hovered around 350 milliseconds at this resolution. In normal, stable network conditions, and given around 25 Lego pieces fitting comfortably in-frame at a time, our machine is capable of processing up to $25\ pieces/image \times \frac{1000\ ms/sec}{400\ ms/image} = 62.5\ pieces/sec.$

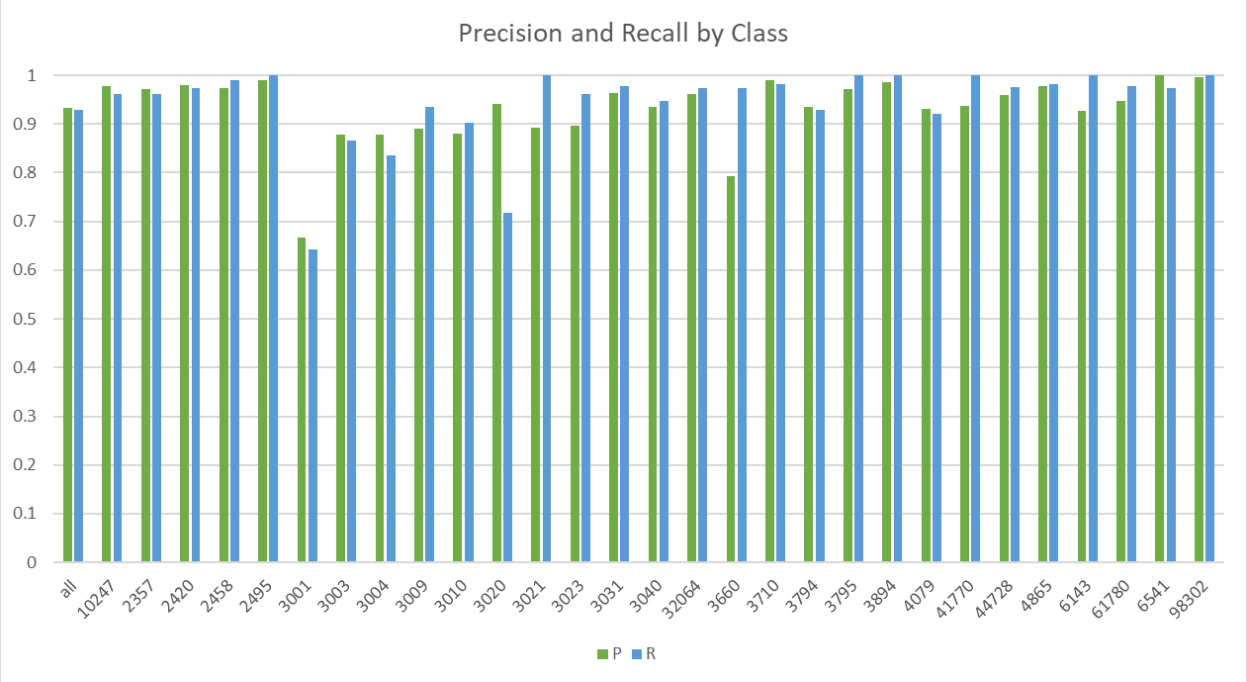

Figure 19: A chart illustrating the precision (P) and recall (R) metrics of the different Lego part numbers covered by our trained model.

The final trained neural network has a precision of 93.4% and a recall of 92.9% over all classes/pieces. Noticeable deficiencies can be seen for pieces that have similar counterparts. For example, both pieces 3001 and 3020 are 2x4 Lego pieces and only differ in their height. Thus, from above, they appear identical. This poses a problem for our setup, where we have one camera aimed down from above, seeing only the top of pieces located in the center of the frame. Overall, our F1 accuracy metric evaluates to 93.1%:

$$F1\ accuracy = \frac{1}{\frac{1}{0.934} + \frac{1}{0.929}} = 0.931 = 93.1\%.$$

Discussion

Improvements in the Mechanical Design

There are several parts of the mechanical design that could be improved. First, the slant of the shaker table needs to be steeper to allow all types of Legos to slide down it. If the shaker table is made steeper the hopper will need to be raised slightly higher to account for the front of the shaker table being higher. The belt of the conveyor belt also needed to be remade. It was sewed slightly too tight, so it just needs to be made slightly larger. The camera mount still needs to be made. A cardboard box was used at the design show to mount the camera above the conveyor belt. The biggest improvement that can be made to the mechanical portion of the project would be better motor control circuits. Currently, the Raspberry Pi PWM library is very jittery and causes the servo attached to the hopper to jump around when it should be held at a set position. Also, the DC motors were just attached to MOSFETs so their speed could not be controlled, it was just an on or off in terms of DC motor control. It would greatly improve the conveyor belt if a motor controller was used to control its motor. Currently the 5 V and 12 V sources that are available to run the motor are too fast for the neural network to easily detect a give Lego piece on the conveyor belt. A motor controller capable of running the motor at different speeds would solve this problem.

Improvements in Machine Learning

While YOLOv5 has spent its time at the top of many object recognition dataset competitions, its age is beginning to show. The YOLO-X network, released in 2021, is showing accuracy rates which improve upon the YOLOv5 network [8]. Another framework which is rivaling the CNN, known as the transformer, has also shown improved performance over YOLO networks. The Microsoft SWIN (Shifted Windows) transformer is at the top of this list. While the SWIN transformer was passed over for the initial phase of this project due to its hardware requirements, the accuracy of this architecture could serve as a boost to the machine. Further area for improvement in the machine learning model lies in the training approach. A new paper even suggests that vision transformers such as SWIN are showing improved accuracy more largely due to the training parameters than the architecture shift [9].

Improvements in Synthetic Data

With some changes, the synthetic data generation process could be improved to produce more true-to-life annotated images, improving the accuracy of our trained neural network. The Blender material applied to the Lego pieces could be improved by updating the shading to better match the spectral behavior of the plastic used in real Lego pieces. For example, it could better account

for the dramatically increased reflection observed when viewing a real Lego piece from an off-angle perspective. The Blender material could also greatly benefit from mimicking the random scuffs, fingerprints, and scratches seen on the surface of a real Lego piece.

The generation script as a whole could be improved by better supporting Lego pieces that are actually combinations of multiple part numbers, and are typically never seen disassembled. For example, part numbers 3679 and 3680 are snapped together to create a 2x2 base with a turn plate, shown in figure 20. While the upper part and lower part could be included during synthetic data generation, the script currently has no support for snapping them together and treating them as one whole piece, nor for creating a piece made up of more than a single color. Additionally, support could be added to generate data capable of training a neural network designed for use with multiple simultaneous camera angles. A setup with multiple camera angles would greatly improve the machine's capability to distinguish between Lego pieces that have different overall forms but appear identical when viewed from particular angles.

Figure 20: A lego piece formed by part numbers 3679 (2x2 turn plate upper) and 3680 (2x2 turn plate lower), currently unsupported by our synthetic data generation script.

Finally, the Blender scene itself could be improved by making it more closely match our final design. Due to difficulties encountered working with the thick rubber conveyor belt we originally purchased for this project, it was swapped out for thinner fabric, which appears significantly different from the original shiny rubber material emulated by the synthetic data. Additionally, our final arrangement and number of LEDs changed towards the end of the project and hence was not accounted for in the generated synthetic data. Updating the synthetic dataset to include these changes would allow the data to more closely match the real world, and hence further improve the accuracy of our trained neural network.

Conclusion

The work done has demonstrated the full functionality of identifying Lego pieces physically out of a pile, by showing the utilization of the YOLOv5 neural network model. Inspired by the algorithm creator, we were able to train a significant amount of efficient data to raise the detection accuracy of the objects to 93.1%, by synthesizing images within standards. We performed the smart use of the Raspberry Pi as both our web server and the controller to the hardware components. Being served by the Pi, the user interaction was achieved with the approachable and explanatory website interface. On the mechanical side, we successfully adopted 3D printing and laser cut technology and constructed the printed models in optimized dimensions with the help of circuit controls. With every piece of design combined, the project reached a sign of success by achieving the initial goals as indicated in the project description at the beginning of the semester.

Acknowledgments

The team wishes to express their gratitude to several parties whose support enabled the success of this endeavor:

- Professor Bazargan - the team advisor - for his expert knowledge, advice, and encouragement that guided us in tackling every obstacle;
- Professor Amin, and Jackson Benning - the course instructors - for their supportive feedback throughout the term; and
- Anderson Labs for helping the team source materials in a time when supply was scarce.

References

[1] Li Z, Tian X, Liu X, Liu Y, Shi X. A Two-Stage Industrial Defect Detection Framework Based on Improved-YOLOv5 and Optimized-Inception-ResnetV2 Models. *Applied Sciences*. 2022; 12(2):834.

[2] Xu, Renjie & Lin, Haifeng & Lu, Kangjie & Cao, Lin & Liu, Yunfei. (2021). A Forest Fire Detection System Based on Ensemble Learning. Forests. 12. 217. 10.3390/f12020217.

[3] Wang, C.Y.; Mark Liao, H.Y.; Wu, Y.H.; Chen, P.Y.; Hsieh, J.W.; Yeh, I.H. CSPNet: A new backbone that can enhance learning capability of cnn. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2020), Washington, DC, USA, 14–19 June 2020; pp. 390–391.

[4] Wang, K.; Liew, J.H.; Zou, Y.; Zhou, D.; Feng, J. Panet: Few-shot image semantic segmentation with prototype alignment. In Proceedings of the IEEE International Conference on Computer Vision (ICCV 2019), Seoul, Korea, 20–26 October 2019; pp. 9197–9206.

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[7] West, Daniel. *The World's First Universal Lego Sorting Machine - YouTube*. 3 Dec. 2019, https://www.youtube.com/watch?v=04JkdHEX3Yk.

[8] Zhang Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. *Yolox: Exceeding yolo series in 2021*, 2021.

[9] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. *A convnet for the 2020s*, 2022.