Raspberry Tie

Verification of Structural Connectors with Simpson Strong-Tie

Faisal Alajmi
faisal.alajmi@ucdenver.edu

Mohammed Alali
mohammed.2.alali@ucdenver.edu

Phillip Koffman
phillip.koffman@ucdenver.edu

Christian McCabe
christian.mccabe@ucdenver.edu

Abstract

Simpson Strong-Tie is a leading company in the manufacture of structural connection components, the use of which spans countless residential, commercial, and industrial projects. Occasionally, the installation of these components can go awry, whether that be from misalignments, structural defects, incorrectly sized parts, etc. The task given to us by Simpson is to develop a device to recognize and characterize an incorrect installation of their connectors, particularly their joist hangers. Being able to efficiently verify the correct installation of a joist hanger not only allows a technician to get more done, but it increases the overall safety of the project as well.

To do this, our team has developed a visual sensor that relies on an image processing algorithm. A user would be able to take a simple photograph of the joist hanger of interest, and the sensor would output whether that particular installation falls within tolerance of a safe and correct install. While the end result (a simple 'GO' or 'NO-GO' readout on a screen) seems underwhelming, our image processing algorithm takes into account a myriad of factors not readily seen by the naked eye. Our hope is our device makes construction a simpler and safer task.

Design Methodology

Our project consists of two subsystems: an image sensor and an image processing program, each with their own subcomponents. Our first subsystem is an image sensor. Based on stakeholder feedback and internal discussion, our image sensor must provide a durable housing for field use, full HD resolution, an ability to communicate wirelessly with a host machine for processing, and a rechargeable battery capable of being used for a full work day. A Raspberry Pi 4 Model B fits these requirements. It is a robust embedded system with a plethora of aftermarket accessories and support at a reasonable price point. An 8-megapixel Sony IMX219 sensor was chosen to fulfill the need for full HD resolution.

Our image processor went through several prototypes and design changes to meet the challenges this project sought to solve. Our first prototype was a Python-based convolutional neural network, a branch of machine learning that is commonly applied to analyze visual imagery. While significant work was done attempting to refine this program, ultimately the amount of training data needed to improve the recognition component of the program proved too much.

Our second prototype used a MATLAB image processing suite called 'Raspberry Pi Support from MATLAB'. This suite allowed for wireless communication with a host machine.

Connecting to a wireless network via secure shell WiFi allowed for the acquisition of data from the Raspberry Pi's camera as well as the visualization and analysis of said data. This method of communication allowed for rapid changes to the sensor's settings and is the method we used most for testing.

Our image processing design for our machine-learning prototype follows the general roadmap illustrated below:

Figure 1 – Project pipeline

- Data Ingestion: Collect joist hanger images from Simpson Strong-Tie's hanger documentation.
- Explore and Preprocess Data: Reference a user-curated library to obtain image commonalities, then apply preprocessing techniques to prepare images for the machine learning model.
- Model Training: Implement a pre-trained model from the Keras library and feed it with our dataset.
- Model Evaluation and Testing: Use a test dataset to evaluate the performance of the model and calculate how accurate the prediction is.

Engineering Documents

Machine-learning implementation sequence:

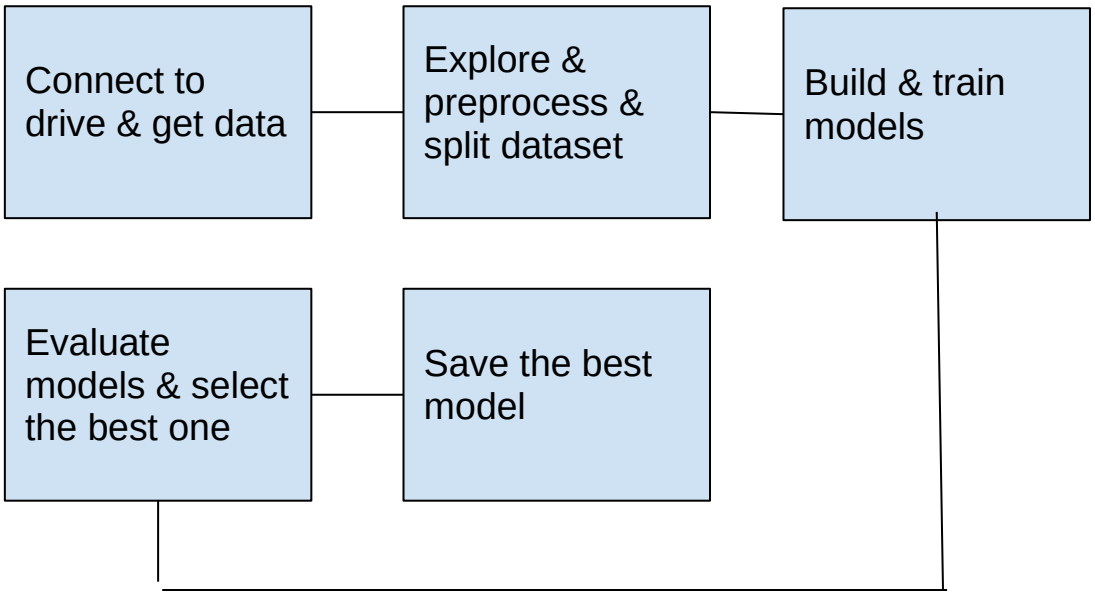


Figure 2 – Machine learning implementation

First, we connect a Google Drive to Colab to get the dataset:

```
import matplotlib.pyplot as plt
```

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive
```

```
!cp -r /content/drive/MyDrive/hangar.zip /content/
```

```
[ ] !unzip /content/hangar.zip

    Archive:  /content/hangar.zip
       creating: hangar/
       creating: hangar/good_hangar/
      inflating: hangar/good_hangar/2.jpeg
      inflating: hangar/good_hangar/107.jpg
```

Figure 3 – Get the dataset from Drive

Explore and Preprocess and Split Dataset:

In this section, we explore data to get the total number of images and see if the data balanced or not, and split data into train, test, and valid datasets. We then convert the image to NumPy array to paper it to be used in math and machine learning model operation. We used data augmentation which is a technique used to increase the amount of data by adding slightly modified copies of already existing data.

Below figure is a sample of augmented data:

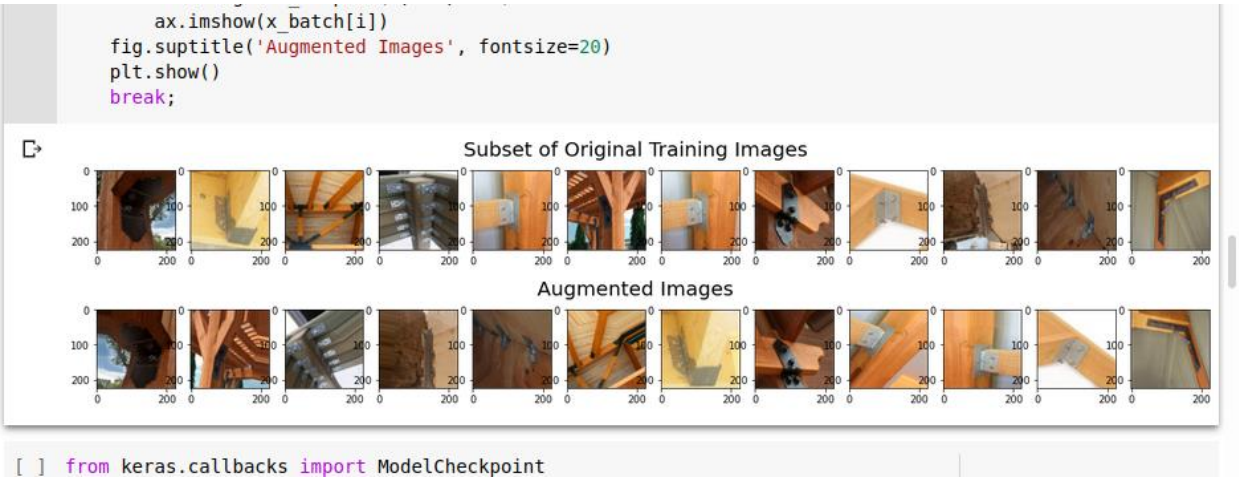


Figure 4 – Sample of augmented data

```
] path ="/content/hangar" ; path

   '/content/hangar'

] import splitfolders  # or import split_folders
  splitfolders.ratio(path, output="dataset", seed=1337, ratio=(.8, .1, 0.1)) # default values

   Copying files: 292 files [00:00, 3306.68 files/s]
```

Figure 5 – Code sample (image numbers)

```
    hangar_files = np.array(data['filenames'])
    class_weights = class_weight.compute_class_weight('balanced',
                                                      np.unique(data['target']),
                                                      data['target'])
    hangar_targets = np_utils.to_categorical(np.array(data['target']), 2)
    return hangar_files, hangar_targets, class_weights

# load train, test, and validation datasets
train_files, train_targets , class_weights_train = load_dataset('/content/dataset/train')
valid_files, valid_targets, class_weights_train_val = load_dataset('/content/dataset/val')
test_files, test_targets, class_weights_train_test = load_dataset('/content/dataset/test')

# load list of classes names
hangar_names = [item[20:-1] for item in sorted(glob("/content/dataset/train/*/"))]

# print statistics about the dataset
print('There are %d total Hangar categories.' % len(hangar_names))
print('There are %s total Hangar images.\n' % len(np.hstack([train_files, valid_files, test_files])))
print('There are %d training hangar images.' % len(train_files))
print('There are %d validation Hangar images.' % len(valid_files))
print('There are %d test hangar images.'% len(test_files))
```

```
There are 2 total Hangar categories.
There are 292 total Hangar images.

There are 233 training hangar images.
There are 28 validation Hangar images.
There are 31 test hangar images.
```

```
[ ] from PIL import ImageFile
    ImageFile.LOAD_TRUNCATED_IMAGES = True

    # pre-process the data for Keras
    train_tensors = paths_to_tensor(train_files).astype('float32')/255
    valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
    test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|██████████| 233/233 [00:02<00:00, 109.34it/s]
100%|██████████| 28/28 [00:00<00:00, 149.11it/s]
100%|██████████| 31/31 [00:00<00:00, 86.14it/s]
```

Figure 6 – Code sample - convert image to tensor

After preparing the dataset, we build the model. First, we build a basic model to be the reference for other model's performance:

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.models import Sequential
from keras import optimizers
input_shape = train_tensors[0].shape
from keras.callbacks import ModelCheckpoint , EarlyStopping


model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                 input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(2, activation='sigmoid'))


model.compile(loss = 'categorical_crossentropy',
              optimizer=optimizers.RMSprop(),
```

Figure 7 - Code sample - basic model

The below figure is the model architecture:

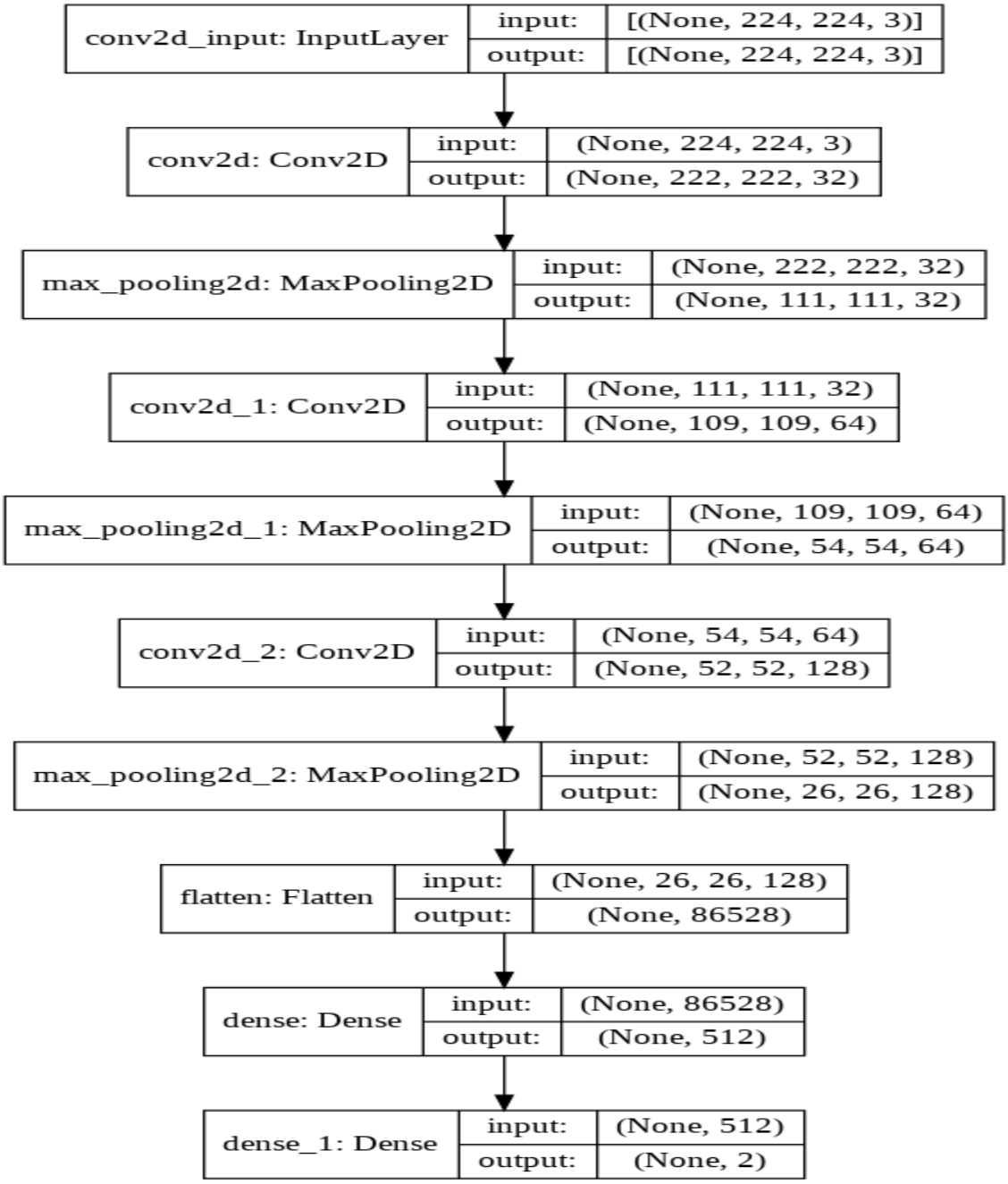


Figure 8 - The model architecture

After this, we set model hyperparameters, and train the model then visualize the result:

```
batch_size = 32
num_classes = 2
epochs = 30
input_shape=(224, 224, 3)
```

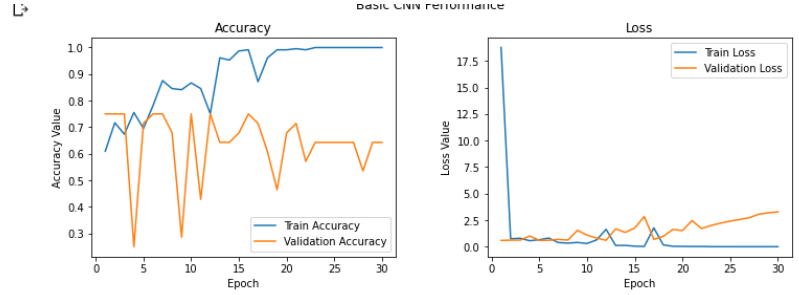
```
[ ] checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.hdf5',
                                   verbose=1, save_best_only=True)
```

```
history = model.fit(train_tensors, train_targets,
          validation_data=(valid_tensors, valid_targets),
          epochs=epochs, batch_size=batch_size,
           callbacks=[checkpointer], verbose=1)
```

```
Epoch 1/30
8/8 [==============================] - 33s 330ms/step - loss: 26.9446 - accuracy: 0.6085 - val_loss

Epoch 00001: val_loss improved from inf to 0.58729, saving model to saved_models/weights.best.from_
Epoch 2/30
8/8 [==============================] - 1s 131ms/step - loss: 0.7380 - accuracy: 0.7025 - val_loss:

Epoch 00002: val_loss did not improve from 0.58729
```




```
[ ] # get index of predicted dog breed for each image in test set
    hangares_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0))) for tensor in test_

    # report test accuracy
    test_accuracy = 100*np.sum(np.array(hangares_predictions)==np.argmax(test_targets, axis=1))/len(hang
    print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 80.6452%
```

Figure 9 - Code sample - the model result

To improve the result we add a dropout layer to the previous model. Dropout is a regularization method that gives the model ability to train a large number of neural networks with different architectures in parallel. It works by randomly ignoring or “dropping out” some number of the model layers during training:

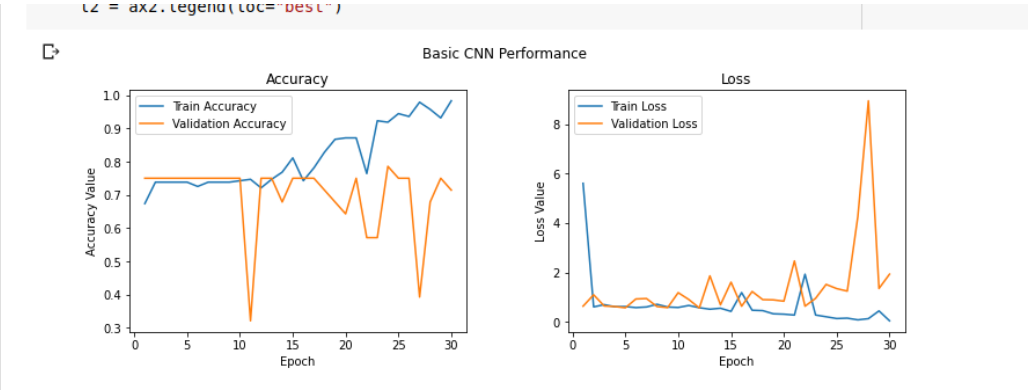


Figure 10 - Code sample - the model result after adding drop out

To increase accuracy, we use a transfer learning technique, which is a machine learning method where a pre-trained model is developed for a task and reused as the starting point for a model on a second task because a lot of general pattern in images have some similarity:

```
from keras.applications import vgg16, vgg19,inceptionResNetV2
from keras.applications.resnet import ResNet152, ResNet50
from keras.applications.mobilenet import MobileNet
from keras.layers import Input, Flatten, Dense, Dropout, GlobalAveragePooling2D, GlobalM
from keras.models import Model
import keras
def model_maker():
    base_model =MobileNet(include_top=False, input_shape = (IMG_WIDTH,IMG_HEIGHT,3))

    for layer in base_model.layers[:]:
        layer.trainable = False # Freeze the layers

    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    custom_model = base_model(input)
    custom_model = GlobalAveragePooling2D()(custom_model)
    custom_model = Dense(64, activation='relu')(custom_model)
    custom_model = Dropout(0.2)(custom_model)
    predictions  = Dense(NUM_CLASSES, activation='sigmoid')(custom_model)
```

Figure 11 - Code sample - transfer learning

We used MobileNet neural network and this is model diagram:

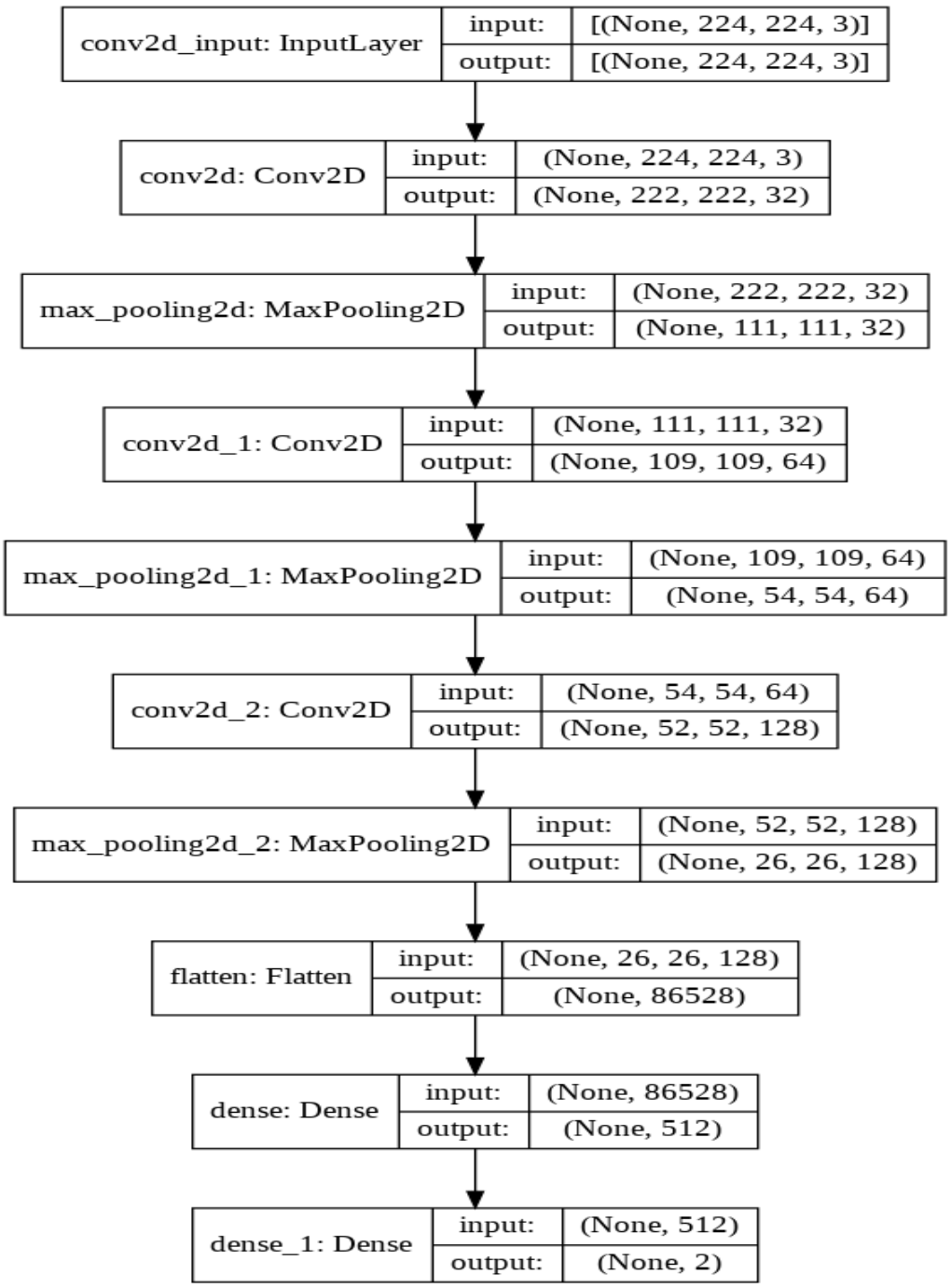


Figure11 - MobileNet neural network model diagram

This enhances the model result and also we do not have an overfitting problem:

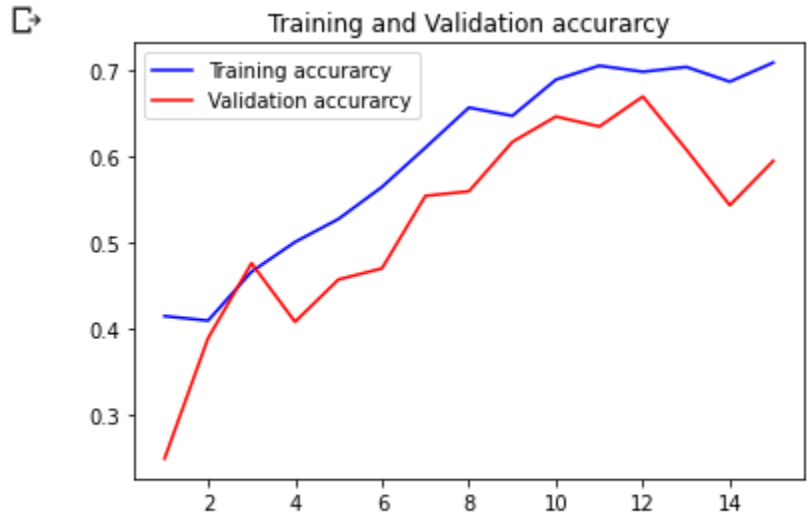
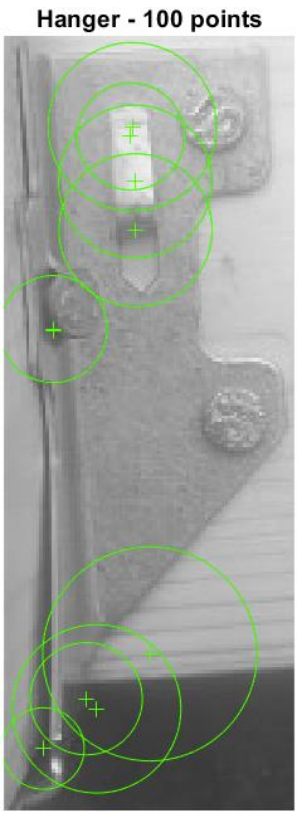


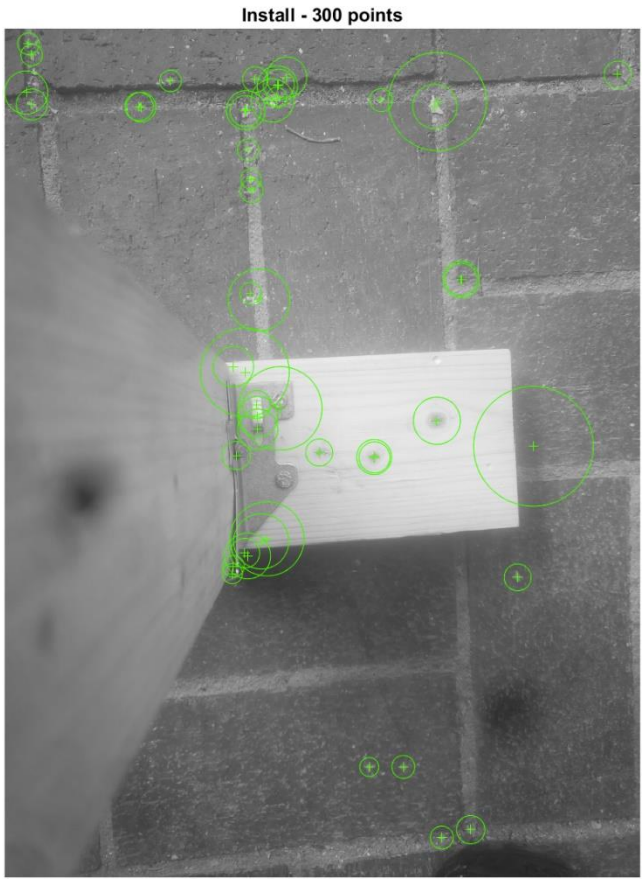
Figure 12 - MobileNet result

MATLAB-based Image Processing Prototype

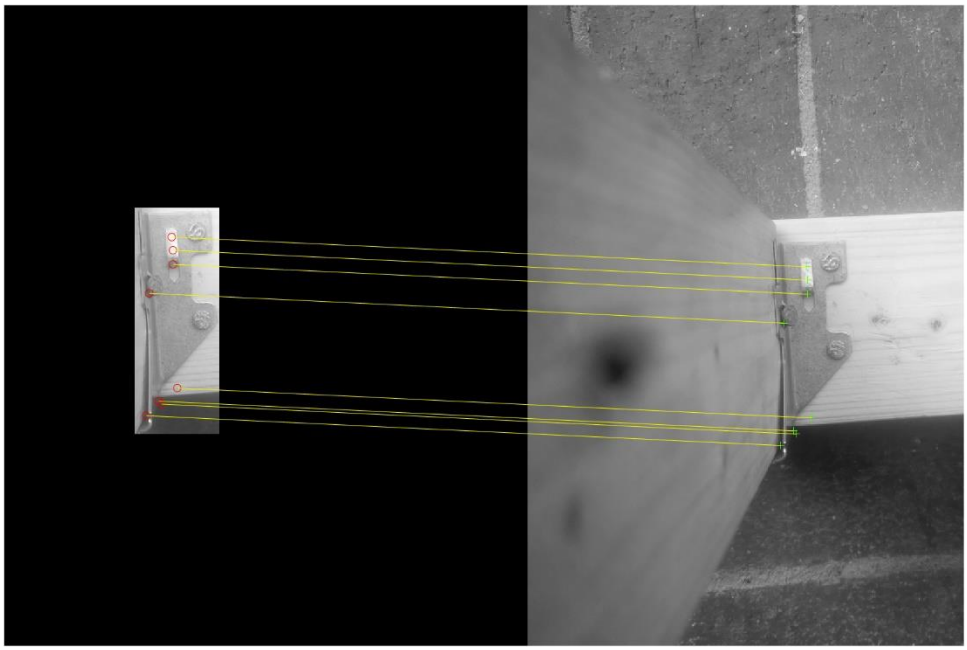
The MATLAB-based image processing program is relies on feature matching and line detection. These algorithms are included in MATLAB's 'Raspberry Pi Support from MATLAB' image processing suite.

In short, the program would detect a reference, then calculate relevant geometry based on the model of hanger. The first method tested was point feature matching. To obtain the object of reference, the program must know what it is looking for. Using a Speeded Up Robust Features (SURF) algorithm, the outstanding features of a given image are detected and mapped. We first fed an image of our object of interest into the SURF algorithm, then fed an image obtained from our sensor and let the algorithm choose prominent points.

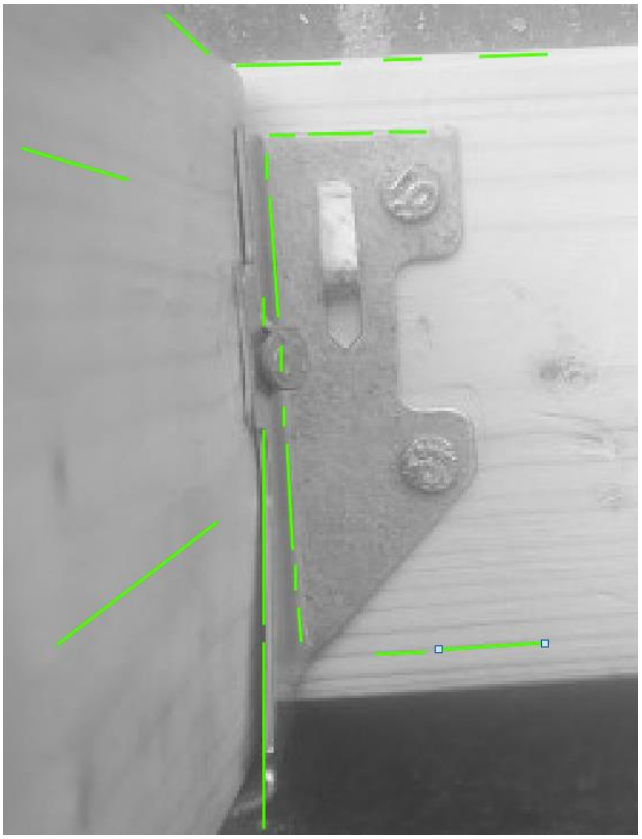


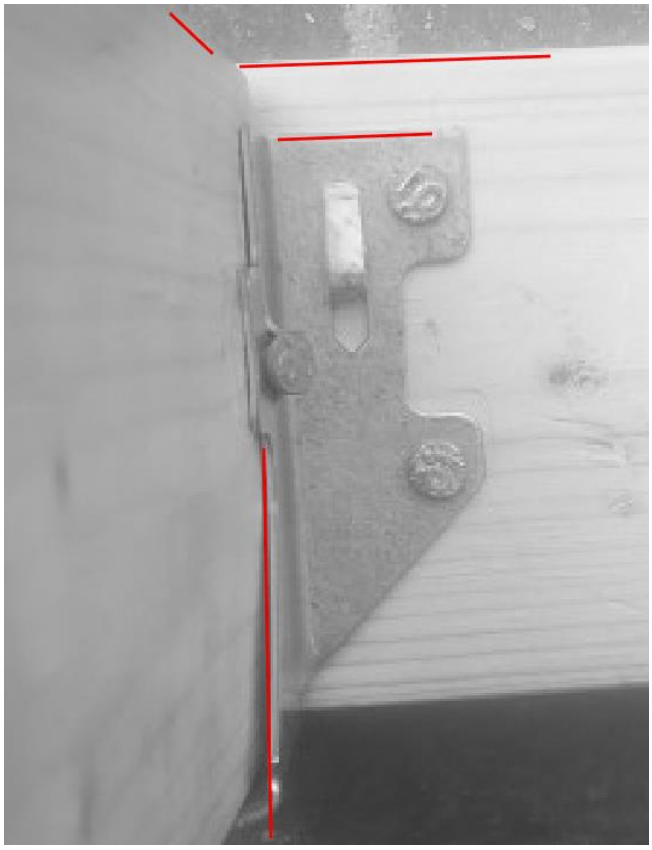


We then program MATLAB to determine commonalities between the two images. MATLAB references a database of previously tested and processed hangers to determine which hanger model it is analyzing.



A Hough Transform is then run on the image. This detects prominent, straight lines in an image. From the results of the Hough Transform, we create a system of lines of best fit around our reference object. These lines describe the positions of the beam, joist, and hanger. Vectorizing these lines allows for the calculation of a dot product between the vectors to obtain the angles between our reference lines. And from these reference lines, we can determine if the hanger/beam/joist interface is installed properly, as least geometrically.





Gantt Chart

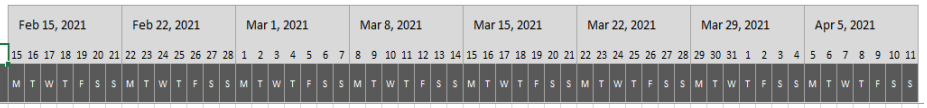
Simpson Strong Tie - Team 8

Christian McCabe, Phillip Koffman, Faisal Alajmi, Mohammed Alali

Project Start: Tue, 1/19/2021

Display Week: 5

TASK	ASSIGNED TO	PROGRESS	START	END
Building Prototype				
Task 1	Research commercial image processing software (Faisal)	100%	2/15/21	2/17/21
Task 2	Request go/no-go images from Evan (Phill)	100%	2/17/21	2/19/21
Task 3	Maintain/improve initial MATLAB prototype software (Team)	100%	1/25/21	3/1/21
Task 4	Put together hanger testing rigs x10 (Phill, Chris)	100%	2/20/21	2/21/21
Task 5	Implement team software onto Raspberry Pi (Team)	100%	2/22/21	2/25/21
Testing Prototype				
Task 1	Load training data into IP program (Chris)	100%	2/15/21	3/1/21
Task 2	Test program with GO hanger setups (Phill, Chris)	100%	2/15/21	3/7/21
Task 3	Improve GO hanger recognition (Team)	100%	3/8/21	3/14/21
Task 4	Test program with NO-GO hanger setup (Phill, Chris)	100%	3/15/21	3/22/21
Task 5	Improve NO-GO hanger recognition (Team)	100%	3/23/21	3/30/21



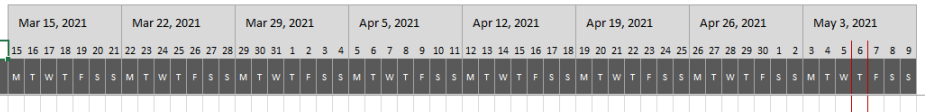
Simpson Strong Tie - Team 8

Christian McCabe, Phillip Koffman, Faisal Alajmi, Mohammed Alali

Project Start: Tue, 1/19/2021

Display Week: 9

TASK	ASSIGNED TO	PROGRESS	START	END
Task 4	Put together hanger testing rigs x10 (Phill, Chris)	100%	2/20/21	2/21/21
Task 5	Implement team software onto Raspberry Pi (Team)	100%	2/22/21	2/25/21
Testing Prototype				
Task 1	Load training data into IP program (Chris)	100%	2/15/21	3/1/21
Task 2	Test program with GO hanger setups (Phill, Chris)	100%	2/15/21	3/7/21
Task 3	Improve GO hanger recognition (Team)	100%	3/8/21	3/14/21
Task 4	Test program with NO-GO hanger setup (Phill, Chris)	100%	3/15/21	3/22/21
Task 5	Improve NO-GO hanger recognition (Team)	100%	3/23/21	3/30/21
Presentation				
Task 1	Prototype complete (Team)	100%	2/15/21	4/5/21
Task 2	Confirm design/implementation with Evan, Mr. Selman (Team)	100%	2/2/21	4/5/21
Task 3	User interface between device and computer (Team)	100%	2/15/21	4/29/21
Task 4	Project demo (Team)	100%	2/15/21	5/4/21



Computer Design Tools

The design tools we used came in the form of programming languages: Python and MATLAB. Because commercially available products met our requirements, we had no need to develop a custom embedded system to run our sensor. We did find that our host machine required minimum specifications to execute our MATLAB algorithm in a timely manner. However, the machine we discovered this with is several years old and is unlikely to be used in the field.

Most of our processing was done on a GPU, which while not necessary, greatly speeds up implementation. The model used was a Tesla K80, compute 3.7 with 2496 CUDA cores.

Patent and Standards Research

There are currently no devices or patents on the market that cater to Simpson's needs. There is however a plethora of information on image processing. Most of these programs and devices focus on recognizing faces but the same techniques of recognizing patterns and contour lines is applicable to identifying our hangers. In fact, it is much easier for an algorithm to recognize a hanger rather than a face.

Our device is one of a kind on the software side. As mentioned, there were already a lot of projects involving our Raspberry single chip board that implement the attachable camera for facial recognition, so we did nothing new in terms of the hardware. The Raspberry Pi and its attachments can legally be used for anything the purchaser wants and our software is unique so this is a fully legal device with respect to patents.

The only standards we followed are the ones outlined in the datasheets for the hangers. Simpson developed these standards themselves from stress testing in their laboratories. We could not cover every parameter of the standards that were provided but we made sure our device detected the relatively more important ones like "Heel Height" and correct nail installation. Our device is inside of a case, so it's protected from ESD which is an IEEE requirement. Since this is a new device there are no established rules for building it.

Proof of Concept

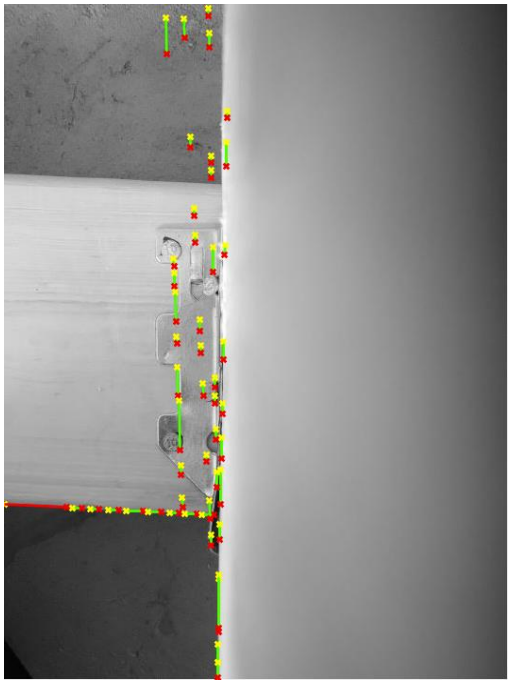
In order to test and verify the accuracy of our system, we must first establish a baseline of what it is measuring. Our sponsor at Simpson Strong-Tie, Mr. Evan Hammel, provided us with a

company installation guide that outlines the basic dimensions and fitment of their joist hangers – these measurements constitute the first control model of our tests. Our image processing program will attempt to recognize these correct measurements and any deviation outside a defined tolerance level specified to us by Mr. Hammel will result in a failed test.

Our second control model involves the joist-beam interface. Ideally, the joist meets the face of the beam at a perpendicular angle. Too much deviation from this orthogonal interface results in a decrease in load-bearing capability of that particular hanger. Again, we rely on the documentation provided by Mr. Hammel as our control data. Our image processing program will attempt to recognize the correct joist-beam fitment of a correctly installed interface, then we will test a fitment outside the specified tolerance level.

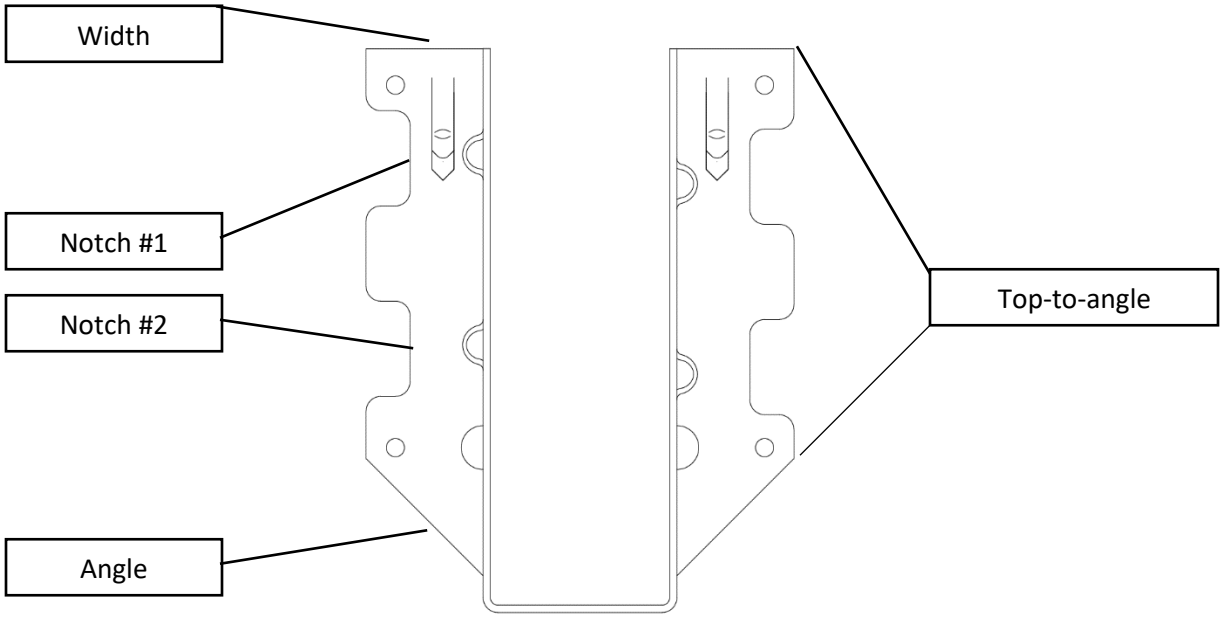
Our final control model involves hanger defects. Obviously, a defective hanger will decrease the load-bearing capability of the joist it bears. However, these defects are often minute and difficult to detect by the naked eye. As such, with a correctly installed joist/beam/hanger as our control, the image processing program will attempt to recognize gross deformities. The deformities we create to test will be subjective in nature but will have different "levels" of deformation. From this test, we will attempt to see at what point our image processing program fails to recognize a deformed hanger.

Test #1

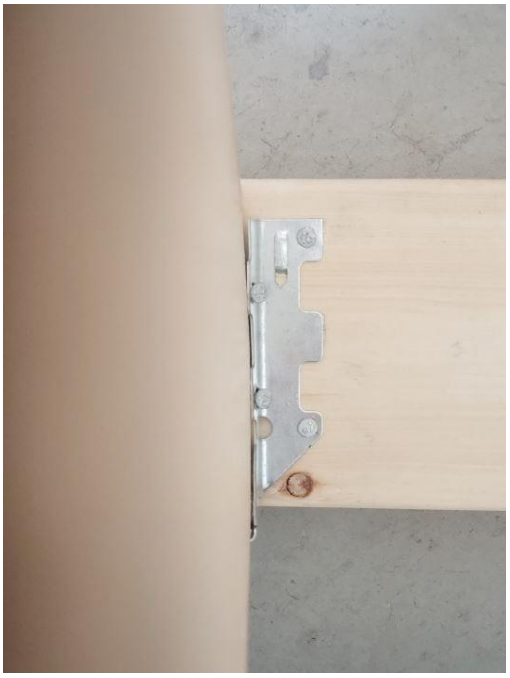


From a control distance of 40 cm from the hanger, the image processor was able to measure known lengths of different segments of the hanger.

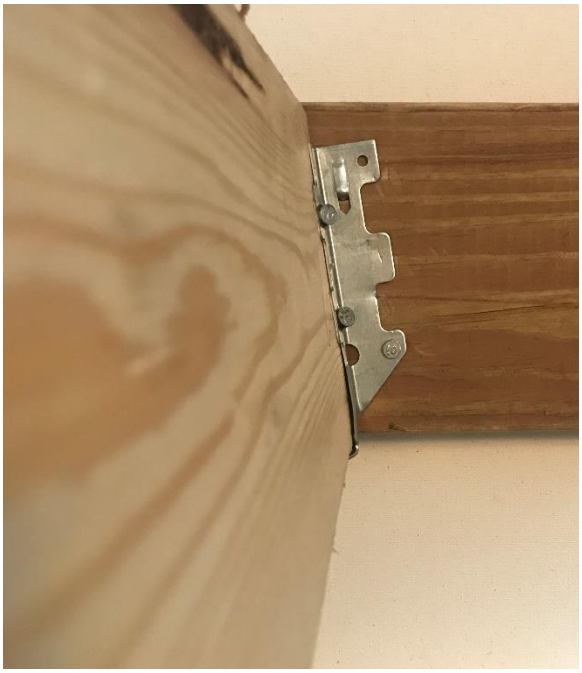
	Control length (cm)	Measured length (cm)	% Error
Hanger width	2.5	2.45	2%
Notch #1	2.0	1.98	1%
Notch #2	2.0	1.98	1%
Angle	3.2	3.11	2.8%
Top-to-angle	9.0	8.92	0.88%



Test #2



Correct installation

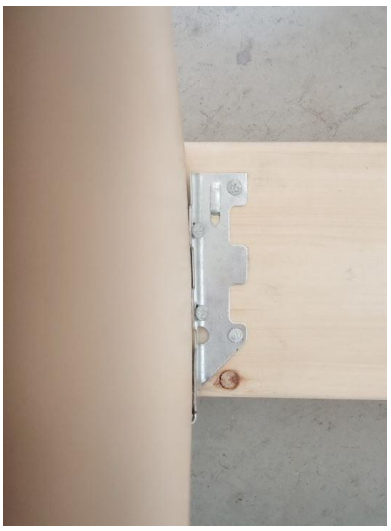


Twisted installation

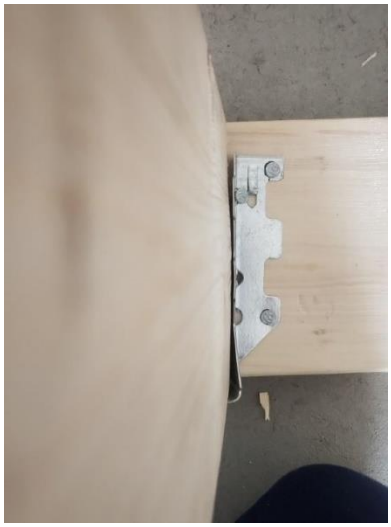
Using the beam as our 0˚ reference, the image processor detected rotation (measured at the top of the hanger) at less than 1˚. However, the program had difficulty choosing the correct feature with which to compare to the reference. Rotational installations above 3˚ were easily detected.

Rotation (degrees)	Detection rate (%)	False positive due to wrong feature detected
0	99	x
0.5	98	x
1	95	x
1.5	90	x
2	90	x
2.5	95	x
3	99	x
3.5	100	
4	100	
5	100	
6	100	
7	100	

Test #3



Correct installation



Level 1

Level 2

Level 3

The image processor was able to consistently detect a Level 2 and Level 3 deformity. However, it detected a Level 1 deformity only 70% of the time. Using a different scan method in the program (right-to-left) may alleviate these errors. The program also had a false positive rate of 1% when testing the control installation. This may be due in part to handheld camera placement.

Level of deformity	Detection rate (%)	False positive
Control	99	x
1	70	
2	97	
3	96	

Project Tasks and Responsibilities

Faisal Alajmi:

Responsible for commercial image processing software research. Established himself as the subject matter expert regarding the type and model of hangers used during testing. Above all, wrote and debugged the Python-based prototype code with Mohammed.

Mohammed Alali:

Curated training image data for the Python-based prototype. Drafted outlines for classwork deliverables. Debugged MATLAB code with Chris and Phill. Most importantly, collaborated equally with Faisal on the Python-based prototype code.

Phillip Koffman:

The primary interface between the team and our stakeholder, Mr. Evan Hammel. Built, organized, and implemented testing aides. Helped write and debug MATLAB-based image processing prototype code with Chris. Helped build and test the image sensor with Chris.

Christian McCabe:

Managed team documents, deliverables, and presentations. Maintained the team's budget and Gantt Chart. Helped write and debug MATLAB-based image processing prototype code with Phill. Built and tested image sensor with Phill.