Groceries Identification and usage estimation with deep learning

Vinod Kumar
vkumar328@gatech.edu

Nitin Chauhan
nitinchauhan@gatech.edu

ABSTRACT

Ceres is a Groceries storage and management appliance. Ceres is designed to identify and measure the groceries stored in it's selves. These capabilities allow several value added applications to be built leveraging the information gathered. The solution that is built inside ceres can be used in inventory management, large scale groceries storage, supermarkets etc. Further the algorithms can be retrained for identification of consumer products or spares. The hardware is generic and the software is specific but can be retrained to be used in other applications. In this project we use the solution in our Groceries management appliance: Ceres.

Some of the applications that can be built on top of our solution is monitoring stock, predicting consumption, automatic ordering, store space management etc.

KEYWORDS

Deep learning, object identification, groceries management, computer vision, load sensing

1. WHAT IS THE PROBLEM?

There are millions of Stock Keeping Units (SKU) in the inventory. An item may come in different brands, composition, flavour, packaging etc. It is hard to design a single deep learning algorithm with continuously growing millions of labels. Product identification using computer vision comes with its inherent challenges owing to the limitations of deep learning like occlusion, skewed object, lighting conditions etc. The deep learning model is usually large making it hard to update from remote locations. The amount of data required to identify an object is large.

If the products are consumed in parts but not as an unit, it is harder to keep track of the consumption. This is an especially useful feature in a consumer application like Groceries management system. A solution that can identify, extract the position of its placement as well as measure products on a shelf would make strides in solving the existing problems we stated.

2. WHY IS THE PROBLEM IMPORTANT?

In the context of large stores, thousands of man hours are spent every day in assessing stock, standing in the queue in the billing counter, searching for unaccounted stock etc. An automated system that identifies and keeps track of products has the potential to cut down on human intervention and associated costs.

In the context of a groceries management appliance in a household, there is an everyday problem of running out of groceries, stocking too many items and leading to expired groceries.

3. SOLUTION

Our solution consists of a software solution that processes captured images from the cameras, performs pre-processing, detects objects in the image, identifies the items, extracts features and matches them with the internal database of the objects. The hardware components include cameras, door switch sensors and weight sensing mat that would provide coordinates, 2D shape and overall weight exerted by the objects placed on it.

The information from the hardware sensors would be fused with the information from the image processing algorithm to map objects on to unique SKU Ids. Once, the object is recognized and measured, several value added services may be built over it as explained in the previous section.

3.1 ARCHITECTURE

Our solution is built on NVidia Jetson Nano Single board computer running Ubuntu 16.04 OS. Since, we have several hardware sensors that interact with the algorithms in real time, we used the Robot Operating System (ROS) to integrate the hardware modules with software. The provisioning of data to the Web UI and Android UI is done with the Web application that resides outside the ROS.

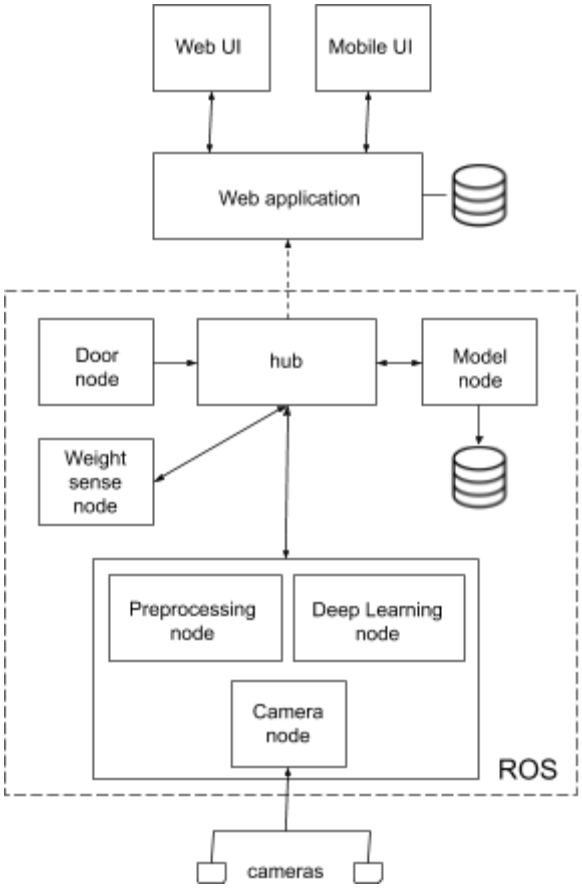

Figure 1: Architecture of Ceres

3.1.1 ROBOT OPERATING SYSTEM (ROS)

ROS Kinect provides packages that are installed over Ubuntu OS. Ubuntu is not designed for a Real time solution. The processing paradigm is not optimal for embedded system performance. The ROS packages simulate an embedded system environment on Ubuntu and to an extent provide real time performance. ROS also provides design patterns that are tailored for hardware software interaction. ROS is structured in a graph form with several nodes interacting through messages. We predominantly use Publisher-subscriber and Action Client-Server patterns in our solution. We have six nodes and each of these contain logic to perform a particular task. They process and either broadcast or selectively pass on the information to the other nodes as configured.

3.1.1.1 DOOR NODE

The door of the shelf is attached to electromagnetic switches that close or open when the door is closed or opened. The Door node listens to the GPIO on the hardware for the signal from the switch and broadcasts when its state changes. GPIO on Jetson Nano is a hardware pin that is mapped to a file in the OS. When the signal is HIGH on the GPIO pin, the content of the corresponding file that maps the GPIO pin changes, for example from string '0' to '1'. We are interested in the OFF to ON state that indicates that the door was closed.

The door node is a 'publisher'. When the door is opened and subsequently closed, it is assumed that an item in the shelf was used. A message is generated and published on the 'door_activity' topic. The door node is in a continuous loop listening to the switch status until ROS is shut down.

3.1.1.2 HUB NODE

The Hub node is as the name hints a hub where the logic for routing the messages and the execution is written. The hub node is an Action client. Action client-server is a design pattern used when a client requests an action to the server that takes several seconds to execute. The client does not usually wait for the execution to complete on the server and can continue with other executions until the response from the server arrives. The client contains 'feedback_callback' to listen to intermediate feedback from the server during the execution. It also has a 'done_callback' that collects the results from the server.

The hub client also subscribes to the 'door_activity' topic. When a message is published by the door node, the hub client raises a request to the camera node to capture images of the shelf. When the capture server completes the capturing, the hub client raises a request to the detect node to perform pre-processing, object detection and identification. The hub node passes the processed data to the model node to store in the database. It then posts the data to the web server.

3.1.1.3 CAMERA NODE

Camera node is an Action server. In the present state of our prototype, we have two cameras connected to the Jetson Nano board. Since, Jetson Nano has only one MIPI camera interface, we use a camera multiplexer. The camera server is responsible for addressing a particular camera through GPIO pins and executing the 'gstreamer_pipeline' provided by NVidia to capture an Image of 2464 x 3280. The image is stored in a shared folder location known to all the nodes that need access to it. Since, two

cameras are required to capture the entire width of a shelf, the image name indicates the shelf id and if the image is of the right side or the left side of that shelf.
The cameras are placed on the cabinet door facing inward. Since, the items are kept in close proximity to the camera, a wide angle camera was chosen. The wide angle distorts the image and reduces the clarity. The pre-processing steps are contained in the camera node. The preprocessing steps are discussed in section 3.2.1. After the pre-processing, the hub client receives the image location in the 'done_callback' function.

3.1.1.4 DETECT NODE

The detect node is an Action server, that is responsible for detection and identifying the items on the self from the pre-processed image. Once the item is identified, the properties of the item are returned to the hub client 'done_callback' function. The details of the algorithm are mentioned in section 3.2.2.

3.1.1.5 WEIGHT SENSOR NODE

The Weight sensor node is an Application server that is responsible for scanning through the weight matrix and capturing the pressure exerted on each square cell of the 16x16 matrix. The pressures are denoted as values between 0 and 200 in a matrix form. The pressure map is converted and processed as an image. The image map is returned to the 'done_callback' function of the hub client.

3.1.1.6 MODEL NODE

The properties of the items like starting weight, current weight, purchase_date, SKU, brand, category etc are stored as python objects in the Model node. The properties of the objects are manipulated when the weight of the respective item changes. When the items on the shelves are identified after every scan, the identified item properties are compared with the existing objects. If there is an object that does not match the properties of any items in the current scan, it is removed. If there is a new item identified, an object is added along with the item properties. We have not yet implemented permanent storage of these objects in the database. However, when the model node completes the object creation, it returns hub node the properties of all the items in the shelf in a JSON format. The hub node sends an HTTP POST to the Web server. The server extracts each individual item and stores it in MongoDB to be served to the UI clients.

3.1.2 WEB APPLICATION

The server is created with Node JS and the web client is written in EJS. The mobile app has been written using React Native with the use of some external libraries. Both web client and mobile client have the ability to asynchronously update their UI based on changes in the model at the server. Section 3.5 discusses more on the implementation.

3.2 IMAGE PROCESSING

Image processing entails removing distortions with calibration, merging images from two cameras per shelf, object detection and object identification.

3.2.1 PRE-PROCESSING

The captured image was calibrated to remove distortion using the openCV functions. We remove the radial and tangential distortion by using the chess board pattern provided by OpenCV. Several images of a chess board 8x8 grid pattern are captured from different angles. The 'findChessboardCorners' function returns the corner points. The function 'calibrateCamera' generates a calibration matrix that can be used with function 'undistort' to get a reasonably undistorted image.

Figure 2: Orignal, calibrated and cropped image

After undistortion, the images are warped at the edges making the outer portion unusable. So, we crop the edges and retain only the center portion of size 1376x1376 pixels.

3.2.2 DEEP LEARNING

We have experimented with various solution architectures. We found hierarchical architecture where properties of the objects are extracted first and identity narrowed down based on these properties. next to work better. We used a few common groceries in our experiments. We trained a small sized identifier specialised for a particular grocery item e.g Cereal identifier, Pasta Identifier, Tin Can identifier etc. All the objects that are detected in the detection

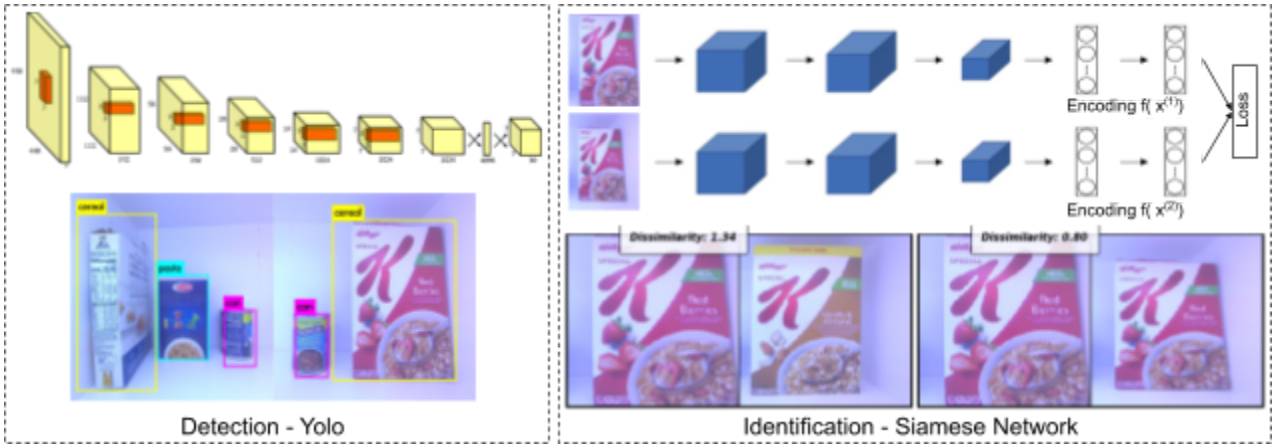

Figure 3: Yolo, Siamese network and results

modules are individually passed to the respective micro identifier networks to further identify the SKUs.

3.2.2.1 OBJECT DETECTION

We used Yolo architecture[1] for object detection since it is comparatively lightweight and allows for a wide range of object classes. Yolo determines the location of the objects as well as the classification of those objects present in the image in a single neural network. Yolo considers object detection as a single regression problem, from image pixels to bounding box coordinates and class probabilities.

Yolo divides the image into SxS (13x13) grid. for each cell in the grid, it produces B (5) bounding boxes. Each bounding box is responsible for determining if it contains an object or not. And if it contains an object, it also spits out a confidence score as to how confident it is. for each bounding box, Yolo calculates the class probabilities for each bounding box object. Five numbers represent each bounding box. The X and Y coordinate of the object wrt to the cell it is in, The height and width (wrt the entire image) and the confidence score (if it contains the object or not). For a grid cell, we predict C class probabilities (for the C classes we are predicting for). The confidence prediction represents the IOU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, Pr(Classi |Object). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B. The score for each cell-bounding box combination is given by:

$$Pr(Classi|Object)*Pr(Object)*IOU = Pr(Class)*IOU$$

These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object. The key aspects of the loss functions are the differential weight for confidence predictions from boxes that contain objects and boxes that don't contain objects during training and we predict the square root of the bounding box width and height to penalize error in small object and large object differently.

A CNN architecture similar to GoogLeNet is used. The Inception modules are replaced with 1x1 reduction layers followed by 3x3 convolution layers.

The CNN had 24 convolutional layers followed by 2 FC layers. We used the pre-trained Yolo V3 and re-trained the last two layers with our dataset. The final layer predicts both the bounding box and the class. These values are normalized to fall between 0 and 1.

3.2.2.2 OBJECT IDENTIFICATION

We use the siamese network[2][3][9] for object identification. The architecture of our model consists of two identical neural networks. During training each network takes an image as an input. The network generates feature vectors (embeddings) $f(x^{(1)})$ $f(x^{(2)})$, and determines the similarity of both images $x^{(1)}$ and $x^{(1)}$ through a loss function. We would like the dissimilarity to be high for images of different class (different SKUs in our case) and small for images of the same class.

The objective of the siamese architecture is not to classify input images, but to differentiate between them. A triplet loss function takes as input a positive example P, a negative example N and an anchor A.

The loss can be formalized as:

$$L(A,P,N) = max(\ \| f(A) - f(P) \|^2 - \| f(A) - f(N) \|^2, 0\ )$$

We want the difference of the euclidean distances between $f(A) - f(P)$ and $f(A) - f(N)$ to be ≤ 0. if this is greater than zero then we take the max so we get a positive loss. The overall cost function for our neural network can be a sum over a training set of these individual losses on different triplets:

$$J = \sum_{i=1}^{m} h(A^{(i)}, P^{(i)}, N^{(i)})$$

The metrics would include accuracy, euclidean distance and the weighted L1-norm between the two feature vectors of the pair input.

We collect sample images from several SKU's. We generate encodings for all these images and store it. When we get an input from the object detector, we encode the image and compare it with encodings of all SKU samples. In figure 3 we see that dissimilarity between the first pair of images is 1.34 and the second pair is 0.80. The SKU corresponding to the lowest dissimilarity is believed to be the identity of the input image.

For the Cereal identifier network, We created a dataset of 300 samples of 20 unique SKU of cereals. The size of the dataset is too small to be used in real world applications but is sufficient to examine the efficacy. In our experiments we noticed an accuracy of between 85-90% varying based on image position and orientation with respect to the camera. A thorough dataset with large number of samples in several orientations would significantly increase the accuracy.

3.3 WEIGHT SENSING

The key material of the weight sensing matrix is the Velostat [7]; a polymeric foil ingrained with carbon to make it electrically conductive.

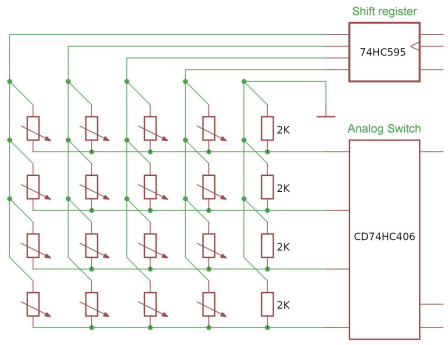

Figure 4: Sensor matrix block diagram

Applying force on this material decreases its resistance measurably. The sensing matrix is a three layered sheet with Velostat sandwiched between two sheets of electrodes.

Each point where the electrodes touch the velostat layer can be used as a pressure dependent voltage dividers. Every pressure-dependent voltage divider in our matrix consists of a variable resistor (sensor) and a fixed resistor. The sensors are arranged in columns and rows of 16. The high sides of all sensors in a column are connected to one output of a shift register. The low sides of all sensors in a row are connected to a fixed ground-resistor and an analog switch input. A MCU controls the shift register and the analog switch in a common scanning strategy: Rows are connected one at a time to a MCU analog input pin by the analog switch and their voltage is measured while columns are energized one at a time by the shift register. This way we are able to measure all 256 sensor voltages with only one ADC and a few digital pins. We use the ADC on the MCU to convert the analog voltage levels to digital values. The hardware specifics and the flow of data is explained in section 4.

The pressure resistance curve is however not linear across the pressure range. The slope is roughly linear between 100g and 500g but flattens thereafter.

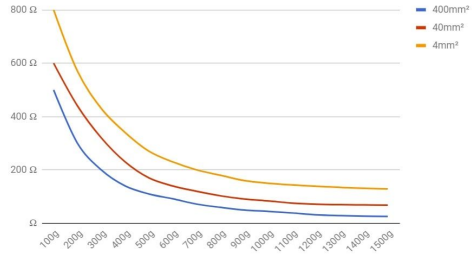

Figure 5: Load Vs Resistance curve

The choice of 2K ohm resistance as a fixed register translates to voltages between 0-5V and digital values of values between 0 and 255.

When a box is placed on the sensing matrix, it forms a rectangular area of pressure on the sensor matrix. Most of the objects do not exert equal force across the surface area leading to a rough rectangular blob.

Figure 6: Raw value between 0-255 and select mask

Figure 6 shows raw values on the left and a threshold mask to extract the pressure values is shown on the right. The mask is applied on the raw values to extract only significant pressure values which are then summed up. The summed pressure is then mapped to an empirically plotted weight pressure curve. With multiple objects placed on the sensing matrix, each blob is treated as a seperate object. So, it is important for the blob to cover the entire object base and not in smaller blobs.

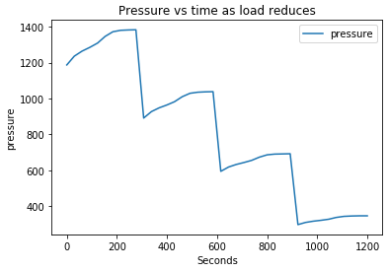

Figure 7: Pressure changes when load reduced gradually

One of the limitations of the pressure sensor is that it takes about 3 minutes for the pressure to stabilize. Figure 7 shows the pressure curve when a 500g weight was reduced by 100g four times every 5 minutes. At 300 seconds, the load placed on the matrix is 400g but the pressure starts to increase and reaches saturation of 1038 after 500 seconds.

3.4 PREDICTION

Since we do not have a product mature enough to perform real life experiments, we simulated data based on general consumption patterns. We simulated the weight of fast consumables like cereals as well slow consumables like coffee across days. We fit the curves with Sklearn 'PolynomialFeatures' and 'LinearRegression' functions. Since, this is a simulated data, testing would not yield any meaningful results.

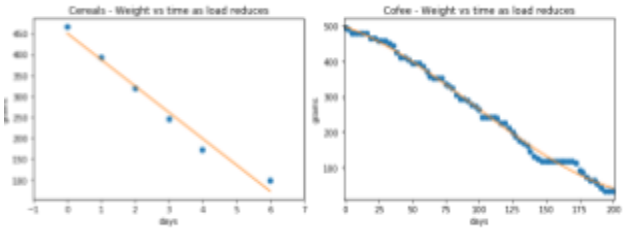

Figure 8: Simulated usage curve - weight vs days

3.5 USER INTERFACE

The UI has a Client-Side and Server-Side. EJS (Embedded JavaScript) is used as the template language on the client side which generates HTML markup along with CSS 3, Bootstrap 4, JavaScript, & jQuery. Pusher has been used to receive certain real-time events from the server. Mobile app has been written using React Native along with external libraries.

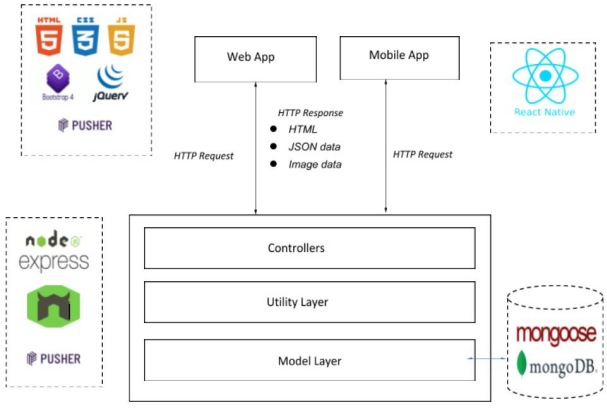

Figure 9: Client Server Architecture

A request is sent to the home route ("/") which is handled by the homeController. The homeController queries the CabinetShelf database and sends the information of all the items as a JSON response. This response is received on the client-side and is used to populate the images and the relevant information of each item on each of the images. This information can be viewed on the web-app by hovering the cursor over each item. On the mobile app, the same information can be observed by tapping on the respective items.

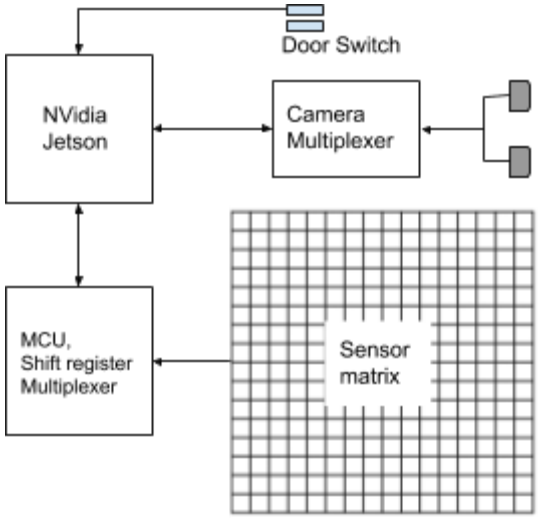

Figure 10: Hardware Architecture

4. HARDWARE

NVidia Jetson has one MIPI CSI camera interface. We needed to connect 8 cameras. So, we use a Camera multiplexer. Camera Multiplexer is a board

that can address one camera at a time through I2C and gather image data. The sensor matrix has a dedicated controller board that has Shift registers to togger each electrode, a multiplexer to read voltage and direct it to the MCU. The ADC on the MCU converts to digital information and sends it through USB port of the NVidia Jetson computer.

5. GOALS:

The goals we set at the beginning of the project and the progress are as below:

75% Goals:

- Hardware setup
- ROS pipeline
- Deep Learning model (Baseline)
- Deep Learning model (Accurate)

Progress and lessons learnt:
Hardware setup was more complex and time consuming than we anticipated. The choice of the camera and lense with properties that suit our application like focal length, sensor size, resolution, communication interface, cost etc were critical. We tried several cameras before arriving at a reasonable choice. Arriving at ways to connect multiple cameras with multiplexers was another major hurdle. The design and coding of the ROS pipeline with the right architecture to facilitate flow of data went smooth. Before using the images from the camera, we had to heavily pre-process the image due to major distortion with wide angle lenses. This step requires significant effort. Deep learning model architecture and baseline was developed without major hassle since we used different standard architectures. However, the accuracy of the model depended entirely on the data. We had anticipated support from the entrepreneurship cell in opening doors to outlets for collecting data. However, we were not able to get any outlets to permit us to gather data in the form of images of groceries. We used the images from the internet and performed common data augmentation. The images from the internet were captured in only front view or side view. The items are rarely placed on the shelf in a way that matches these standard product images. We gathered some sample products and captured images from different orientations. This led to higher accuracy but was overfitting for unseen products. The model is only as good as the quantity and quality of data we use to train. This leaves room for gathering a lot more data and training.

100% Goals:

- Android application
- Integration

We were able to develop a basic android application that shows the image of the shelves and properties of each item. We were able to integrate all the modules and form a pipeline from end to end.

125% Goals:

- Web application

We were able to develop a web version of the android application for us to view and debug. We also wanted to create an online profile for each cabinet and sync the cabinet information on to the online database but this was not achieved due to lack of time and the task being of lower priority.

6. REFERENCES:

[1] You Only Look Once: Unified, Real-Time Object Detection by Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi (2015)
[2] Siamese Neural Networks for One-shot Image Recognition, Gregory Koch et. al
[3] Improving Siamese Networks for One Shot Learning using Kernel Based Activation functions, Shruti Jadon and Aditya Arcot Srinivasan
[4] Blind Geometric Distortion Correction on Images Through Deep Learning by Xiaoyu Li et. al
[5] Distortion Robust Image Classification using Deep Convolutional Neural Network with Discrete Cosine Transform by Md Tahmid Hossain et al
[6] A Hierarchical Grocery Store Image Dataset with Visual and Semantic Labels by Marcus Klasson et al
[7] Low-Cost Pressure Sensor Matrix Using Velostat, S.S. Suprapto et.al
[8] Locating Objects Without Bounding Boxes, Javier Ribera et al
[9] Siamese Network Features for Image Matching, aroslav Melekhov et al
[10] Large scale classification in deep neural network with Label Mapping, Qizhi Zhang
[11] A deep learning pipeline for product recognition on store shelves, Alessio Tonioni et al.