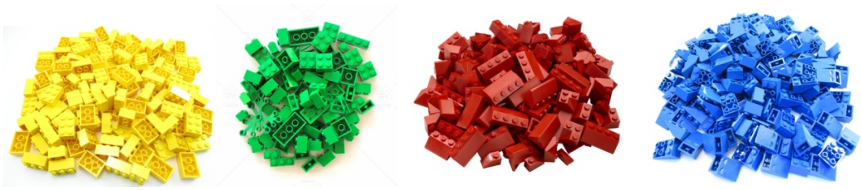


The Bricks Sorter

Final Capstone Project Report

Ikram Ikramullah (ikrambabai@gmail.com)

Table of Contents

Project Overview

It is fascinating to watch a child's growth over time as they meet various milestones gaining proficiency in little skills that we all cheer at those early stages. From uttering the first words to crawling and walking; these are all great events that every parent cheer about. As an infant turn into a toddler and further on, the child goes through some very crucial brain development stages. Research has shown that most of the brain development happens during the early stages of life. It is therefore crucial for parents (and teachers) to consider new innovative and fun ways of engaging the little ones in activities that help them grow their brains to their full potential.

One of the most fascinating of these learning techniques are the called the Legos (more generic name is Building Blocks). These Lego collections play a crucial role in the development of problem-solving and STEM skills. These building blocks train the kids on how to train their brains on building objects they observe the real-world around them in an organized fashion by combining smaller blocks and pieces together. From a simple toy car to a rather 'sophisticated' spaceship that kids love to make, these bricks and building blocks play crucial role in providing both fun and extremely educational brain-development experience.

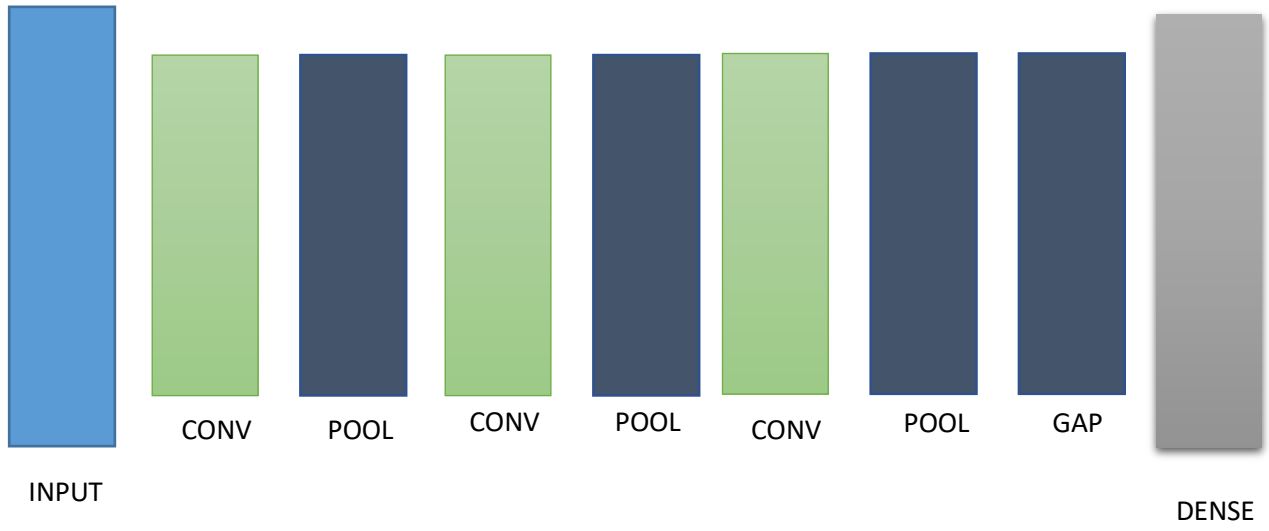
However, this learning experience can quickly go south if in a considerably large collection, those various of bricks, the plates, the tiles and many other smaller and pointy pieces either go missing, get mixed up with each other (and the other toys in the "toybox"). Worse yet, imagine those pointy objects coming under your feet – that whole 'learning experience' can easily turn into something else. Therefore, organizing these pieces and keeping them in the right containers is a common house-hold challenge that needs a good fix. So, here I am with the solution.

In this project, I've created a Deep Machine Learning model trained to recognize various Lego pieces architecture around Convolutional Neural Network (CNN). The bricks, tiles, plates and many other pieces are each separately recognizable by the model. Not just those higher classes, but further these can be put into more fine-grained classes by the standard sizing species (e.g. 1x2, 2x2, 4x4 bricks, tiles and plates). The model can be put on a raspberry pi (for example) connected to it camera module overlooking the passing Lego pieces with further signals a step motor roaming a slide to the correct container for the intended piece.

Solution Description

The idea is that by the time the game of Lego is over, the bricks and all other pieces go back to their right places (jar or whatever way is deemed workable for the family) so that the next attempt on the Lego experience is educational and stress-free. To achieve this, I have trained a machine-learning model using CNN (Convolutional Neural Network) by feeding it various classes of bricks

in training, testing and validation sets. The neural network architecture is a powerful technique that allows for various epochs (iterations) and well-defined model layers for extracting out key features from an image. The idea is to divide the set of images into training sets which will never override with the testing and validation sets. The neural network reads images in the form of 4D tensors fed to its input layers which then passes is over to a series of convolutional and pooling layers before being passed to a GAP. Following is the high-level of the my neural network architecture.



Input Layer:
Responsible for learning presenting the input images to the convolutional layers in the form or a readable 4D 'tensor'. More details on this in the architectural discussion.

Convolutional Layer:
Convolutional layers are responsible for summarizing the features in the image coming in from the input layer. A convolution is the application of a filter to an input (such as in image) that results in 'activation'. If applied repeatedly (like I've chosen above), this filter or layer produces a map of 'activations' called feature map. The feature maps detect the locations and strengths of a detected feature in the input such as images.

Pooling Layer:
The features in the form or a feature map output by the convolutional layer has one problem – the features are very sensitive to the locations of the features in the input image. Before a second layer of convolution is applied, these features need to be down sampled – The pooling layers provide just data. This is known as 'local translation invariance'. See the application architectural section for some more details.

Dense Layer:
A final 'dense' layer reads from the previous layer to produce one of the labels (categories) for our bricks. The SoftMax function throws out a value between 0 and 1 in this layer that depicts one of the categories the prediction is made against.

Benchmark Model

There isn't any existing model on this class that I can compare mine with. However, the accuracy of my model should be better than a naïve model. My samples in each category among the 11 categories of ['1_by_1', '1_by_2', '1_by_2_triange', '1_by_3', '1_by_4', '1_by_6', '1_by_8', '2_by_2', '2_by_3', '2_by_4', '2_by_6']. These are the categories I have chosen see details below. This means the accuracy of my model should be better than 9% for a naïve classifier with 11 classes.

Evaluation Metrices

The best evaluation matrix for this project would be to see the accuracy of the output of the machine that puts an object in its respective class.

Dataset and Input

For each of the categories above, I took squared pictures in at least four angles of each of the distinct colored piece. For example, see Figure 2 on various angles for a 2x6 brick. The total images I collected are close to a 1000 with approximately 90 in each of the 12 categories.

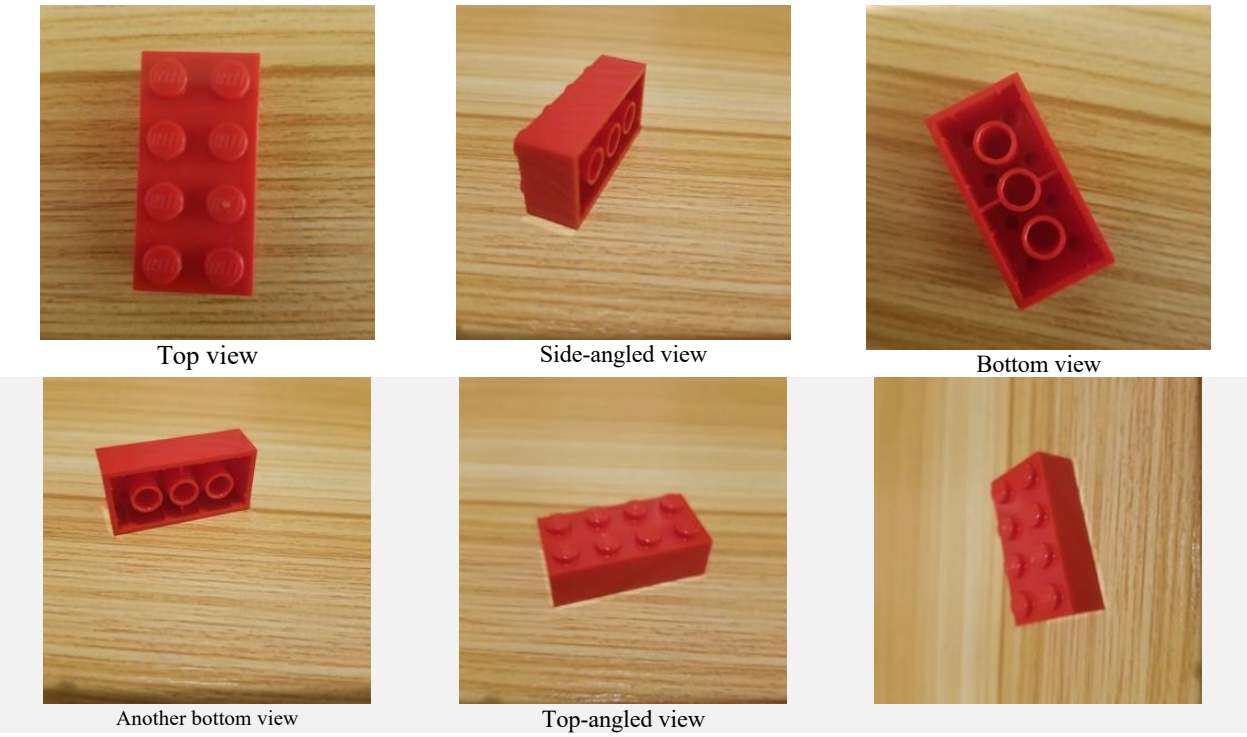


Figure 2: Depicting 5 angles a CNN should see to make features out for a 2x4 brick class

The input to my CNN Trained Brick Sorter modal will be an image of a Lego piece. The it needs to sort. Another input will be to let the application know the type of sorting needed – by color,

block-type or a combination of the two. The block-type can be one of the three: a brick, a plate or a tile.

Brick: Brick is a regular build block and is the most basic type of building block. It measures certain dimensions. It can get connected from the top as well as from the bottom with another component.

Plate: A plate is same as a brick except it is much thinner – 3 plates make up one brick in height. Just like a brick, a plate can also connect to components to its top and bottom.

Tile: Tile is same as a plate except it can only get connected from the bottom – the top side of a tile is a smooth surface having not studs. Usually these type of building blocks are used for making up of the smooth surfaces such as table tops etc.

There are other specialized building blocks too of course, but these three types of building blocks make up for most of the collection of the building blocks and that's where I will concentrate more on.

Note: the size of a bricks is measured by the number of studs that fit on its short side x the number of studs that fit on its longer side. A tile, though doesn't have any studs on it, is also measure using the same rule. i.e. A plate with 2 studs on its shorter side and 3 on the longer side will be measured as 2x3 plate. A tile of the same size (though it won't have any studs) will also be measured the same way: 2x3 tile.

The images aren't yet ready – but I have the blocks which I will take the pictures of, convert to an acceptable dimension and feed in for the training.

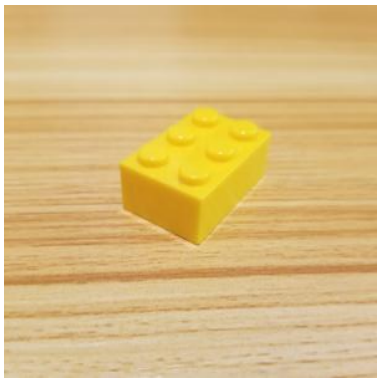
The input to the training will therefore be the following.

1. Bricks
    a. 1x1 Bricks (4 angles)
    b. 1x2 Bricks (4 angles)
    c. 2x3 Bricks (4 angles)
    d. And some more sizes and colors (each in 4 angles)
2. Plates
    a. 1x1 Plates (4 angles)
    b. 1x2 Plates (4 angles)
    c. 2x3 Plates (4 angles)
    d. And some more sizes and colors (each in 4 angles)
3. Tiles
    a. 1x1 Tiles (4 angles)
    b. 1x2 Tiles (4 angles)
    c. 2x3 Tiles (4 angles)
    d. And some more sizes and colors (each in 4 angles)

The input to our learning model to get trained on to make those fascinating decisions will be many raw images of various colors of those bricks / Legos – each brick will be captured in various angles (normal position, turned sideways, turned upside down and so on) for the machine to properly learn.

Legos Basic Blocks Classifications

Initial classification:
Unlike I had thought at the time of starting the project proposal where considered the major pieces to be the bricks (the most common Lego peace which with other pieces both from the top and the bottom), the plates (1/3 the size of brick – like bricks, they can connect from top as well as bottom) and the tiles (same height as tiles but can only connect from the bottom), turned out that tiles in a typical set or collection are just too few. Most of the classes were actually made up from bricks and plates in a rather more size-based classifications. See figure 1 below.



Brick

Plate

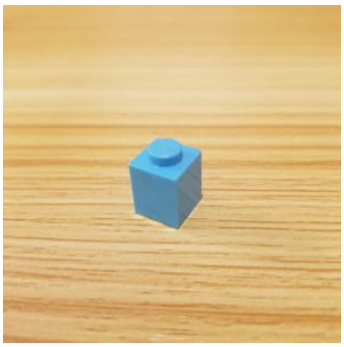
Tile

As you can see, 3 plates (or tiles) make up one brick in height while a tile doesn't have the studs on its top so it cannot connect to other pieces from there.

A more realistic classification:
What should be a sensible ordering if our data sets aren't really balanced in a realistic collection in a household or a brick shop? What are the actual or realistic classes that would bring maxiumum order to the collection? I poured out my collection of bricks (and ordered a new set of Lego Classic

[2] as well) – and here were the typical classes make up for the chaos when mixed up. Note, there are other classes too like tires, base plates and more specialized pieces. But they do not add to the complexity or mix-ups as they are very easy to spot.

Follow are the categories I found most common (and best candidates for sorting). So instead of doing the bricks/plate/tile classification, I went ahead and did more challenging and detailed classification of all sizes.

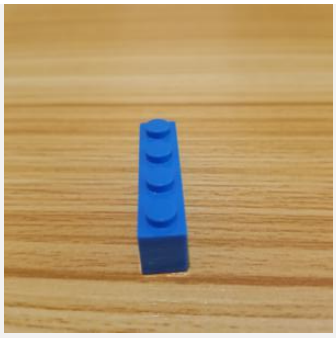


1x1 Bricks

1x2 Bricks (Triangles)

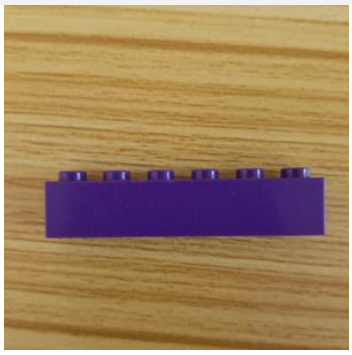
1x2 Bricks

1 x 3 Brick

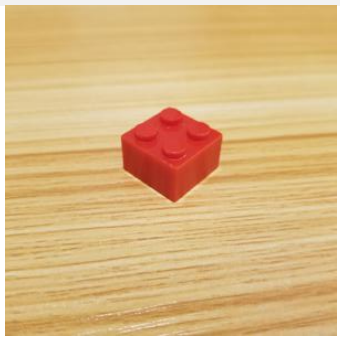


1 x 4 Brick

1 x 4 Brick

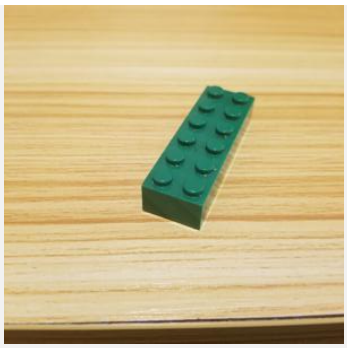


1 x 6 Brick

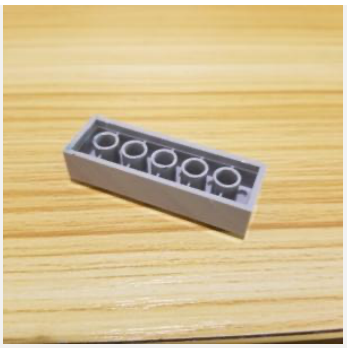


2 x 2 Brick

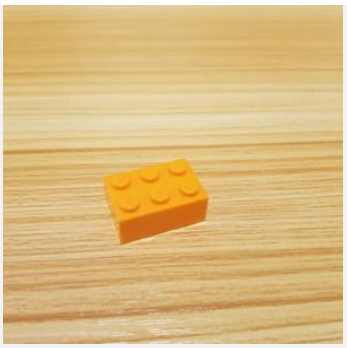
2 x 4 Plate



2 x 6 Brick



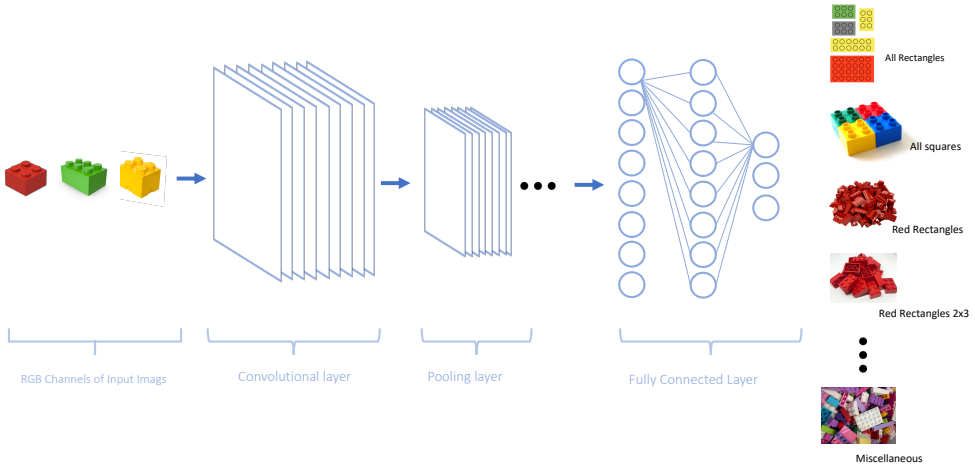
2 x 6 Plate



2 x 3 Brick

Algorithms and the Training Architecture:

Following is the design of my CNN. As you can see, I have used three sets of convolution + max-pool layers followed by a flattening, and a dropout layer. Finally, the dense layer outputs the one of the category of the bricks through SoftMax activation.



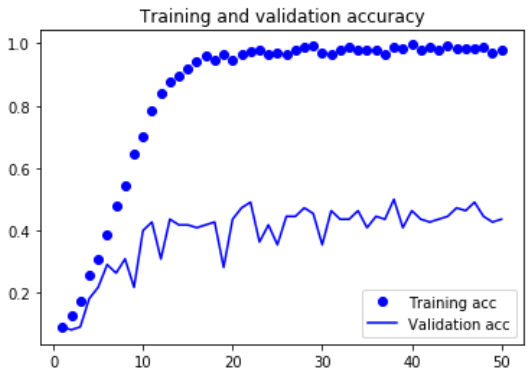
Here is the model's summary.

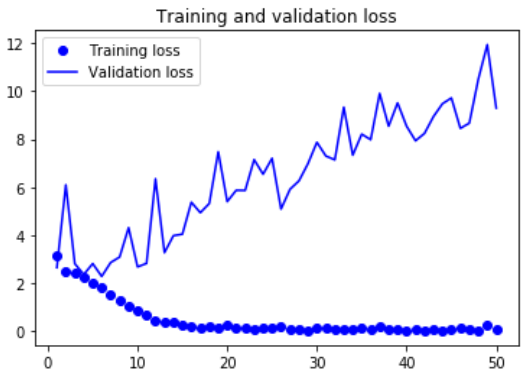
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 64)        0
```

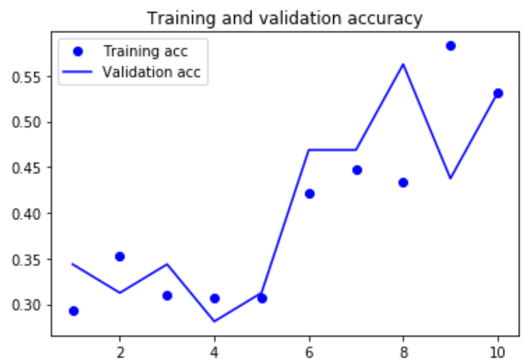
```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 18432)             0
_________________________________________________________________
dropout (Dropout)            (None, 18432)             0
_________________________________________________________________
dense (Dense)                (None, 212)               3907796
_________________________________________________________________
dense_1 (Dense)              (None, 133)               28329
=================================================================
Total params: 4,176,957
Trainable params: 4,176,957
Non-trainable params: 0
```

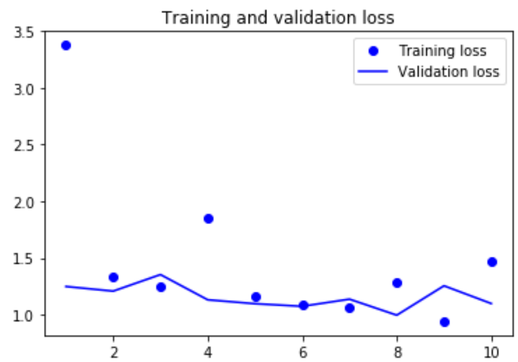
There are about 4million params all of which are trainable. I had setup an AWS xlarge instance to train. An epoch of 200 took about two hours to get completely trained.

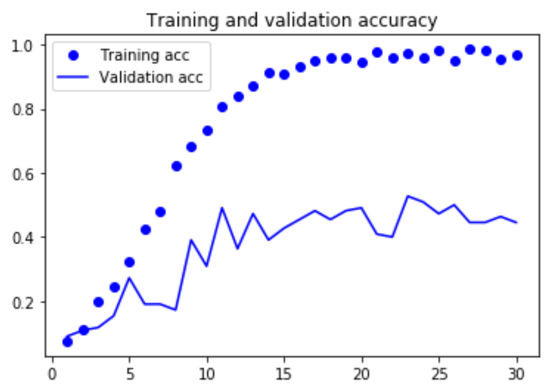
Here are some visuals of the various epoch I used to train the mode. Notice the difference between the smaller epoch sizes and bigger. Bigger epoch sizes really made a difference in better accuracy for the model.

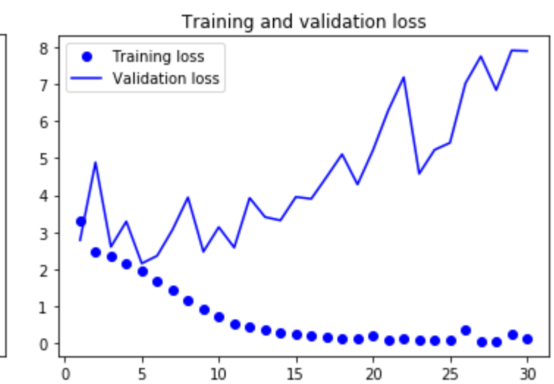


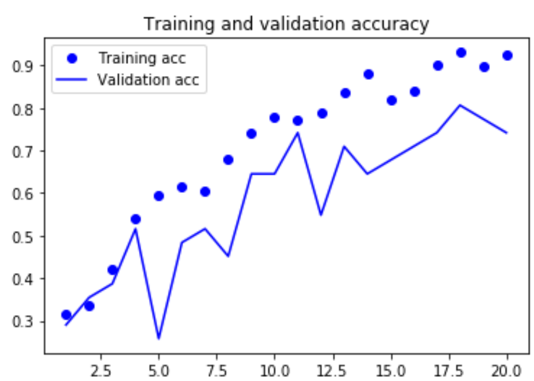


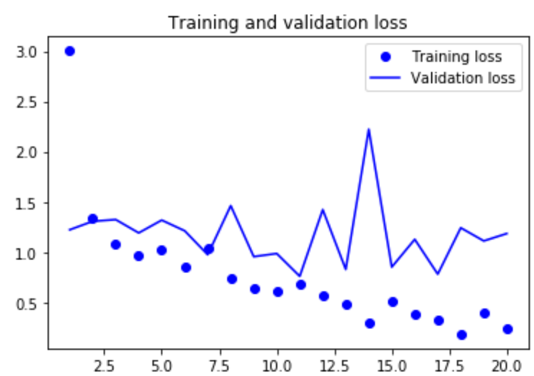

Training and validation accuracy
Training acc
Validation acc

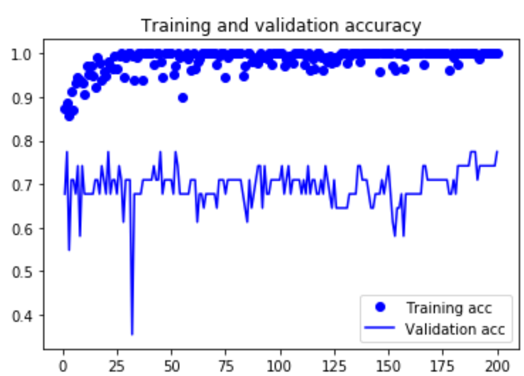

Training and validation loss
Training loss
Validation loss

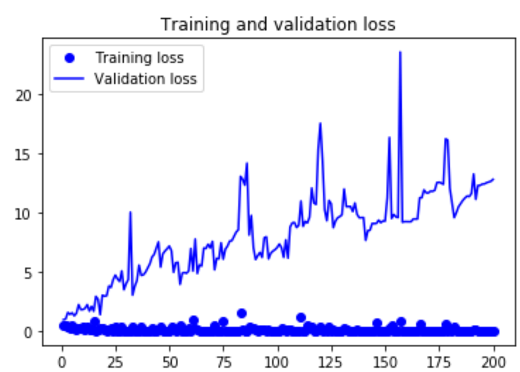

Training and validation accuracy
Training acc
Validation acc


Training and validation loss
Training loss
Validation loss


Training and validation accuracy
Training acc
Validation acc

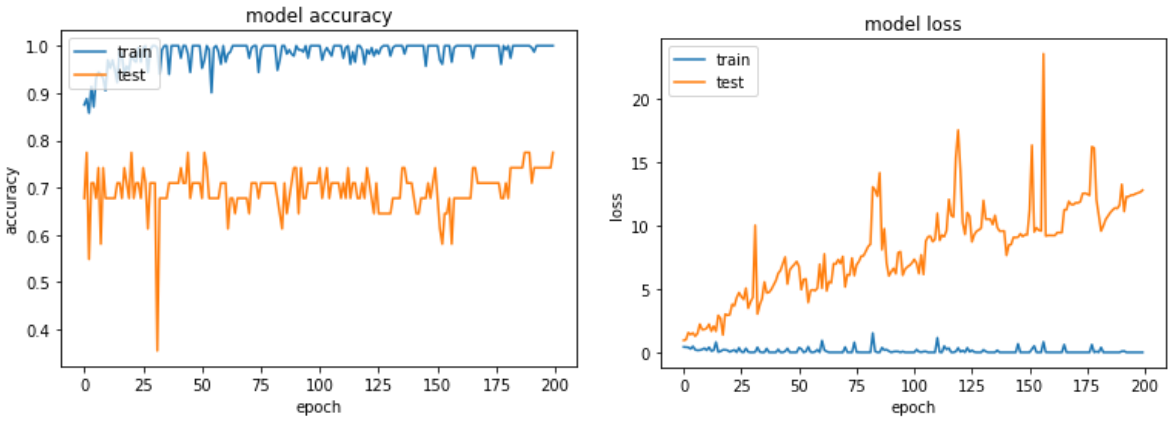

Training and validation loss
Training loss
Validation loss


Training and validation accuracy
Training acc
Validation acc


Training and validation loss
Training loss
Validation loss

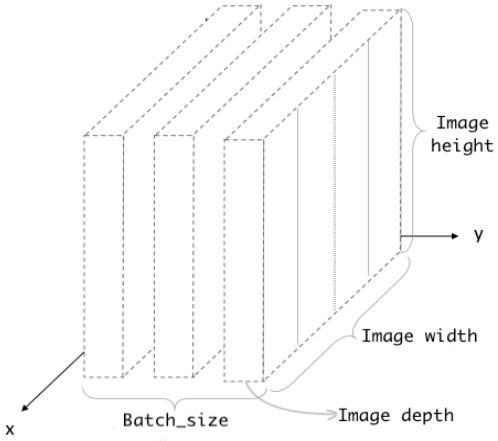
Model Accuracy

Follow are the plots I have completed in the notebook. The following two graphs show model's accuracy and the loss. In the accuracy image, we can see that the training accuracy is much higher than the testing accuracy. This 200-size epoch ran for about 1 hour where it tried to improve upon the previous epochs parameters, but after a while, there was no more learning possible.



Algorithm Parameters and Model Justification

The algorithm is compiled with various parameters which I changed periodically to reach to a certain point of accuracy. The test accuracy was in the 60s with the initial settings. The best I could reach with changes in the parameters discussed below was 77% (a recompilation and training now has it around 75% which is submitted in the html download of the notebook).



Before we go to the algorithm parameters is is important to understand how the input is fed into the input and consecutive layers of the model. The above figures shows the 4D tensor that goes into for the training through the CNN. The batch size it the number of files / imags that should are

sent to the network for see in one batch. The image itself is nothing but 2D array of numbers representing the pixels (height and width). Image depth is the number of channels the image is holding. For example for RGB, the value is 3 and for grayscale it is 1.

Following are the key parameters in the algorithm used which did improve or change the testing accuracy of the application.

1. Epoch: This greatly improved my accuracy. Initially with 20 Epochs, I was kinds of discouraged by the level of learning and the accuracy results. I then went on to 200 on AWS xlarge instance. It took time but that's where things getting improved the most against this parameter.

2. Batch size: The batch size didn't do much except the training time differences on various sizes of the gpu machine I had been using.

3. Kernel Size: Kernels of sizes (2,2) and (5, 5) in intermediate and ending Conv2D layers gave me bad results of the test accuracy. The best was (3,3). Kernel size is the part of the picture that the model / filters are asked to looked at once point in time. Obviously too big or too small of a window will not be a good idea (3,3) fit ok for me.

4. Optimizer: While the "adam" and "rmsprop" gave very similar results, sgd was a really bad choice throwing the accuracy into 30%.

Feature Maps Visualizations:

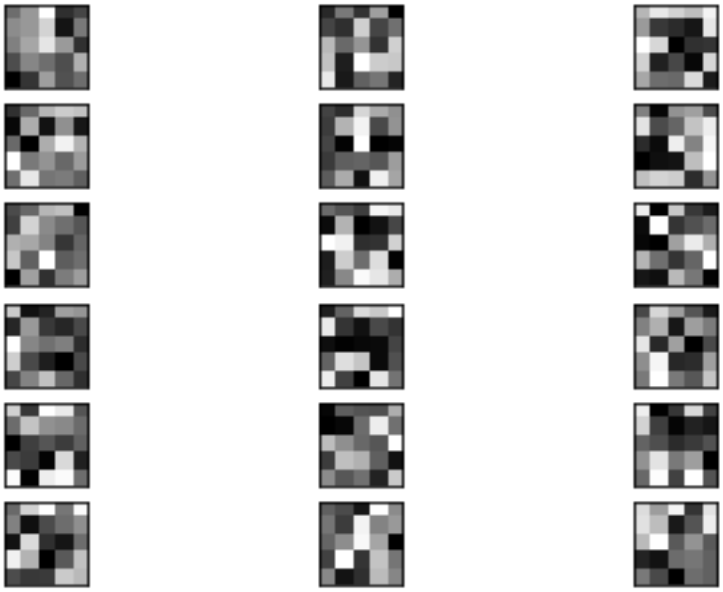
In order to do a deeper dive into the model itself, I wrote functions that given a Conv2d layer, will render on the notebook what that layer actually viewed.

The algorithm I wrote above in the design of the model has 4 sets of Conv2d/Maxpools layers. In order to see what features or characteristics of the input image were seen by each of the Conv2D layer, it is important to be able to slice the model till that layer, let it predict what it sees – and then display the rendered features.

The convolutional layers near the input layers are trained will features size of 32 as you can see in the model definition. Later, this increases down the sets. The idea is to that the layers near the input should capture very specific features while those closer to the output capture the rather more generic features of the image supplied.

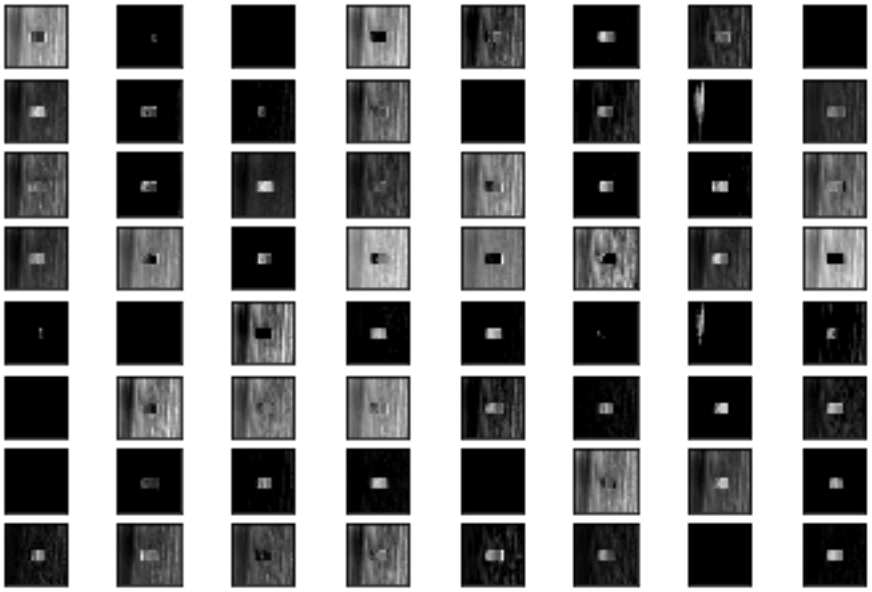
The functions I've written to display these features from the feature map picks the given Conv2D map, creates a new model from out of it, and take the string location to the file it needs to predict. After taking all this information, I render top 6 features out of that layer. As you can see, in the nearer to input layers, the more specific features are being collected. Following is a render from my 1st Conv2D layer (2nd in the list of total layers, hence the parameter 2 to the function).

First 6 features for the first Conv2D layer:



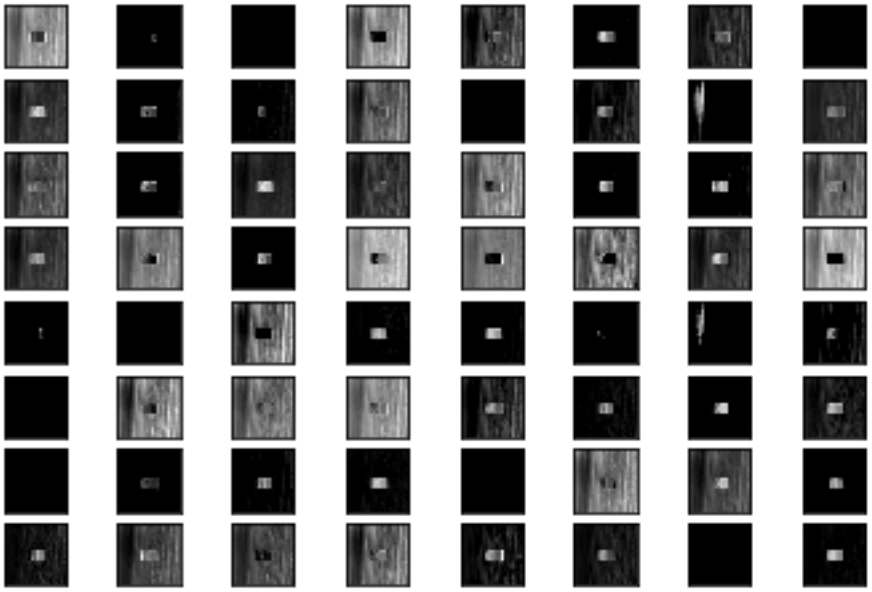
The 1_by_1 image I provided as seen by the 1st Conv2D layer. As we can see, the feature map is looking into very specific characteristics of the image like edges, the center point of the image etc.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_72_input (InputLayer) [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d_72 (Conv2D)           (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d_72 (MaxPooling (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_73 (Conv2D)           (None, 109, 109, 64)      18496
=================================================================
Total params: 19,392
Trainable params: 19,392
Non-trainable params: 0
_________________________________________________________________
Feature_maps dimention (1, 109, 109, 64)
```



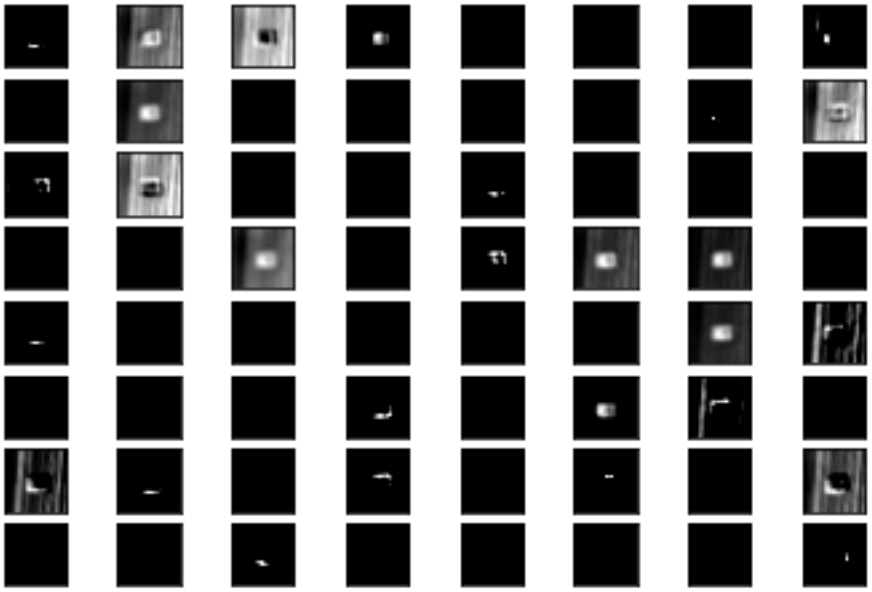
Following is the features rendering from the 2nd Conv2D layer (4th on the index of all layers in the model). Here, the total number of filters that were asked as you can see from the model definition where 64. With more filters, we can see the image is getting smoother as the model move towards reading the more generic attributes of the image.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_72_input (InputLayer) [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d_72 (Conv2D)           (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d_72 (MaxPooling (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_73 (Conv2D)           (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_73 (MaxPooling (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_74 (Conv2D)           (None, 52, 52, 128)       73856
=================================================================
Total params: 93,248
Trainable params: 93,248
Non-trainable params: 0
_________________________________________________________________
Feature_maps dimention (1, 52, 52, 128)
```



Finally the rendering of the 3rd and Final Conv2D layer (6th on the index of all layers in the model). As you can see in the model definition, filters in this layer are 128 and the image gets smoother.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_72_input (InputLayer) [(None, 224, 224, 3)]     0
_________________________________________________________________
conv2d_72 (Conv2D)           (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d_72 (MaxPooling (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_73 (Conv2D)           (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_73 (MaxPooling (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_74 (Conv2D)           (None, 52, 52, 128)       73856
_________________________________________________________________
max_pooling2d_74 (MaxPooling (None, 26, 26, 128)       0
_________________________________________________________________
conv2d_75 (Conv2D)           (None, 24, 24, 128)       147584
=================================================================
Total params: 240,832
Trainable params: 240,832
Non-trainable params: 0
_________________________________________________________________
Feature_maps dimention (1, 24, 24, 128)
```



These characteristics are very interesting – See layers of model looking at the image is quite a powerful concept and root to the new advancements in deep learning.

Looking into Resent50

I also tried seeing what bricks look like to the famous Resnet50 model which is pre-trained on 1000 categories of objects. Apparently bricks, aren't one of the 1000 categories the resnet50 is trained against – but some very interesting results came out. It predicted (see the code/html) that most of my bricks were close to the category of 'prayer mats' – if you look at the shape of a prayer mate, it does actually look like a brick.

Here are the models of resnet50 I used. It shows how deep the network is and cope with various objects in those 1000 categories.

```
Model: "resnet50"
```

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	[(None, 224, 224, 3)	0	
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	0	input_4[0][0]
conv1_conv (Conv2D)	(None, 112, 112, 64)	9472	conv1_pad[0][0]
conv1_bn (BatchNormalization)	(None, 112, 112, 64)	256	conv1_conv[0][0]
conv1_relu (Activation)	(None, 112, 112, 64)	0	conv1_bn[0][0]
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	0	conv1_relu[0][0]
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	0	pool1_pad[0][0]
conv2_block1_1_conv (Conv2D)	(None, 56, 56, 64)	4160	pool1_pool[0][0]
conv2_block1_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_1_conv[0][0]
conv2_block1_1_relu (Activation	(None, 56, 56, 64)	0	conv2_block1_1_bn[0][0]

conv2_block1_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block1_1_relu[0][0]
conv2_block1_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_2_conv[0][0]
conv2_block1_2_relu (Activation	(None, 56, 56, 64)	0	conv2_block1_2_bn[0][0]
conv2_block1_0_conv (Conv2D)	(None, 56, 56, 256)	16640	pool1_pool[0][0]
conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block1_2_relu[0][0]
conv2_block1_0_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block1_0_conv[0][0]
conv2_block1_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block1_3_conv[0][0]
conv2_block1_add (Add)	(None, 56, 56, 256)	0	conv2_block1_0_bn[0][0] conv2_block1_3_bn[0][0]
conv2_block1_out (Activation)	(None, 56, 56, 256)	0	conv2_block1_add[0][0]
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16448	conv2_block1_out[0][0]
conv2_block2_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block2_1_conv[0][0]
conv2_block2_1_relu (Activation	(None, 56, 56, 64)	0	conv2_block2_1_bn[0][0]
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block2_1_relu[0][0]
conv2_block2_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block2_2_conv[0][0]
conv2_block2_2_relu (Activation	(None, 56, 56, 64)	0	conv2_block2_2_bn[0][0]
conv2_block2_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block2_2_relu[0][0]
conv2_block2_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block2_3_conv[0][0]
conv2_block2_add (Add)	(None, 56, 56, 256)	0	conv2_block1_out[0][0] conv2_block2_3_bn[0][0]
conv2_block2_out (Activation)	(None, 56, 56, 256)	0	conv2_block2_add[0][0]
conv2_block3_1_conv (Conv2D)	(None, 56, 56, 64)	16448	conv2_block2_out[0][0]
conv2_block3_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block3_1_conv[0][0]
conv2_block3_1_relu (Activation	(None, 56, 56, 64)	0	conv2_block3_1_bn[0][0]
conv2_block3_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block3_1_relu[0][0]
conv2_block3_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block3_2_conv[0][0]
conv2_block3_2_relu (Activation	(None, 56, 56, 64)	0	conv2_block3_2_bn[0][0]
conv2_block3_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block3_2_relu[0][0]
conv2_block3_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block3_3_conv[0][0]
conv2_block3_add (Add)	(None, 56, 56, 256)	0	conv2_block2_out[0][0] conv2_block3_3_bn[0][0]
conv2_block3_out (Activation)	(None, 56, 56, 256)	0	conv2_block3_add[0][0]
conv3_block1_1_conv (Conv2D)	(None, 28, 28, 128)	32896	conv2_block3_out[0][0]
conv3_block1_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block1_1_conv[0][0]
conv3_block1_1_relu (Activation	(None, 28, 28, 128)	0	conv3_block1_1_bn[0][0]
conv3_block1_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block1_1_relu[0][0]
conv3_block1_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block1_2_conv[0][0]
conv3_block1_2_relu (Activation	(None, 28, 28, 128)	0	conv3_block1_2_bn[0][0]
conv3_block1_0_conv (Conv2D)	(None, 28, 28, 512)	131584	conv2_block3_out[0][0]
conv3_block1_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block1_2_relu[0][0]

conv3_block1_0_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block1_0_conv[0][0]
conv3_block1_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block1_3_conv[0][0]
conv3_block1_add (Add)	(None, 28, 28, 512)	0	conv3_block1_0_bn[0][0] conv3_block1_3_bn[0][0]
conv3_block1_out (Activation)	(None, 28, 28, 512)	0	conv3_block1_add[0][0]
conv3_block2_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block1_out[0][0]
conv3_block2_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block2_1_conv[0][0]
conv3_block2_1_relu (Activation	(None, 28, 28, 128)	0	conv3_block2_1_bn[0][0]
conv3_block2_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block2_1_relu[0][0]
conv3_block2_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block2_2_conv[0][0]
conv3_block2_2_relu (Activation	(None, 28, 28, 128)	0	conv3_block2_2_bn[0][0]
conv3_block2_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block2_2_relu[0][0]
conv3_block2_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block2_3_conv[0][0]
conv3_block2_add (Add)	(None, 28, 28, 512)	0	conv3_block1_out[0][0] conv3_block2_3_bn[0][0]
conv3_block2_out (Activation)	(None, 28, 28, 512)	0	conv3_block2_add[0][0]
conv3_block3_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block2_out[0][0]
conv3_block3_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block3_1_conv[0][0]
conv3_block3_1_relu (Activation	(None, 28, 28, 128)	0	conv3_block3_1_bn[0][0]
conv3_block3_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block3_1_relu[0][0]
conv3_block3_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block3_2_conv[0][0]
conv3_block3_2_relu (Activation	(None, 28, 28, 128)	0	conv3_block3_2_bn[0][0]
conv3_block3_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block3_2_relu[0][0]
conv3_block3_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block3_3_conv[0][0]
conv3_block3_add (Add)	(None, 28, 28, 512)	0	conv3_block2_out[0][0] conv3_block3_3_bn[0][0]
conv3_block3_out (Activation)	(None, 28, 28, 512)	0	conv3_block3_add[0][0]
conv3_block4_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block3_out[0][0]
conv3_block4_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block4_1_conv[0][0]
conv3_block4_1_relu (Activation	(None, 28, 28, 128)	0	conv3_block4_1_bn[0][0]
conv3_block4_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block4_1_relu[0][0]
conv3_block4_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block4_2_conv[0][0]
conv3_block4_2_relu (Activation	(None, 28, 28, 128)	0	conv3_block4_2_bn[0][0]
conv3_block4_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block4_2_relu[0][0]
conv3_block4_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block4_3_conv[0][0]
conv3_block4_add (Add)	(None, 28, 28, 512)	0	conv3_block3_out[0][0] conv3_block4_3_bn[0][0]
conv3_block4_out (Activation)	(None, 28, 28, 512)	0	conv3_block4_add[0][0]
conv4_block1_1_conv (Conv2D)	(None, 14, 14, 256)	131328	conv3_block4_out[0][0]
conv4_block1_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block1_1_conv[0][0]

conv4_block1_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block1_1_bn[0][0]
conv4_block1_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block1_1_relu[0][0]
conv4_block1_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block1_2_conv[0][0]
conv4_block1_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block1_2_bn[0][0]
conv4_block1_0_conv (Conv2D)	(None, 14, 14, 1024)	525312	conv3_block4_out[0][0]
conv4_block1_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block1_2_relu[0][0]
conv4_block1_0_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block1_0_conv[0][0]
conv4_block1_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block1_3_conv[0][0]
conv4_block1_add (Add)	(None, 14, 14, 1024)	0	conv4_block1_0_bn[0][0] conv4_block1_3_bn[0][0]
conv4_block1_out (Activation)	(None, 14, 14, 1024)	0	conv4_block1_add[0][0]
conv4_block2_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block1_out[0][0]
conv4_block2_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block2_1_conv[0][0]
conv4_block2_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block2_1_bn[0][0]
conv4_block2_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block2_1_relu[0][0]
conv4_block2_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block2_2_conv[0][0]
conv4_block2_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block2_2_bn[0][0]
conv4_block2_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block2_2_relu[0][0]
conv4_block2_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block2_3_conv[0][0]
conv4_block2_add (Add)	(None, 14, 14, 1024)	0	conv4_block1_out[0][0] conv4_block2_3_bn[0][0]
conv4_block2_out (Activation)	(None, 14, 14, 1024)	0	conv4_block2_add[0][0]
conv4_block3_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block2_out[0][0]
conv4_block3_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block3_1_conv[0][0]
conv4_block3_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block3_1_bn[0][0]
conv4_block3_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block3_1_relu[0][0]
conv4_block3_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block3_2_conv[0][0]
conv4_block3_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block3_2_bn[0][0]
conv4_block3_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block3_2_relu[0][0]
conv4_block3_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block3_3_conv[0][0]
conv4_block3_add (Add)	(None, 14, 14, 1024)	0	conv4_block2_out[0][0] conv4_block3_3_bn[0][0]
conv4_block3_out (Activation)	(None, 14, 14, 1024)	0	conv4_block3_add[0][0]
conv4_block4_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block3_out[0][0]
conv4_block4_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block4_1_conv[0][0]
conv4_block4_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block4_1_bn[0][0]
conv4_block4_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block4_1_relu[0][0]
conv4_block4_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block4_2_conv[0][0]
conv4_block4_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block4_2_bn[0][0]

conv4_block4_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block4_2_relu[0][0]
conv4_block4_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block4_3_conv[0][0]
conv4_block4_add (Add)	(None, 14, 14, 1024)	0	conv4_block3_out[0][0] conv4_block4_3_bn[0][0]
conv4_block4_out (Activation)	(None, 14, 14, 1024)	0	conv4_block4_add[0][0]
conv4_block5_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block4_out[0][0]
conv4_block5_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block5_1_conv[0][0]
conv4_block5_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block5_1_bn[0][0]
conv4_block5_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block5_1_relu[0][0]
conv4_block5_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block5_2_conv[0][0]
conv4_block5_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block5_2_bn[0][0]
conv4_block5_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block5_2_relu[0][0]
conv4_block5_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block5_3_conv[0][0]
conv4_block5_add (Add)	(None, 14, 14, 1024)	0	conv4_block4_out[0][0] conv4_block5_3_bn[0][0]
conv4_block5_out (Activation)	(None, 14, 14, 1024)	0	conv4_block5_add[0][0]
conv4_block6_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block5_out[0][0]
conv4_block6_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block6_1_conv[0][0]
conv4_block6_1_relu (Activation	(None, 14, 14, 256)	0	conv4_block6_1_bn[0][0]
conv4_block6_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block6_1_relu[0][0]
conv4_block6_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block6_2_conv[0][0]
conv4_block6_2_relu (Activation	(None, 14, 14, 256)	0	conv4_block6_2_bn[0][0]
conv4_block6_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block6_2_relu[0][0]
conv4_block6_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block6_3_conv[0][0]
conv4_block6_add (Add)	(None, 14, 14, 1024)	0	conv4_block5_out[0][0] conv4_block6_3_bn[0][0]
conv4_block6_out (Activation)	(None, 14, 14, 1024)	0	conv4_block6_add[0][0]
conv5_block1_1_conv (Conv2D)	(None, 7, 7, 512)	524800	conv4_block6_out[0][0]
conv5_block1_1_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block1_1_conv[0][0]
conv5_block1_1_relu (Activation	(None, 7, 7, 512)	0	conv5_block1_1_bn[0][0]
conv5_block1_2_conv (Conv2D)	(None, 7, 7, 512)	2359808	conv5_block1_1_relu[0][0]
conv5_block1_2_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block1_2_conv[0][0]
conv5_block1_2_relu (Activation	(None, 7, 7, 512)	0	conv5_block1_2_bn[0][0]
conv5_block1_0_conv (Conv2D)	(None, 7, 7, 2048)	2099200	conv4_block6_out[0][0]
conv5_block1_3_conv (Conv2D)	(None, 7, 7, 2048)	1050624	conv5_block1_2_relu[0][0]
conv5_block1_0_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block1_0_conv[0][0]
conv5_block1_3_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block1_3_conv[0][0]
conv5_block1_add (Add)	(None, 7, 7, 2048)	0	conv5_block1_0_bn[0][0] conv5_block1_3_bn[0][0]
conv5_block1_out (Activation)	(None, 7, 7, 2048)	0	conv5_block1_add[0][0]

```
conv5_block2_1_conv (Conv2D)    (None, 7, 7, 512)    1049088     conv5_block1_out[0][0]
__________________________________________________________________________________________________
conv5_block2_1_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block2_1_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_1_relu (Activation (None, 7, 7, 512)    0           conv5_block2_1_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_2_conv (Conv2D)    (None, 7, 7, 512)    2359808     conv5_block2_1_relu[0][0]
__________________________________________________________________________________________________
conv5_block2_2_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block2_2_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_2_relu (Activation (None, 7, 7, 512)    0           conv5_block2_2_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_3_conv (Conv2D)    (None, 7, 7, 2048)   1050624     conv5_block2_2_relu[0][0]
__________________________________________________________________________________________________
conv5_block2_3_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block2_3_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_add (Add)          (None, 7, 7, 2048)   0           conv5_block1_out[0][0]
                                                                 conv5_block2_3_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_out (Activation)   (None, 7, 7, 2048)   0           conv5_block2_add[0][0]
__________________________________________________________________________________________________
conv5_block3_1_conv (Conv2D)    (None, 7, 7, 512)    1049088     conv5_block2_out[0][0]
__________________________________________________________________________________________________
conv5_block3_1_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block3_1_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_1_relu (Activation (None, 7, 7, 512)    0           conv5_block3_1_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_2_conv (Conv2D)    (None, 7, 7, 512)    2359808     conv5_block3_1_relu[0][0]
__________________________________________________________________________________________________
conv5_block3_2_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block3_2_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_2_relu (Activation (None, 7, 7, 512)    0           conv5_block3_2_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_3_conv (Conv2D)    (None, 7, 7, 2048)   1050624     conv5_block3_2_relu[0][0]
__________________________________________________________________________________________________
conv5_block3_3_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block3_3_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_add (Add)          (None, 7, 7, 2048)   0           conv5_block2_out[0][0]
                                                                 conv5_block3_3_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_out (Activation)   (None, 7, 7, 2048)   0           conv5_block3_add[0][0]
__________________________________________________________________________________________________
avg_pool (GlobalAveragePooling2 (None, 2048)         0           conv5_block3_out[0][0]
__________________________________________________________________________________________________
probs (Dense)                   (None, 1000)         2049000     avg_pool[0][0]
==================================================================================================
Total params: 25,636,712
Trainable params: 0
Non-trainable params: 25,636,712
```

Other Tools and Techniques Used

Apart from the obvious challenge of training a Convolutional Neural Network to identify features out of these pictures for a correct classification, I wrote two utilities to convert my captured images to a correct sizes (see convert.py in the github rep) and to distribute the pictures into train, test and valid folders (see spread.py in the github repo). The spread utility is very useful in not doing any manual distribution where we have to change the train-test-valid rations for different results.

Model Testing and Results

I achieved an accuracy of 77 percent for my mode. The predictions were good (though I wish I could achieve even better) – but this percentage is well above my naïve benchmark model.

Refinements and Future Enhancements

We could use other off-the-shelf pre-trained models and use parts of their layers to have them identify the bricks better. Further research into other architectures using various optimizers such

as *RMSprop* and working with or dropping the dense layers may result in better performance worth the trying.

- Work with other dense layers (drop if necessary).
- Use optimizers other than *RMSprop such as Adam and SGD.*
- Model setup with 20 epochs

Another enhancement I would like to do would be to have this trained model move over to raspberry pi, attached it with a camera module, a setup of step motors etc. to actually let a stream of bricks flow into a conveyer belt for an eventual automatic sorting. That will be a great work to extend this project on.

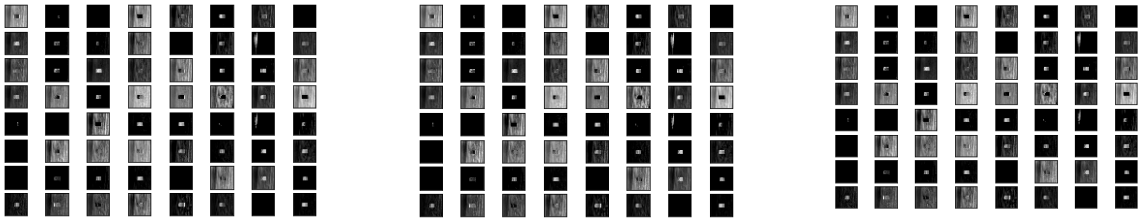
Challenges:

The major challenge I faced was the decision to whether do a brick/plate/tile classification or a more practical sized based classification. I had initially thought that a brick/plate/tile type classification will be more fun. But given the very uneven distribution of these classes (bricks are found the most in typical Lego sets, while plates and tiles are too few), I fell for the sized based classification as I could formulate a benchmark model easily that I can improve upon. With an uneven distribution, it would have been challenging to form and compare with a benchmark classification on accuracy.

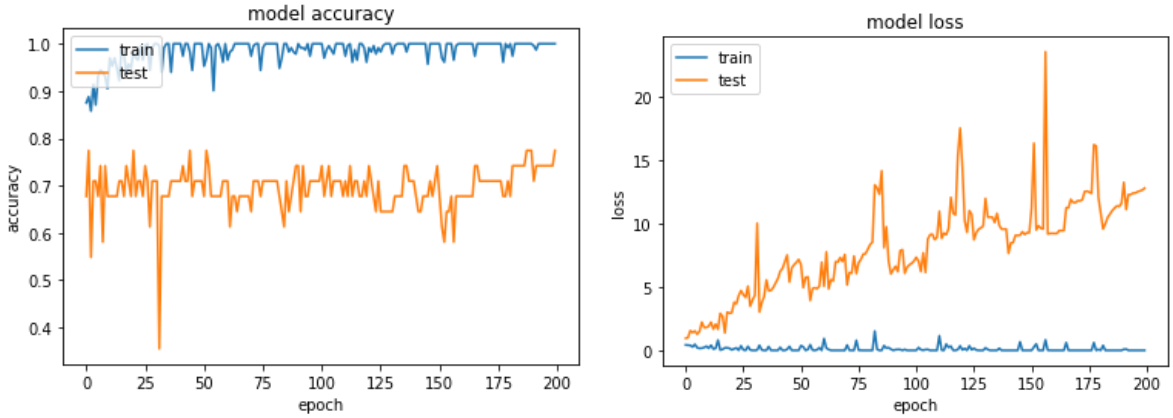
Second challenge that I faced was the very few features that a Lego brick could exhibit. Unlike a human or animal face, the bricks are so much similar to each other. The studs on top of the edges (and their number on each brick), the bottom of the image with holes which are different for each size and that is all. The rest of a brick is literally a smooth surface with not much information. With very fewer features, my classification faced challenges.

Conclusion

Computer vision has always been an exciting area for me. With the advent of recent innovations in deep neural networks has enabled this dream to be much more possible than a few years back. Enabling computers to ‘see’ just like humans is an exciting accomplished.



With varying level of changes in the algorithm parameters and choice of optimizers, I as able to train a Deep learning Convolutional Neural Network model to identify various classes of bricks in Lego boxes. Was it perfect? I do not think so. Image processing is an all-evolving field and further enhancements can be done to any model. But given the following results on an epoch of 200 with the best model is choose relative to loss, I think this model should serve good on a small scale Lego recognizer engine.



The future enhancements could be one where the model sits on a raspberry-pi or Arduino while a camera reads the image of a passing brick while the model classifies and routes it to it correct ‘bucket’. I will definitely pursues this for these extensions and improvements.