AdaConv: an Inverse Lego Project

Weichen (Tony) Zheng
zweichen@seas.upenn.edu

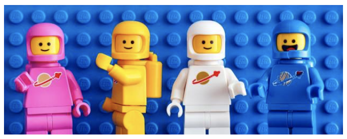
Yiting (Ellen) Hua
ellenhua@seas.upenn.edu

Hanyu (Sylvie) Sun
sunhanyu@seas.upenn.edu

Weiyi Tang
tangwy@seas.upenn.edu

I. Introduction

LEGO, a robust collection of plastic construction toys manufactured by the LEGO Group, is a great target to study for a computer vision machine learning project. It's natural for the Lego fans like us to ask: if we toss a bunch of Lego pieces on to the ground and took a photo, will the computer be able to recognize each piece captured? Our project seeks to dig deep into this question using object detection model.



Data is the foundation of any machine learning task. In this project, we generate synthetic data as there is no readily available labeled data, and annotating data within the time frame of this project is impossible. Hence we choose to generate datasets using a 3D modeling tool: Blender. For object detection, we test the use of Faster R-CNN in this task. We also introduce a novel adaptive convolutional layer (AdaConv) and we propose replacing the RoI pooling layer in the original Faster R-CNN model with an AdaConv layer, which we hypothesize would reduce information loss and improve performance of the model.

In this paper, we will discuss in details of our data generation process, our non-deep learning approach, deep learning approach as well as our reasoning behind the decisions. We will also show the analysis on the results.

II. Related Works

Our project aims to solve an object detection problem. Current state of the art object detection frameworks include Fast R-CNN[11], Faster R-CNN[10], Mask R-CNN[1], Single Shot MultiBox Detector(SSD)[3] and You Only Look Once(YOLO)[2]. Fast R-CNN improves speed and performance from traditional object detection frameworks by making a single stage training and inference network. Faster R-CNN, proposed in 2015, further improves the performance. It generates regions of interests (RoI) using Region Proposal Networks(RPN), and passes the RoIs to Fast R-CNN model for location and class prediction. The detailed architecture will be explained in later sections.

Mask R-CNN extends Faster R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition. SSD discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. YOLO frames object detection as a regression problem to spatially separated bounding boxes and associated class probabilities.

According Huang et al. at Google Research[12], Faster R-CNN in general has a relatively higher accuracy in object detection, especially for small objects. YOLO and SSD, on the other hand, are famous for detecting real-time images and thus though having a higher speed at the expense of accuracy. First, since our project doesn't need to provide masks for our lego bricks, we ruled out Mask R-CNN. Also, for this particular project we aim to identify and locate objects on still images and we value accuracy over speed, we finally decided to implement Faster R-CNN as our deep learning baseline model.

Faster RCNN uses max pooling to turn the feature map of each region of interests into the same shape. Nonetheless, information is often lost in max pooling. In a paper by Springenberg et al. "Striving for Simplicity: the All Convolutional Net," the author shows that replacing max pooling layer with convolutional layers with larger stride can stabilize and improve models.[9] This technique is also used by Radford et al. in their deep convolutional GAN. In their paper "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," they also show that using all convolutional net can stabilize learning.[7] These arguments motivate use to find an alternative to RoI pooling.

III. Methods

A. Data Generation

To train an object detection model that can detect LEGO pieces, we need training images that contain scattered LEGO pieces on the floor with bounding boxes and label for each LEGO piece. Based on our search, there is not any such dataset readily available online. Thus, we choose to generate our own data using LDraw, an open source LEGO CAD software containing CAD models for most LEGO parts, and Blender, a 3D modeling software.

With CAD modeling, we could know the bounding boxes of each piece and the label for each piece for each image. As a result, the generated images can be automatically annotated. In addition, with CAD modeling, we can randomize the background color, the lighting, and the angle of the camera to add variety to our dataset and improve our model's ability to generalize.

Due to the time limit of the project and the limitation of our computational resources, we do not think training a model that can recognize every LEGO part (thousands of them) is feasible. Hence we use only parts from one LEGO category in our training set. We chose LEGO set 21320 Dinosaur Fossils since one of our members already owns this set. There are around 170 different LEGO bricks in total. We include 6 copies of each brick in our bags of LEGO bricks when generating new images. In our chosen LEGO set, for most types of parts, there are only 4 copies contained in the set. Hence we choose to use only 6 copies of each part as our pool of LEGO parts. Furthermore, using more copies would slow down the speed of data generation.

Using Rebrickable API, we were able to generate a list of LDraw IDs of all parts contained in our dataset. Using this list of LDraw IDS, we are able to import the 3D models of the parts in our chosen LEGO set into Blender.

Blender has a physics simulation functionality that allows us to simulate the dropping of LEGO parts onto a plane. When generating images, we first generate a setting. To generate a setting, we first randomly choose a parameter p uniformly from the range $[0.75, 0.85]$. This is the probability that a part from our pool of parts would be excluded from that setting. With a total of around 170 different types of objects and 6 copies of each object, the total number of LEGO parts in our pool would be $170 \cdot 6 = 1020$. With $p \in [0.75, 0.85]$, the expected number of parts included in each setting ranges between 255 and 153. We also choose the height of the camera uniformly at random from the range $[10, 15]$ and choose the x and y coordinates of the center of the camera viewpoint uniformly at random from $[4.5, 5.5]$. The z coordinate of the center of the camera viewpoint is fixed at 0. This way the viewpoint of the camera would be randomized in a range that it can capture most of the images within the square of $(0,0,0), (0,10,0), (10,0,0), (10,10,10)$. We also randomize the location of the lighting by choosing its x, y coordinates uniformly at random from the range $[0, 10]$ and z coordinate uniformly at random from the range $[2, 10]$. The rotation of lighting is also randomized. However, since the lighting is not directed, the rotation of the lighting does not affect the generated images. In all settings, a plane is placed horizontally at $z = 0$. For each setting, we also randomize the color of the plane by choosing each of its RGB values uniformly at random from $[0, 255]$.

After generating a setting, we then iterate over the pool of LEGO parts. For each part, we first generate randomly choose x_r and y_r uniformly from $[0, 10]$ and z_r uniformly from $[10, 30]$. Then with probability p, we will exclude it by placing it at coordinates $(1000 + x_r, 1000 + y_r, 1000 + z_r)$. With probability $1 - p$ we place it at coordinates (x_r, y_r, z_r). In addition, we also randomize the orientation of the parts uniformly at random from all possible orientations. We also randomize their colors by choosing each value of their RGB uniformly at random from the range $[0, 255]$. Processing all pieces, we begin the simulation of the pieces. Since all pieces were placed above the plane, they would fall onto the plane. After 200 frames, the drop would be finished and we use the camera to take 5 different images form 5 different angles. This is done by moving the camera in a circle around the center of its viewpoint after the drop and generating image at 5 different frames during the movement of the camera.

For each of the image, we use the object data of each LEGO part in that frame to generate the bounding box. We use script from online [13] to generate bounding boxes of the different parts. Figure 1 shows an example of the generated images.

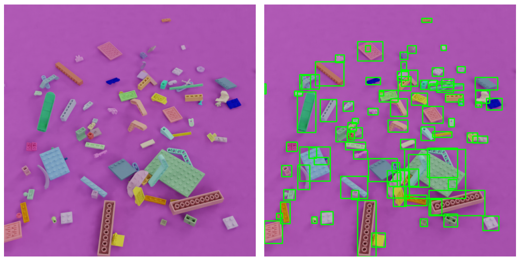


Fig. 1: One example of the generated images

We generated a total of 24,000 images, each of size 560 × 560 × 4 (RGBA). For each of the models we've tried, we use a subset of the images. Details will be explained in the respective sections. For data processing, we convert the images to size 560 × 560 × 3 by removing the alpha value, which is 255 across all images. We then normalize the images with mean=$[0.485, 0.456, 0.406]$, std=$[0.229, 0.224, 0.225]$.

B. Non deep learning baseline

For our non deep learning baseline model, we implemented a linear regression model and a simple convolutional model to predict the number of LEGO pieces in an image. Ideally, to better compare the results between baseline models and advanced models, the models should output locations and classes of identified LEGO pieces in the images, which would vary from image to image. However, due to the limitation of non-deep baseline models, the two models can only produce output of a fixed size. Therefore, we decide to let the baseline models to predict the total number of LEGO pieces in an image.

For both models, we input normalized images of size 560 × 560 × 3 and output a number indicating the number of pieces in the image. For the two baseline model, we use a total of 1800 images which are split into train, validation and test set with ratio 0.8:0.1:0.1. The performance of the models on the test set is similar to the train set so we deem such number of train images to be sufficient. For both models, we use batch size 1.

Linear Regression: For our Linear Regression model, we flattened the image and plugged it into the model, which consists of 1 linear layer.

Simple convolutional model: To compare the performance, we also implemented a simple deep neural network. The model consists of two convolutional layers followed by leaky ReLU and maxpool respectively, and then two linear layers. TABLE I shows the hyperparameters of the two models.

Hyperparam	Linear Reg	Simple Conv
Optimizer	torch.optim.Adam	torch.optim.Adam
Learning Rate	0.0001	0.0001
Loss Function	MSELoss	MSELoss
Num Epoch	5	5

TABLE I: Hyper parameters of non-deep/simple models

C. Deep learning baseline

For our deep learning baseline, we implemented the Faster R-CNN architecture. (Fig. 2)

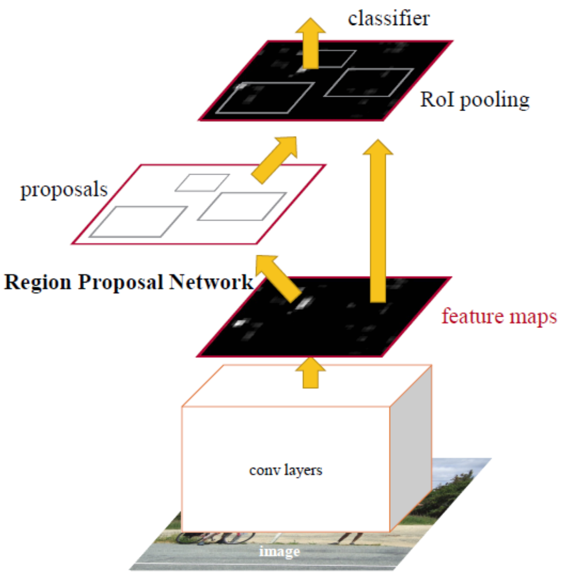


Fig. 2: Visualization of Faster R-CNN model from original paper

Faster-RCNN model is composed of multiple parts: VGG or ResNet convolutional layers as feature extractors, a Region Proposal Network(RPN), an non-maximum suppression operation, an RoI Pooling layer, and a Fast R-CNN detector. For training it also requires an anchor target creator and a proposal target creator.

Faster R-CNN first uses a traditional state of the art convolutional model such as VGG or ResNet to extract feature maps of images. In our model, we use ResNet since ResNet has a simpler architecture than VGG and is thus faster than VGG16.[11]

Since our images have size 560×560, we use 3 convolutional layer from ResNet to extract a feature map of 70×70. This means the feature stride would be 8. Using more layers would increase the feature stride to 16, which would be larger than some of the bounding boxes. As a result some of the smaller objects might be missed. Hence we only use 3 convolutional layers from ResNet.

The extracted feature maps would be shared by different parts of the network. RPN is then used to generate region of interests proposals. RPN first uses a convolutional layer to preprocess the input feature map. It then generates proposals based on base bounding boxes called anchors. These anchors are pre-determined by a list of scales and list of height-to-width ratios that are passed in as hyper-parameters. In our implementation, we use 2 different scales: 1 and 2 and 3 different ratios: 0.5, 1, and 2. We use a base box size of 16 since most of the boxes have dimension 20. See figure 3 for a histogram of the dimension distribution.

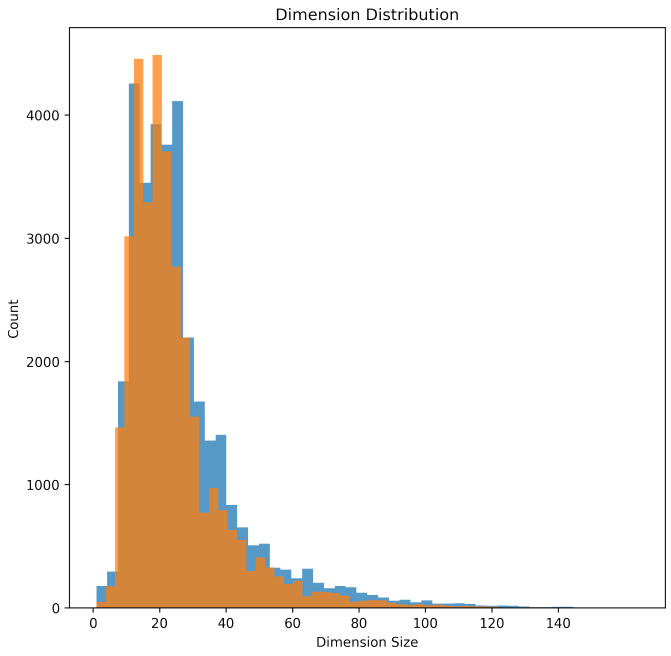


Fig. 3: A distribution of the dimensions of the ground truth bounding boxes of a subset of our dataset

As shown in the figure, most of the bounding boxes have dimension centered at 20. Hence we use base box size 16. The scale of 1 and 2 would set the anchor sizes to be 16 and 32 respectively.

To calculate the coordinates of the anchors, we first calculate the heights and widths of an anchor from a combination of scale and ratio values. The height and width of the anchor box are given by the following equation, where h_{box} is the height of the anchor bose s_b is the base size, s is the cale, and r is the height to width ratio.

$$h_{box} = s_b \cdot s \cdot \sqrt{r}$$

$$w_{box} = s_b \cdot s \cdot \sqrt{\frac{1}{r}}$$

With this equation, we have that

$$\frac{h_{box}}{w_{box}} = \frac{s_b \cdot s \cdot \sqrt{r}}{s_b \cdot s \cdot \sqrt{\frac{1}{r}}} = \sqrt{r^2} = r$$

which satisfies the ratio requirement. The area of the box is

$$h_{box} \cdot w_{box} = s_b^2 \cdot s^2 \cdot \sqrt{r} \cdot \sqrt{\frac{1}{r}} = s_b^2 \cdot s^2$$

which is the desired area value. We then calculate the coordinates of the boxes from the dimensions at each pixel of the feature map. Let x_l and y_l be the coordinates of the top left corner of the anchor box (lower valued coordinates) and x_h and y_h be the coordinates of the bottom right corner of the anchor box (higher valued coordinates). Let x_c and y_c be the coordinates of the center of an anchor box. We have

$$x_l = x_c - \frac{w_{box}}{2}$$

$$y_l = y_c - \frac{h_{box}}{2}$$

$$x_h = x_c + \frac{w_{box}}{2}$$

$$y_h = y_c + \frac{h_{box}}{2}$$

With 2 different scales and 3 different ratios, there are 6 different anchor boxes for each pixel on the feature map. Let $k = 6$ denote the number of anchor boxes for each pixel. We use a convolutional layer of kernel size 1 to map each pixel in the feature map to a vector of size $2k$. This represents the background score and object score of each anchor box at each pixel. The object score indicates how likely the anchor box contains an object of interests. We use another convolutional layer of kernel size 1 to map each pixel in the feature map to a vector of size $4k$. This represents the offsets and scales for each anchor box.

There are 4 different components in each offsets and scales vector for an anchor box: two offsets for x and y and two scales for height and width. Let o_x and o_y be the offsets and let s_h and s_w be the scales. These offsets and scales are used to adjust anchor boxes in the following way. Let h and w be the original height and width of the anchor box. Let x_c and y_c be the original coordinates of the center of the anchor box. Let h', w', x'_c, and y'_c be the height, width, and center coordinates of the adjusted bounding box. We have

$$h' = e^{s_h} \cdot h$$

$$w' = e^{s_w} \cdot w$$

$$x'_c = h \cdot o_x + x_c$$

$$y'_c = w \cdot o_y + y_c$$

We can then turn the adjusted heights, widths, and center coordinates into box coordinates using the equation above. These adjusted anchor boxes are candidate proposals. We then sort them based on their predicted object scores. We take the first 16,000 candidate proposals. The original implementation takes the first 12,000 candidate proposals. Since there are more objects in each image in our dataset than in many other object detection datasets, we increase the number of candidate proposals we consider.

These candidate proposals are then filtered using non-maximum suppression. Non-maximum suppression filters out overlapping bounding boxes. It calculates the intersection over union (IoU) of each pair of boxes. The IoU of two boxes is the area of their intersection divided by the area of their union. In practice, it is calculated by first finding the coordinates of the bounding boxes by taking the maximum of the lower valued box coordinates and the minimum of the higher valued box coordinates. We then use the intersection coordinates to calculate the height, width, and then the area of the intersection. We then calculate the area of the union by first calculating the area of the two boxes, and then by principle of inclusion and exclusion, the area of the union is the sum of the areas of the two boxes subtracted by the area of the intersection. We then divide the area of the intersection by the area of the union to calculate the IoU.

Non-maximum suppression take a set of candidate proposals and their predicted scores. Then in each iteration, it takes the highest scoring candidate proposal from the set and remove all candidate proposal that has IoU value above certain threshold with the selected candidate proposal. It repeats this process until no candidate proposals remain. It then return the list of selected proposals ranked by their scores.

With these selected candidate proposals, we choose the top 4000 proposals. In the original implementation 2000 proposals were chosen. However, again since our dataset contains more objects in each image than usual, we increase the number of proposals we choose.

These chosen proposals would be our regions of interests (RoI). When evaluating the model, we pass all RoI to the RoI pooling layer. However, during training, randomly sample RoI from the RoI generated by RPN.

The RoIs are then passed to the RoI pooling layer. This layer takes the input RoI, finds the area on the feature map that corresponds to that RoI, and then converts the area to a fixed size using max pooling. We use torchvision's roi_pool operator in our implementation.

After RoI pooling, the proposed regions would be converted into a batch of smaller sized feature map with the same size. We can hence use the same layers to process each feature map. We use two fully connected layers to map each feature map to a vector of size 4096 as per the original implementation. We then use a linear layer to map the vector to a vector of size $p + 1$ where p is the number of different objects. This vector represents the probability of the RoI containing an object of a certain label or not containing any object. We use another linear layer to map the vector of size 4096 to a vector of size $4 \cdot (p + 1)$. This vector contains the offsets and scales for adjusting the RoI box to fit the object of a certain type. To generate the final predicted bounding box, we take all RoI that are classified as some object and adjust the bounding boxes of these RoI according to the predicted offsets and scales for the predicted label.

To train the model, we need to create targets for both the RPN and the Fast R-CNN classifier. For RPN, we calculate the IoU between each anchor and each ground truth bounding box. We label an anchor as positive if it has an IoU of over 0.7 with any ground truth bounding box. We label an anchor as negative if it has an IoU of less than 0.3 with all ground truth bounding boxes. These thresholds are taken from the original Faster R-CNN implementation. We ignore other anchors since they are ambiguous. For location targets, we calculate the offsets and scales required to transform a positive anchor to a target ground truth bounding box using the inverse of the equations discussed before. We ignore the predicted locations of negative anchors.

For the Fast R-CNN classifier, during training, we sample a fixed number of RoIs from the proposed RoIs from RPN. We first assign ground truth bounding boxes to RoIs according to their IoU values. The bounding box with the largest IoU with an RoI is assigned to that RoI. If an RoI has an IoU value of at least 0.5 with its assigned bounding box, then the label of that bounding box is assigned to that RoI. Otherwise the RoI is considered negative and is assigned the label corresponding to no object. For location target, we again calculate the offsets and scales required to transform a positive RoI to its assigned grounding truth bounding box using the inverse of the equations shown above. Again predicted locations for negative RoIs are ignored.

We use cross entropy loss for the classification tasks and smoothed L1 loss for the offsets and scales prediction tasks as in the original implementation. We take a weighted sum of these four losses as the final loss of the model.

For training, we tried 2 different types of training. In the first type, we train all parts of the model together. In the second type, we first train the RPN only by setting the weight of the losses of the Fast R-CNN to be 0. We then train the entire model together with the trained RPN.

We only support batch size 1 since all implementations we found, including the original implementation, only support batch size 1.

Unlike many implementations found online, our implementation ensures that all model computations are done within PyTorch using CUDA. Many implementations require libraries such as CuPy to do computation outside PyTorch with CUDA.

D. Advanced deep learning approach

In the original Faster R-CNN architecture, the RoI pooling is done using an adaptive max pooling layer to convert feature maps of different sized RoIs into the same size. However max pooling can lead to information loss. Hence we propose a new type of layer: adaptive convolutional layer (AdaConv). AdaConv performs the same shape conversion as adaptive max pooling, but uses convolutional operations rather than max pooling operations. Previous papers such as "Striving for Simplicity: the All Convolutional Net" and "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" have shown that replacing max pooling layer with convolutional layers with larger stride can stabilize and improve models.[7][9]

To convert feature maps of different size to a fixed size, we use various different sized convolutional layers. Specifically, we use all kernel size combinations with both height and width ranging from 1 to 5 with stride sizes same as the kernel size and no padding. In addition, we also use 3 convolutional layers with kernel sizes $(2,1)$, $(1,2)$, and $(2,2)$ and stride sizes same as kernel sizes and no padding as shrinking layers.

Model	TrainLoss	TrainAcc	TestLoss	TestAcc
LinReg	1460.311	0.0125	217.260	0.0167
SimpleConv	21.691	0.0799	26.444	0.0833

TABLE II: Performance of non-deep/simple models

Given an image of size h and w that need to be converted into size h_p and w_p, we first check to see whether $h > 5 \cdot h_p$ or $w > 5 \cdot w_p$. If yes then use the shrinking layers to shrink the size of the image. If both $h > 5 \cdot h_p$ and $w > 5 \cdot w_p$ holds then we use the shrinking layer with kernel size $(2,2)$. Otherwise we use the shrinking layer with kernel size 2 in the dimension that is too larger and kernel size 1 in the dimension that meets the size limit. We keep passing the feature map through the shrinking layers until the condition $h \leq 5 \cdot h_p$ and $w \leq 5 \cdot w_p$ holds.

We then calculate the kernel size required to convert the images. Let h' and w' be the size of the feature map after the shrinking layers. We use the convolutional layer with kernel height

$$h_k = \max\{1, \lfloor \frac{h'}{h_p} \rfloor\}$$

and kernel width

$$w_k = \max\{1, \lfloor \frac{w'}{w_p} \rfloor\}$$

Before the final convolution, we use an adaptive max pooling layer to convert the size of the feature map to $(h_k \cdot h_p, w_k \cdot w_p)$. Note that when h' or w' is less than 7, $h_k = 1$ or $w_k = 1$ and PyTorch's adaptive max pool would scale up the feature map in that dimension by repeating some rows or columns of the original feature map.

At the end we use a convolutional layer of size (h_k, w_k) to convert the feature map into size (h_p, w_p). We can guarantee the size of the output to be (h_p, w_p) since we have converted the feature map to size $(h_k \cdot h_p, w_k \cdot w_p)$. Using a convolutional layer of size (h_k, w_k) and stride (h_k, w_k) and no padding would ensure that the size of the output is (h_p, w_p).

We only use up to convolutional layer of size 5 because larger kernel might be inflexible. We still use adaptive max pooling in our AdaConv layer, but the information loss is limited since in both dimensions the size is reduced by at most $h_p - 1$ and $w_p - 1$ by max pooling.

In our model, we replace the RoI pooling operator with our own AdaConv layer and hold all other components the same.

IV. Analysis

A. Non-deep Results

Table II summarizes the results of the two baseline models. Linear Regression achieves 0.0125 accuracy score on train set and 0.0167 on test set, while simple convolutional model achieves 0.0799 accuracy score on train set and 0.0833 on test set. Simple Convolutional model perform better than Linear Regression. However, the accuracy for both models are still very low.

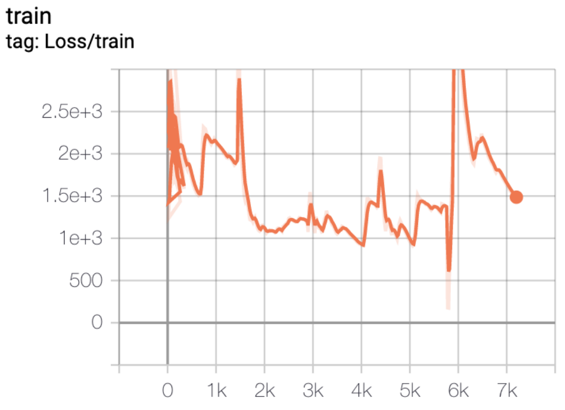


Fig. 4: Training loss curve of Linear Regression Model

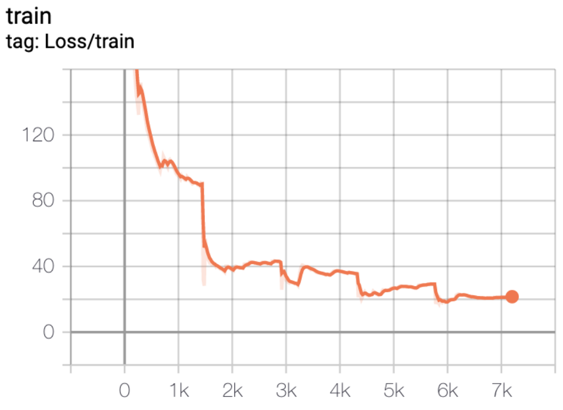


Fig. 6: Training loss of Simple Convolutional Model

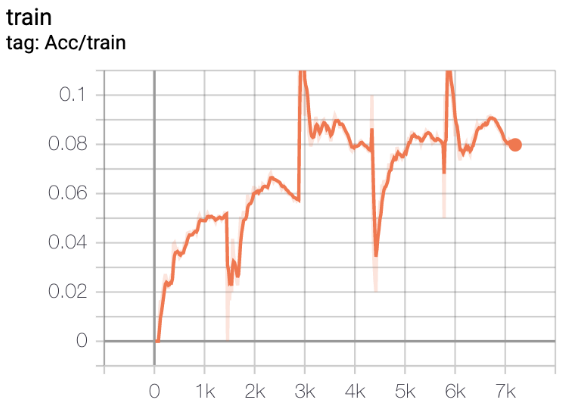


Fig. 7: Training accuracy curve of Simple Convolutional Model

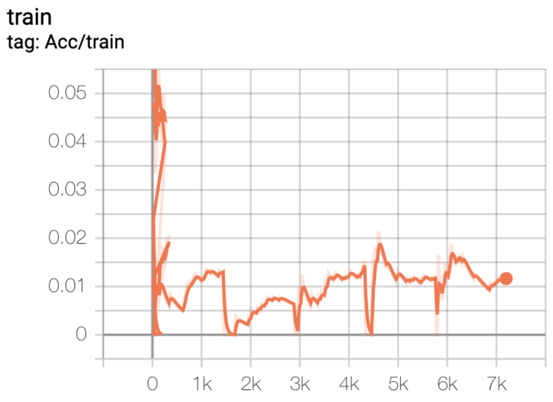


Fig. 5: Training accuracy curve of Linear Regression Model

From the loss and accuracy curve of Linear Regression model we can see that the training is not well, and the model is not learning throughout the steps. In comparison, Simple Convolutional Model has a relatively smoother loss and accuracy curve, indicating that it is training effectively.

We hypothesize that there are several reasons for Simple Convolutional Model performing better : 1. linear regression is relatively simple model, making it unable to make good predictions on such complex image. 2. The training set has a class imbalance problem, making linear regression overfitting the data and predict numbers around 30 for most of the trials.

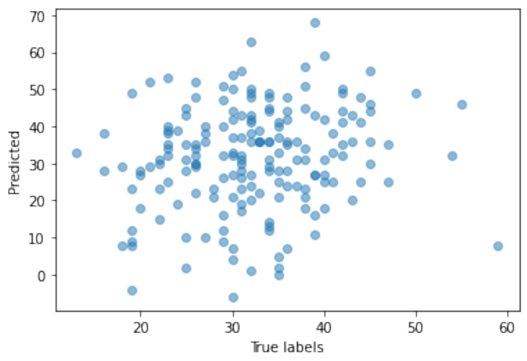


Fig. 8: Scatter plot of test result of Linear Regression

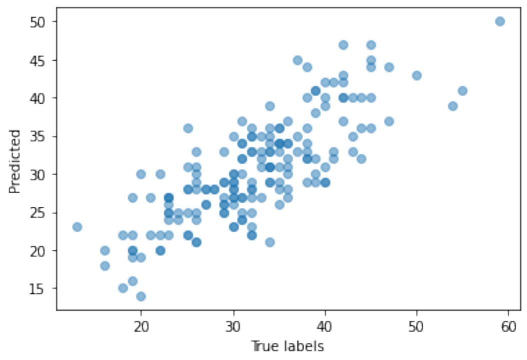


Fig. 9: Scatter plot of test result of Simple Convolutional Model

From the scatter plot we can see that Linear Regression model tends to predict numbers that have the highest probability: around 30-50, regardless of the actual image. It shows that the Linear Regression model inclines to predict the more popular classes in order to minimize its loss. The scatter plot of Simple Convolutional Model shows a much better pattern. Points are scattered around the line y=x, which means that most predictions are around the ground truth numbers. Thus, we conclude that Simple Convolutional Model better extract and learn from information of the image, and is thus better at predicting the number of objects in images than Linear Regression, which is expected.

From the results both models, we can see that in general simple/non-deep models are not competent enough for predicting the number of objects in an image, let alone its inability to locate and classify objects. We will then look into the performance of object detection models.

B. Performance Metric

For our deep models, we use mean average precision (mAP) as our evaluation metric. It is the standard metrics used in many object detection tasks such as PASCAL VOC, ImageNet, and COCO challenges. It is also the metric used by the original Faster R-CNN paper. [10] mAP measures the average precision across all recall levels, taking into account all classes, the confidence score, the classification accuracy and the detection accuracy. Specifically, it calculates the average interpolated precision scores across 11 recall levels [0, 0.1, ..., 1.0]. The interpolated precision score is simply the highest precision score at a given recall level across all classes. The interpolation is used to reduce the impact of ”wiggles” on precision recall curve. [15] mAP score has a range of 0 to 1, with 1 being the highest score.

C. Deep Results

We use 1200 images as our development dataset. We use 900 images for training and 300 images for evaluation.

The training curve of our implementation of Faster R-CNN with learning rate 1e-5 is shown in figure 10.

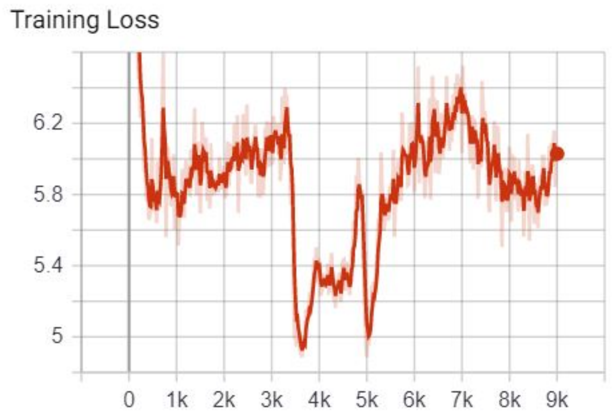


Fig. 10: Training Loss of the Original Faster R-CNN with Learning Rate 1e-5

As shown in the graph, the training is very unstable. It is likely that the learning rate is too high. Hence we also tested the model with learning rate 1e-6. Figure 11 shows the resulting training curve.

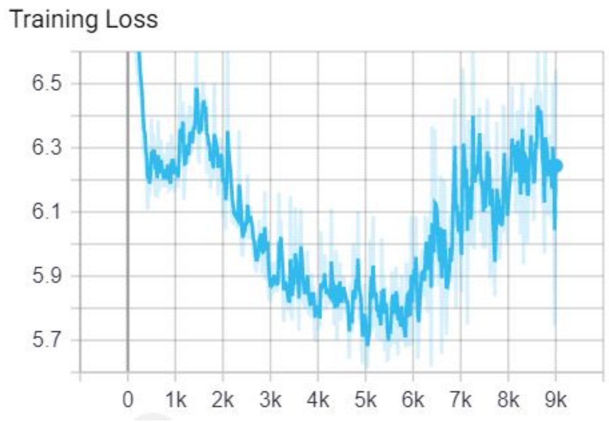


Fig. 11: Training Loss of the Original Faster R-CNN with Learning Rate 1e-6

The training loss is more stable than with learning rate 1e-5. However, after 5k steps, the loss starts to increase. We further test a smaller learning rate of 1e-7. Figure 12 is a comparison of the training curve of learning rate 1e-6 with the training curve of learning rate 1e-7.

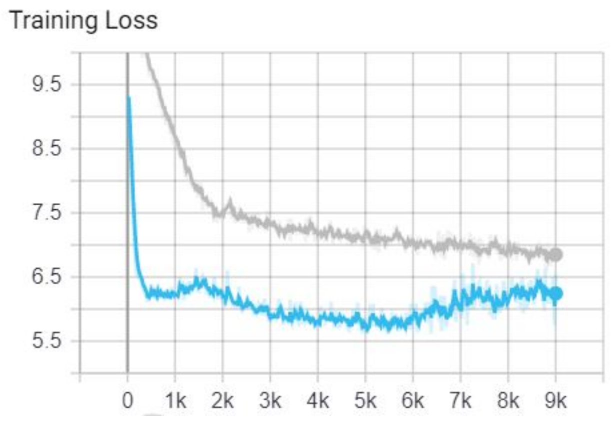


Fig. 12: Training Loss of the Original Faster R-CNN
Light Blue: 1e-6, Grey: 1e-7

As shown, in figure 12, while learning rate 1e-7 gives a more stable training curve, the loss plateaus before it reaches the same level of loss as learning rate 1e-6.

Although the loss plateaus, the model fails to generate any reasonable predictions. The resulting model using learning rate 1e-6 has mAP score of 0% even on the training dataset.

We then tested our second method of training. We first train the RPN only by setting the weights of the loss of Fast R-CNN to 0. We train the RPN using learning rate 1e-6 for 5 epochs. After training the RPN for 5 epochs, we found that the training curve does not quite plateaus. Hence we train the RPN for another 5 epochs. Figure 13 shows the training curves.

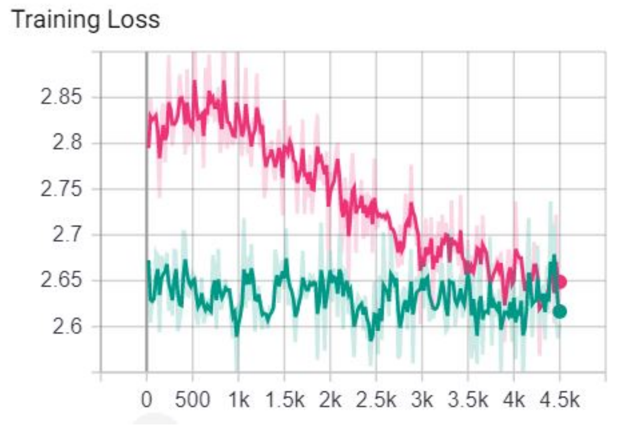


Fig. 13: Training Loss when training RPN only
Magenta: first 5 epochs, Green: second 5 epochs

As shown in the figure, the second 5 epochs does not further improve the model. Therefore we continue further training using the resulting model from the first 5 epoch of RPN training. Figure 14 shows the resulting training curve compared with the training curve of training the entire model directly with learning rate 1e-6.

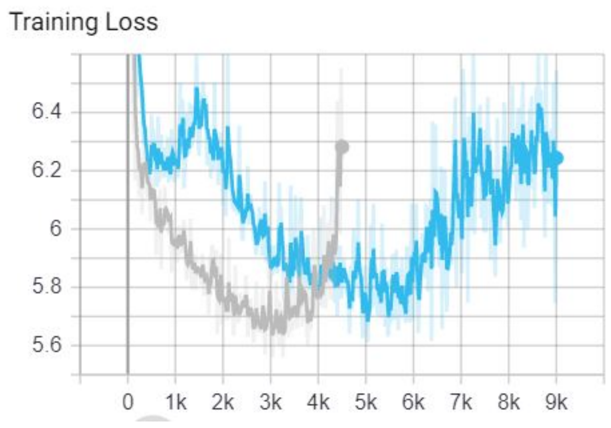


Fig. 14: Training loss from training the model with trained RPN (grey) compared with the training loss from training the entire model directly (light blue)

Initially, the loss of the continued training model stays below the loss of training the entire model. However we see that after 3,000 steps, the loss starts to increase again. The mAP score of the resulting model stays at 0%.

We then tested adding more channels to the RPN feature map. We increased the number of channels of the RPN convolutonal layer from 128 to 256. We then again trained the RPN of the model only. Figure 15 shows the resulting training loss.

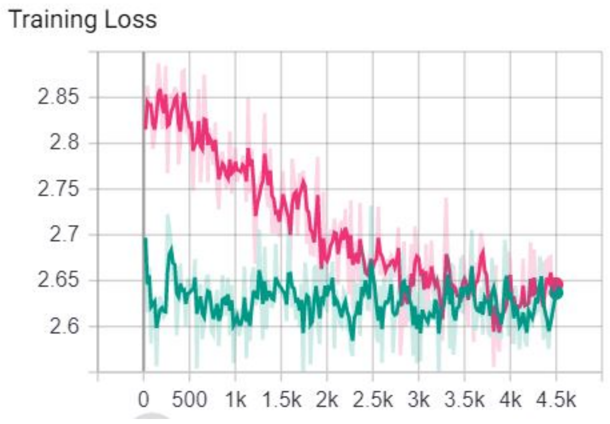


Fig. 15: Training Loss when training RPN only with 256 channels
Magenta: first 5 epochs, Green: second 5 epochs

Again the second 5 epochs does not help improve the RPN. Furthermore, compared with figure 13, the increase in the number of channels does not decrease the training loss. This suggests that increasing the number of channels in RPN does not really improve the model. However, we still continued training the entire model based on the trained RPN. Figure 16 shows the resulting training curve in comparison with the continued training with 128 channels in RPN.

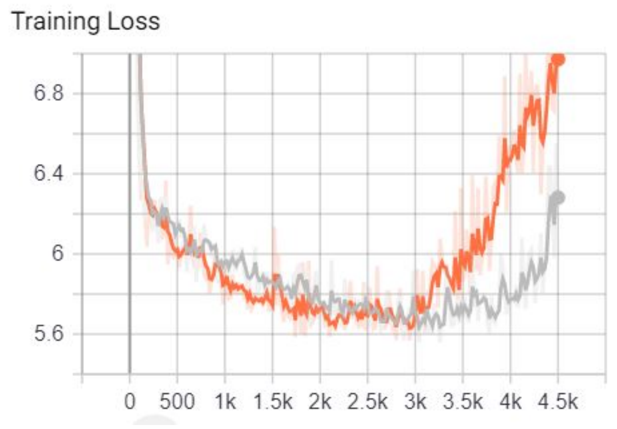


Fig. 16: Training loss from training the model with trained RPN with 256 channels (orange) compared with the training loss from training the model with trained RPN with 128 channels (grey)

The figure shows that with 256 channels, the loss of the model increase at a even faster rate after 3000 steps. The resulting model, unfortunately, still fails to predict any reasonable bounding boxes and have mAP score 0%.

We now turn to AdaConv and compare the results of using AdaConv vs using RoI pooling. Again we first train the model using AdaConv with learning rate 1e-6. Figure 17 shows the resulting training curve compared to the original Faster R-CNN model.

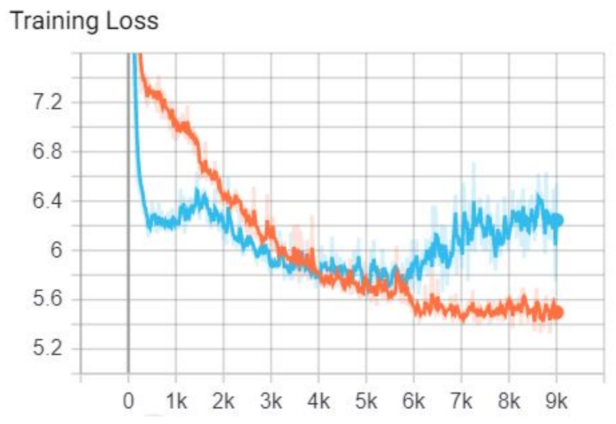


Fig. 17: Training loss from AdaConv with learning rate 1e-6 compared to the training loss of the original Faster R-CNN

As shown in the figure, initially the original Faster R-CNN model learns faster than our model using AdaConv. This might be due to the fact that AdaConv uses many convolutional layers while RoI pooling does not require any parameters that need to be learned. Hence AdaConv requires more steps to learn appropriate models. After 5000 steps, the loss of the original Faster R-CNN model starts to increase but the loss of our model using AdaConv continues to decrease. This shows that using AdaConv instead of RoI pooling indeed stabilizes the learning and prevents the loss from increasing.

We also took the trained RPN and further trained it using our model with AdaConv. The resulting training curve is shown in figure 18.

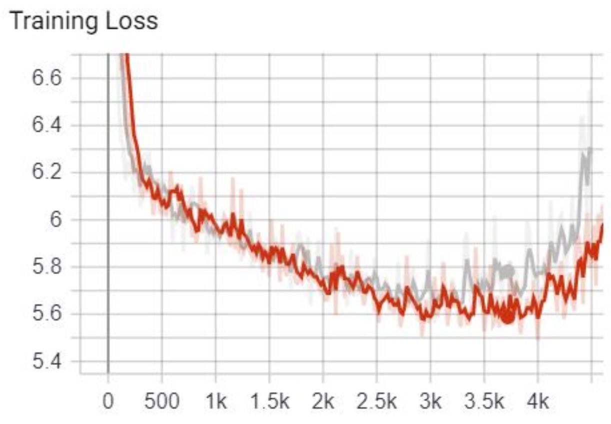


Fig. 18: Training loss from training our model with trained RPN with 128 channels (red) vs training loss from training the original Faster R-CNN model with trained RPN with 128 channels

The training curves show that our model using AdaConv is able to achieve a loss slightly less than the loss achieved by the original Faster R-CNN model.

Although our models still fail to achieve good results on the mAP score, these training curves do show that using AdaConv layer instead of RoI pooling can help stabilize training.

D. Exrternal Implementation of Faster R-CNNs

Since our implementations fail to achieve good mAP scores, we want to see how other implementations Faster R-CNNs performs in this particular case (e.g. detecting a single piece of Lego with a lot of classes). We found a suitable implementation of Faster R-CNN by Roboflow that is compatible with our custom dataset. Here is the link to the implementation: *Faster R-CNNs implementation by Roboflow*

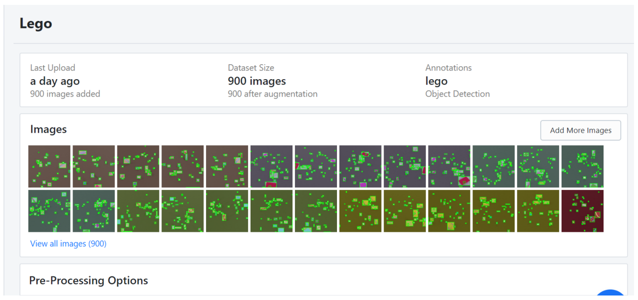
In order to use this model with custom dataset, we first need to preprocess our dataset to convert it to the required format. Images need to be labelled XML file, since this model only accepts XML file. So we wrote a python script to convert box.txt which contains coordinates of bounding box of each Lego piece to XML file. Then utilized a useful website called Roboflow (as you can see in figure 19) to pre-process data. After uploading images and corresponding XML file, we can have a dataset health check. It will give us class balance, number of total annotations, and average annotations in each image. So we are able to remove or add new images to balance classes. We can also resize the images to any size we want. In this case, we want 416, so we can just keep it. Apart from that, data augmentation operation is also available to use. Roboflow will help us flip, rotate, changing brightness and so on. After all this pre-process operations are done, we could export dataset in particular format. In this case, TFRecords format is required.

Parameters	Value
Batch size	12
Training rate	0.0002(>2000 steps), 0.00002(<2000 steps)
Momentum	0.9
Number of step	4000

TABLE III: Parameters Choice

Result	Value
DetectionBoxes_Precision/mAP	0.0135
DetectionBoxes_Precision/mAP (large)	0.8000
DetectionBoxes_Precision/mAP (medium)	0.0652
DetectionBoxes_Precision/mAP (small)	0.0062
BoxClassifierLoss/classification_loss	0.9898
BoxClassifierLoss/localization_loss	0.8267
RPNLoss/localization_loss	0.5291
RPNLoss/objectness_loss	0.0833
Total_loss	2.4290

TABLE IV: Training Results


Lego

Last Upload
a day ago
900 images added

Dataset Size
900 images
900 after augmentation

Annotations
lego
Object Detection

Images

Pre-Processing Options

Fig. 19: Roboflow Website

Then we need to choose a specific model and pipeline file. As for model, faster_rcnn_inception_v2 was selected, and the parameters we set in pipeline file can be seen in TABLE III. We also tuned some parameters to have a better result. In order to find minimal point more quickly and accurately, we schedule training rate, that is to say, before 2000 steps, we use a relative large training rate to allow loss decrease to somewhere close to global minimal point, then we decrease the training rate to a relative small value to find more accurate global minimal point. This operation will also decrease the vibration when the we are around the global minimal point. We also use momentum here to accelerate the converge rate of SGD, because momentum will consider the previous update state to help modify the direction of derivative, which will slightly move it towards the global minimal points.

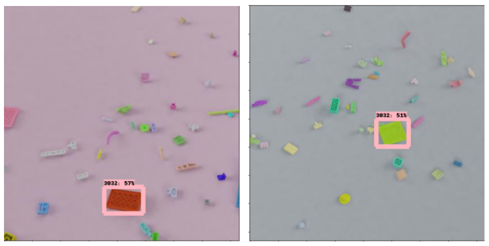


Fig. 20: Result Images

After training, statistical results are in TABLE IV, and the output images can be seen in figure 20, when we fed test images to trained model.

As the result shown above, mAP is relatively low, and during training, total loss will first decrease from 7.5 to 2.6, then after 800 steps, loss will oscillate between 2.5 and 2.9, resulting in that the result is not what we want. According to this performance, we think it's either we should tune parameters to improve this model or Faster R-CNNs is not good at detecting small objects with a mass of classes due to the complex background, occlusion, low resolution and tiny difference between classes.

E. Result Analysis

Our project fails to make good predictions for the LEGO detection and identification task. Even when using the external implementation, the model fails to achieve a good mAP score. Below are two main challenges posed by the LEGO detection task in comparison with traditional object detection tasks.

- LEGO pieces are small in size. The ground truth bounding boxes in our dataset are generally very small with dimensions around 20 pixels as shown in figure 3. This means that in order not to miss certain objects of interests, we cannot increase the feature stride too much. Therefore, our feature extractor is limited to three convolutional layers and our feature map can only have 128 channels since using more convolutional layers would lead to larger feature stride, causing some objects to be missed in the feature map. This limits the variation of features our model can learn from.
- At a high level sense, LEGO features are quite similar. Pre-trained models such as VGG and ResNet are often trained to extract high level features to distinguish images of different classes at a high level such as cats and dogs. However, in these datasets, LEGO pieces are likely all categorized as the same class. Therefore, the extracted features by ResNet might not be enough to distinguish LEGO pieces from other LEGO pieces. Furthermore, many LEGO pieces contain similar features such as circular stubs and straight corners. Hence distinguishing these pieces might require counting the number of features. However, counting might not be something the feature extractor using pre-trained models can achieve. It would be able to detect a circular stub, but it might not be able to actually count the number of stubs to distinguish a longer brick from a shorter brick. For instance, when multiple stubs appear in the image, at one layer a feature map might contain the signal for each of the stubs. In the next layer, however, the multiple signals from the stubs might be merged into one signal by max pooling. As a result, the model can predict the existence of a feature, but it might not be able to count the number of features.

In comparison, non-deep models are unable to predict variable number of bounding boxes based on the input. Our two deep models, Faster R-CNN and Faster R-CNN with AdaConv, both fail to achieve good mAP scores. However, figure 17 and 18 both suggest that Faster R-CNN with AdaConv is able to learn better and more stably than the original Faster R-CNN. Using AdaConv might especially be beneficial in LEGO detection task. As mentioned before, being able to count the number of features is important in distinguishing LEGO piece from LEGO piece, but max pooling might bury the signal of multiple features and merge multiple signals into one. When replacing RoI pooling with our AdaConv layer, we are replacing max pooling operations with convolution opterations. When a convolutional layer encounters multiple signals, it takes the weighted sum of the signals. As a result, more feature signals would lead to a larger resulting signal, unlike with max pooling. Hence, our model using AdaConv performs better than the original Faster R-CNN model.

V. Discussion

As mentioned at the end of our previous section, our project faces certain challenges not present in traditional object detection tasks: small object sizes and similar features. Furthermore, there are much more targets in one single image. In the future work, these challenges need to be further addressed.

First, for the model, we would look into ways to adjust the model architecture to better fit our task. As mentioned, max pooling would prevent the model from learning to count the number of features. Hence we would remove the max pooling layer in the feature extractor and replace them with down sampling convolutional layers.

Furthermore, if we have enough time and computational power, we would retrain the ResNet feature extractor so that the feature extractors would learn more features specific to LEGO pieces. This would allow the feature extractor to extract features that can be used to distinguish LEGO pieces from LEGO pieces instead of general features used to distinguish ImageNet classes.

In addition, it is also possible to improve upon the current AdaConv layer. Our current design still utilizes max pooling as part of our layer since all of our convolutional layers inside AdaConv has the same strides as their kernel sizes and no padding. If possible, we would replace the max pooling layer with more intricately designed convolutional layers with variable strides and padding such that it can convert feature maps of variable size into the same output size without using adaptive max pooling.

Also, one another improvement worth exploring in future works is to change the way we generate data. Since we are randomizing each pieces' RGB value, there are time when our pieces look ”fake” compared to real pieces. While such data generation might make the model more robust, such unrealistic colors might present an additional challenge to our model, especially to the pretrained layers which are pretrained on natural ImageNet images. If time and resources permit, the best data generation method would be manually taking photos of real Lego pieces and then annotate all the pieces on the photo. However, such task would be very time consuming. Hence a possible middle-ground method would be finding all possible real colors of LEGO pieces and real texture of floors and randomly use these colors and textures instead of purely random colors in data generation.

Since our models are trained on synthetic images, in future research, it would also be necessary to test and potentially increase the model's robustness on real images. As mentioned above, the synthetic images do not look completely the same as real images, especially the colors. The model might need extra tuning with a small real image dataset.

In addition, we would utilize another advantage of synthetic data: the ability of manipulate pieces in the way we want. We could adopt an adversarial training method. For example, we could deliberately position certain pieces to fool the model. We could also set the lighting and the camera angle specifically to cause the model to misclassify. We could also put similar pieces together to train the model's ability to differentiate them. We would look more into the area of Domain Randomization[14]. In addition, we would take real photos of scattered LEGO pieces and mix them in our dataset, to improve the model's performance on real images.

References

[1] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick, Mask R-CNN, Computer Vision and Pattern Recognition (cs.CV), 1703.06870.

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, You Only Look Once: Unified, Real-Time Object Detection, Computer Vision and Pattern Recognition (cs.CV), 1506.02640.

[3] Wei Liu, Dragomir Anguelov, Dumitru Erhan et al., SSD: Single Shot MultiBox Detector, Computer Vision and Pattern Recognition(cs.CV), 1512.02325.

[4] Jacob Sullivan , Machine learning LEGO image recognition: Using virtual data and YOLOv3. https://towardsdatascience.com/machine-learning-lego-image-recognition-using-virtual-data-and-yolov3-f12e0544012.

[5] Nichols, Nicole, and Robert Jasper. ”Projecting Trouble: Light Based Adversarial Attacks on Deep Learning Classifiers.” arXiv preprint arXiv:1810.10337 (2018).

[6] Su, Jiawei, Danilo Vasconcellos Vargas, and Kouichi Sakurai. ”One pixel attack for fooling deep neural networks.” IEEE Transactions on Evolutionary Computation 23.5 (2019): 828-841.

[7] Radford, Alec, Luke Metz, and Soumith Chintala. ”Unsupervised representation learning with deep convolutional generative adversarial networks.” arXiv preprint arXiv:1511.06434 (2015).

[8] Huang, Jonathan, et al. ”Speed/accuracy trade-offs for modern convolutional object detectors.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[9] Springenberg, Jost Tobias, et al. ”Striving for simplicity: The all convolutional net.” arXiv preprint arXiv:1412.6806 (2014).

[10] Ren, Shaoqing, et al. ”Faster r-cnn: Towards real-time object detection with region proposal networks.” Advances in neural information processing systems. 2015.

[11] He, Kaiming et al. “Deep Residual Learning for Image Recognition.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): n. pag. Crossref. Web.

[12] Ross Girshick. ”Fast RCNN”. ICCV 2015.

[13] Huang, Jonathan, et al. ”Speed/accuracy trade-offs for modern convolutional object detectors.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[14] https://blender.stackexchange.com/questions/7198/save-the-2d-bounding-box-of-an-object-in-rendered-image-to-a-text-file. Web.

[15] Tobin, Josh, et al. ”Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World.” IEEE/RSJ International Conference on Intelligent Robots and Systems. 2017.

[16] Everingham, Mark, et al. ”The PASCAL Visual Object Classes (VOC) Challenge.” International Journal of Computer Vision. 2010.