Object Detection on LEGO Images

Zhengyuan Xu
University of Pennsylvania
Philadelphia, PA
zhengyx@seas.upenn.edu

Xuanyi Zhao
University of Pennsylvania
Philadelphia, PA
xuanyizh@seas.upenn.edu

Po-Yuan Wang
University of Pennsylvania
Philadelphia, PA
poyuanw@seas.upenn.edu

Shuqi Zhang
University of Pennsylvania
Philadelphia, PA
sshuqi@seas.upenn.edu

1. Introduction

In the past several decades, attempts and progress have been made on visual recognition tasks such as image classification and object detection. However, little of the classification and detection tasks are implemented to combine with LEGO. We first use methods such as transfer learning to implement classification on single LEGO bricks and evaluate its accuracy. Then two YOLO structures are applied on composed LEGO bricks to give bounding box of each brick as well as its corresponding brick type. We expect our ideas and experiment results may contribute to future deep learning classification model design for reverse engineering.

An overview of the remainder of the paper is as follows. Section 2 presents the detailed methods of simple feed forward neural network, convolutional neural network, transfer learning on ResNet-18 and YOLO. Section 3 shows how exactly we implement methods with LEGO bricks. Section 4 demonstrates the outcome of the YOLO model training. The last section is a discussion of the results and possible improvements.

2. Related Work

2.1. Logistic Regression

Logistic regression has been used in handling categorical data generally as a baseline. In the core of the logistic regression is the function it uses: the logistic function or the sigmoid function. Logistic regression uses an equation as the representation, similar to what linear regression does. The input values are multiplied with the weights, just like a linear regression. However, unlike the linear regression, it has an activation function at the end. Before the final output the calculation is fed into the logistic regression which bounds all output values within the range [0, 1]. A logistic regression can also be interpreted as a single neural network with no hidden layers whose activation function is a sigmoid function. Researchers usually use this kind of model at the beginning of a task to evaluate the complexity of a dataset.

2.2. Convolution Neural Network

CNN in deep learning is a key concept, especially in the image processing field. CNN was popular in the 1990s but went out of fashion with the occurrence of support vector machines (SVM). The work in [8, 9, 5] has shown the potential of convolutional layers in computer vision tasks. However, without the support of massive computation power, shallow architectures using CNN showed only little boost in the model performance. It was only until recently that powerful graphics processing unit(GPU) are introduced In 2012, which brought people's attention back to CNN in deep neural network architecture. With the publication of the well-known AlexNet[1], researchers shifted their attention back to CNN. CNNs have developed from artificial neural networks (ANN), also known as neural networks (NN). The idea of a neural network is inspired by the brain. Researchers have observed that biological learning systems are built of complicated interconnected neurons. Imitating the process of how brains learn, a neural network is a framework that combines several complex units to complete the work. Each unit takes responsibility for a single part.

A typical CNN structure contains four layers. A con-

volution layer is essentially a filter that extracts the features from the given input. The word convolution implies the calculation method in CNN. Different from the so-called ANN, CNN achieves sparse connectivity using convolutional calculations instead of the normal matrix multiplication. This change significantly speeds up the calculation of the trained model. Another layer is known as pooling. The most famous pooling is called max pooling. The functionality of this layer maximizes the input size while maintaining the feature given by the previous layer. Reducing the input size can increase the calculation speed. Rectifier linear units (ReLU) are the third classic layer. As an activation function, ReLU helps bring nonlinear features to the linear output, thus increasing the accuracy of the prediction. The last layer is the fully-connected layer which combines all of the learned features from the previous layers and provides a final prediction.

2.3. Transfer Learning

Transfer learning has been an important measure in the field of machine learning. Researchers and engineers have been using this technique to resolve the issue of data dependence in deep learning[12]. Deep learning methods often require a strong dependence on large amount of training data, compared to other traditional machine learning models. As deep learning models are learning, they try to find the latent patterns of data through a huge amount of training data. The number of training data available are usually very limited in amount in most tasks, as collecting and labeling the data require substantial human labor. The introduction of transfer learning provides a solution when there is insufficient training data. For image-related tasks, recent deep learning models like VGG networks[11] and residual networks[4] have achieved high scores on the ImageNet[2] object classification tasks.

2.4. ResNet-18

As mentioned in the previous section, transfer learning is a common technique used in image classification. Residual network(ResNet)[4] scored the first place on the ILSVRC 2015 classification task and has been a helpful base of transfer learning in other image classification tasks. The network outperforms many other architecture as a result of the application of deep residual framework. It addresses the problem of degradation in deep learning by letting the layers in the model fit a residual mapping, instead of fitting on the original, unreferenced mapping. In our experiment we applied a 18-layer residual network as our architecture of transfer leaning.

2.5. YOLO

You only look once (YOLO)[10] is a real-time object detection system. YOLO is known for its fast speed on detection and also good accuracy. On a Pascal Titan X it processes images at 30 FPS and has a mAP of 57.9% on COCO test-dev.

YOLO mainly includes three steps. First it splits image with grids and then each grid predicts bounding boxes and corresponding confidence value. By this step, we can have a knowledge of where is the possible grid that contains an object. The thrid step is for each grid to predict a class possibility. Therefore, we know not only whether this grid has an object, but also what object it contains.

YOLO-v1[7] mainly utilizes regression idea and applies simple networks to directly finish both classification and location tasks very quickly. After that, YOLO-v2[6] and YOLO-v3[10] further improve the detection accuracy and reduce the training time tremendously, which speed up the application of object detection in industries.

3. Methods

3.1. The Dataset

There are two datasets we use for our work. The first one is a collection of single LEGO bricks. It contains snapshots of 50 different LEGO bricks taken from 800 different angels using the script produced by [3]. Some of the examples are given in Figure 1. Note that screenshots of a single brick are taken from multiple angles and some of the angles are harder than others. We mainly used this dataset as a early baseline in evaluating the difficulty of our final task.

The other dataset we use is a collection of composite LEGO objects, where each LEGO object is composed of three to five single LEGO bricks. There are a total of 20 different LEGO objects assembled using 37 different single LEGO bricks. These snapshots are taken and labeled manually by members of our group. Some examples from the dataset are given in Figure 2. We used this dataset for our final task: composite LEGO objects detection and classification.

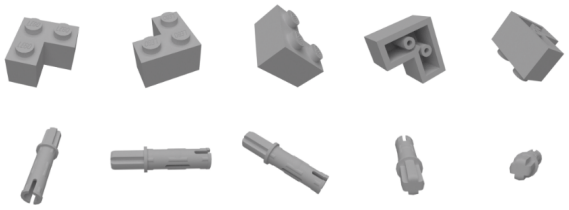


Figure 1. Examples of single LEGO bricks. The names of the two bricks shown are: 2357 Brick(top) and 11214 Bush 3M friction with Cross axle(bottom).

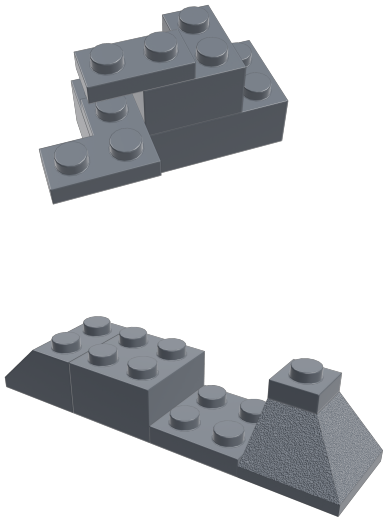


Figure 2. Examples of composite LEGO objects.

3.2. Evaluating the Task Difficulty

Before approaching the final task of LEGO brick detection and classification, we want to first evaluate the feasibility of our potential solution to the task. As the final dataset contains various composite LEGO objects that consist of three to five single LEGO building bricks, we start with tasks that involve only single LEGO bricks first. Therefore we introduce the sub-task: Single LEGO Brick Image Classification. We assume that we are able to train a model that can achieve excellent scores on this sub-task. Only when this assumption is proven to be true can we further our goal into detecting and classifying of composite LEGO images.

To be more specific, we assume that each brick has a latent representation that is semantically meaningful. These representations are likely to be containing physical information of brick, like shape, dimension, and position of a brick in the image. By extracting these semantic representations using methods like CNN we may be able to distinguish them from each other. Success on this sub-task will confirm our assumption that the LEGO images are possible to be learned by some models and we could proceed with our final task starting from whatever models we found efficient for the sub-task. We also assume this task is harder than usual 2-D image classification tasks like [2]. The reason behind this is that the images and object classes are taken from real life scenarios where the distinguishing features of an object might be exposed well in most images, for examples, the face of cats and dogs help significantly in recognizing them. However, many of our LEGO images are taken from angles that distort the original 3-D object in a 2-D projection. Hence we assume our models could learn some 3-D features based on considerable amounts of pure 2-D images of LEGO bricks from different perspectives.

From Figure.1 it is clear that some images of the bricks could be rotated to an angle so that some of the distinct features of a brick is not easily visible. Note that this situation somewhat resembles the setup of a composite LEGO object image where some bricks are hidden or not rotated to an angel that is hardly recognizable. Lots of the data we have are shot from angels that make the brick barely identifiable by human. We are hoping that the model will learn some latent representations of bricks through massive training images shot from a variety of angles. In the following sections we show some of our approaches to the single brick classification problem.

3.3. Data Pre-processing

For all methods we experimented with, either deep learning methods or other tradition machine learning methods, we perform some image preprocessing first. We first used the same procedure for preprocessing data in all models, then we realized that some models might need more or less preprocessing than others. So we have specified what we have done before training the model.

3.4. Traditional Machine Learning Models on the Sub-Task

We resize the image to 128 by 128 pixels. The images in both datasets are in three RGB color channels. Then we normalize the three channels with $\mu = 0.485, 0.456, 0.406$ and $\sigma = 0.229, 0.224, 0.225$ for red, green and blue channels respectively. No other preprocessing was done in this step.

We tried logistic regression as our initial naive approach to the sub-task. We implemented our logistic regression using the PyTorch framework in Python. The logistic regression method is implemented using a single linear layer without any activation function, and trained using stochastic gradient descent. For this method we trained the logistic regression model on a GeForce 2080 Ti graphic card for 15 epochs. We did not train it for a prolonged period of time because we believe the performance of this model would not increase drastically even if trained for a longer period of time. We also experimented with hyper-parameters to see if better performance can be achieved. For this model we used learning rate $= 0.001$, and batch size $= 32$. The results using this model can be seen in Figure5. We experimented with different set of hyper-parameters but it turned out that this simple machine learning method is not very sensitive to hyper-parameter changes. We also discovered that using a higher resolution, surprisingly, decreased the training and validation accuracy of the model.

3.5. Deep Learning Models on the Sub-Task

In this section we discuss the deep learning methods we tested. We tried simple feed forward neural networks, convolutional neural networks and applied transfer learning on ResNet18 model.

3.5.1 Simple Feed Forward Neural Networks

First we built a simple feed forward neural network. After tuning the hyper-parameters like the number of layers, the number of neurons in each layer, learning rate and batch size, the chosen network structure is 3 hidden layers with [2048, 1024, 256] neurons for each. Also, for the selected model we used learning rate $= 0.001$, and batch size $= 32$. The results of this model can be seen in Figure5. As we expected, this model performs much better than Logistic Regression, which implies that more complex neural networks in deep learning might be able to further improve the classification results.

3.5.2 Convolutional Neural Networks

Given the fact that simple feed forward neural networks have a much better baseline than traditional machine learning, we decide to explore the convolutional neural networks in order to further improve our subtask accuracy. We started from implementing two convolution layers and two fully connected layers. After the initial attempt, we decide to increase layers to learn more features from large number of pixels. However, too many layers might cause over-fitting problem and longer training time, so we end up with sticking to three fully connected layers. In addition, after tuning the hyper-parameters on validation set, we chose learning rate $= 0.001$, and batch size $= 32$. Also, we applied a softmax followed by a logarithm on the last layer's output, because it is faster and has better numerical properties. The results of CNN beat that of simple feed forward neural networks and can be seen in Figure5.

3.5.3 Transfer Learning

We also tried transfer learning on ResNet18 pretrained model. The ResNet architecture [4] was proposed to solve the degradation problem in a plain deep network. It achieved high score on image classification tasks. We used the 18-layer residual net architecture for the model and initialize the model parameters with weights pretrained on ImageNet[2]. The initial learning curves are given in Figure.5. This model does require a little bit of hyper-parameter tuning in order to get the high validation score noted in the figure. We initially obtained high score on the training set. This indicates that the model is able to overfit the training data. As the model is able to learn all the variance in the training set, our next step is to reduce overfitting. Then we performed random flip and random clip on the training images to obtain the final validation accuracy shown in Figure.5.

3.6. YOLO on the Dataset

3.6.1 YOLO v3 Theory and Novelty

The YOLO v3 algorithm implements a single Neural network to the Full Image. The YOLO network divides the image into several regions and predicts bounding boxes and probabilities for each region. These bounding boxes are also given a weight by the predicted probabilities.

In comparison to the previous version (YOLO v1, YOLO v2), a novel deeper architecture of feature extractor called Darknet-53 was proposed in this version.This feature extractor contains of 53 convolutional layers, each followed by a batch normalization layer and Leaky ReLU activation layer.There is no pooling layer in this architecture but a convolutional layer with stride 2 is used to downsample the feature maps, which can help in preventing loss of low-level feature when we are doing pooling in normal convolutional neural network.

3.6.2 The Architecture of the YOLO v3

YOLO v3 uses a variant of Darknet, which originally has 53 layer network trained on Imagenet. For the task of detection, 53 more layers are stacked onto it, giving us a 106 layer fully convolutional underlying architecture for YOLO v3. This is the reason behind the slowness of YOLO v3. Before selecting the parameters, we first calculate some important features in the YOLO v3 architecture. The Input dimension of the image is 416*416*3. The output layer dimension is (class numbers + boundary box parameters + confidence score) * number of Anchors, which in our case , we have a dimension of (37+5)*3 = 126. Figure 3 [10] shows the structure of the darknet-53 model:

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 3. Darknet-53 Architecture

For each of the three YOLO layers, we need to determine the anchor value. The formula is (input size * scale parameter) * (input size * scale parameter) *((number of classes + number of parameters of boundary box + one confidence score)* number anchors used). The scale parameters are $\frac{1}{32}$, $\frac{1}{16}$ and $\frac{1}{8}$.

For scale parameter $\frac{1}{32}$, the input size is 416/32=13. Our bounding box parameter is 5 and number of classes is 37. After applying all numbers to the equation above, it becomes 13*13*126. With the same idea, for scale $\frac{1}{16}$, it becomes 26*26*126. For scale $\frac{1}{8}$, it becomes 52*52*126.

Below is the anchor number (mask selection) we set for different yolo layers.

1st YOLO layer: 116,90, 156,198, 373,326

2nd YOLO layer: 30,61, 62,45, 59,119

3rd YOLO layer: 10,13, 16,30, 33,23

3.6.3 Tiny-YOLO-v3 and its Architecture

Tiny YOLO-v3 is a shorter version of YOLO-v3, with less layers and therefore, faster speed to train and use with the cost of accuracy. We use tiny-YOLO-v3 as a model to compare the result with normal YOLO-v3. A comparison of two models is shown in section 4. Figure 4 shows the detailed structure of tiny-YOLO.

```
   layer   filters  size/strd(dil)      input                output
   0 conv     16       3 x 3/ 1    416 x 416 x   3 ->  416 x 416 x  16 0.150 BF
   1 max               2x 2/ 2    416 x 416 x  16 ->  208 x 208 x  16 0.003 BF
   2 conv     32       3 x 3/ 1    208 x 208 x  16 ->  208 x 208 x  32 0.399 BF
   3 max               2x 2/ 2    208 x 208 x  32 ->  104 x 104 x  32 0.001 BF
   4 conv     64       3 x 3/ 1    104 x 104 x  32 ->  104 x 104 x  64 0.399 BF
   5 max               2x 2/ 2    104 x 104 x  64 ->   52 x  52 x  64 0.001 BF
   6 conv    128       3 x 3/ 1     52 x  52 x  64 ->   52 x  52 x 128 0.399 BF
   7 max               2x 2/ 2     52 x  52 x 128 ->   26 x  26 x 128 0.000 BF
   8 conv    256       3 x 3/ 1     26 x  26 x 128 ->   26 x  26 x 256 0.399 BF
   9 max               2x 2/ 2     26 x  26 x 256 ->   13 x  13 x 256 0.000 BF
  10 conv    512       3 x 3/ 1     13 x  13 x 256 ->   13 x  13 x 512 0.399 BF
  11 max               2x 2/ 1     13 x  13 x 512 ->   13 x  13 x 512 0.000 BF
  12 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  13 conv    256       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 256 0.089 BF
  14 conv    512       3 x 3/ 1     13 x  13 x 256 ->   13 x  13 x 512 0.399 BF
  15 conv    126       1 x 1/ 1     13 x  13 x 512 ->   13 x  13 x 126 0.022 BF
  16 yolo
[yolo] params: iou loss: mse (2), iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.00
  17 route  13                                     ->   13 x  13 x 256
  18 conv    128       1 x 1/ 1     13 x  13 x 256 ->   13 x  13 x 128 0.011 BF
  19 upsample                2x    13 x  13 x 128 ->   26 x  26 x 128
  20 route  19 8                                   ->   26 x  26 x 384
  21 conv    256       3 x 3/ 1     26 x  26 x 384 ->   26 x  26 x 256 1.196 BF
  22 conv    126       1 x 1/ 1     26 x  26 x 256 ->   26 x  26 x 126 0.044 BF
  23 yolo
```

Figure 4. Tiny-YOLO Architecture

As shown in the figure, tiny-YOLO only has 24 layers in total and contains two yolo layers instead of 3 as compared to YOLO-V3.

4. Experiment Results

For our sub-task, we split our whole processed single bricks data into 70% training set, 20% validation set and 10% test set. For our object detection task on composite LEGO objects, we show the YOLO prediction results on validation data and display real-time object detection results on new composite models with used bricks.

4.1. Traditional Machine Learning Method on Single LEGO

The training results using the logistic regression model can be seen in Figure5. We also experimented with different sets of hyper-parameters but it turned out that this simple machine learning method is not very sensitive to hyper-parameter changes, as no other set of hyper-parameters would give a significantly higher training score. We observed that if images are resized to a higher resolution, say, 256 by 256 pixels, the validation accuracy of the model are actually decreased. We attribute this decline of performance to the problem of overfitting. When the resolution is higher the model tends to overfit the training set and therefore act worse on the development set. Also we noted that although the

training accuracy keeps going up, the validation accuracy is upper-bounded to around 0.33.

Model	Accuracy	Loss
Logistic Regression	0.3304	0.0318

Table 1. The validation accuracy and validation loss on the Single LEGO brick task using logistic regression.

4.2. Deep Learning on Single LEGO

We experimented with several deep learning models on the single LEGO brick dataset. Note that our initial training results of the transfer learning model in 5 are high, but the validation results are inferior to those of CNN we trained from scratch. We think the reason is that the resnet18 architecture has a lot more model parameters than the CNN we built. Hence the fine-tuning process of the pre-trained resnet18 model might need more time. Then we increased the training steps from around 6k to 17k and obtained a training accuracy of nearly 1.00, as shown in Figure.5. However the validation accuracy of this model is only 0.92. Then in order to solve the overfitting issue we randomly flipped and cropped the training images to create more data for training. Then the model performance on the validation set grows to around 0.96 as shown in Table 2.

Model	Accuracy	Loss
Feed Forward Neural Networks	0.6646	0.0084
Convolutional Neural Networks	0.7084	0.0031
Transfer Learning on ResNet18	0.9647	0.0015

Table 2. The validation accuracy and validation loss on the Single LEGO brick task using different deep learning models.

Figure 5. Training accuracy curves for the four models on sub-task

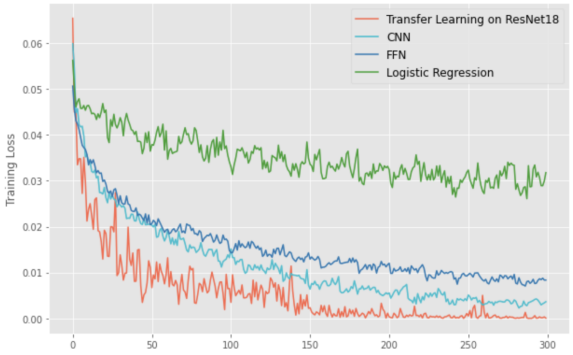


Figure 6. Training loss curves for the four models on sub-task

4.3. YOLO on Assembled LEGO

Model	Intersection over Union	Loss
YOLO-v3	0.7309	0.9252
Tiny YOLO-v3	0.6169	1.8613

4.3.1 YOLO Data Preprocessing

Our Dataset includes 20 different assembled lego model, using from 37 kinds of lego bricks. We Split the data set into 8:2 of training and validation set. We use a script to label all the image data with bounding box, and output the label in YOLO format, the label format is a text file that include Class label, x,y,width, height in for every bounding box in the training image.

4.3.2 YOLO Data Augmentation

1. Pre-train Stage Augmentaion: Since we are using different combination of lego bricks from the 37 classes of bricks, it would improve the learning performance since bricks in the image are overlapped in different ways and also in different orientations, viewing angles.

2. Training Stage Augmentation: In order to prevent over-fitting and increase the generalization of the model, we also tuned some augmentation parameters in the YOLO to augment the image data in different ways during training.Parameters such as jitter, hue, exposure and angle were taken into consideration to best fit our training data set properties.

4.3.3 Hyper-parameter Tuning

Considering YOLO in different learning rates and training batch numbers, we use a step policy and scale strategies to view how the YOLO models perform under these parameters.

YOLO v3	IoU	Loss
200 batch, lr=0.01	0.3212	1271
600 batch, lr=0.01	0.5702	513.37
800 batch, lr=0.01	0.6733	85.69
1000 batch, lr=0.001	0.7129	43.53
2000 batch, lr=0.001	0.7309	0.9252

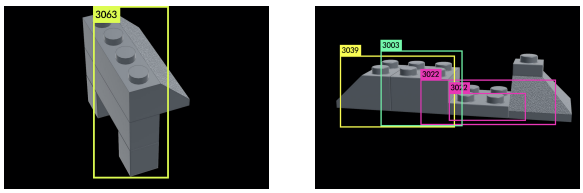
Table 3. Hyperparameter tuning of YOLO v3

Tiny YOLO v3	IoU	Loss
200 batch, lr=0.01	0.3374	36.25
600 batch, lr=0.01	0.5450	4.29
800 batch, lr=0.01	0.4002	3.94
1000 batch, lr=0.001	0.5304	3.59
2000 batch, lr=0.001	0.5950	2.17

Table 4. Hyperparameter tuning of Tiny YOLO v3

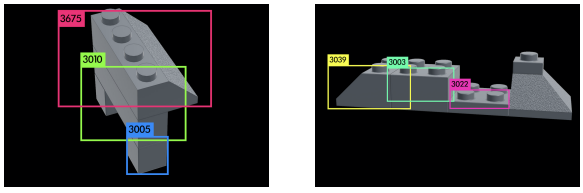
4.3.4 Prediction Results

Below shows the prediction results of Tiny YOLO-v3 and YOLO-v3 on the same pictures. As the results, we can see that Tiny YOLO-v3 have relatively more missing prediction and wrong prediction phenomena. Moreover, in Prediction 2, although both models predict the correct bounding boxes, however, the bounding box in the Tiny YOLO-v3 has a much larger size than the truth label, which might result from the bounding box error not converging enough but the class prediction error converging faster. Generally, YOLO-v3 performs better than Tiny YOLO-v3 on our dataset.



(a) Tiny YOLO-v3 Prediction 1 (b) Tiny YOLO-v3 Prediction 2

Figure 7. Tiny YOLO-v3 prediction results



(a) YOLO-v3 Prediction 1 (b) YOLO-v3 Prediction 2

Figure 8. YOLO-v3 prediction results

4.3.5 Mission impossible by YOLO-v3

Even though YOLO-v3 performs much better than Tiny YOLO-v3, it still has limitations on LEGO object detection. As shown in Figure 9, from the view of bottom, the detection performance is not very good. This is because there are billions of LEGO brick types but lots of them have exactly the same view from the bottom angle. Specifically, in Figure 9, there are some overlapping predicted areas which have multiple predictions for one brick. To solve this problem, we might need to make quality control on training data in the future in order to make sure only images that are human recognizable are used.

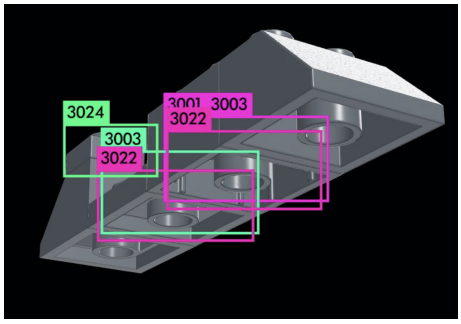


Figure 9. Bad result sample of YOLO-v3

5. Analysis and Discussion

Below are some analysis and discussion of our experiment results.

5.1. Sub-Task Models Hyperparameter Sensitivity Analysis

After tuning hyperparameters on our sub-task models, we achieved relatively high test accuracy on 50 classes single brick classification. During the training process, we found that deep learning methods on the specific dataset are very sensitive to hyperparameter choices like batch size and learning rate. For example, appropriate selections of learning rate and batch size in CNN and transfer learning on ResNet18 can lead to high training accuracy up to 97%. Inappropriate choices will just get about 70% training accuracy. Also, by randomly flipping the input data we improve our validation accuracy results from 70% to above 90%.

5.2. YOLO Training-phase Configurations

Jitter = 0.2 The Jitter parameter is used to control the crop percentage of the image, since our model already include bricks overlapping each other, if we increase the Jitter too much, some bricks might be cropped, which

might affect the training performance, therefore we decrease the Jitter to 0.2.

Angle = 10 Since our training image data are pretty limited, so we would like to increase the angle parameter, which would randomly alter the angle of the image to create the effect of different orientations and view angles of the lego model.

Saturation = 1.5, Exposure = 1.5 Both parameters are used for controlling the light augmentation effects. Since our light source of testing image are mostly the same, so we decided to maintain the default values, not to increase them.

Hue = 0.5 Since we are using gray bricks to assemble the Lego models, we would want the hue to have a larger value, in order to randomly modify the RGB values in the images.
We only use gray bricks to see if we could generalize the model to be able to detect the bricks even with slight different in colors, textures and orientations.

Momentum = 0.9 The optimizer used in the YOLO v3 is the Momentum optimizer, we didn't alter the default value 0.9. Although using momentum usually takes longer training time to converge compared to the Stochastic Gradient Descent applied in traditional Convolution Neural Network, however, the way it modifies the learning rate would be beneficial when training complicated data set, which in our case, training 37 classes of bricks. Learning rate: We increased the starting learning rate to 0.01 and use the step policy to modify the learning rate with a scale of 0.1 when the iteration reaches 1000 and 2000. The max batches was set to default 6000 batches, however, the model usually converges when the iterations reaches 3 4000 iterations.

Anchors Design Anchors is a feature that's been implemented in the YOLO v3 and has similar logic as how the Faster-RCNN's anchor works. There are 9 anchors in YOLO v3 to choose from, if the anchor is too big, then small objects will be hard to capture while if the anchor is too small, then big objects will be hard to detect, therefore, in different yolo layers, the different scale of anchors were selected to improve the model on detecting multi-scale features.

Loss functions

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} 1_{i}^{\text{obj}} \sum_{c \in \text{ classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2
\end{aligned}
$$

As we can see above, the first two lines of the loss function is the bounding box error, while the second portion of the loss function is the class predictions error and confidence error. Using logistic regression to predict class instead of softmax could improve the multi-label classification performance. A way to modify the loss function to stabilize the training performance is to increase the bounding box error coefficient and decrease the confidence score error coefficient.

5.3. YOLO-v3 and Tiny YOLO-v3 Comparison

5.3.1 Model Learning Speed - *Shallow vs. Deep*

Since Tiny YOLO has much less convolutional layers (around 24 layers) compare to the YOLO v3 (106 layers), it doesn't need much training time to achieve a decent result. If we take a look at Table 3 and Table 4, it is obvious that in the first 600 batches of training, Tiny YOLO v3 already achieve an average loss as low as 4.29 while the YOLO v3 is still as high as 513.37, which is nearly 100 times more than the Tiny YOLO v3. The reason might be due to the fact that it takes much longer time to train when using more complicated and deeper learning model.

5.3.2 Training Performance- *Gradient Vanishing*

We know that gradient vanishing happens when the hidden layers increase, therefore, in YOLO v3, residual blocks are implemented to avoid such problems and even improve the training performance as iterations increases. In the Tiny YOLO v3, we may see that the average loss is stuck in the range of 2 to 4 as the iterations

increases since 24 hidden layers is still a pretty big number of layers, and if the loss is lowered to a certain point (2 to 3 in our case), the weights might not learn effectively. The vanishing gradient issues might also emerge as iterations increases, and since there are no residual layers or additional mechanism to tackle such issue, the YOLO v3 would beat the Tiny YOLO performance-wise eventually.

5.3.3 Bounding Box Predictions - *Multi-Scale Detection Issue*

If we take a look at Figure 7 and Figure 8, the class prediction result on the (b) lego models from both YOLO models are all accurate, however, the bounding boxes generated by Tiny YOLO-v3 has much larger sizes than the ground truth label bounding boxes, which means the bounding box errors didn't converge as good as the class prediction errors. The YOLO-v3 generated much reasonable sizes of bounding boxes.

Possible reasons might be due to the effect of the yolo layers in the YOLO-v3 model. In layer 82 ,94,106, which are also called yolo layers, selects different scales of anchors to generate bounding boxes, in other words, they could effectively detect or extract features in different scales and predict a more reasonable height and width of bounding boxes compare to Tiny YOLO-v3. Tiny YOLO-v3 also has yolo layers but only 2, and the anchors choice is less than the YOLO v3 model, which is why it might predict a pretty accurate location of the object, but the bounding box size predictions are less accurate.

5.4. Challenges and Possible Improvements

We have use both YOLO v3 and Tiny YOLO-v3 to train on the Lego data set, and did data augmentation during both pre-trained stage and training phase, hyperparameter tuning while training using different iterations. Mostly, after 1000 to 2000 iterations, both models can predict some bounding boxes correctly from the static images or video stream ; However, there is still some challenges that have been restricting the model's performance. Here we provide some possible future improvements based on our prior tuning and training experiences.

5.4.1 Anchor Training Problem

Since the Default Anchor was trained on the Coco dataset, if we pre-design and retrained our Anchor before training on the Lego dataset, then the bounding box error will be smaller in the first few iterations and converge to the minimum faster, which would then increase the training performance and speed.

Possible approaches include using K means cluster with the IOU as a distance metric, to train the anchor size before implementing the model. Since the larger the IOU, the less the Bounding box error is, therefore inverse IOU may be a good distance metric to apply K means. Figure 10. shows an example of bounding box size misprediction.

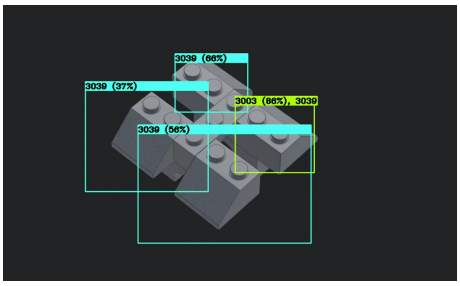


Figure 10. Wrong bounding box size of YOLO-v3

5.4.2 Image Resolution Problem

Although our image seems to be clear in human eyes, but the resolution of 416*416 might not be good enough if we are dealing with smaller bricks or small edges, doubling the size of the image height and weights is a feasible approach but would lead to a longer training time.

5.4.3 Model Complexity

In the first 1000 iterations we may see that Tiny YOLO converges faster than the YOLO v3, the possible reasons might be due to the complexity of the YOLO v3 model. Since the Feature extraction architecture and the convolutional layers and structure are far more complicated Tiny YOLO, therefore it might take longer time to update all the weights in each layer to a point which could generalize the model, if given enough iterations (Batch size =64, iterations ¿3000), We could see the YOLO v3 has the average loss converge to a much lower value, with higher IOU, while Tiny YOLO would be stuck in a range and couldn't achieve a smaller loss.

5.4.4 Optimizer Learning Problem

In the YOLO series model, Momentum is used as the optimizer, the advantage is it alters the learning rate base on previous learning experience, which could be beneficial when dealing with complicated model, however if

the model is not complicated enough, it may lead to the optimizer learning much slower than Stochastic Gradient Descent optimizer. Therefore the value of the Momentum optimizer should be tuned carefully in order to boost the performance.

5.4.5 RGB Color Channel Problem

Since we are using gray scale image, but the model was designed for taking RGB color channel into training, therefore, when the testing image has some slight color changes, the model would be more sensitive to the change, which might increase the bounding box prediction error and lower the IOU and confidence score of the specific class.

Possible solution is to restructure the model layers to convert all RGB images to gray scale images , in other words, there will be only one channel for each image, this way, color won't be consider as such an major role when doing feature extractions and bounding box predictions.

5.5. Future Steps

Based on what we have got so far, we can detect some different LEGO bricks in composite LEGO models. In order to extend our model, as well as improve the IoU and reduce the loss further, we can collect more composite LEGO picture data with more brick types. In addition, if there are lots of time available, we can try Mask-RCNN to improve the detection accuracy. Detecting LEGO bricks completely in every LEGO models is an impossible mission, because there are over 400 billion bricks in the LEGO world and plenty of them are exactly the same when seen from specific angles. Therefore, what we can do and should do in the future on the LEGO detection is performing quality control on the data collection process. Make sure only images that are human recognizable are used.

In the LEGO world, there are thousands of elements that can be used to assemble composite LEGO objects. However, there are only 50 different pieces in our single LEGO brick image classification task and 37 different pieces in our composite LEGO object images. The model may have the potential to learn more latent representations of these images if more images of LEGO pieces are added to the dataset. Although transfer learning results suggest that we have captured the latent representations in the single brick classification task alone, the YOLO model on detecting and classifying LEGO objects in composite objects shows that there is still space for more improvement in this kind of images. In addition, as this is only an initial attempt to detecting and classifying images in the LEGO world, all of the images in the dataset have the same color. In reality, nevertheless, there are over 50 different colors in LEGO pieces. In the future, LEGO pieces with different colors should also be added to the dataset.

Also, one should perform quality control on the data collection process of these LEGO images. Some of the training images of the LEGO objects are shot from the bottom and thus provide little information of the actual LEGO pieces. Although detecting an object partially hidden from the camera is one of the inevitable tasks of the model, we do not believe a model can make predictions well when there is just too little useful information. As a result, one should make sure that most, if not all, of the images in the dataset are shot from valid angles.

References

[1] I. S. A. Krizhevsky and G. Hinton. Imagenet classification with deep convolutional neural networks, 2012.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[3] J. Hazelzet. Images of lego bricks, 2020. data retrieved from Kaggle, `https://www.kaggle.com/joosthazelzet/lego-brick-images`.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[5] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009.

[6] A. F. Joseph Redmon. Yolo9000: Better, faster, stronger, 2016.

[7] R. G. A. F. Joseph Redmon, Santosh Divvala. You only look once: Unified, real-time object detection, 2016.

[8] Y. LeCun, Fu Jie Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–104 Vol.2, 2004.

[9] H. Lee, R. Grosse, R. Ranganath, and A. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. page 77, 01 2009.

[10] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[12] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu. A survey on deep transfer learning, 2018.