Where are my LEGOs?

Convolutional Neural Networks for Brick Detection

André Charneca (338621), Bartul Kovacic (342814), Tomás Feith (342553)
Supervisor: Paolo Ienne - School of Computer and Communication Sciences
Machine Learning (CS-433), EPFL, Switzerland

***Abstract*—To study the feasibility of a LEGO detection app, we created a pipeline for the large-scale generation of synthetic data. A dataset consisting of 2,000 synthetic images, 72 real images, 722 negative samples, and 16,304 augmentations of these were used to train a YOLOv4 model for object detection of 10 distinct LEGO pieces. We found the model performed well for inference of synthetic data (mAP∼98% and IoU∼86%), but the performance on real data had low to moderate success, due to the simulation-to-real gap. It was concluded that, using this methodology, such an app would not be fully feasible, but we present several directions to explore in future research.**[1]

I. Introduction

Have you ever had the fun of building a LEGO set taken away from you by the enormous amount of time it takes to find each piece? The objective of this project is to train a deep learning model to correctly identify specific LEGO bricks from video or images. This project will serve as a proof of concept for an application in which the user would input the ID of the brick they want to find, point the camera at the pile and the application would display the piece's location in the camera feed. 10 different LEGO bricks were considered for this object detection task (Figure 2).

Our contributions are summarized as follows:

1) We create a large training dataset consisting of synthetic data, generated using Unity's Perception [1] module;
2) We develop a pipeline to improve the data, using a series of data augmentation techniques and adding real images and negative samples scraped from the web (see Section II-C for background on the choices for the pipeline);
3) We test the feasibility of a LEGO detection application, using a YOLOv4 object detection model to detect specific LEGO pieces, with low to moderate success.

II. Related Work

A. Object Detection Models

There is a large number of object detection models used in current research. These can be mainly divided into two categories: one-stage detectors (like YOLOv4, from Bochovskiy *et al.* (2020) [2], or RetinaNet, from Lin *et al.* (2018) [3]) and two-stage detectors (like Mask R-CNN, from He *et al.* (2018) [4]). The key difference between these is that, while a one-stage detector can detect the bounding boxes and classify them in one pass through the network, a two-stage detector needs to pass through the network twice, once to detect the boxes and once to classify them. This difference means that one-stage detectors have a higher inference speed, while two-stage detectors have higher localization and recognition accuracy, as is analyzed by Soviany and Ionescu (2018) [5].

[1]All the code and materials can be found at https://github.com/tomas-feith/LEGO-Finder.git

B. Synthetic Data

The crux of most object detection projects is the difficulty of getting training data. This data has to consist of thousands of images and accompanying files with the bounding boxes and labels of the objects in each image. Manual labeling is possible, but not with datasets of this size. Synthetic generation of training data, using virtual models of the desired objects is more efficient in this case.

In recent years, some frameworks for generating realistic synthetic data have been developed for specific tasks, such as aircraft detection (Liu *et al.* (2020) [6]) and healthcare sensors (Dahmen *et al.* (2019) [7]). The methods for data generation are specific to each task, but one thing is common among most of these: models trained only with synthetic data do not perform well with real data. However, as was shown by Borkman *et al.* (2021) [1], the usage of a mix of synthetic and real-world images can boost accuracy. They found that using only 760 real-world images lead to a mean Average Precision (mAP) of 0.450 ± 0.020, using only 400,000 synthetic images lead to an mAP of 0.381 ± 0.013 but using simultaneously the 760 real-world images and the 400,000 synthetic ones lead to an mAP of 0.684 ± 0.006. This stems from the fact that, although synthetic images allow for much larger data sets than real images at a much smaller cost, they are not perfect matches to their real-world counterparts and real images are needed to close that gap.

C. Data Augmentation and Negative Samples

Data augmentation is a widely used technique to increase the quality and variance of a training dataset. Several techniques, and the reasoning behind them, were explored by Naveed (2021) [8]. These methods mainly operate by applying combinations of mixing and deleting images. Mixing and deleting works because by removing and changing portions of the object the model learns how to identify it from partial views, thus understanding the overall structure.

However, it should be pointed out that the methods presented by Naveed were designed for object recognition, not detection, and the latter is the relevant one for this project. The conversion of some of these methods from recognition to detection is not trivial, and only a selection of them could be used.

Another way of increasing the dataset size is using Negative Samples. These are images of objects that the model wasn't trained to identify . However, these tend to be assigned to a random class with high confidence. Li *et al.* (2020) [9] studied an approach called Negative-Aware Training (NAT), which purposefully introduces negative samples into the training dataset. They found that this leads to a more robust network and better performance on negative samples in testing. Given that the introduction of negative samples into the training dataset is a cheap process as there is no need for labeling, NAT is a simple and worthwhile additional method to implement.

III. Models and Methods

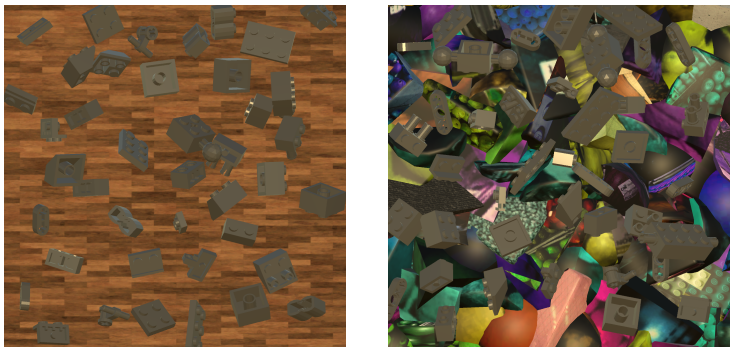
A. Training Data

1) Generating Synthetic Data: Due to the sizeable amount of LEGO assets that exist online, the majority of the training data used was synthetically generated.

This was done using the software Unity with the Perception package [1]. This package allows for the automatized generation of training images, along with the labeling and bounding box information. It also allows for randomized parameters in each image, which in this project were: brick position and rotation, background, lighting intensity, and hue.

For the identification of the parts (through their LEGO ID), Rebrickable [10] was used, and the CAD models were extracted from LDraw [11].

Two separate batches of 1000 images featuring 23 different pieces (10 of which will be used as classes for the model, see section III-A5) were generated. The first consists of the pieces in front of different floor textures as backgrounds. The second batch attempts to make the model more robust to background variations, with a completely random texture set as background, that is distinct in each image. Examples are shown in Figure 1.



(a) Floor background. (b) Random background.

Fig. 1: Synthetic images generated with Unity Perception [1].

2) Data Augmentation: To increase the variance of the dataset, several techniques of data augmentation were applied: color jittering, cutouts, gaussian blur, gaussian noise, gray-scale, and Laplacian filter. These allowed to increase the dataset 9-fold[2], making the model more robust and forcing it to learn the overall structure of the pieces. Figure 5 shows an example of the data augmentations.

3) Negative Samples: Given the final model is meant to identify specific LEGO pieces from a video of a large set of pieces, it is likely that in a large number of frames, the desired piece will not be visible. As such, it is necessary to train the model so that it does not identify a piece when none is present. For that, NAT was used, as it leads to the reduction of misidentifications in the absence of a class. For this purpose, images that did not contain objects of any class were added to the training set. Specifically, since most LEGO pieces will be searched for on a flat surface, we considered a mixture of wooden, stone, and metal textures, as well as geometric shapes like squares, triangles, and blocks, to prevent over-fitting to simple shapes.

[2]Cutouts were applied randomly to every bounding box, and this was done 3 times per image.

4) Real Images: Real images of the LEGO pieces, including 13 pieces that were not used for detection, were included in the dataset. The amount of these was low (around 70 images), due to the manual labeling and bounding box identification that had to be performed. However, as stated in Section II-B, a small number of these images can be very beneficial.

5) Pieces Used: The synthetic images generated consisted of 23 pieces. However, for training, only a subset of these was considered, due to the computational resources and time available. The pieces used, as well as their IDs, are displayed in Figure 2.

Fig. 2: Pieces used for training the model.

B. Models Considered

Several models were considered for this project. The three main contenders were YOLOv4, RetinaNet, and Mask R-CNN.

From these 3, Mask R-CNN was discarded because it was too computationally expensive, as it is a two-stage detector. This makes it inadequate for real-time detection, running at 5 FPS with a high-end GPU. Furthermore, as the labeling needs to be with the true shape of the object, and not just a bounding box, it would not be possible to label a big enough training set without a tremendous investment in man-hours, since Unity Perception can only draw bounding boxes.

For the two remaining options, Bochkovskiy *et al.* [2] showed that YOLOv4 leads to a better Average Precision (AP) score when compared to RetinaNet (43.0% vs 37.3%), with only a slight decrease in speed (31 FPS vs 31.3 FPS). Therefore, YOLOv4 was the model chosen for this project.

C. Accuracy Metric

There are two main problems regarding an accuracy metric for video detection, the final goal of this project. Firstly, there is a lack of a labeled video testing set. Any attempt to label a video would entail an enormous amount of frames, rendering this unfeasible in the time frame of this project.

Secondly, there isn't a standard way of measuring the accuracy of video object detection. Reviewed by Zhu *et al.* (2020) [12], there are several metrics proposed for accuracy measurement, like mean Average Precision (mAP) or Average Delay (AD), but there isn't one standard way of measuring it.

The main problem here is the first, and the lack of a labeled testing video completely prevents a quantitative measurement of accuracy in videos. As such, all that can be performed is a qualitative measurement of the testing videos. However, testing can still be done on the static images used for training, using the standard metric mAP.

D. Training

The training of the YOLOv4 model was done in several stages. To reduce training time, a transfer learning approach was used. Pre-trained weights were loaded initially (available on YOLOv4's GitHub[3]), and these weights allow for better object detection right away, despite not being trained specifically for LEGOs.

At the end of each training stage, the model's weights were saved, and loaded at the beginning of the next stage. The stages were:

- *Stage I - Floor backgrounds*: 1000 synthetic images with floor textures as backgrounds (see Figure 1a), plus 8000 augmentations (see Section A).
- *Stage II - Random backgrounds*: 1000 synthetic images with fully randomized backgrounds (see Figure 1b), plus 8000 augmentations.
- *Stage III - Real images and negative samples*: 76 real images of LEGO pieces, plus 304 augmentations (here, the augmentations were gaussian noise and blur, gray-scale, and color jitter). 722 negative samples of images randomly scraped from the web, including floor textures, actual bricks, geometrical shapes, etc.

IV. Results

A. Accuracy Evolution During Training

To monitor the model evolution over each stage, the Intersection over Union (IoU) and mAP were measured at the end of each stage. To measure them, the data was divided into training and testing with a 70/30 split. At each stage, the test data consists of a subset of the images used in that stage alone. The results obtained are below.

	IoU (%)	mAP (%)
Stage I	90.56	99.65
Stage II	83.22	97.97
Stage III	86.43	98.83

TABLE I: Values of mAP and IoU on test data, for each of the training stages.

B. Inference Tests

After training the model, inference was performed on 3 videos (links available in the GitHub repository), with more pieces than the ones the model is trained for. Figures 3 and 4 show some results.

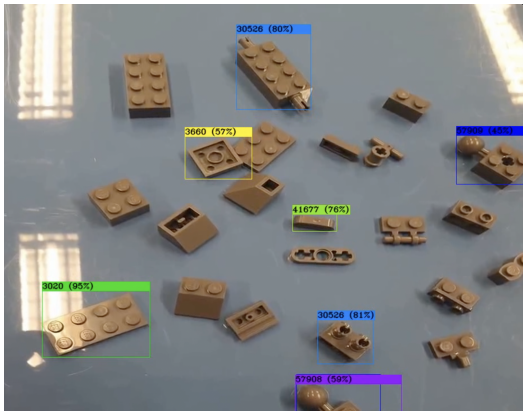


Fig. 3: Test on a real image. Some used pieces aren't detected and the pieces 30526 (top center) and 3660 (center) are wrong.

[3]https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects

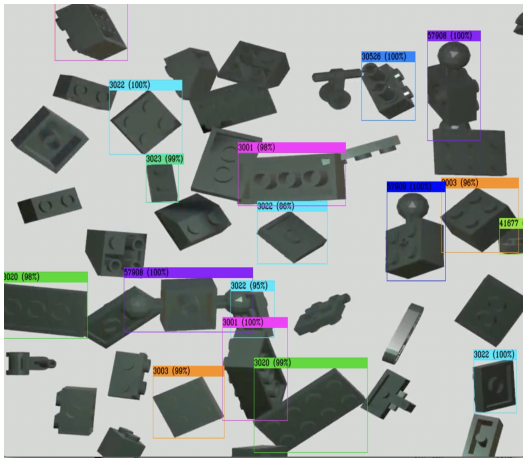


Fig. 4: Test on a synthetic image. Some used pieces are not detected, but all detected pieces are correct (maximum recall).

V. Discussion

The results in Figure 3 show moderate success in the correct detection of the pieces. The model misses 5 out of the 11 present used pieces. Out of the 7 detected pieces, 2 of them are not part of the ones chosen for this model, but the other 5 are correctly identified. In the video where it's taken from[4], the labels change frequently on some bricks, indicating that some of the used pieces might be too similar to each other. However, this is a problem with detecting LEGO bricks in general, as most are just slight variations of each other, with different sizes or features.

Another characteristic of LEGO bricks that makes the learning process more difficult is the fact that, at certain angles, some pieces are indistinguishable (for example, pieces 3001 and 3020, or 3003 and 3022). This renders the correct detection of these pieces down to luck, at these angles. This can be observed in certain frames of this video[5], taken with a much larger amount of bricks.

In Figure 4 the results are much more satisfactory. In this video[6], we see that the labeling is much more consistent, and the confidence of each detection is higher than in the previous ones. This makes sense, based on the results in Table I. The values achieved of mAP and IoU are extremely satisfactory. The slight decrease from Stage I to Stage II is due to the usage of more complex images on Stage II, with the random background. The results for Stage III aren't particularly informative, as they were mainly for negative samples, to increase robustness. Nevertheless, mAP around 98% and IoU around 86% is very high, and if they translated directly to the real images then it would be a clear success. The fact that they didn't implies over-fitting of the model to synthetic data, and that it is not realistic enough. The sim-to-real gap could not be bridged. The bottleneck for this might have been the type of file the LEGO models were available in (`.dat` files), which are very rudimentary and don't allow for realistic shading and lighting in Unity.

VI. Summary

A YOLOv4 model was trained for object detection of 10 different LEGO pieces, using a dataset of synthetic images with several randomized parameters (created in Unity, using the Perception package [1]), negative samples, and a small number of real images

[4]drive.google.com/file/d/1-G2dfGAPo0TkKUq2jBjldczDaIKo6zBs/view
[5]https://drive.google.com/file/d/1BvhTnzwFEv9b5cjvnP5Z6dAHgEhv_TbZ/view
[6]drive.google.com/file/d/1-EX9BB6Jv2JKJYoVTnefhKa7LYRKjGY5/view

of the pieces. Data augmentation techniques were applied to these images to improve the robustness of the model (see Section A). While the results were very satisfactory on synthetic data, the model didn't perform as well on real data, with low to moderate success, depending on the usage.

VII. Application Feasibility

This project was meant to serve as a proof of concept for a LEGO identification application. However, the results seen before are not very promising, and there still seem to be several hurdles to surpass before this app is feasible.

In terms of accuracy, even though the model performed very well on synthetic data, it did much worse on real images of LEGO pieces. Since, ultimately, it is meant to detect the latter, it is not yet working well enough to be useful. Fortunately, there are several options to explore that can improve this (discussed in section VIII). As discussed in Section V, the fact that most pieces look alike, and that some are identical from certain angles, will make this application only useful for some distinct-looking pieces, or require the user to get different angles or shuffle the pieces.

Another major obstacle to the potential development of the app is the computational resources it requires. This proof of concept was trained for 10 pieces, on a system with 4 NVidia Xp GPUs and each stage of the training took about 10h. However, for it to be reasonably useful it would need to be trained for hundreds of pieces. This would be very computationally expensive, but it could be conducted offline and only once since for inference only the weights are needed and the app could come preloaded with them. The major problem is that even for inference YOLOv4 needs a GPU, if it is to run in real-time. Most mobile devices can't meet this requirement, which would render the app virtually useless. It is possible to use Tiny-YOLO, which is significantly faster, but this leads to worse accuracy and, even like this, it is not capable to achieve real-time inference solely on CPU power.

Overall, the results show that while it is possible to get some results using YOLOv4 for LEGO detection, there is still a lot of work to do before it can be turned into a real app for public use.

VIII. Next Steps

Several directions can be explored in future research. These mainly lie in improved synthetic data, more real-world images, and improved data augmentation.

1) Improved Synthetic Data: One very necessary improvement is the quality of the synthetic data generated, to reduce the sim-to-real gap. Namely, in terms of the texture and smoothness of the objects.

The texture of the objects needs work because the shadows and lighting are too flat, as Unity generates one uniform shade per surface. This is not how objects behave in reality, and so to achieve better results this would have to be corrected.

The smoothness of the objects also needs to be improved, specifically for curved surfaces, which can be seen in Figure 5i where the polygons are visible for the sphere near the center of the image. This needs to be addressed in future work because it might be misleading the model into searching for this kind of polygonal pattern in objects, which will not exist in real-world bricks.

One final point about the synthetic data generated is regarding the size of the pieces. In this project, several parameters about the pieces were randomized to achieve maximum variance: the piece angle, the lighting color, intensity and position, the background. However, the research conducted points to the fact that YOLO is also sensitive to the piece size. So in an eventual improvement on this work, it is recommended to add the piece size as another randomized parameter for the synthetic data, as this will most likely improve detection accuracy.

2) More real images: The usage of real images extracted from the web has its caveats. It requires manual filtering of which images are good enough to be used, and it leaves the research dependent on the availability of images on the web. These can, and should, be avoided by using images taken during the project and, by splitting the images taken, it is possible to create a training set and a good testing set.

As was stated before, the main problem with this is labeling and bounding box identification. Manual labeling is an extremely time-consuming process, especially if there are many pieces in one picture. The best alternative to this seems to be crowd-sourcing, via tools like Amazon's Mechanical Turk [13].

So, if there is funding and/or manpower for labeling real pictures, it is highly recommended to do so in future research.

3) Improved Data Augmentation: Only a subset of the methods introduced by Naveed (2021) [8] was used because, as explained in section II-C, some methods didn't have a trivial conversion from classification to detection. Nevertheless, it can still be done. For example, Bouabid and Delaitre (2020) [14] proposed a MixUp method for object detection, which generates a new image by mixing pixel values of two randomly selected images

This should be implemented in future research, as it is stated to improve robustness and generalization power. Similar to MixUp, there may be other versions of the methods explored by Naveed which can be explored, thus improving the quality of the training dataset.

References

[1] S. Borkman, A. Crespi, S. Dhakad, S. Ganguly, J. Hogins, Y.-C. Jhang, M. Kamalzadeh, B. Li, S. Leal, P. Parisi, C. Romero, W. Smith, A. Thaman, S. Warren, and N. Yadav, "Unity Perception: Generate Synthetic Data for Computer Vision," 2021, arXiv:2107.04259.

[2] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 2020, arXiv:2004.10934.

[3] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal Loss for Dense Object Detection," 2018, arXiv:1708.02002.

[4] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," 2018, arXiv:1703.06870.

[5] P. Soviany and R. T. Ionescu, "Optimizing the trade-off between single-stage and two-stage object detectors using image difficulty prediction," 2018, arXiv:1803.08707.

[6] W. Liu, J. Liu, and B. Luo, "Can synthetic data improve object detection results for remote sensing images?" 2020.

[7] J. Dahmen and D. Cook, "Synsys: A synthetic data generation system for healthcare applications," *Sensors*, vol. 19, no. 5, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/5/1181

[8] H. Naveed, "Survey: Image Mixing and Deleting for Data Augmentation," 2021, arXiv:2106.07085.

[9] X. Li, X. Jia, and X. Jing, "Negative-Aware Training: Be Aware of Negative Samples," in *ECAI*, 2020.

[10] "Rebrickable API Documentation," https://rebrickable.com/api/v3/docs/, accessed: 12-21-2021.

[11] "LDraw.Org - LDraw.Org homepage," https://www.ldraw.org/, accessed: 21-12-2021.

[12] H. Zhu, H. Wei, B. Li, X. Yuan, and N. Kehtarnavaz, "A Review of Video Object Detection: Datasets, Metrics and Methods of video object detection; deep learning-based video object detection," *Applied Sciences*, vol. 10, p. 7834, 11 2020.

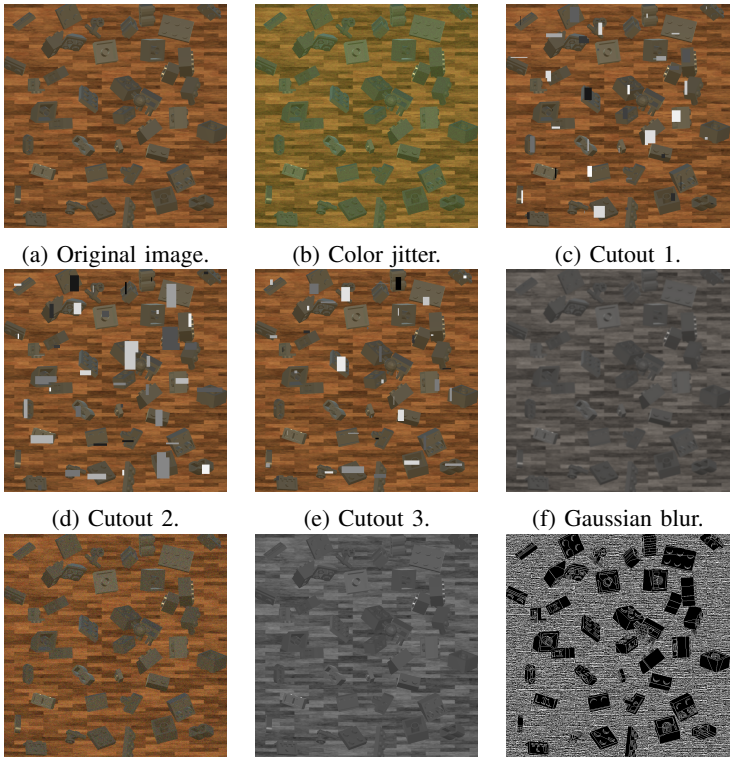
[13] K. Crowston, "Amazon Mechanical Turk: A Research Tool for Organizations and Information Systems Scholars," in *Shaping the Future of ICT Research. Methods and Approaches*, A. Bhattacherjee and B. Fitzgerald, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 210–221.

[14] S. Bouabid and V. Delaitre, "Mixup Regularization for Region Proposal based Object Detectors," 2020, arXiv:2003.02065.

APPENDIX

A. *Data Augmentation*

Below we present an example of the application of data augmentation to one of the synthetic images generated, with a floor-pattern background.



(a) Original image. (b) Color jitter. (c) Cutout 1.

(d) Cutout 2. (e) Cutout 3. (f) Gaussian blur.

(g) Gaussian noise. (h) Grayscale. (i) Laplacian filter.

Fig. 5: Example of the results of data augmentation.