Toward understanding chaotic sets of Legos

Martin Dahl, Antoine Escoyez, Arvid Linder
CS-433 Machine learning, EPFL

***Abstract*—YOLO trained on our own synthetic data is used to detect specific Lego bricks in a pile of Legos. mAP was improved by 23% on average compared to related work. A pipeline was created for generating synthetic data of Lego piles with bounding-boxes automatically labeled using 2D images of bricks and various backgrounds.**

I. Background

This project tackles a problem dear to the heart of engineers between the ages of zero and infinity: finding the last piece of Lego for your Millennium falcon. Having a pile of Lego bricks, a specific individual brick can be surprisingly difficult to find, especially when the Legos in the pile are uniformly colored. Some bricks are also difficult to distinguish having only small differences setting them apart as in Figure 2. An example of the messy situation is shown in Figure 1. In this report, a method is suggested for assisting in the task of finding individual Lego bricks in a messy batch of Lego bricks. This could be turned into a smartphone application for use when building with Legos: the user asks for a specific brick, aims the camera towards the pile, and lets the application work until it suggests a brick that could be what the user was looking for.

Figure 1. Chaotic set of Legos.

In a fashion the problem is paradoxical: a model needs to be trained to help humans detect pieces, but to train the model a large amount of data needs to be labeled which means pieces need to be detected by a human in the first place. This task is very time-consuming and expensive if labelers are paid. To tackle this without a large amount of resources for labeling, the dataset must be generated in a synthetic way. Nvidia forecasts that synthetic data will dominate over real data before 2030 [2], which confirms the struggle of finding enough data and time to label it. Synthetic data is being more widely used in other machine learning applications such as autonomous vehicles [3]. With an initial survey of previous projects attempting to detect Legos, generating data will be the biggest focus of the project. Several similar projects have made attempts of detecting Legos. The most famous is arguably the *universal Lego sorting machine* by Daniel West [4], however, the Lego detector in West's project only classifies Lego bricks one at a time under great lighting. More similar to this project is the Deeepwin repository [5] which uses *Mask R-CNN* and a synthetically generated dataset with 2000 images. The results were mixed having good accuracy for sparsely placed bricks but issues arose when the bricks were increasingly tightly placed, something that will be apparent in this project. State of the art detectors for general object detection such as *YOLO* [6] and Mask-R-CNN [7] (as used by Deeepwin) are available within handy frameworks. Since YOLO is know for its speed at inference time, we chose this model and further describe its pipeline in the following section.

Figure 2. Brick 3023 left of 32028. Taken from [1].

II. Method

A. Constraints

10 classes of bricks were included. These were selected as they are considered the most common bricks and yet it can be quite a challenge to differentiate some of them: 2540, 3001, 3003, 3004, 3020, 3021, 3022, 3023, 3039, 3660.

B. Data

The main part of building a good object detector is to find and label enough data to be able to train a model. There was no time to manually label a lot of images, taht is why the main approach was to build a synthetic dataset. This was performed by pasting cutouts of individual Lego bricks on a background.

First, images of bricks in different orientations were rendered using Maya and CAD models. This was performed using the same approach as in the Kaggle Lego dataset [8].

Examples of the result when rendering the brick images are shown in Figure 3. Thus, generating bounding-boxes for each brick in the images was simple.

Figure 3. **Left**: Kaggle dataset giving different angles of brick 3023. Note that these are a small fraction of all angles available and that the actual brick images only had one brick per image. **Right**: Cutouts of real bricks taken with real camera.

A major concern was making the model recognize bricks in real images after only training on synthetic data. Since the YOLO model used for detection was already pre-trained on real data, training on synthetic data mainly fine-tuned the model to the shapes of the Legos. However, to avoid over-fitting to ”deep” synthetic features some images of real bricks were added, an example of real brick cutouts is seen in Figure 3. These images were created by manually cutting out the bricks from images of Legos found on Google images. The resulting images were similar to the ones generated with CAD models but depicting real bricks acquired with a real camera. This was a time-consuming task, so the set of real images was kept to 20 instances per brick class on average.

Various backgrounds were selected in an attempt to bring diversity to the dataset. The goal was to avoid training the model on specific features in a background, while still keeping the collection of backgrounds simple. Two types of backgrounds were used: one without a lot of features to simulate for example a tabletop or a floor, and another with more colors and features to mimic a carpet or pile of unknown bricks.

Next, the real and rendered brick cutouts were pasted on the different backgrounds. In design of this process some things were of main concern: having a sufficient but not excessive overlap of bricks (and thus overlap of bounding-boxes), if placement should be random (uniform or biased to some area of the background), and finally if the brick sizes should differ. A very important aspect is also the distribution of the number of class instances over all images in the generated dataset, this distribution was kept uniform for all brick classes.

In the first set ”Set A”, bricks were pasted randomly on the background. The reasoning behind a random placement is considering if real Lego bricks are laying on a floor, they are most likely randomly distributed and may very well overlap each other. For this reason, the size of the bricks was also randomized (between 1/10 and 1/5 of the image size). This was done to mimic pieces at different distances. The result was a set of images with an unrealistic look as seen in Figure 4. However, the main goal was to create images

Figure 4. One image from the training set with a lot of color and diverse backgrounds.

where the desired features of the 3D-images (e.g. the shape from different angles) were given in diverse settings.

In the second set ”Set B”, only simple backgrounds were used and brick placement was performed in a fashion to guarantee evenly placed bricks across the background: the background was split into 25 smaller grids where each of them would contain zero or one brick. The coordinates for brick placement within each of these smaller parts were randomized. The maximum amount of pieces in these images was set to 25, giving a maximum of one piece in each grid. This resulted in images of many more pieces but with no pieces overlapping. Figure 5 displays this variant of synthetic data.

C. Data augmentation

To make the dataset more diverse and the model more robust image augmentations were performed. Both the bricks and the complete images (backgrounds with bricks) were augmented. Three augmentations were used for the complete images: 5x5-kernel blur, motion blur, and Gaussian noise with random intensities. Kernel blur was implemented in order to smooth the edges between inserted bricks and the background to in a sense mimic light diffraction. Motion blur to mimic effects from camera movement and finally Gaussian noise to remove the unrealistic look of completely uniform colors as seen in the Kaggle images in Figure 3. The generation of images resulted in a wide range of images from very clear to images where detection would be difficult even for a person.

In addition to augmentation of the complete images, the bricks were augmented before being pasted on the backgrounds. The rendered brick images from Maya were completely gray, so a problem in using those directly would be over-fitting the model to gray bricks. To make the model more robust to differences in colors the colors were shifted randomly for each synthetic brick. This still resulted in non-realistic-looking bricks, but with the prospect of reducing the model dependency on color for bricks. An example of a dataset-image with bricks shifted in colors can be seen in Figure 4.

Another approach was tested by simplifying the synthetic data to look more like the real test data. In this approach, all

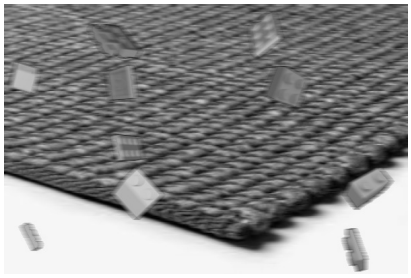

Figure 5. One image from the black and white training set.

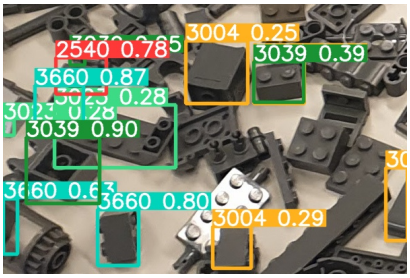

Figure 6. Inference when running YOLO trained on Set A.

Figure 7. Inference when running YOLO trained on Set B.

dataset images were converted to gray-scale with some Kernel and motion blur. The resulting images are exemplified in figure 5.

D. Model for detection and classification

Two models were investigated: YOLO and Mask-R-CNN as cited previously. The decision of using YOLO was motivated by 3 reasons:

- YOLO is designed to be fast and exists in a light version (v5s) which was used. Thus it is suitable for use on a mobile device.
- YOLO only requires labeling of bounding-boxes while Mask-R-CNN is designed to also predict masks of an object, something that is not required for the purpose of this project.
- YOLO is well known and does not need further explanation.

The models were trained on the YOLOv5s architecture using 100 epochs and a batch size of 32 for all datasets. Furthermore an 80,10,10 train,val,test split was used. The training ran on Google Colab Pro which trained on 8000 images (80% of 10000) for 100 epochs in 4 hours.

III. Results

YOLOv5s was run on two synthetic datasets created as described in previous section:

- **Set A**: Random placement of bricks on backgrounds, kernel blur of complete image, shifted colors. In total 10000 images with about 200,000 instances of bricks uniformly distributed over the 10 classes. Most images looked like the example in Figure 4.
- **Set B**: Placement of bricks restricted to a 5x5 grid on the background with no more than 1 brick per grid. Colors set to grayscale. In total 10000 images with about 140,000 instances of bricks uniformly distributed over the 10 classes. Most images looked like the example in Figure 5.

In Figure 6 and 7 the result of detecting with YOLO trained on Set A and Set B respectively is visible. The image used in these figures is completely independent of training, no part has been used in any way for training. For Set A the model becomes more conservative in its detections but also misses some clear (for a person) bricks such as the 3003 in the right side of Figure 6, something the model trained on Set B does not miss. However, Set B gives considerably more false positives such as the 3023's.

In Figures 8 and 9 the precision and recall can be found for YOLO trained on Set A and Set B respectively. It is clear that for Set B the precision reached saturation after only 10 epochs, while Set A required 50. For recall Set A never reaches full saturation but Set B reaches saturation after approximately 20 epochs. For Set B the saturation is at a level of 100% for both precision and recall, Set A only reaches about 95% for precision saturation. This means that Set A induces both false positives and negatives while Set B does not on considering test metrics. The mAP is 0.8 and 1.0 for Set A and Set B respectively as seen in Figure 11.

Observe also Figure 10 which depicts Legos detected with YOLO trained on Set A. In this case, the Legos were more sparsely placed and no bricks were missed by the model compared to in Figure 6 where at least half of the bricks in the image were not detected. The results were similar for Set B.

IV. Discussion

The results show that YOLO trained on Set A is conservative in detecting bricks, while YOLO trained on Set B is less conservative giving more false positives. The test accuracy is however significantly higher for YOLO training on Set B, which could be explained by the fact that bricks never overlap each other in this set, and thus the model can learn each brick instance. In Set A there are several random overlaps that creates unique instances every time the dataset

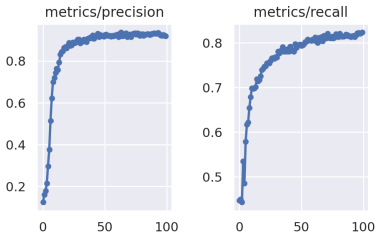

Figure 8. Metrics results for YOLO training on Set A.

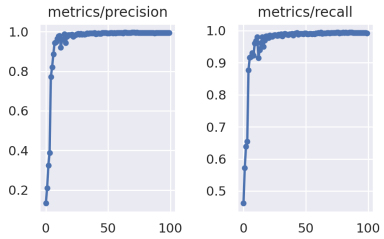

Figure 9. Metrics results for YOLO training on Set B.

is created. This leads to state a significant problem with the dataset: the synthetic dataset is created with a limited number of images of bricks in different orientations. The same brick is at risk of appearing both in the test and the training set which unfortunately was not accounted for in this project. To make the project mature for real usage on Lego bricks, more research must be done on generating convincing synthetic datasets for this application. Potentially the entire YOLO can be trained on a giant synthetic dataset of Legos, but then the problem of overfitting to synthetic features is even higher: the pre-trained YOLO is already trained on real images which eliminate some of the issues with synthetic data such as lack of lens effects and convincing lightning. The training that was performed in this project was merely a fine-tuning to the shapes of Legos. Another problem can be the fact that the dataset was constrained to 10 classes: distinguishing between a brick included in the dataset and an excluded brick can be difficult for the model, leading to false positives.

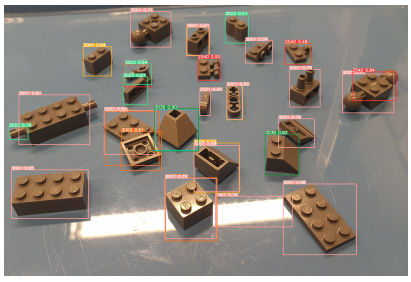

Figure 10. Inference when running YOLO trained on Set A on sparsely placed Legos.

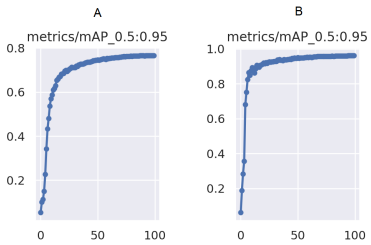

Figure 11. mAP when training YOLO on Set A and B respectively. NOTE the different y-axis range!

Compared to the Deeepwin repository this project achieved a higher mAP score (0.74 vs 0.80 and 1.0). Unfortunately, this comparison can not be made with full confidence because of the problem with reoccurring instances of specific bricks described above. However, considering that Deepwin only trained on 1280 images compared to 8000 images in this project, the improved mAP in this project can still be stated with some confidence.

A. *Suggested further research*

One big improvement would be to completely generate the synthetic data using actual 3D piles of Lego in a program such as Blender [9]. That would allow for not having to speculate about what augmentations to use for convincing 2D bricks pasted on a background but instead, focus on generating enough data and testing models.

V. Conclusion

A pathway for generating a synthetic dataset with automatic labeling was created, enabling creating two datasets with 10000 images and about 170,000 instances of bricks each. mAP was improved compared to the Deeepwin repository with 23% on average. However, from an absolute standpoint the accuracy is still low for tightly placed bricks given by a purely visual examination of the results. More work is required for generating completely convincing synthetic data to rule out the aspects of bad synthetic data affecting the results. Then, if not before, another model instead of YOLO can be considered.

Acknowledgements

Thanks to Professor Paolo Ienne at EPFL for the trust in us to take on this project.

Disclaimer

A part of this project utilised Google image search to find pictures of Lego bricks and consequently the rights of these do not belong to the authors. However, this project is a non-profit course-project and thus it was deemed appropriate as a way of testing the concept of blending real with rendered bricks.

REFERENCES

[1] "Brickowl," https://www.brickowl.com/, accessed: 2021-12-19.

[2] "Nvidia synthetic data," https://blogs.nvidia.com/blog/2021/06/08/what-is-synthetic-data/, accessed: 2021-11-09.

[3] S. I. Nikolenko, "Synthetic data for deep learning," 2019.

[4] "Universal lego sorting machine," https://www.theverge.com/2019/12/11/21011792/lego-ai-universal-sorting-machine, accessed: 2021-12-20.

[5] "Deeepwin lego-cnn," https://github.com/deeepwin/lego-cnn, accessed: 2021-11-09.

[6] "Yolo," https://github.com/ultralytics/yolov5, accessed: 2021-11-09.

[7] "Mask r-cnn," https://github.com/matterport/Mask_RCNN, accessed: 2021-11-09.

[8] "Kaggle lego database," https://www.kaggle.com/rtatman/lego-database, accessed: 2021-12-19.

[9] "Blender python api," https://docs.blender.org/api/current/index.html, accessed: 2021-12-21.