AutoSimulate: (Quickly) Learning Synthetic Data Generation - Supplementary Material

Harkirat Singh Behl[1], Atilim Güneş Baydin[1], Ran Gal[2], Philip H.S. Torr[1], and Vibhav Vineet[2]

[1] Univeristy of Oxford, Oxford, UK
{harkirat,gunes,phst}@robots.ox.ac.uk
[2] Microsoft Research, Redmond, USA
{rgal,vibhav.vineet}@microsoft.com
https://harkiratbehl.github.io/autosimulate

1 CLEVR Blender

The different simulator parameters optimized for the Clevr experiment and the underlying distributions are shown in Table 1. The validation set of 30 images is composed of synthetic images generated with a particular simulator configuration which involved a particular lighting configuration, quality of images, image size, location and material of images. This is shown in the right column of the last row of Fig.1. This configuration is hidden during the simulator training and only the validation set is available.

Table 1. Distributions over Clevr simulator parameters

Parameter	Distribution	Range
Number of Samples (Quality)	Categorical	2, 128, 512
Number of Bounces (Quality)	Bernoulli	8, 128
Image Size	Categorical	32x32, 128x128, 256x256
Ambient Light Intensity	Gaussian	0 to 100
Back Light Intensity	Gaussian	0 to 100
Each Object Location (3 Objects)	Gaussian	-10 to 10
Each Object Material (3 Objects)	Bernoulli	Rubber, Metal

The test set and validation set are generated from the same hidden simulator. The simulator is initialized from a different configuration than the validation simulator. During training, the simulator learns to generate data which is optimal for training a neural network for perforimng well on the validation set. Figure 1 shows the evolution of the simulator with training. It can be seen that the simulator starts from poor lighting, quality and location of objects and improves itself during training, along with improving the segmentation performance on the validation set.

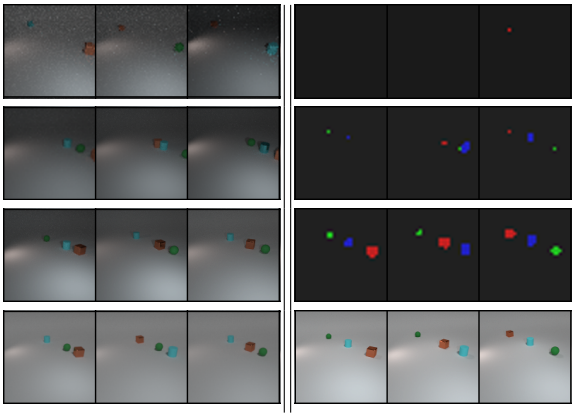

Fig. 1. Evolution of CLEVR simulator during training using AutoSimulate. `Left`: rows show the images generated by the simulator at different iterations during simulator training. It can be seen that the simulator evolves from poor lighting and low quality (Row 1 and 2) to good lighting and quality (Row 4) and also learns the object positions, as are in the validation set. Row 4 shows images generated from the final trained simulator. These images are used to train a semantic segmentation model. `Right`: semantic segmentation model output on 3 validation images (Row 4) across simulator training iterations. Row 3 provides outputs from the final trained segmentation model.

2 Photorealistic Renderer

We briefly describe the rendering technique used to generate photo-realistic images in the main set of experiments. There are three essential components to the rendering system. First component is acquiring accurate 3D CAD models of objects; which involves assigning accurate geometric properties, along with material, texture and color to the objects. Then, these 3D objects models are rendered within a realistic scene, which is essential for capturing the context around these objects. A realistic scene should also have accurate geometric shapes, background objects, along with accurate materials, texture and lights.

Second component is the proper placement of these objects and camera within a scene such that the images renderer are plausible. Objects are placed in the scene such that rigid body properties are not violated, for example, they should not intersect with each other. A physics engine (Nvidia PhysX) is used to generate possible object arrangements. A set of camera positions is randomly generated for each object arrangement.

Finally, in order to generate photorealistic images, accurate simulation of reflectance and light properties, including shadows, soft shadows, inter-reflections etc, is necessary. Physically based rendering method, for example Arnold [1], is then used to generate highly photorealistic images. This system uses ray tracing which relies on shooting rays through within a scene. The quality of rendered images depends on the number of rays sampled and the number of bounces of lights from (diffused and specular) surfaces. A user can control the quality of

images by specifying the values for these parameters. More details about data generation with the simulator can be found in the work of Hodan et.al., [2] and Arnold [1].

Implementation Details For calucation of the inverse hessian vector product using the Conjugate Gradient (CG) algorithm, we used the `fmin_ncg` function from python library `scipy.optimize` [3]. Furthermore, a standard implementation of Grid Search from Scikit-learn Python library [3] was used to tune the hyper-parameters for all the baselines and our algorithm, and the best results are reported for every algorithm. The size of the dataset K generated in each iteration is tuned over the values $20, 50$ and 70. The simulator learning rate α is tuned over the values $0.01, 0.1$ and 10. For LTS, the number of datasets generated at each iteration is tuned over the values $5, 10, 40$ and 50. The weight decay regularization λ of the hessian is tuned over 0.1 and 10. We observed a standard deviation of around 3% in the Test mAP with this hyper-parameter tuning.

2.1 More Ablation Studies

We now show another experiments to compare and evaluate the robustness and performance of different algorithms.

Effect of Network Architecture In this experiment we evaluate the effect of the network architecture on simulator training for different methods. In particular, we show results using two networks: Faster-rcnn with Resnet50-fpn backbone and YOLO with 112 layers in Table 2. We observe that our methods is much better than all the other baseline methods across different network architectures, thereby demonstrating the generality of the method.

Table 2. Effect of Network Architecture

Method	mAP (Faster-rcnn)	mAP (Yolo)
REINFORCE (LTS)	51.8	37.2
Bayesian Optimization	46.0	37.5
Random Search	50.3	36.8
Ours	**55.1**	**45.9**

3 Gradient of Expectation for different distributions

We discuss a technique to compute the term $\frac{d}{d\psi}\,\mathbb{E}_{s\sim p_\psi}[f(s)]$ which is the gradient of expectation of a function over a distribution, with respect to parameters of

[3] https://het.as.utexas.edu/HET/Software/Scipy/generated/scipy.optimize.fmin_ncg.html

the distribution. For continuous distributions we can directly use REINFORCE [6], as follows:

$$\frac{d}{d\psi} \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)] = \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)\frac{d}{d\psi}\log p_\psi(s)] \tag{1}$$

Gaussian Distribution For the mean of Gaussian distribution, we have $\frac{\partial}{\partial\psi}\log p_\psi(x) = \Sigma^{-1}(x-\psi)$, where ψ is the mean and Σ is the covariance matrix. Putting this into Eq. 1 we get:

$$\frac{d}{d\psi} \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)] = \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)\,\Sigma^{-1}(s-\psi)] \tag{2}$$

For discrete distributions we can derive it similarly [5, 4]. Here we give the derivations for Bernoulli and Categorical distributions which we used in our experiments.

Bernoulli Distribution Let s take value 1 with probability ψ and value 0 with probability $1-\psi$, defined as $p_\psi(s) = \psi^s(1-\psi)^{(1-s)}$. The gradient over expecation term can be computed as:

$$\begin{aligned}\frac{\partial}{\partial\psi} \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)] &= \frac{\partial}{\partial\psi}(1-\psi)f(0) + \frac{\partial}{\partial\psi}\psi f(1)\\ &= f(1) - f(0)\end{aligned} \tag{3}$$

Categorical Distribution Let the parameter be $\psi = (\psi_1, \psi_2, ..., \psi_K)$ s.t $\psi_1 + \psi_2 + ... + \psi_K = 1$ where ψ_i is the probability of seeing element i. Let $[s=i]$ be 1 if $s=i$ and 0 otherwise, then:

$$\begin{aligned}p_\psi(s=i) &= \prod_{i=1}^{K}\psi_i^{[s=i]}\\ \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)] &= \sum_{i=1}^{K}[s=i]\psi_i f(s)\\ \frac{\partial}{\partial\psi_i} \mathop{\mathbb{E}}_{s\sim p_\psi}[f(s)] &= \frac{\partial}{\partial\psi_i}[s=i]\psi_i f(s)\\ &= f(i)\end{aligned} \tag{4}$$

References

1. Georgiev, I., Ize, T., Farnsworth, M., Montoya-Vozmediano, R., King, A., Lommel, B.V., Jimenez, A., Anson, O., Ogaki, S., Johnston, E., et al.: Arnold: A brute-force production path tracer. TOG (2018)

2. Hodaň, T., Vineet, V., Gal, R., Shalev, E., Hanzelka, J., Connell, T., Urbina, P., Sinha, S., Guenter, B.: Photorealistic image synthesis for object instance detection. ICIP (2019)
3. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. JMLR (2011)
4. Rezende, D.J., Mohamed, S., Wierstra, D.: Stochastic backpropagation and approximate inference in deep generative models. In: ICML (2014)
5. Schulman, J., Heess, N., Weber, T., Abbeel, P.: Gradient estimation using stochastic computation graphs. In: NIPS (2015)
6. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning (1992)