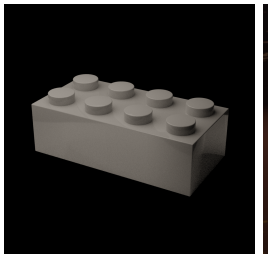
Assignment 2

Batch Size of 3

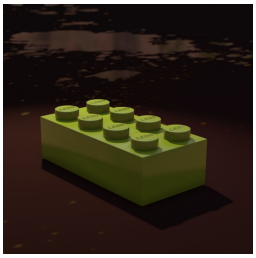
October 29, 2023

1 Improvement on the dataset

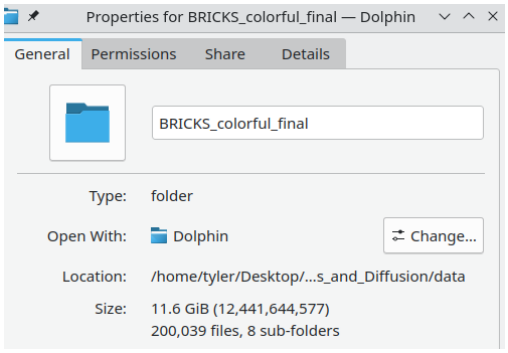
We updated the dataset by incorporating new computer-rendered LEGO images, introducing diverse and expanded data. Unlike the previous dataset, which featured bricks with only slight hue variations, a null background and just bricks, the updated collection encompasses various LEGO classes, such as rectangular bricks, sloped bricks, plates, walls, and miscellaneous (irregular shapes). This new dataset accurately represents LEGO colors and incorporates realistic shading, accompanied by a dynamic background featuring both diffuse and glossy spots. Furthermore, we've introduced increased variability in lighting conditions and camera angles.



(a) Old Data Example



(b) New Data Example



(c) Dataset

In our new dataset, we included 10,000 to 20,000 images of each category for a class. Our dataset includes a class (i.e. bricks, plates, slopes), and a category of the class (i.e. brick-1x1, slope-1x2). Note that some files have some extra images.

```
\dataset (200,000 datapoints)
    \bricks (40,000)
        - bricks-1x1 (10,000 images)
        - bricks-1x2 (10,000 images)
        - bricks-2x2 (8,000 images)
        - bricks-2x4 (12,000 images)
    \miscellaneous (30,000)
        - arch-1 (10,000 images)
        - fence (10,000 images)
        - flags (10,000 images)
    \plates (40,000)
        - plate-1x1 (10,000 images)
        - plate-1x2 (10,000 images)
        - plate-2x2 (10,000 images)
        - plate-2x4 (10,000 images)
    \slopes (60,000)
        - slope-1x2 (20,000 images)
        - slope-2x2 (20,000 images)
        - slope-2x4 (20,000 images)
    \wall (30,000)
        - wall-2 (10,000 images)
        - wall-4 (10,000 images)
        - wall-6 (10,000 images)
```

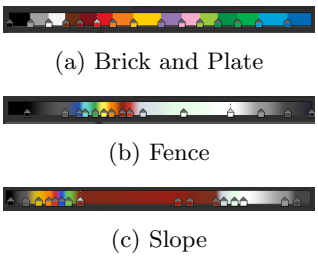
(a) Brick 1x1 (b) Brick 1x2 (c) Brick 2x2 (d) Brick 2x4 (e) Plate 1x1

(f) Plate 1x2 (g) Plate 2x2 (h) Plate 2x4 (i) Wall 2 (j) Wall 4

(k) Wall 6 (l) Flag (m) Fence (n) Arch 1 (o) Slope 1x2

(p) Slope 2x2 (q) Slope 2x4

Figure 2: Examples of our dataset



(a) Brick and Plate

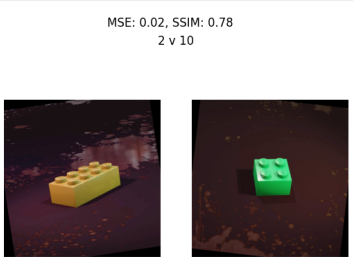
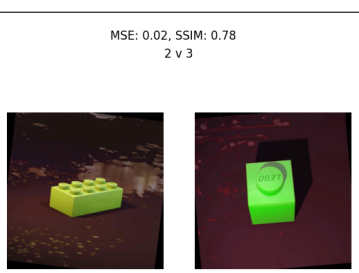
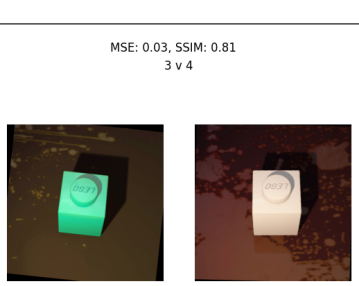
(b) Fence

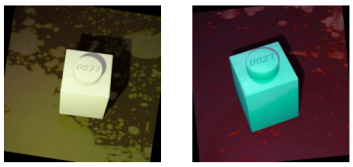
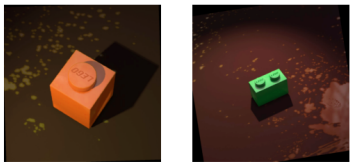
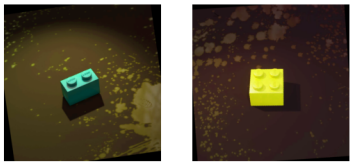
(c) Slope

Figure 3: Color gradient for LEGO pieces

2 Degree of Similarity

Original Images
1 
2 
3 
4 
5 
6 
7 
8 
9 
10 

Compared Pairs (with aug)	LPIPS	MSE	SSIM	Tyler	Zhi	Alan
	0.491	0.01	0.82	0.4	0.3	0.6
	0.528	0.02	0.78	0.6	0.7	0.6
	0.528	0.02	0.78	0.5	0.4	0.4
	0.522	0.03	0.81	0.8	1.0	1.0

Compared Pairs (with aug)	LPIPS	MSE	SSIM	Tyler	Zhi	Alan
MSE: 0.05, SSIM: 0.66 4 v 5 	0.550	0.05	0.66	0.8	0.9	1.0
MSE: 0.04, SSIM: 0.70 5 v 6 	0.473	0.04	0.70	0.8	0.7	1.0
MSE: 0.03, SSIM: 0.75 6 v 7 	0.474	0.03	0.75	0.4	0.7	0.6
MSE: 0.02, SSIM: 0.79 7 v 8 	0.559	0.02	0.79	0.6	0.6	0.6
MSE: 0.03, SSIM: 0.78 8 v 9 	0.569	0.03	0.78	0.8	1.0	0.8

Only from "Compared Pairs (with aug)"

- Average MSE: 0.025, Average SSIM: 0.687
- Average LPIPS: 0.52127 +/- 0.01112
- Average Tyler: 0.55
- Average Zhi: 0.63
- Average Alan: 0.68

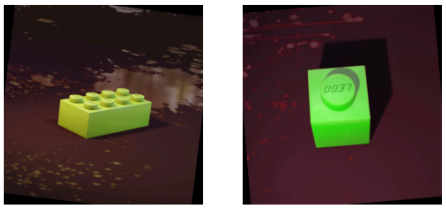
All comparisons from Tyler/Alan/Zhi

First Image/Second Image	2	3	4	5	6	7	8	9	10
1	8/10/8	4/6/4	4/8/5	4/6/4	4/6/3	6/4/5	6/6/5	8/8/7	6/6/3
2		4/4/4	4/4/3	4/4/3	4/4/3	6/4/6	6/4/5	6/4/5	6/6/7
3			8/10/10	8/10/8	8/10/8	4/8/7	4/6/5	4/6/6	6/8/7
4				8/10/9	8/10/9	4/8/6	4/6/5	6/8/8	4/6/3
5					8/10/7	4/6/6	4/6/5	4/6/6	4/6/3
6						4/6/7	4/6/5	4/6/3	4/6/4
7							6/6/6	6/6/7	6/6/7
8								8/8/9	8/10/7
9									8/10/9

If we example all of the results for the similarity, we can definitely say that for all situations MSE is not effective at all with our dataset. This is due to the fact that MSE compares every pixel value, which will always be different in this case since we have noise in the background with different colored legos in different rotations and locations. By looking at SSIM and LPIPS we have a more concrete comparison to a human similarity.

On average, we can say that the human similarities are very close to each other. However, one person's average is closer to one comparison than the other, on one hand Alan's average is closer to SSIM, on the other hand Tyler's average is closer to LPIPS.

MSE: 0.02, SSIM: 0.78
2 v 3

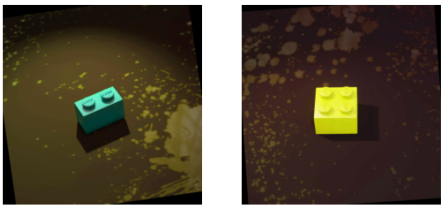


- **LPIPS agree with human**: In the case of (2 v 3), the measurement given by LPIPS (0.528) agree more with the human similarity (around 0.5) than SSIM, which gave it a very high similarity score of 0.78. Having a lower similarity here makes sense, the two pieces are at a different angle, the color is different, and the shape is different.

MSE: 0.04, SSIM: 0.70
5 v 6

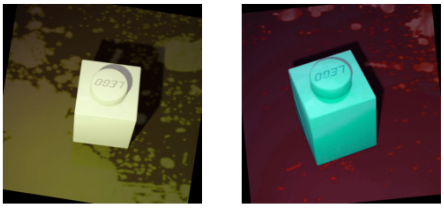
- **SSIM agree with human**: In the case of (5 v 6), the measurement given by SSIM agree more with the human similarity (around 0.85) than LPIPS, which gave it a lower similarity score of (0.473). If we examine the two images, we can see that the rotation is only slightly off with the color changed, thus the two legos are very alike in that case. Thus, having a higher similarity score would make sense for these two images.

MSE: 0.02, SSIM: 0.79
7 v 8



- **Both agree with human**: In the case of (7 v 8), the measurement given by both SSIM, LPIPS, and the humans are mostly around the same, although SSIM is slightly higher than the rest. Regardless, the two pieces although different, show that they are close by a similar color and size relative to the viewer.

MSE: 0.05, SSIM: 0.66
4 v 5



- **Both disagree with human**: In the case of (4 v 5), we can say that on average, the human similarity slightly does not agree with SSIM and LPIPS. If we examine the two images, we can say that the color and rotation is different. However, the piece is essentially the same, thus leading to a high human rating (around a 0.9). The metrics seem to disagree with that sentiment, LPIPS gave it a 0.55 and SSIM gave it a 0.66, both of which is a lower relative score for both metrics. For us, the LEGO shape plays a huge role in deciding if it is a similar piece, this might not be how SSIM and LPIPS measure the similarity.

Problem 2: GANs

a. Architecture and Hyperparamters

We used a DCGAN implementation from PyTorch. They used the celeba dataset to train their DCGAN and it was trained with these following hyperparameters:

	Params for DCGAN Paper
Resolution	64x64x3
Latent Space Dim	100
Epochs	5
Learning Rate	0.0002
Beta1 (Adam)	0.5
Batch Size	128

The paper for DCGAN suggested the hyperparameters of learning rate = 0.0002 and $beta_1 = 0.5$ so the only hyperparameter that was changed from the source code was the number of epochs. This depended on how many datapoints (images) were in the dataset. The celeba dataset has about 200k images so 5 epochs were sufficient for them. Our dataset also had 200k images but with augmentation it doubled to 400k. We did not measure the time it took to train the model, but we know that it took more than 5 hours to train from scratch.

Issue: Epochs was too low

We thought that 3 epochs would be enough from the ratio of images to epochs. However, it was not enough to train the model as seen in the follow image:

We assume that this is due to the variation between the legos is higher than the faces in celeba.
Due to the two issues above, we have 200k images and decided to raise the number of epochs to 40 and thus our hyperparameters are:

	Params for our DCGAN
Resolution	64x64x3
Latent Space Dim	100
Epochs	40
Learning Rate	0.0002
Beta1 (Adam)	0.5
Batch Size	128

Issue: Resolution

We tried changing the dimensions of the data to a higher resolution but we couldn't resolve some immediate errors that occurred relating to the dimensions of some vector. Thus the resolution was kept at 64x64.

Issue: Seeding

Heres the order of operations:

$$\text{Seeding} \rightarrow \text{Training dataset augmentation and creation} \rightarrow \text{Fixed noise} \rightarrow \dots$$

This fixed noise was used to keep track of how the legos turned out over time and since the dataset batching and augmentation happened before the fixed noise, when trying to load the trained model, the fixed noise would look different without the training dataset. The easy fix would be to change the order at which it happens:

$$\text{Seeding} \rightarrow \text{Fixed noise} \rightarrow \text{Training dataset augmentation and creation} \rightarrow \dots$$

Takeaways

- Keep track of both the discriminator and generator. Getting a checkpoint of the model at some location is a good decision. There are many times when the discriminator or generator wins over the other and drives the model off the cliff. Sometimes it is generally hard to avoid the wrong gradient step for both models, thus having checkpoints and going back to that gives a good impression of what happened at that iterative step. Moreover, you can recover a model from taking the wrong step as well by saving checkpoints.

- We noticed that adding noise to the background helped the results. Originally we had a white background, and was able to generate the following with 100 epochs (left image). Compared to the new images we generated (right image), the original images performed worse on our GAN.

b. Example Images

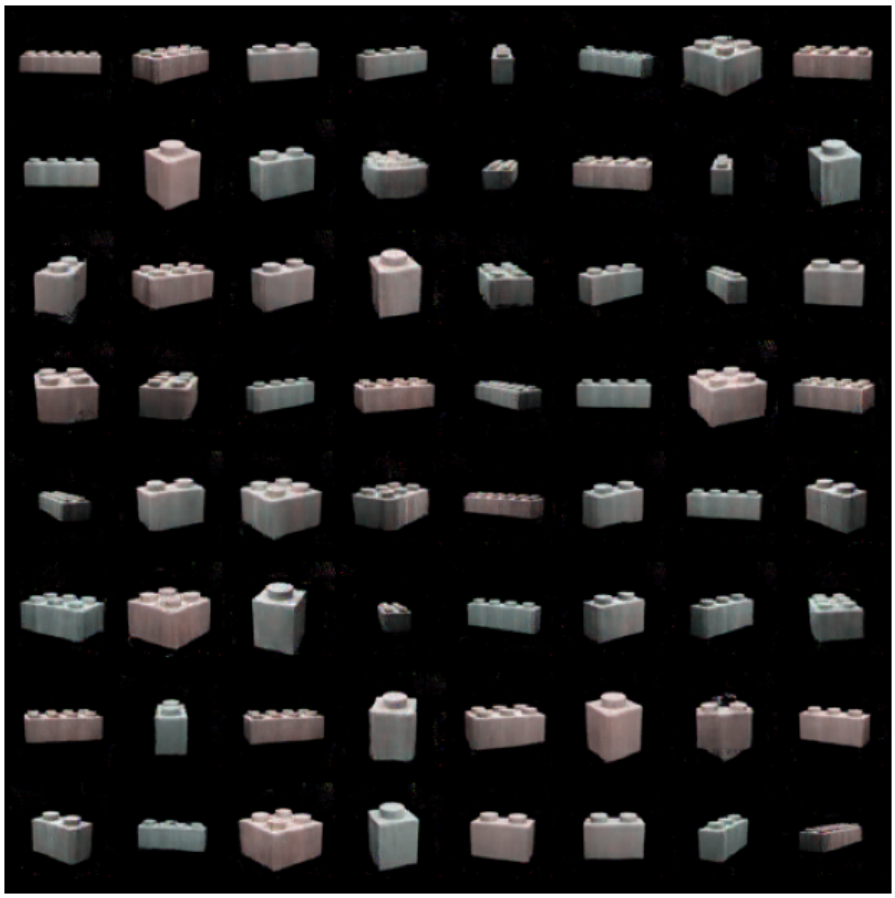
These are generated with the new dataset of legos (200k images with colorful background + 200k augmented images).

With the GAN theres seems to be a lot of legos that are miscolored with multiple colors. Compared to the VAE dataset, the GAN also had more trouble with properly defining the edges of the Legos.

More Example Images

These are only the lego bricks class, since it's a much narrower selection of shapes, the GAN results are able to generate much more realstic results with more accurate edges.

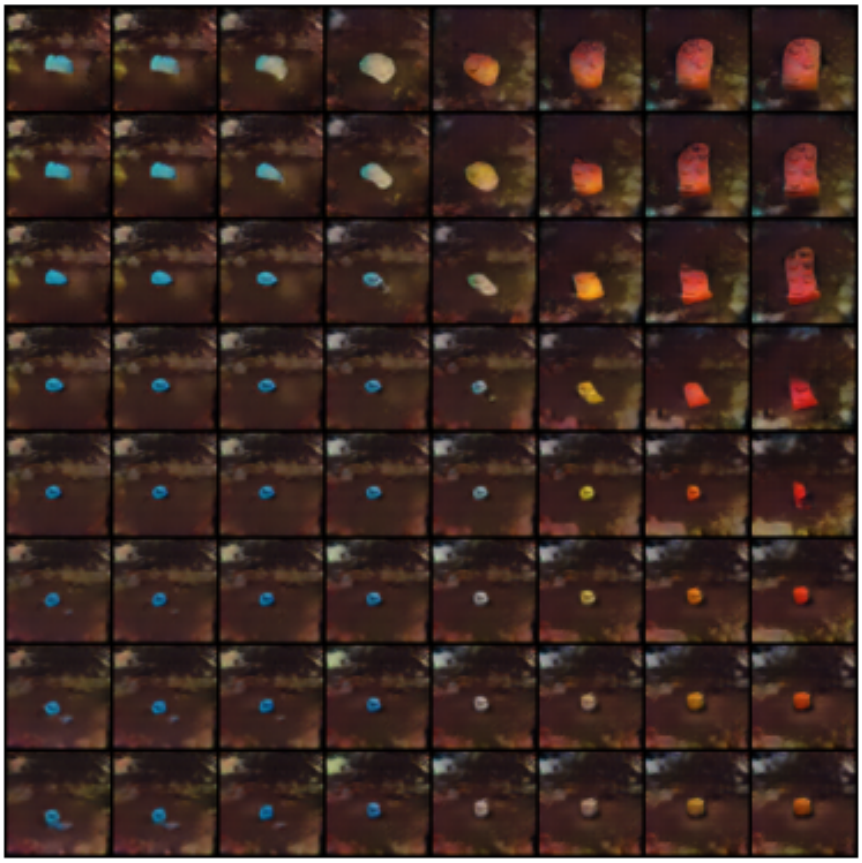
We also decided to train a DCGAN on the VAE dataset to compare:



These legos have much more defined studs and edges as compared to the VAE legos however, like the above result there are legos that have multiple colors and there is still a bunch of noise that is discoloring the legos.

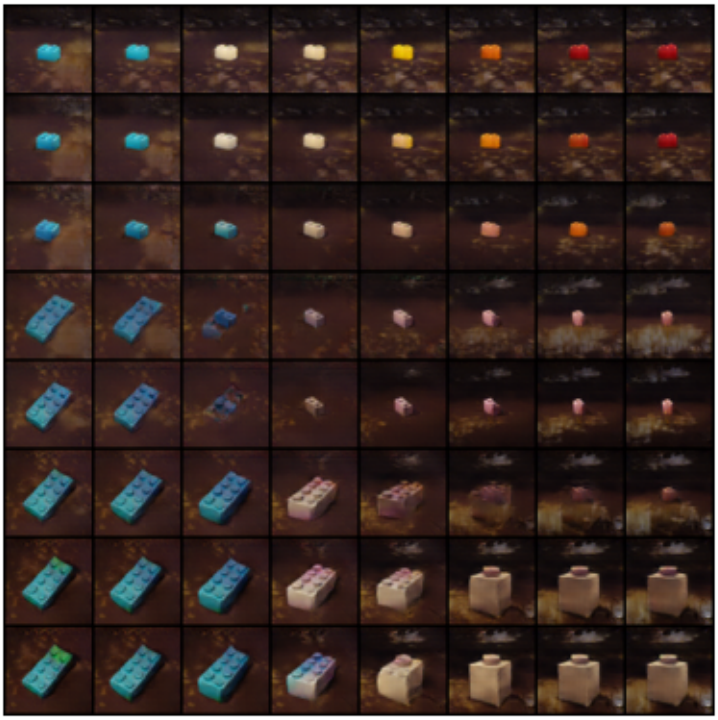
c1. Interpolation

Here is an interpolation between 4 different latent vectors:



Issue: Lego Inconsistency

Interpolating between these 4 legos gave us legos that weren't the same as the input ones (see the bottom left is green tipped instead of pink tipped):

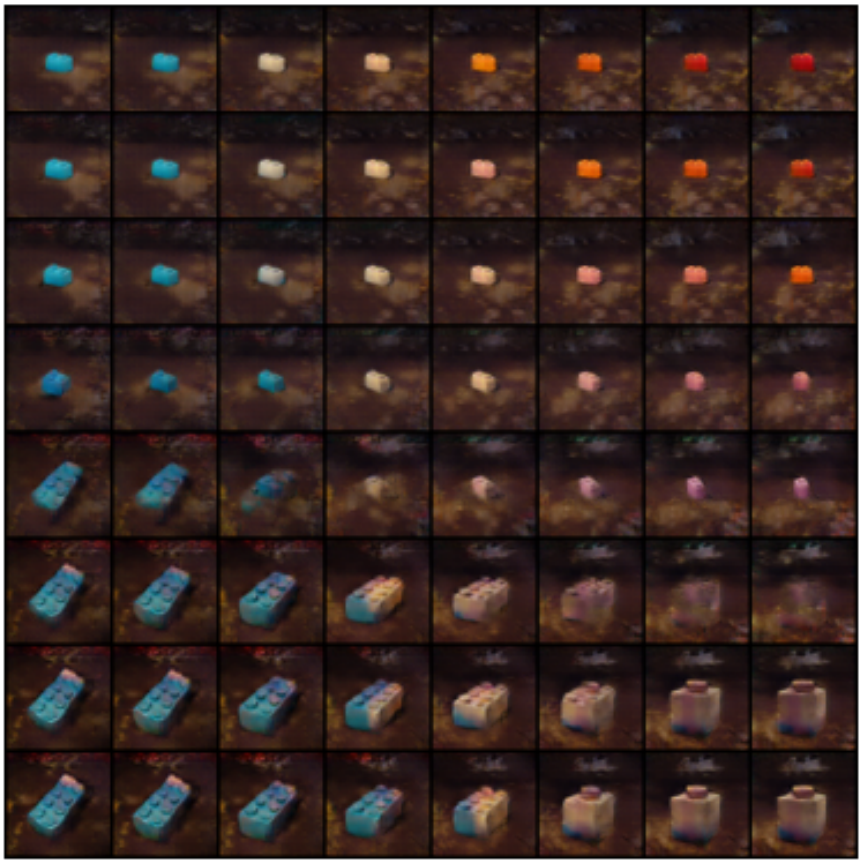


The network takes a B (batch) sized of vector of latents and converts it to batch of images

$$(B \times d_z \times 1 \times 1) \rightarrow_G (B \times 3 \times 64 \times 64)$$

The fix was to individually send each image through the network and combine the images afterwards:

$$(1 \times d_z \times 1 \times 1) \rightarrow_G (1 \times 3 \times 64 \times 64)$$



As you can see the bottom left lego is now pink tipped instead of green tipped.

c2. Interpolation Video

Generating many different latent vectors and interpolating between them in a video. See Box or Github for the mp4 file.

c3. Adding Constants to Latents

I was interested to see what happened if I continuously added a constant to the latent space and do it for 8 steps. The legos start to morph to different legos as expected but they all eventually end up at a terminal lego when the floats in the latent gets too big:

Problem 3: Diffusion Models

a. Architecture and Hyperparamters

We used a Denoising Diffusion Probabilistic Model (DDPM). The original hyperparameters for the DDPM repo that we used were:

	Params for repo DDPM
Resolution	128x128x3
Time Steps	1000
Training Steps	700k
Learning Rate	8e-5
Ema Decay	0.995
Gradient Accumulate Every	2

We decided to keep the resolution the same as the GAN to keep comparisons more fair. The only hyperparameter that we changed was the training steps since 700k was going to take way too long and 10k gave good results already. The model took us around 5 hours to train from scratch.

	Params for repo DDPM
Resolution	64x64x3
Time Steps	1000
Training Steps	20k
Learning Rate	8e-5
Ema Decay	0.995
Gradient Accumulate Every	2

Issue: Augmentation

We Realized that the diffusion model has its own Dataloader inside the pip package and thus we were not able to apply the augmentations directly to the tensors when they were loaded in. Thus this diffusion model was trained without the augmented images.

Issue: Taking to long to train

Started training the DDPM without changing the hyperparameters and it was 2k/700k steps in and it had taken 5 hours. The reason why it was taking so long was because it was doing FID Evaluations: they would take forever since each image had 250 timesteps and there were 1500 evaluations. I calculated that would it would take approx 60 days to train with evaluations. The fix was to just turn off the evaluations by setting it to False.

Takeaways

- If you are using a prebuilt library for your diffusion model, do more research of what your model is actually doing in code so that you can more easily modify what it is doing to suit what you want to achieve.

b. Example Images

Here is 25 images sampled randomly from the latent space:

The diffusion model picked up the defined hard edges of a lego, the studs, the correct number of studs for the blocks (usually) and the characteristics of the different classes (slopes, walls, plates, bricks). The images are very reasonable and most of them would pass for actual legos. Compared to the GAN generated images, the legos are more well defined than the GAN's: the edges and much sharper, and there are consistent colors for the pieces.

More Example Images

Like the GAN section, here is a diffusion model trained on the dataset we used for the VAE:

Same conclusion as the one on the new dataset but this time the model managed to capture the rendering issues present in between the studs.

Issue: VRAM Issues

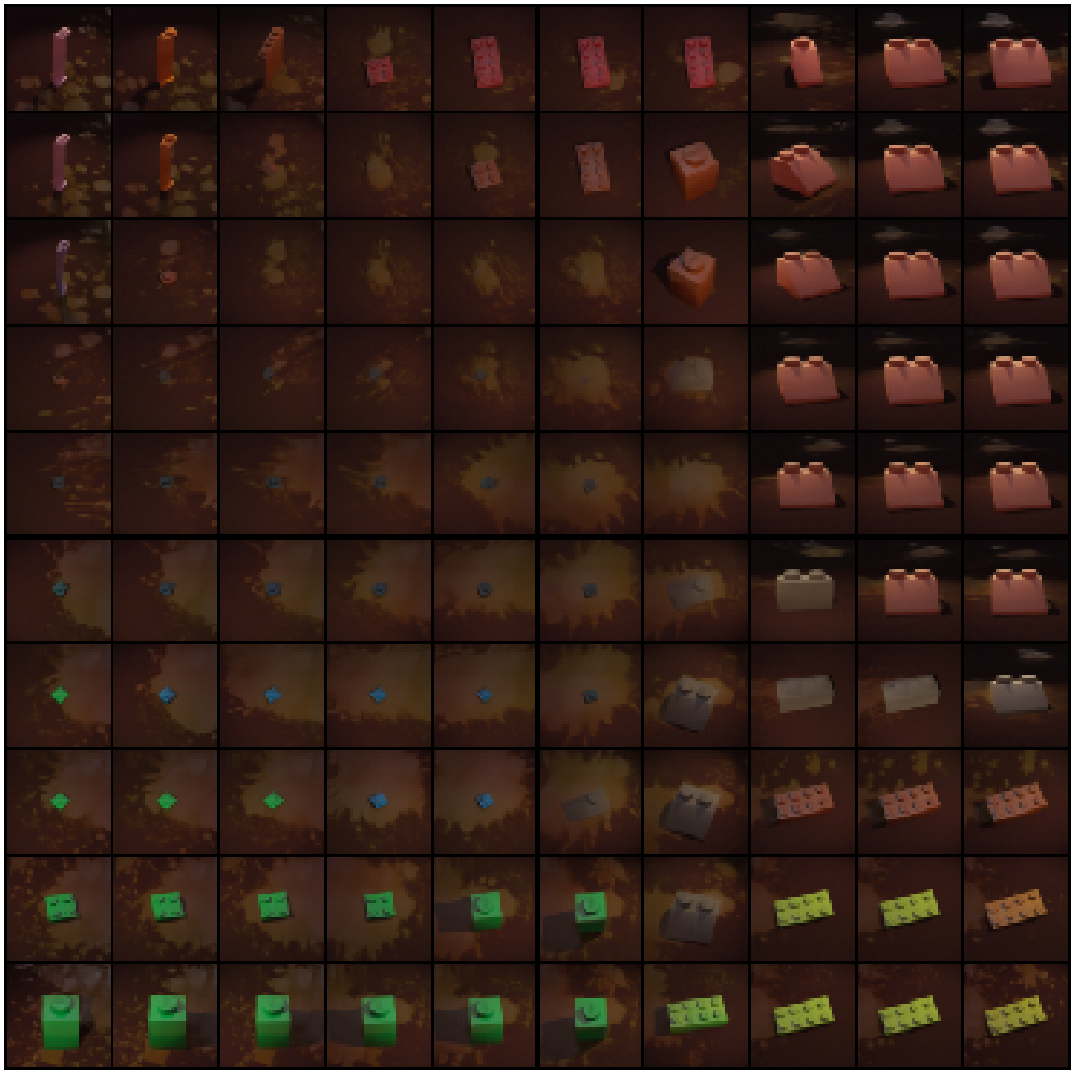
I was running out of VRAM when generating more than 5 images at a time. The fix was to generate one image at a time, bring it to CPU/RAM and clear the cache in the GPU.

c1. Interpolation

Here is an interpolation between two different latent spaces:

Compared to the GAN and VAE, the interpolation between different bricks happens very suddenly. There isn't really good in between bricks, which makes since since the diffusion model generates very well defined images of the legos.

Just like the GAN, here is an interpolation between four different latent spaces:



c2. Interpolation Video

As with the GAN we created a video of the interpolation between many different latent spaces on Box and Github.