VEATGen
Variational Auto-Encoder for Texture Generation

Alexandre Variengien

March 24, 2021

1 Introduction

To create nice visual content using virtual scenes, all shapes must have a texture. The textures are often drawn from an image dataset, however, the choice is often limited. In order to generate arbitrary varied textures, methods using procedural generation are often used. However, they work well for only a small subset of textures such as fire or smoke for instance.

Other methods involve using a real-world image and then sampling sub-images from it to get a realistic, non-repeating tiling.

In this project, I do not attempt at creating a way to tile an object using preexisting model images, instead, I tried to provide a user-friendly way to generate new texture images from a small data-set. My goal was to create the most diverse textures from a little data set of images and to give the user an intuitive, spatially based way to explore the space of images generated to chose the one he or she needs. To achieve this goal, I used a neural network-based technique called Variational Auto-Encoders [1]. They are currently one of the main techniques for image generation with neural networks along with GAN. Even if GANs lead to great results for texture synthesis such as in [2], they are less stable and more computationally expensive than VAE. Moreover, VAE can also lead to reasonable texture synthesis quality [3].

I found many papers that used GAN for texture synthesis but I did not find much work with VAE, so I also thought it was nice to try by myself.

1.1 Presentation of VAE

I used neural networks called Variational Auto-Encoders (VAE) to project the source images in a structured, low-dimensional latent space.

The architecture of such networks is depicted in fig. 1.

A VAE is divided into two parts: the encoder and the decoder.

The goal of the encoder is, given an image, to generate the parameters of a Gaussian distribution in $\mathbf{R}^N$. Then, a sample is generated following this distribution. The decoder then transforms this sample into an image. The intermediate space $\mathbf{R}^N$ is called the latent space of the VAE.

The goal of the whole network is to get the generated image as close as possible to the input image.

However, if it was all, its task would be easy. It could associate each image of the dataset into a narrow region of the latent space with a sharp Gaussian distribution centered on a specific region. The decoder could then perfectly reconstruct the image by recognizing the corresponding region.

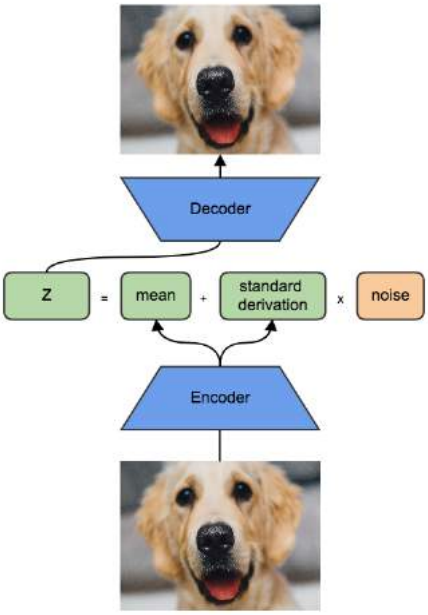

Figure 1: VAE architecture

But our final goal is to get a latent space where every point around the origin can be decoded into a meaningful image by the decoder. In the case described below, in the regions where no image is mapped, the decoder will output random images because these situations were not present during its training. To avoid these behaviour, the loss of the network is decomposed into two terms:

$$\begin{aligned} Mean, Std &= \text{Encoder}(Image_{in}) \\ z &\sim \mathcal{N}(Mean, Std) \\ loss &= \text{ImageDist}(\text{Decoder}(z), Image_{in}) + \lambda * \text{KL}[\mathcal{N}(Mean, Std)||\mathcal{N}(0,1)] \end{aligned}$$

$\text{ImageDist}(\text{Decoder}(z), Image_{in})$ is the reconstruction loss: we want the output image to be as close as possible to the input image. In my experiment, I used binary cross-entropy as a pixel-based distance over the images.

$\text{KL}[\mathcal{N}(Mean, Std)||\mathcal{N}(0,1)]$ is the Kullback-Leibler divergence. It measures the distance between the distribution outputted by the encoder and the $\mathcal{N}(0,1)$ distribution. Thus, we penalize every distribution that are either too sharp or too far from the origin. This way we avoid the case described below: because we force every distribution to be close one from another, every point of the latent space will map to a meaningful image after the decoder.

The tradeoff between reconstruction loss and KL loss is controlled by the hyperparameter λ.

2 Method

2.1 Workflow

To generate images, here are the steps taken by VEATGen :

- Chose a set of images that we want to extend to create new texture.
- Train the VAE with latent space dimension N on the dataset. In practice, I chose $N = 20$.
- Compute the PCA (Principal Component Analysis) of the means outputed by the encoder of the VAE on the images from the data-set.
- Select the d vectors $v_1, ..., v_d$ corresponding to the d greatest singular values of the PCA. In practice I chose $d = 6$.
- Provide a GUI to explore the images outputted by the decoder when we restrain the latent space on $Vect(v_1, ..., v_d)$. In practice, the user can tweak 6 cursors corresponding to the $v_1, ..., v_d$. For each combination, a 2D grid of images is generated. The 2D spatial dimensions of the grid are linked to variation in the two main directions of the PCA (v_1 and v_2) around the values of the cursor the user set for v_1 and v_2. The amount of variation is controlled by a zoom parameter.

We use PCA to reduce the space where the user can explore "interesting" directions, i.e. the ones that explain the most variance from the latent vectors of the dataset. Moreover, this gives an intuitive hierarchical ordering of the cursor: the first dimensions (v_1, v_2) cause great change in the output image while the last ones (v_5, v_6) account for small details.

3 Dataset

I used the Describable Textures Dataset [4]. It is a commonly used dataset for texture generation. It contains 5640 images sorted in 47 categories.

For my experiment, I did not used it as a whole but I selected by hand 2 interesting subsets. The images I used to train my model were a 300x300 region crop in the center of the original image. This square was then reduced to a 50x50 resolution or to 100x100 in the case of the `honey-large` dataset.

3.1 `Honey` dataset

I selected images from the "honeycombed" category. I then selected 30 images from this category that match the criterion: representing a pretty flat surface, even lighting and the minimal occluding object. I also try to include some variance in the images. I chose some real images honeycomb images as well as abstract hexagonal tiling. A subset of the images chosen can be seen in fig. 2.

I chose this category also because it was used by Liam Wynn in a blog post about texture generation with VAE. I used his results as a point of comparison with mine.

I conduct experiment with 50x50 and 100x100 resized images, respectively called the `honey` and `honey-large` datasets. I created the dataset with the 100x100 images to keep the hexagonal tiling appearance that is sometimes lost in the 50x50 versions (e.g. the second example, starting from the right).

3.2 `Cracked` dataset

The structural importance of the hexagons is crucial in the appreciation of the honeycombed textures. Also, the images from this dataset are mainly homogeneous in shape and color. To try an exepriment with more variation in the dataset, I selected by hand a subset of 43 images from

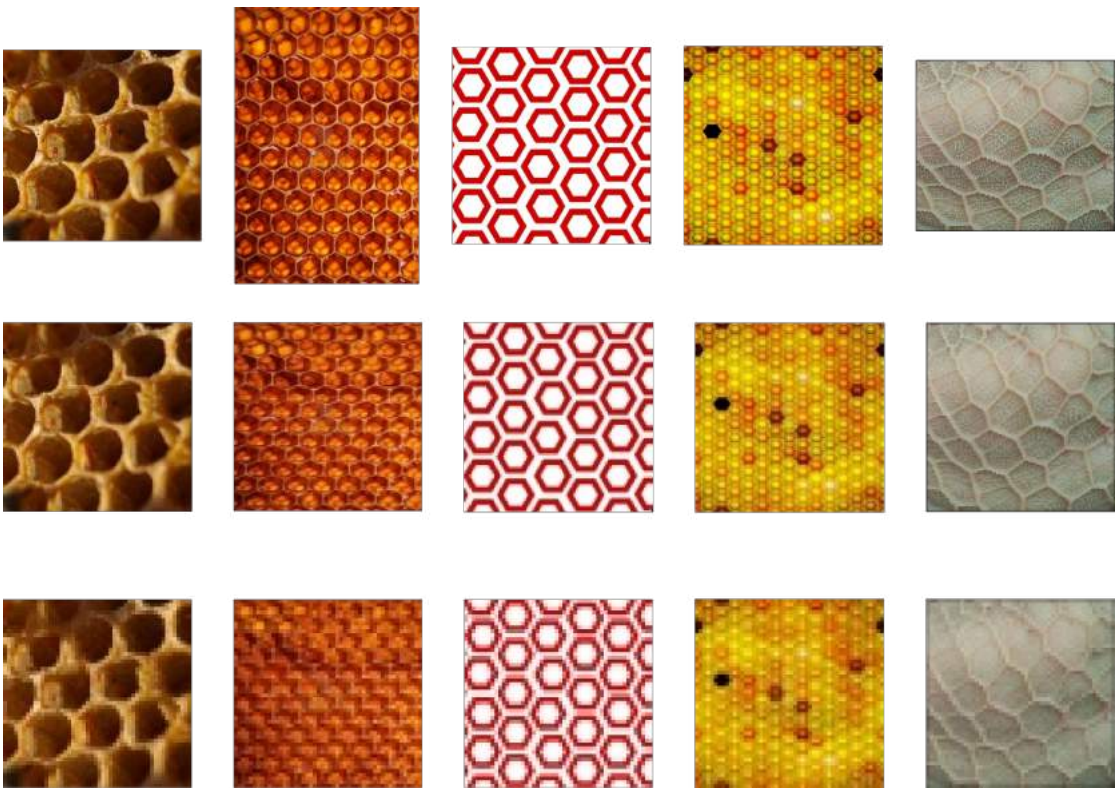

Figure 2: 5 images from the dataset selected in the category `honeycombed` along with their cropped and 100x100 and 50x50 resized versions.

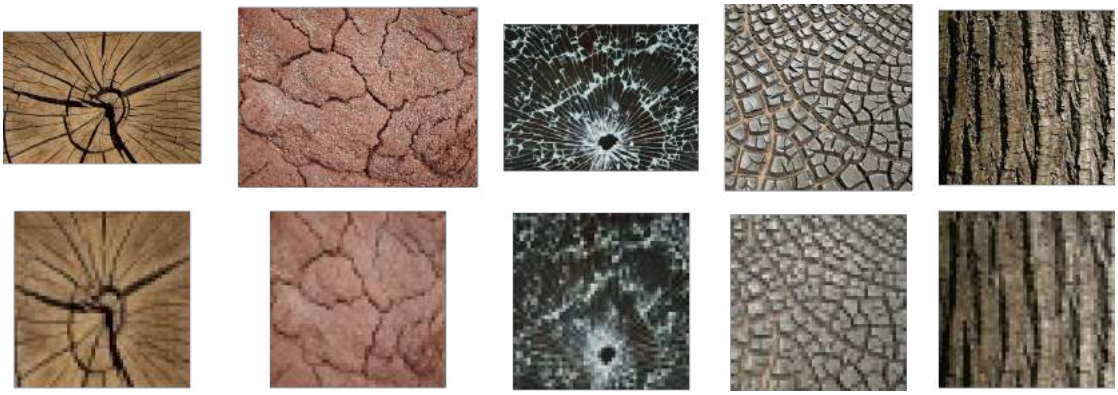

Figure 3: 5 images from the dataset selected in the category `cracked` along with their cropped and 50x50 resized versions.

the `cracked` category. I only used 50x50 resized version. Some example images are visible in fig. 3. As you can see, I included texture from various materials: barks, wood, glass dirt. I was curious to see how the model will map these in its latent space and if we could exploit texture with hybrid materials to create a nice artistic effect.

4 Implementation

4.1 VAE architecture

I used a standard VAE architecture with parameters tweaked by hand to get the best results. The encoder consists of a succession of 2D convolution and, max pooling and fully connected (dense) layers. The decoder is almost the same architecture but in reverse: dense layers followed by deconvolution layers. The architecture for 50x50 images is depicted in fig. 4. There are more parameters in the decoder part to be able to embed more prior to its weights. This will enable it to produce images with sharp details. Whereas for the encoder, there is no need to have too many parameters since the precision on the parameters of the distribution of the latent vector is not crucial.

For the `honey-large` dataset, I used the same basis but I added one deconvolution just before the output image of the decoder. This produce a continuous scaling of the resolution along the data flow in the decoder.

For regularisation, I used dropout on the last dense layer of the decoder. I thought that this will prevent from too much overfitting and add variation to the results during training. I tough that this will also add robustness for small perturbation in the latent space. This last feature will become handy when the user will explore it by hand by tweaking the parameters: we don't want abrupt change in the output image caused by a little movement in the latent space.

4.2 Environment

This project was developed in `Python 3` for its ease of package manipulation.

For the VAE, I used the `keras` (v. 2.3 running on `tensorflow`) implementation that can be found here.

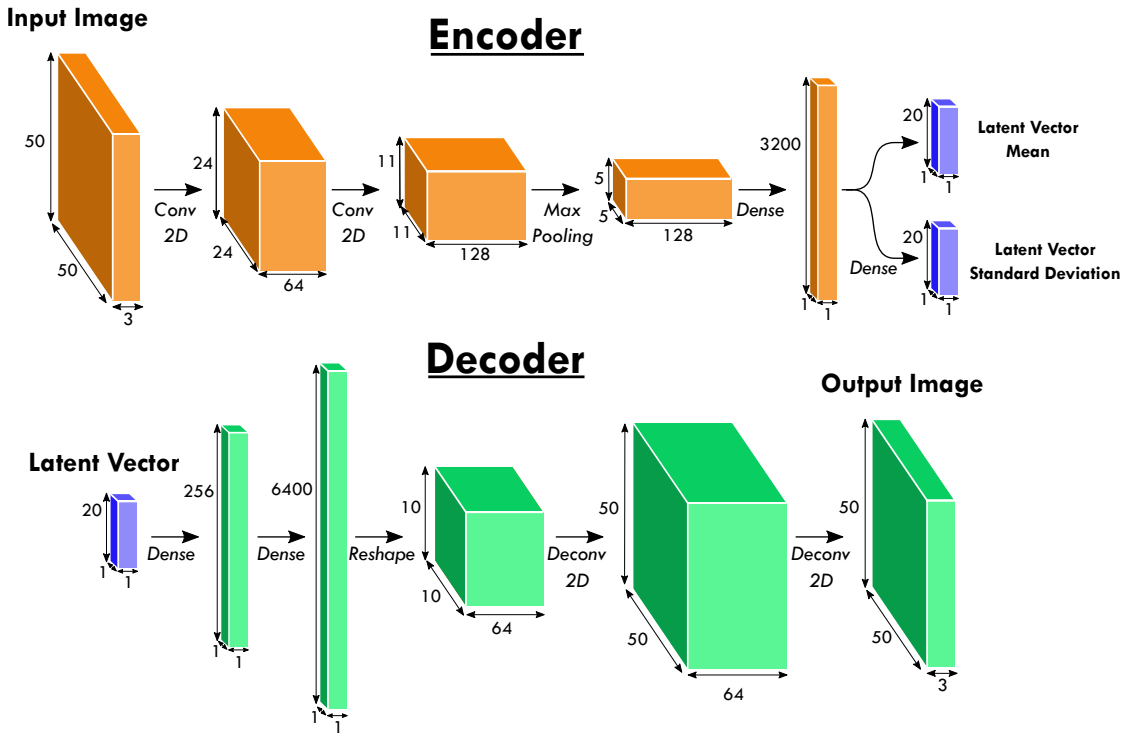

Figure 4: The architecture of the VAE used for both the `cracked` and `honey` datasets.

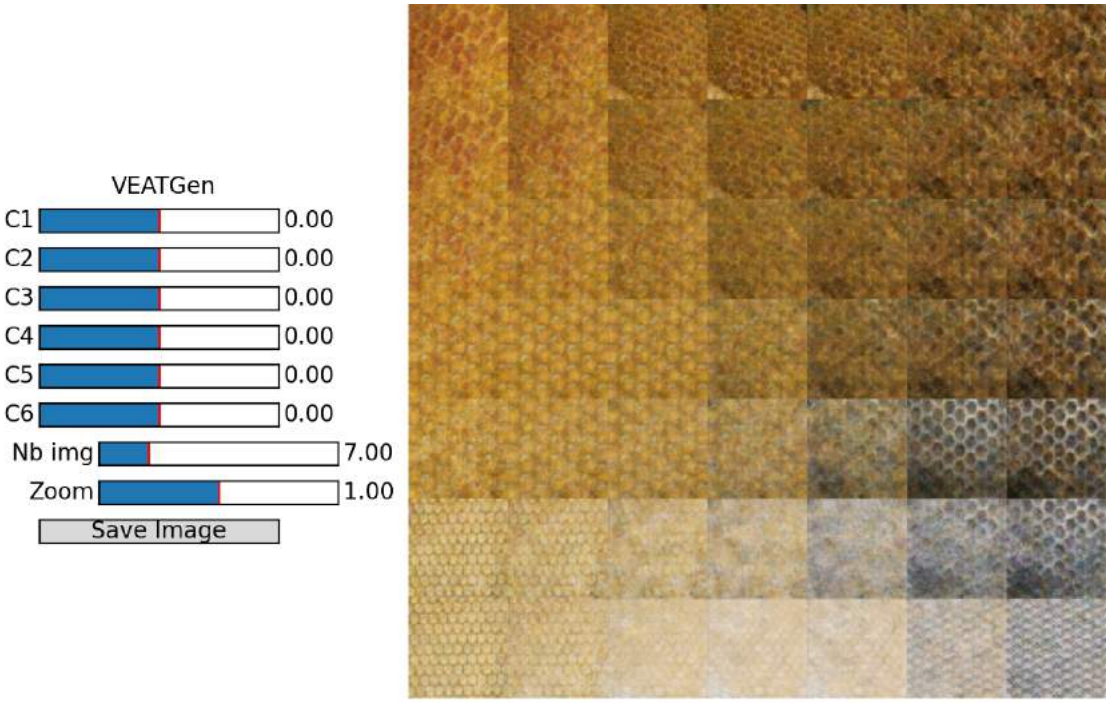

Figure 5: Screenshot of the GUI.

I used the `Google Collab` platform to train the VAE on a GPU. This enables me to test faster many experiments. The pretrained weigths shared are the results of 300 epochs of training. Since the number of images in the datasets is low, the training only took a couple of minutes to get a loss that stabilized.

5 Results

5.1 User interface

I created a simple user interface using `matplotlib` interactive features to ease the exploration of the image space, depicted in fig.5. You can run it with the code provided: each dataset comes with pre-trained parameters and a python file to build the model and run the GUI.

5.2 Results on the `honey` dataset

By exploring the image space we quickly find images really close to the ones in the dataset. The VAE is clearly subject to overfitting. Nonetheless, it doesn't matter too much for us. Indeed, the image of the dataset are the examples in the style we want, it is actually nice to have them embedded in the image space. We want that they are not the *only* images that the VAE can generate.

And it's not! In between the point where we recognized the images from the dataset, we can observe interpolation (see fig. 6) and variation on them, as was expected from the VAE architecture. Instead of creating a simple fade from one image to another, the texture is replaced

Figure 6: The VAE has learnt to interpolate between images in a no trivial way. The intermediate images gives interesting new textures.

by "patches" that match the hexagonal tiling. This creates an organic effect, coherent with the whole dataset: there are other raw textures from the dataset where "patches" are occulted for instance.

The VAE also learned to cluster the images according to their size and to their color. This is coherent with its loss constraints: to limit the interpolation effort, it needs to cluster close images together. This nice clustering effect can be seen in fig. 7 and 8.

5.3 Results on the `cracked` dataset

The phenomenons observed on this dataset are similar to what was observed on the `honey` one, as seen in fig. 9. We can observe the clustering effect as well as the interpolation. Nonetheless, the results seem less convincing, especially in the region transitioning from broken glass to bark (top left): the hole in the glass doesn't blend properly to become the bark. It seems just like a superposition of the two images.

Other transitions between materials are more successful such as the cracks of the wood that transitions to the cracks in the dirt while keeping the whole shape (bottom right).

5.4 Results on the `honey-large` dataset

This dataset leads to more mitigated results. First it was harder to train the model on it. I think that it is due to an increase in the number of parameters that come with the bigger size added in part by the supplementary deconvolution layer in the decoder.

I was constrain to diminish the hyperparameter λ to $1e-4$ to get a pertinent training. For comparison, I used $\lambda = 1$ for `honey` and $\lambda = 1e-2$ for `cracked`. For bigger λ, the KL loss has great importance in the total loss and the model was trapped in a local minimum where it outputted always the same image but the parameters of the distribution were as close as they could to $\mathcal{N}(0,1)$. Diminishing λ, lead to the greater importance of the reconstruction loss. The model couldn't ignore the difference between the outputted image and the one from the dataset and was able to separate them in the latent space. However, because λ was so low, the latent space distribution related to the different images could be far from $\mathcal{N}(0,1)$. This created regions where the decoded image doesn't make sense (e.g. see the top left of fig. 10).

Moreover, despite the higher resolutions, many images seem a bit blurred, especially when we explore far from clearly identified images from the dataset. I think that this is also linked to the problem described above.

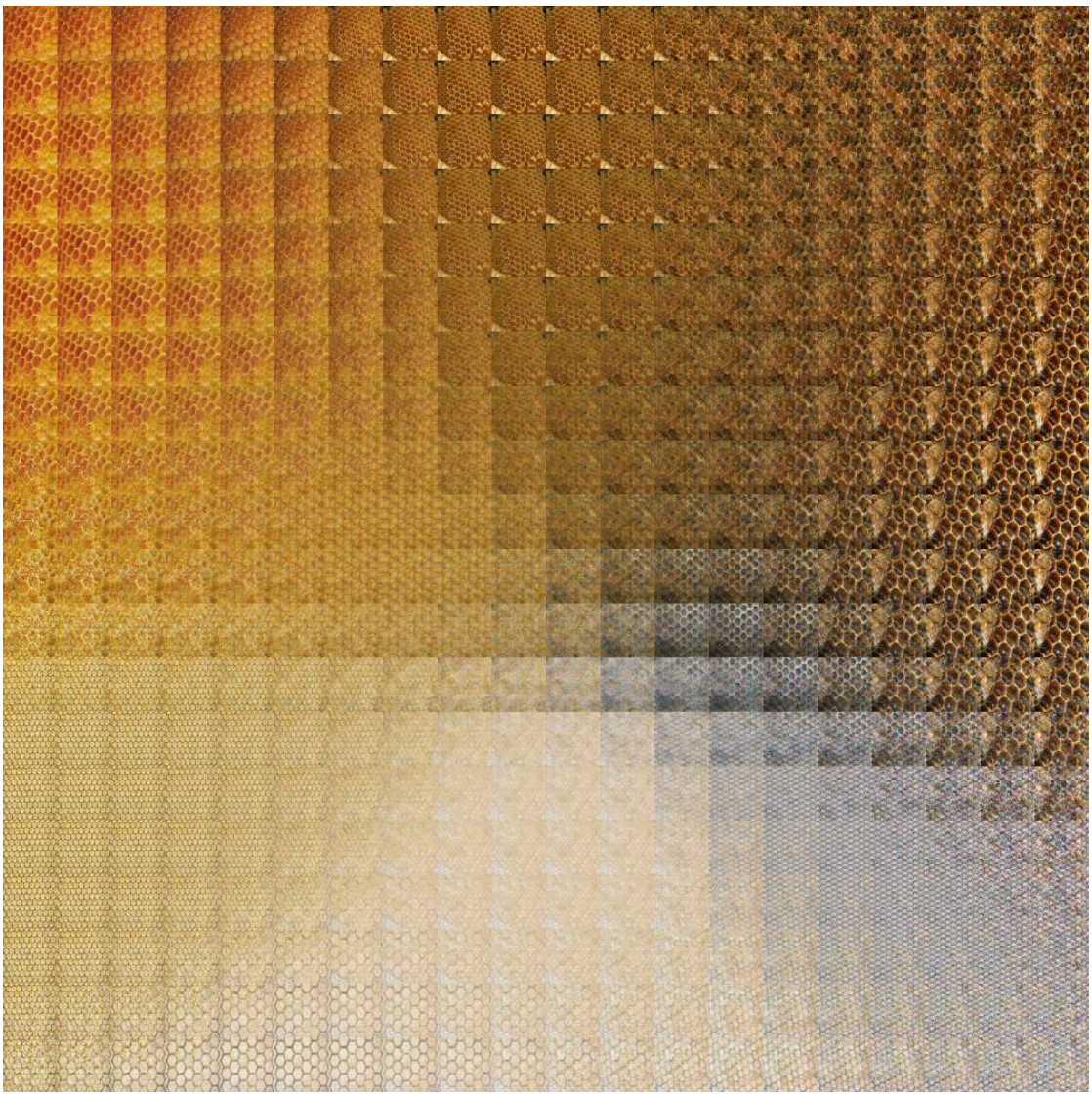

Figure 7: Grid of images generated by the model trained on the `honey` dataset. The two spatial directions correspond to the two main dimensions of the PCA, the others are set to zero.

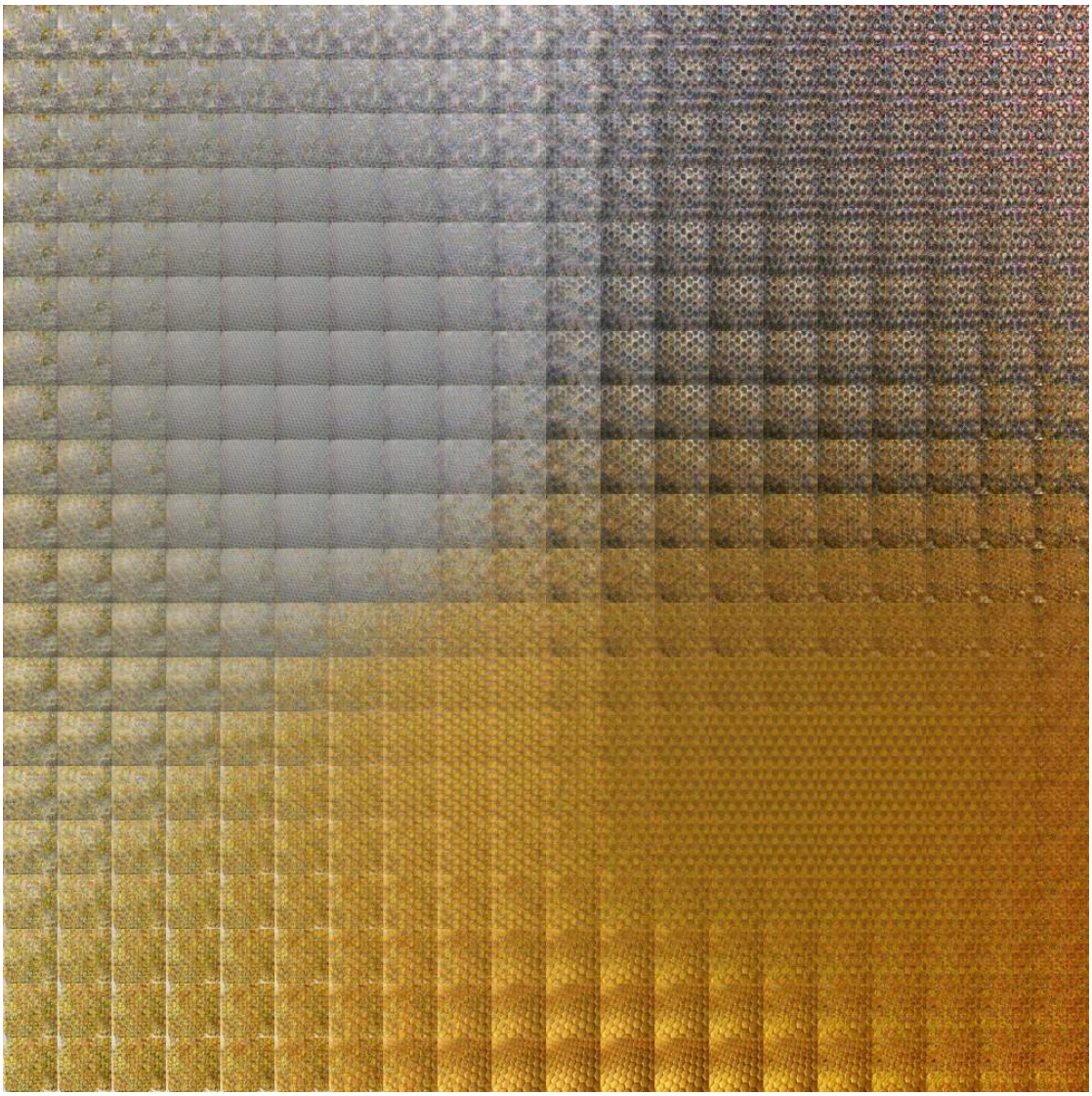

Figure 8: Grid of images generated by the model trained on the `honey` dataset. The two spatial directions correspond to the third and fourth principal components of the PCA, the others are set to zero.

In some places, we can nonetheless observe some great transition between images (bottom of fig. 10), as we saw in fig. 6.

5.5 Comparison

To provide a comparison, the results seems a colored version of the 64x64 black and white images shared by Liam Wynn here, visible in fig. 11. It is thus a real improvement on his work. I also think that I recognize some images from the dataset in the examples he shared. His models seems also subject to over fitting.

6 Conclusion

VAE leads to promising results for texture generation. Even if my current models has several limitations: difficulty to scale up the resolution, overfitting, etc. I was able to construct a meaningful latent space from only several dozens of images. Despite the low number of data and the small amount of computing power used to train the models, some interesting phenomenon can be observed.

VAEs seem to be a great compromise between the image quality, the simplicity to train, and the amount of data required.

For the sake of the demonstration, I used `VEATGen` to synthesize 4 new textures (2 from the `cracked` model and 2 from the `honey-large` model). I used the `make seamless` function in `GIMP` to create a seamless texture usable in a virtual scene. You can see these texture used in a (really) simple 3D scene in fig. 12.

References

[1] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[2] Wenqi Xian, Patsorn Sangkloy, Varun Agrawal, Amit Raj, Jingwan Lu, Chen Fang, Fisher Yu, and James Hays. Texturegan: Controlling deep image synthesis with texture patches. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8456–8465, 2018.

[3] Rohan Chandra, Sachin Grover, Kyungjun Lee, Moustafa Meshry, and Ahmed Taha. Texture synthesis with recurrent variational auto-encoder. *arXiv preprint arXiv:1712.08838*, 2017.

[4] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

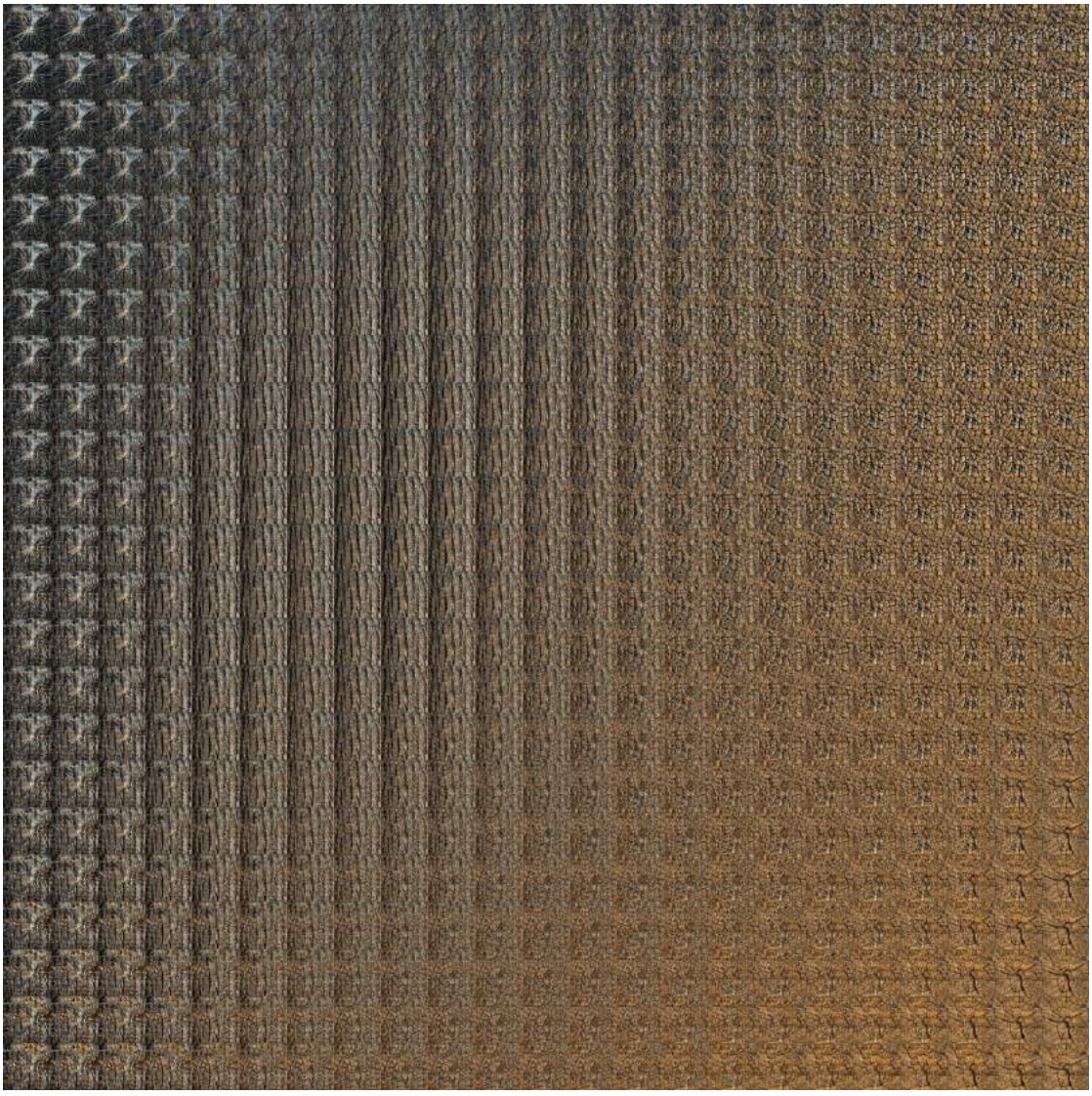

Figure 9: Grid of images generated by the model trained on the `cracked` dataset. The model provides interesting interpolation between materials.

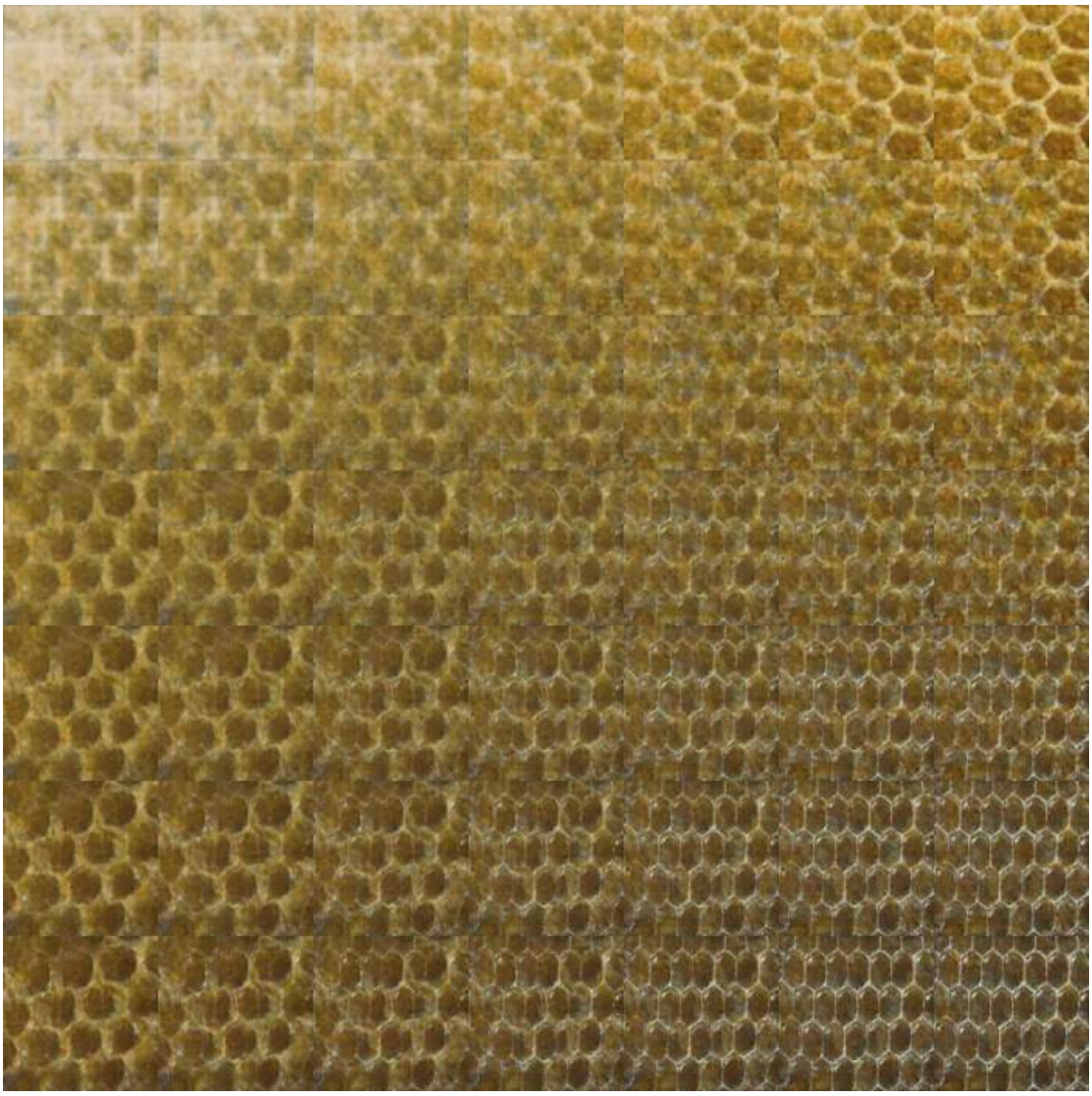

Figure 10: Grid of images generated by the model trained on the `honey-large` dataset. Some of them are blurry and some don't make sense at all (top left corner).

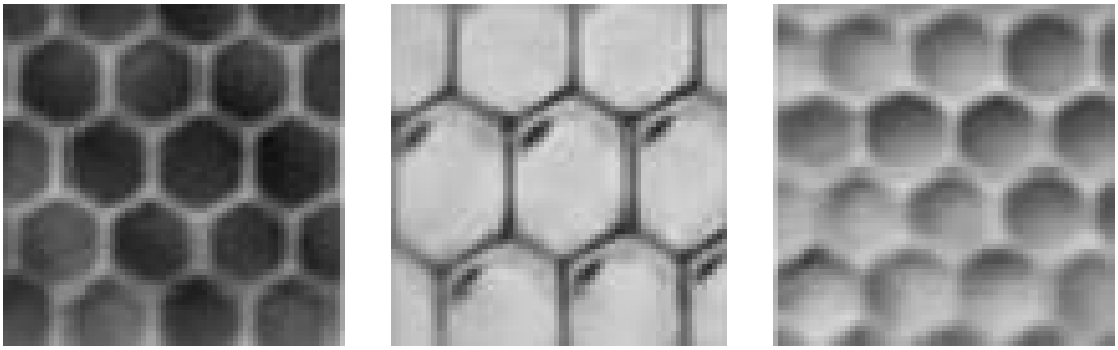

Figure 11: 64x64 black and white images created by Liam Wynn using VAE on the same dataset. Source.

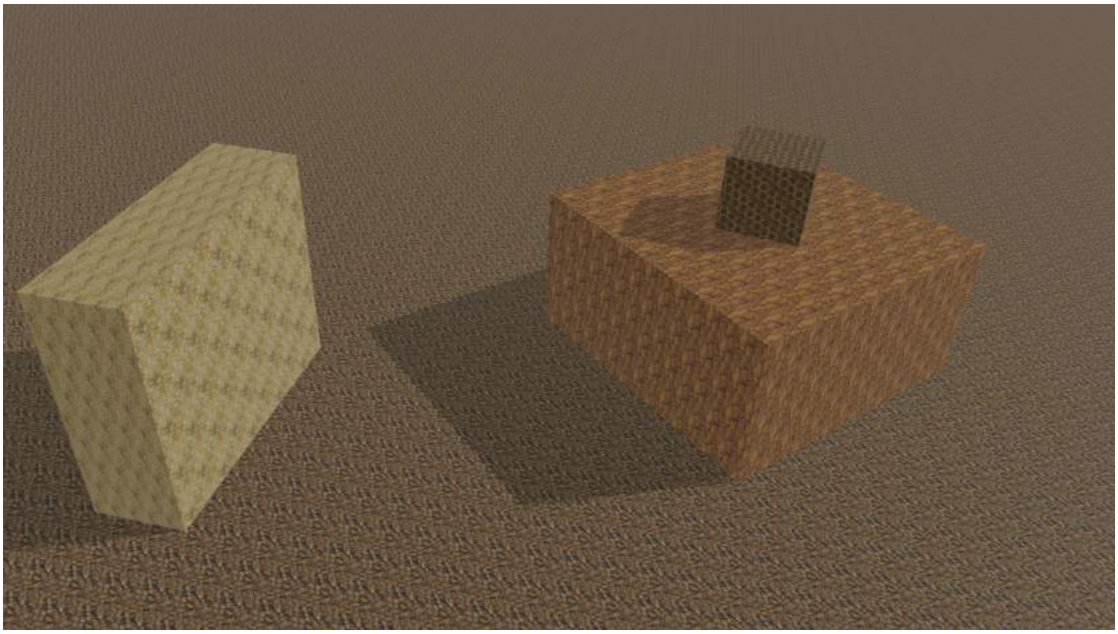

Figure 12: A scene created with `Blender` to illustrate the use of `VEATGen`. I use four textures from `VEATGen`, transformed to become seamless using `GIMP`.