Generating HP Mosaic seeds using generative adversarial network

Raz Saroya

This project submitted for partial fulfilment for Master's in Computer Science

Supervisor: Dr. Adi Shraibman

School of Computer Science

The Academic College of Tel Aviv-Yaffo

Tel-Aviv, Israel

September 2021

Table of Contents

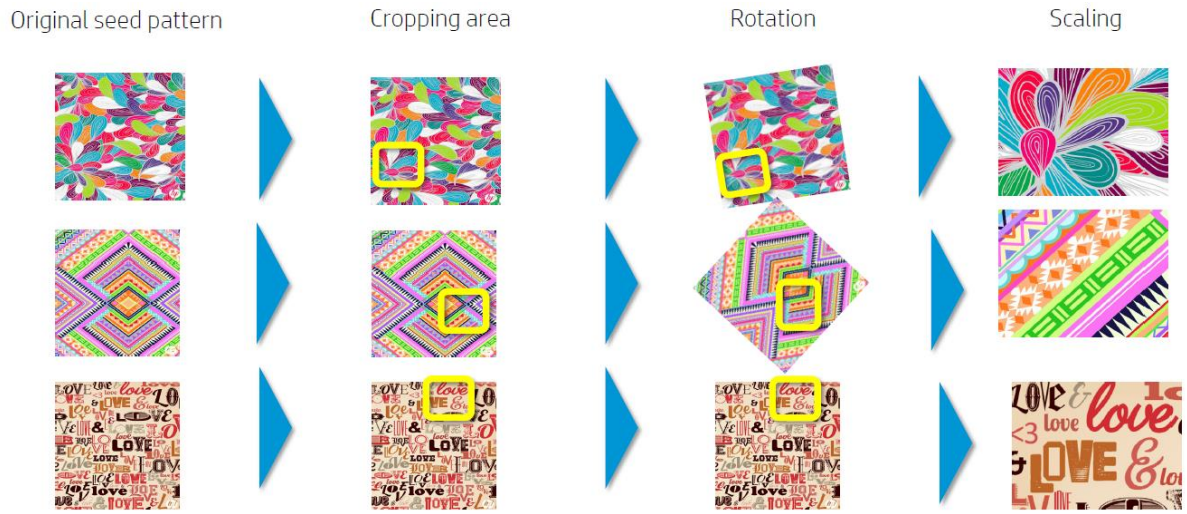
Introduction

HP SmartStream Designer is an Adobe InDesign and Illustrator plugin that let you create sophisticated, custom jobs through a simple, affordable variable data tool optimized for HP Indigo digital presses. It includes exclusive features like HP Collage and HP Mosaic that let you create virtually unlimited design variations.

HP Mosaic let you create automatic designs based on core patterns. It automatically generates a potentially unlimited number of unique graphic designs based on a fixed number of base patterns (also called **seeds**). Those seeds are the heart of the process. HP Mosaic technology uses an algorithm to create designs from those patterns.

As mentioned, the heart of HP Mosaic are one or few base patterns which are vector based graphics (SVG or PDF format). The graphics are varied by rotation, scaling and color change to always create new patterns. The user can also configure the algorithm with commands and choose which colors should be retained or exchanged to the cropping area. With that, potentially millions of variations become unique. To achieve the best results, the base patterns should be as complex and colorful as possible. The more complex the basic patterns are, the results will be better and less similar to each other.

Mosaic algorithm flow -



Here are some examples of HP Mosaic campaigns:

Problem description

The purpose of this project is to help designers create those Mosaic seeds, which are unique and complex base patterns designs fitted to HP Mosaic requirements. The goal is to use machine learning to generate base patterns to feed the Mosaic algorithm with. The generated base patterns should be complex vector patterns in SVG or PDF format, based on a dataset of other seeds and user input.

Today, in order to use the HP Mosaic algorithm, designers have to create their own base patterns or search for the right seeds from various stock photo libraries that fits mosaic design requirements – **complex graphics in vector format that fits to the specific campaign/brand**.

The challenges of the designers today are:

1) **Time consuming** - takes a lot of time to create or find the right base patterns
2) **Non-exclusive** – the same base patterns in stock photo libraries are used by many users. Brands of designers usually worry that **someone else is using the same designs!**
3) **Weak design outcome** - inexperience designers may **not** know how to create or choose the 'right' base patterns that fit the Mosaic requirements and result into weak mosaic designs.
4) **Costly to be unique**: in order to have unique patterns, designers must create the pattern from scratch / edit from stock graphics (take time & high cost)

The solution

Harness the power of machine learning for the goal of creating new mosaic seeds based on a dataset of existing seeds that fit the requirements.

There are few architectures that can generate images using unsupervised learning. Auto Encoders and Generative adversarial networks are two of the best of them.

In this project we will use Generative adversarial networks (GAN in short) for the task.

Generative adversarial networks

What is GAN and how can it help to achieve the goal?

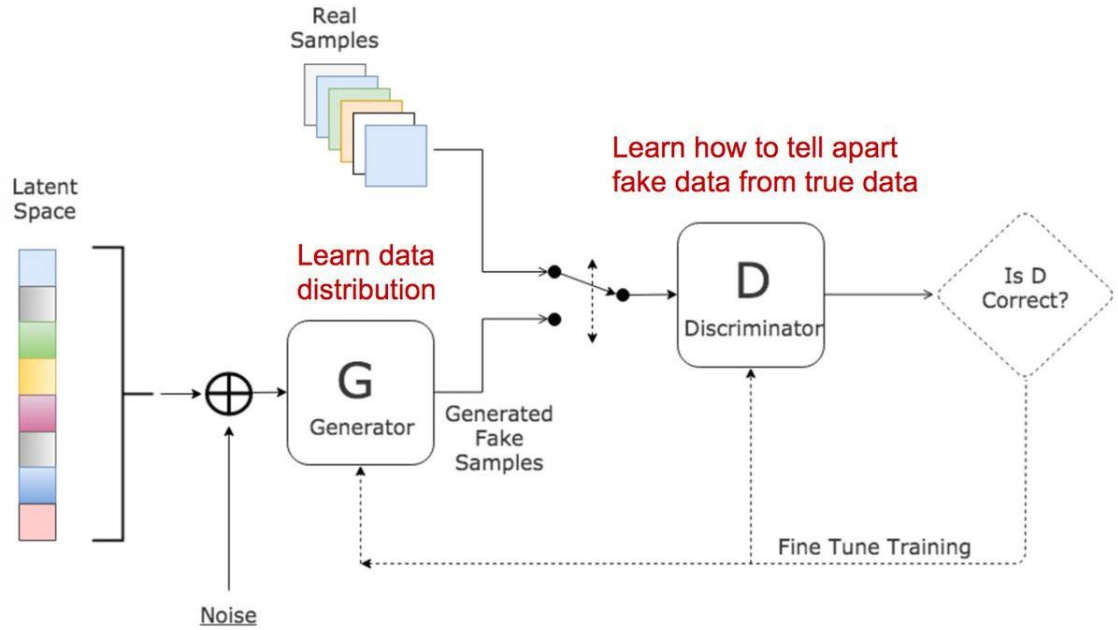
The Goal of a Generative adversarial network is to model what a given dataset looks like (i.e., density estimation), and to be able to generate new examples that "fit" the model. For example, given a dataset of images of cats, the goal is to be able to generate new images of "artificial" cats.

Generative adversarial networks achieve their goal by having two networks that train and compete against each other, resulting in mutual improvisation. The generator misleads the discriminator by **creating compelling fake inputs** and tries to fool the discriminator into thinking of these as real inputs. The discriminator tells if an input is real or fake.

The GAN architecture has two components:

1. Generator - The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
2. Discriminator - The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

General GAN architecture:



There are **3 major steps** in the training process of a **GAN**:

1. Using the generator to **create fake inputs** based on random noise
2. **Training the discriminator** with both real and fake inputs (either simultaneously by concatenating real and fake inputs, or one after the other, the latter being preferred).
3. **Train the whole model:** the model is built with the discriminator combined with the generator.

An important point to note is that the discriminator's weights are frozen during the last step.

The reason for combining both networks is that there is no feedback on the generator's outputs. **The ONLY guide is if the discriminator accepts the generator's output.**

The two models compete in a two-player game, where simultaneous improvements are made to both generator and discriminator models that compete.

We typically seek convergence of a model on a training dataset observed as the minimization of the chosen loss function on the training dataset. In a GAN, convergence signals the end of the two-player game. Instead, equilibrium between generator and discriminator loss is sought.

The minimax objective function

Minimax GAN loss refers to the minimax simultaneous optimization of the discriminator and generator models.

Minimax refers to an optimization strategy in two-player turn-based games for minimizing the loss or cost for the worst case of the other player.

For the GAN, the generator and discriminator are the two players and take turns involving updates to their model weights. The min and max refer to the minimization of the generator loss and the maximization of the discriminator's loss.

The minmax equation between the discriminator and the generator -

$$\min_G \max_D V(D,G)$$

$$V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

As stated above, the discriminator seeks to maximize the average of the log probability of real images and the log of the inverse probability for fake images.

discriminator: maximize log D(x) + log(1 – D(G(z)))

The generator seeks to minimize the log of the inverse probability predicted by the discriminator for fake images. This has the effect of encouraging the generator to generate samples that have a low probability of being fake.

The goal of the minmax game is to find the Nash-equilibria, a state in which no player can improve its individual gain by choosing a different strategy. According to this definition, a Nash equilibrium ($G*$, $D*$) for the GAN minimax problem must satisfy the following for every

$G \in G$ and $D \in D$: $V(G*, D) \leq V(G*, D*) \leq V(G, D*)$.

GAN common problems

GAN has several common failure modes. Although none of these problems have been completely resolved, I will mention few attempts to deal with them.

- **Vanishing Gradients**

  Research has suggested that if your discriminator is too good, then generator training can fail due to vanishing gradients. In effect, an optimal discriminator doesn't provide enough information for the generator to make progress.

  Attempts to handle the vanishing gradients
  - Using the Wasserstein loss function: The Wasserstein loss is designed to prevent vanishing gradients even when you train the discriminator to optimality.
  - Modified minimax loss: The original GAN paper proposed a modification to minimax loss to deal with vanishing gradients.
- **Mode Collapse**

  Usually, you want your GAN to produce a wide variety of outputs. You want for example, a different face for every random input to your face generator.

  However, if a generator produces an especially plausible output, the generator may learn to produce only that output. In fact, the generator is always trying to find the one output that seems most plausible to the discriminator.

  If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject that output. But if the next generation of discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's too easy for the next generator iteration to find the most plausible output for the current discriminator.

  Each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap. As a result, the generators rotate through a small set of output types. This form of GAN failure is called mode collapse.

The following method attempts to force the generator to expand its scope while preventing it from optimizing for a single fixed discriminator:

  - Wasserstein loss: The Wasserstein loss alleviates mode collapse by letting you train the discriminator to optimality without worrying about vanishing gradients. If the discriminator doesn't get stuck in local minima, it learns to reject the outputs that the generator stabilizes on. So the generator has to try something new.
  - Unrolled GANs: Unrolled GANs use a generator loss function that incorporates not only the current discriminator's classifications, but also the outputs of future discriminator versions. So the generator can't over-optimize for a single discriminator.

- **Failure to Converge**

  Because a GAN contains two separately trained networks, convergence is frequently hard to identify.

  Researchers are trying to use various forms of regularization to improve GAN convergence, including:

  Attempts to handle the failure to converge

  - Adding noise to discriminator inputs: See, for example, Toward Principled Methods for Training Generative Adversarial Networks.
  - Penalizing discriminator weights: See, for example, Stabilizing Training of Generative Adversarial Networks through Regularization.

The dataset

For training the GAN properly and getting a variety of unique designs, we had to use as large a dataset as possible for learning stability. I've managed to collect just 400 mosaic base pattens (seeds) in vector format that fits the HP Mosaic algorithm requirements, which is a very small dataset for training a GAN and achieving suitable results. Thus, I needed a larger dataset. In order to increase the dataset size I used data argumentation.

Data augmentation is often performed on the training data to address the issue by position augmentation techniques such as scaling, cropping, flipping, padding, rotation, translation, affine transformation, and color augmentation techniques such as brightness, contrast, saturation.

For the augmentation I used the same tool that process the Mosaic algorithm (HP Smartstream designer), it has features like color shuffle, scaling, rotation, cropping and more. I wrote a script that uses those tools and augmented the 400 images dataset into 4000 images. Each image augmented to 10 different variants.

After the augmentation I converted the dataset of vectors to raster images with RGB channel to fit to the GAN networks.

Samples from the dataset:

GAN Architectures

There are many GAN implementations that attempt to remedy the problems mentioned above.

for image generation task we will use deep convolutional networks for both the generator and the discriminator.

I've tried the most basic architecture for image generation and the most advanced one which are **DCGAN and StyleGAN**.

DCGAN

DCGAN is using deep convolutional neural networks for both the generator and discriminator models. It also contains guidelines for the models and training that result in the stable training of a generator model. It's important because it suggested the constraints on the model required to effectively develop high-quality generator models in practice. This architecture, in turn, provided the basis for the rapid development of many GAN extensions and applications.

DCGAN uses a couple of guidelines, in particular:

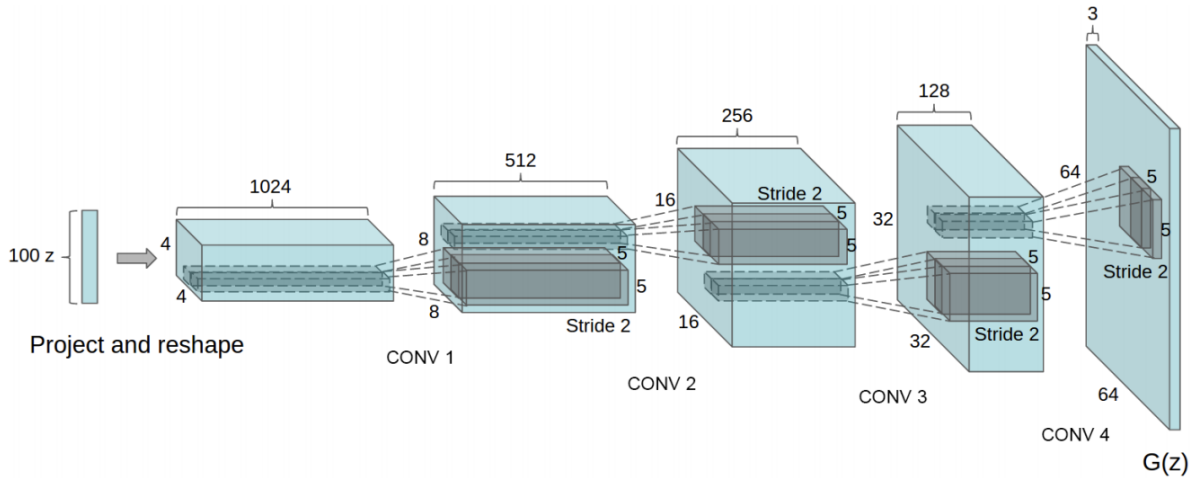
- Replacing any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Using batchnorm in both the generator and the discriminator.
- Removing fully connected hidden layers for deeper architectures.

- Using ReLU activation in generator for all layers except for the output, which uses tanh.
- Using LeakyReLU activation in the discriminator for all layers.

The generator:

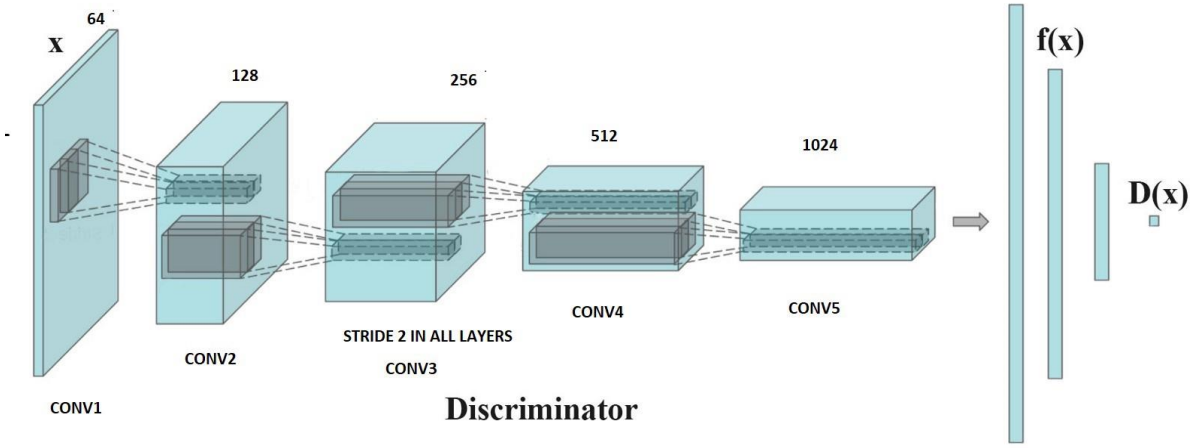
The generator consists of convolution transpose layers followed by batch normalization and a leaky ReLU activation function for up sampling. We'll use a strides parameter in the convolution layer. This is done to avoid unstable training. Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so).

A 100-dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps, A series of four fractionally-strided convolutions converts this high-level representation into a 64X64 pixel image.



The discriminator:

The discriminator also consists of convolution layers where strides are used for doing the downsampling and batch normalization for stability.

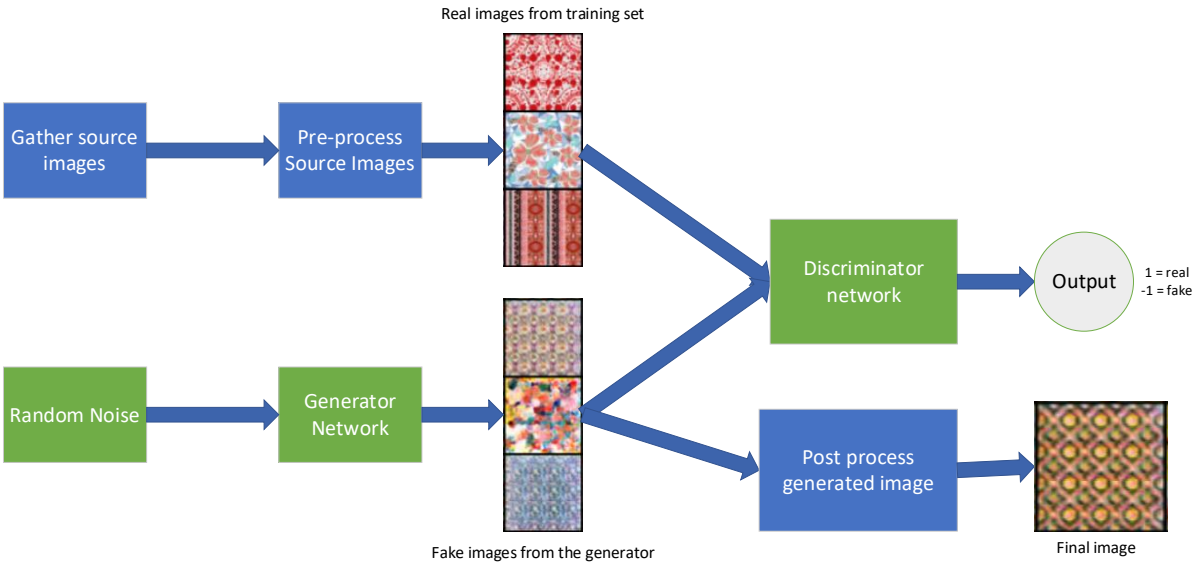


Training

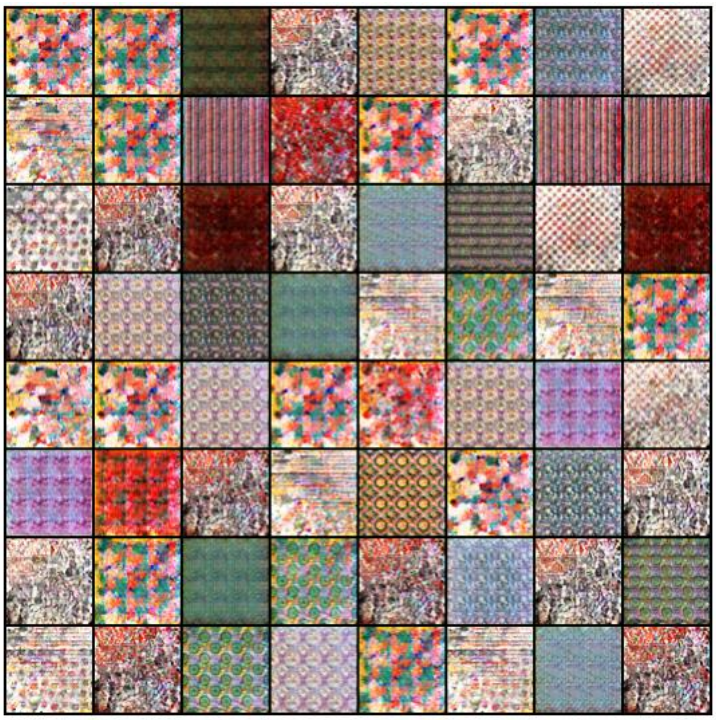
The images were scaled to the size of the desired output (the number of output neurons of the generator and input number of the discriminator), then fed into the Discriminator Network as the "real" images. A set of 100 random numbers are chosen and fed into the Generator Network to create the "fake" images. The job of the Discriminator Network is to determine if the input is real or fake.

I fed the dataset into the DCGAN network with its default settings and ran it for 10000 epochs of 128 batches using "Tesla P100-PCIE-16GB" GPU.

This diagram shows the flow through the system



After the training were ended, the generator network output the following results:



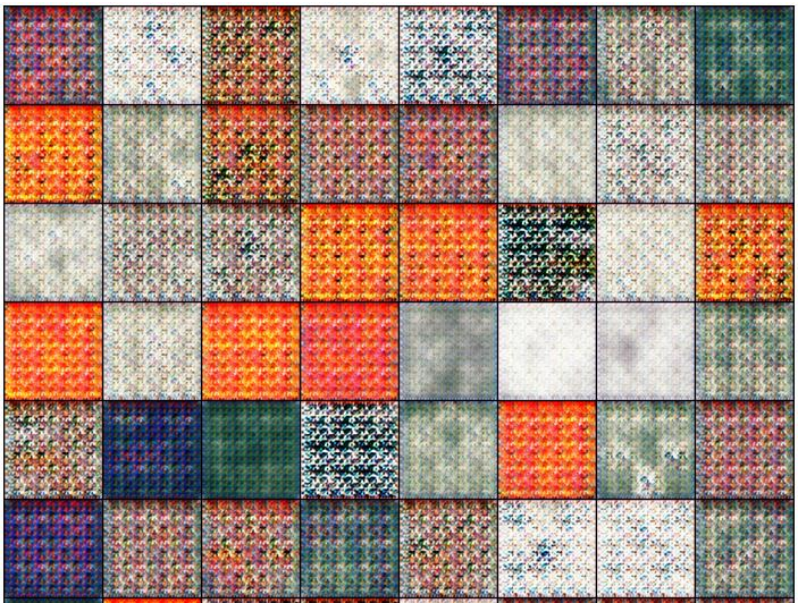
We can see that the generator model learned some of the patterns and colors from the dataset, but the outputs are very noisy and blurry, also the patterns are repeated which indicates that training encountered with mode collapse. I ran the learning process multiple times with dataset changes and different number of batches and epochs, but the result didn't improve much. In general, the DCGAN results were bad comparing to the real images.

I've tried to improve the results by playing with the configurations and the architecture of DCGAN hopefully to stabilize the training process and avoid the mode collapse and hopefully get better results.

Most of the experiments was to play with the architecture setting which are the convolutional stride, padding, layers size values and the number of the layers. All the experiments results ended with worse results than the original DCGAN configurations. The training process was unstable mostly because of the relatively small dataset and the large variety of uniqueness images.

For each experiment I ran the training process for 10000 epochs with different batches sizes (32, 64, 128) which took approximately 24 hours each.

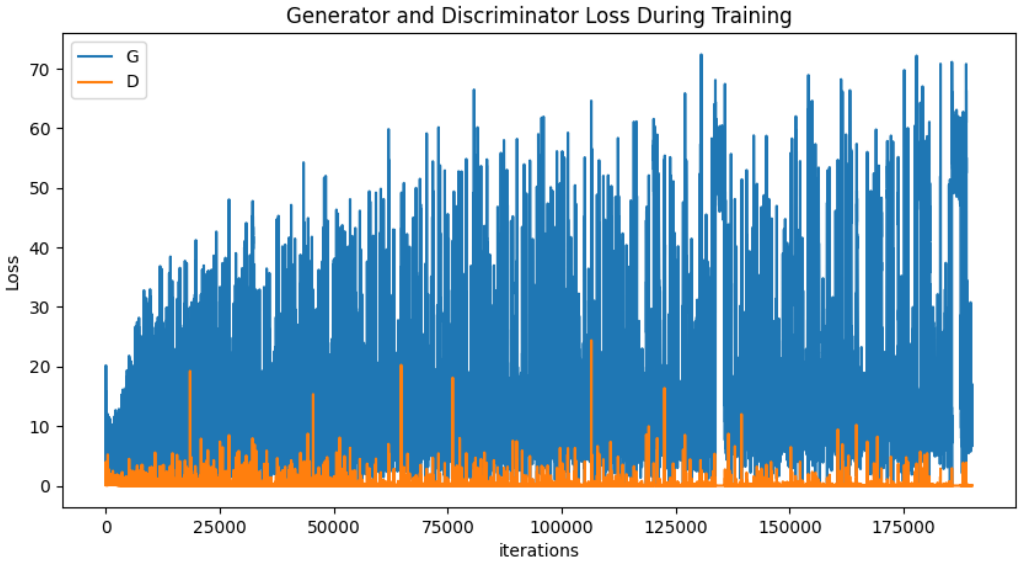
Here is one example of experiment result -



In this test I added 1 layer to both the generator and the discriminator and edited the architecture to output image size of 128X128. (with stride size of 2X2 and padding size of 1)

We can see that the results here were much worse than the original DCGAN architecture. The larger the output images are, the training process became less stable.

We can see how unstable the learning was at the following graph:



StyleGAN

Many improvements to the GAN architecture have been achieved through enhancements to the discriminator model. These changes are motivated by the idea that a better discriminator model will, in turn, lead to the generation of more realistic synthetic images.

As such, the generator has been somewhat neglected and remains a black box. For example, the source of randomness used in the generation of synthetic images is not well understood, including both the amount of randomness in the sampled points and the structure of the latent space.

The Style Generative Adversarial Network, or StyleGAN for short, is an extension to the GAN architecture that proposes large changes to the generator model, including the use of a mapping network to map points in latent space to an intermediate latent space, the use of the intermediate latent space to control style at each point in the generator model, and the introduction to noise as a source of variation at each point in the generator model.

The StyleGAN is an extension of the progressive growing GAN that is an approach for training generator models capable of synthesizing very large high-quality images via the incremental expansion of both discriminator and generator models from small to large images during the training process.

In addition to the incremental growing of the models during training, the style GAN changes the architecture of the generator significantly.

The StyleGAN generator no longer takes a point from the latent space as input. instead, there are two new sources of randomness used to generate a synthetic image: a standalone mapping network and noise layers.

The output from the mapping network is a vector that defines the styles that is integrated at each point in the generator model via a new layer called adaptive instance normalization. The use of this style vector gives control over the style of the generated image.

Stochastic variation is introduced through noise added at each point in the generator model. The noise is added to entire feature maps that allow the model to interpret the style in a fine-grained, per-pixel manner.

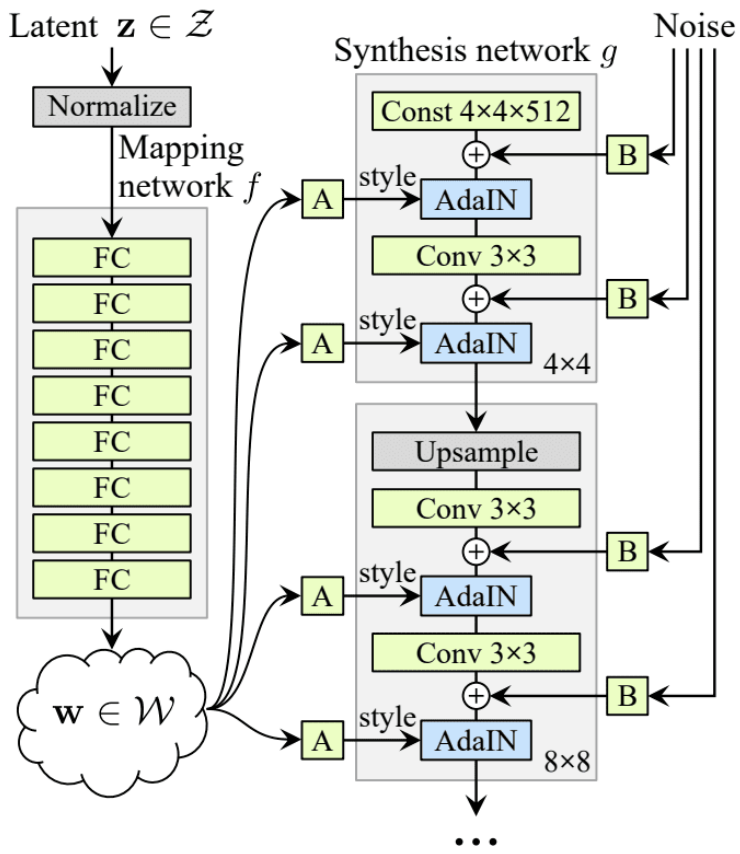
This per-block incorporation of style vector and noise allows each block to localize both the interpretation of style and the stochastic variation to a given level of detail.

The StyleGAN is described as a progressive growing GAN architecture with five modifications, each of which was added and evaluated incrementally in an ablative study.

The incremental list of changes to the generator are:

- Baseline Progressive GAN.
- Addition of tuning and bilinear upsampling.
- Addition of mapping network and AdaIN (styles).
- Removal of latent vector input to generator.
- Addition of noise to each block.
- Addition Mixing regularization.

The image below summarizes the StyleGAN generator architecture.



1. Baseline Progressive GAN

The StyleGAN generator and discriminator models are trained using the progressive growing GAN training method.

This means that both models start with small images, in this case, 4×4 images. The models are fit until stable, then both discriminator and generator are expanded to double the width and height (quadruple the area), e.g. 8×8.

A new block is added to each model to support the larger image size, which is faded in slowly over training. Once faded-in, the models are again trained until reasonably stable and the process is repeated with ever-larger image sizes until the desired target image size is met, such as 1024×1024.

2. Bilinear Sampling

The progressive growing GAN uses nearest neighbor layers for upsampling instead of transpose convolutional layers that are common in other generator models.

The first point of deviation in the StyleGAN is that bilinear upsampling layers are unused instead of nearest neighbor.

3. Mapping Network and AdaIN

Next, a standalone mapping network is used that takes a randomly sampled point from the latent space as input and generates a style vector.

The mapping network is comprised of eight fully connected layers, e.g. it is a standard deep neural network.

The style vector is then transformed and incorporated into each block of the generator model after the convolutional layers via an operation called adaptive instance normalization or AdaIN.

The AdaIN layers involve first standardizing the output of feature map to a standard Gaussian, then adding the style vector as a bias term.

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

The addition of the new mapping network to the architecture also results in the renaming of the generator model to a “synthesis network.”

4. Removal of Latent Point Input

The next change involves modifying the generator model so that it no longer takes a point from the latent space as input.

Instead, the model has a constant 4x4x512 constant value input in order to start the image synthesis process.

5. Addition of Noise

The output of each convolutional layer in the synthesis network is a block of activation maps.

Gaussian noise is added to each of these activation maps prior to the AdaIN operations. A different sample of noise is generated for each block and is interpreted using per-layer scaling factors..

6. Mixing regularization

Mixing regularization involves first generating two style vectors from the mapping network.

A split point in the synthesis network is chosen and all AdaIN operations prior to the split point use the first style vector and all AdaIN operations after the split point get the second style vector.

This encourages the layers and blocks to localize the style to specific parts of the model and corresponding level of detail in the generated image.

The use of different style vectors at different points of the synthesis network gives control over the styles of the resulting image at different levels of detail.

Testing process

The images are fed into the Discriminator Network as the “real” images. A set of 512 random numbers are chosen and fed into the Style Mapper and Generator Networks to create the “fake” images. The result is fed back into the three networks to train them. When the training is done, I post-process the output of the Generator Network to get the final images.

After training the networks for 128 epochs with batches of 64 we have got the following results -

We can see that the images that generated with StyleGAN architecture are still quite far from the training set images but they are less noisy and less repeated than the DCGAN output images that ran with the same settings of epochs and batches. In general, the generated images are still far from being usable.

StyleGAN with Adaptive Discriminator Augmentation

One of the major improvements in StyleGAN ADA is dynamically changing the amount of image augmentation during training.

Image augmentation has been around for a while. The concept is fairly simple. If you don’t have enough images to train a GAN, it can lead to poor performance, like overfitting, underfitting, or the dreaded “model collapse”, where the generator repeats the same output image. A remedy for these problems is image augmentation, where you can apply transformations like rotation, scaling, translation, color adjustments, etc., to create additional images for the training set.

A downside to image augmentation is that the transformations can “leak” into the generated images which may not be desirable. For example, if you are generating human faces, you could use 90° rotation to augment the training data, but you may not want the generated faces to be rotated.

Nvidia found that augmentations can be designed to be non-leaking on the condition that they are skipped with a non-zero probability. So, if most of the images being fed into the discriminator are not rotated for augmentation, the generator will learn to not create images that are rotated

"...the training implicitly undoes the corruptions and finds the correct distribution, as long as the corruption process is represented by an invertible transformation of probability distributions over the data space. We call such augmentation operators non-leaking. — Tero Karras, et al."

The new version of StyleGAN has a feature called Adaptive Discriminator Augmentation (ADA) that performs non-leaking image augmentations during training. A new hyperparameter, p, in the range of 0 to 1, determines how much and how often augmentations are to be applied to both the real images and the fake images during training.

Here is a sample of augmentations with different values for p.

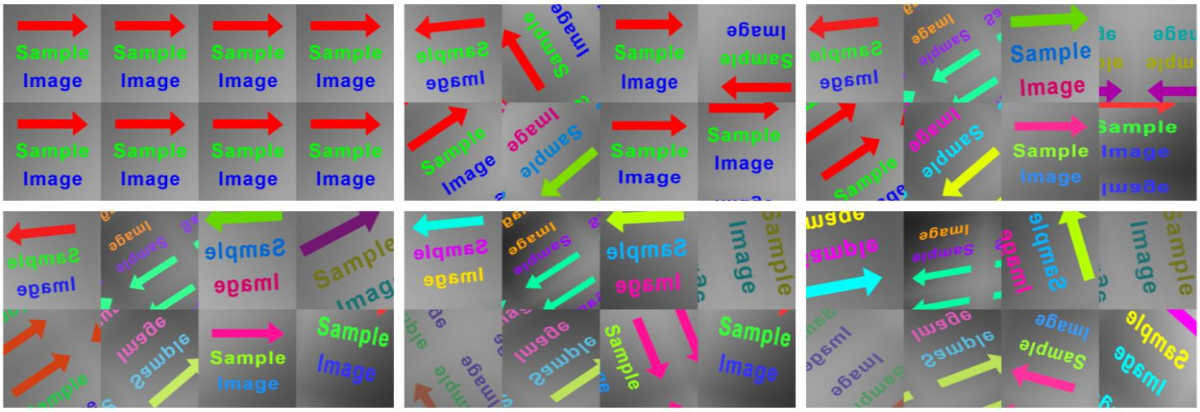


Image Augmentation with (top) p=0.0, 0.2, 0.4 (bottom) p=0.6, 0.8, 1.0, Images by Author

The value p starts at 0 when training begins, and then increases or decreases if the system senses that there is overfitting or underfitting during training.

The heuristic used in the system is based on the sign (positive or negative) of the output of the discriminator during the processing of a batch of generated images. If there are more positive output values than negative, then the trend is towards real images, which is an indication of overfitting. If there are more negative values than positive then the trend is towards fake images, which is an indication of underfitting. The value p is adjusted up or down accordingly after the batch — up for overfitting and down for underfitting.

A target value for p can be set, i.e. 0.7, so there is always a non-zero probability that the augmentations can be skipped, which avoids leaking.

Training StyleGAN with Adaptive Discriminator Augmentation

After training the networks with the same settings as before (128 epochs with batches of 64), the generator output the following images-

Those generated images are far better than the images generated without the augmentation and are already quite similar to our training dataset and can be use as Mosaic seeds after conversion to vectors.

I wanted to go one step further and improve the quality and apply some styles from abstract artworks to the generated images for getting extra special results. For accomplish that I used technique called “transfer learning”.

Transfer Learning

It may not be intuitive, but it’s possible to improve the quality of the generated images by first training the GAN on a different, larger set of images, and then further train the model using the Mosaic seeds. This technique is called Transfer Learning. It was first described as a technique for training NNs in 1976.

“Transfer Learning is technique that uses a pre-trained neural network trained for Task 1 for achieving shorter training time in learning Task 2. [4] — Stevo Bozinovski”

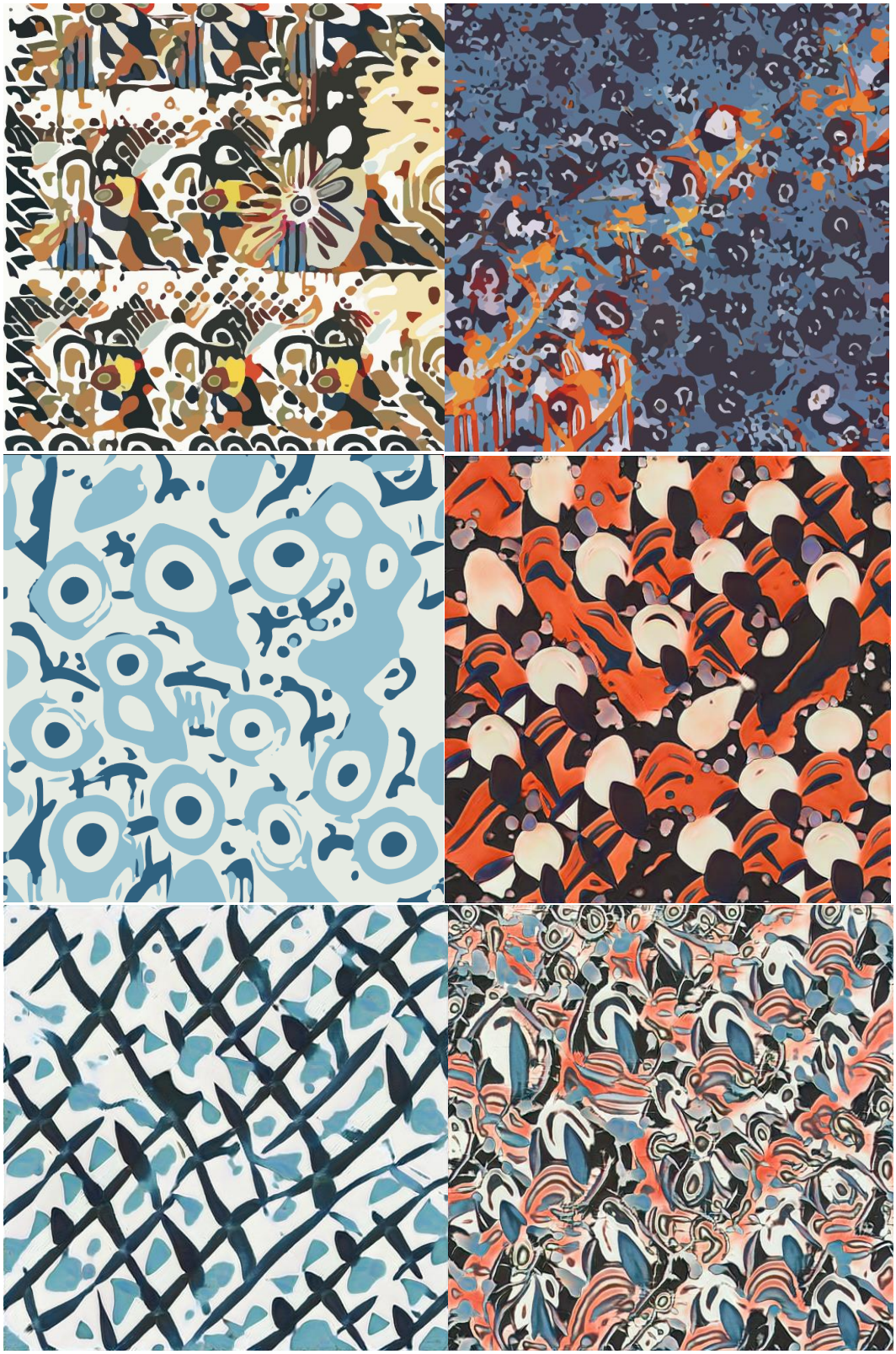
I found a pre-trained GAN network on a public domain that had been trained with tens of thousands of artistic photos from Wikiart, and I used it as my training starting point.

Here are some of the art works used for the pretrained network:

This pretrained network not only improved the quality of the generated images but also added artistic abstract touch from the wikiart dataset. I ran again the training process with the same configurations and number of epochs as before and got the following results -

The results with this technique and the pretrained wikiart network as our starting point were amazing! The generated images were much better, diverse and beautiful from the original dataset thanks to the weighs of the wikiart network.

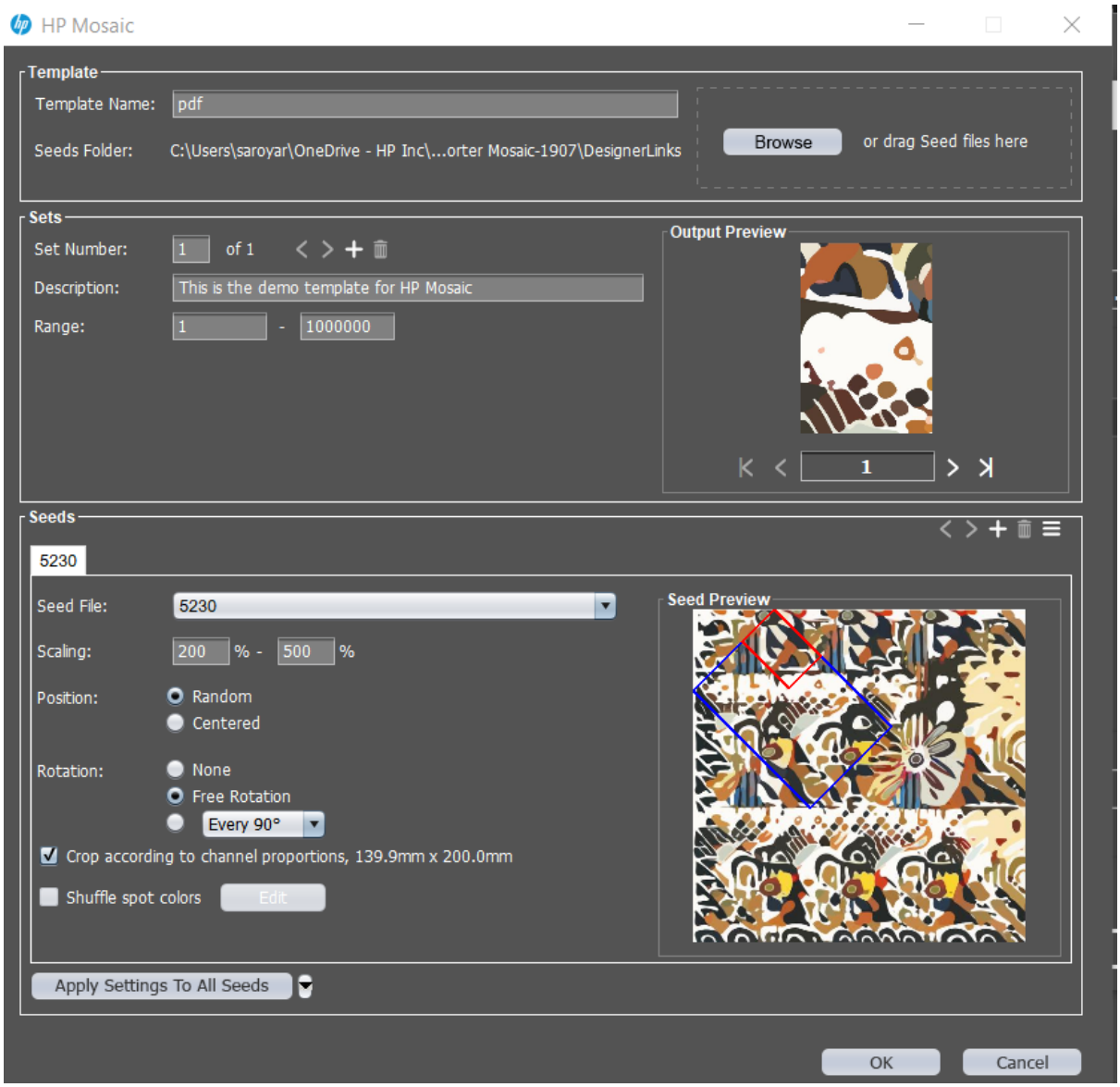
Here are some large samples –



Mosaic generation with the new generated seeds

HP Mosaic algorithm requires the base patterns to be a vector format for not losing quality when transposing and scaling. The conversion of the generated raster images to vector format is done by Adobe Illustrator path tracing algorithm.

The following image is HP Mosaic plugin for Adobe Illustrator, I selected the first sample seed from above and configured free rotation, random position and scaling from 200%-500%.

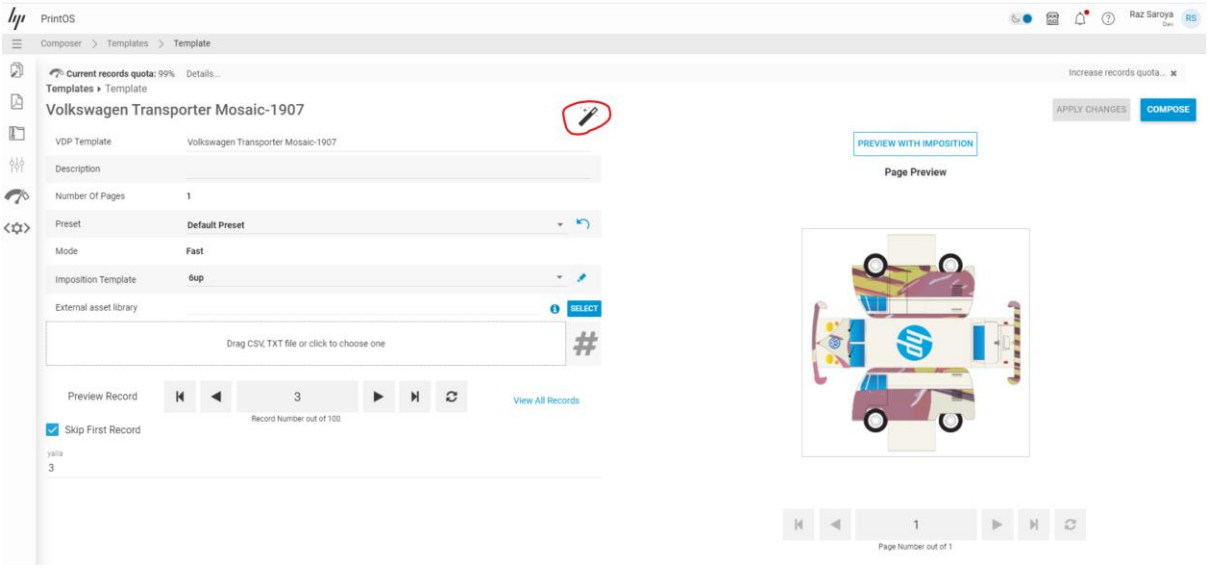


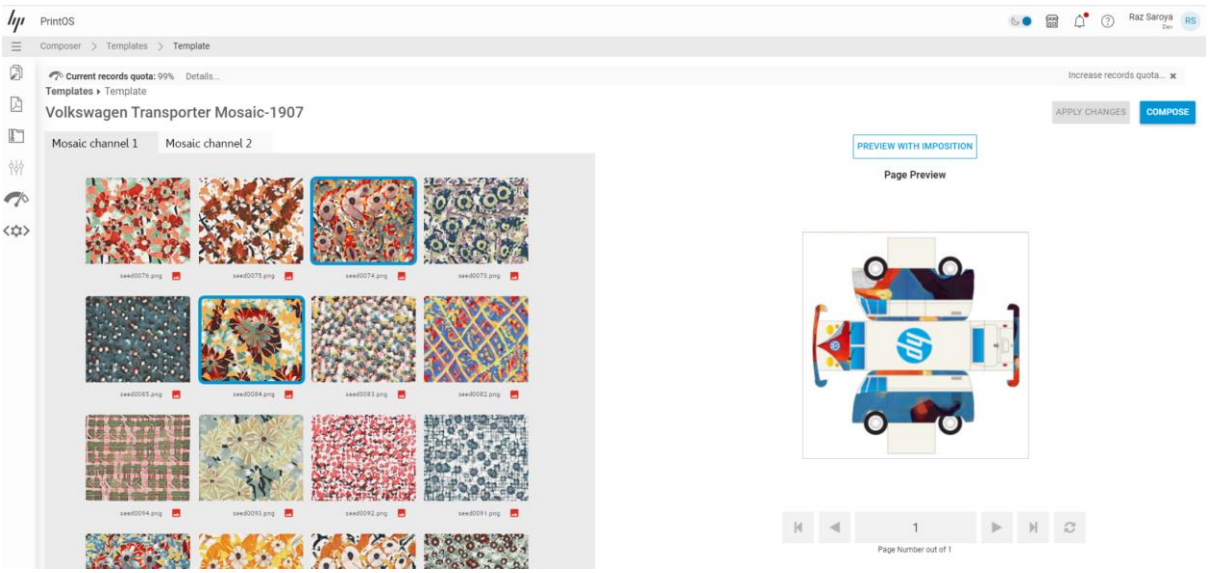
The output of the algorithm with custom template:

Another example of the same template with different seeds:

Changing Mosaic seeds on the flight

The processing of the templates can be very heavy depends on the complexity of the design and number of channels, especially when creating millions of variations. For handle the heavy processing we created a cloud platform for processing jobs on demand. I added a gallery of the generated seeds as a new feature for the application. This feature is available only if the template contains Mosaic channel. With that, the user can browse the gallery and change the Mosaic base pattens before processing, then preview the outcome and process the output.





The results after selecting few seeds from the gallery.

Projecting

The StyleGAN generator model is basically a network of weights, we can try to represent an input image with those weights (projecting an input image into the latent space of the model). We can use this projection to combine the style of an input image with a given seed by learning the input image. For getting only the style of the input image and not trying to represent the actual input image, we need to stop the learning process with half of the loss that we start with.

For example, if we select a seed and learn some company logo, we will get the selected seed with the style of the logo