איליה קוטלוב

321851784

SinGAN: Learning a Generative Model from a Single Natural Image

Introduction:

SinGAN:

The authors of this method propose an innovative pure generative approach for unconditional image generation by learning the internal distribution of exactly one image, from which the model will learn to generate multiple similar images with similar internal distributions. This approach assumes that occasionally a lot of similar details represented in one picture, which are being the same details spread in space and time, I.e in different states. The authors of the method apply a hierarchical observation of the input image by looking at the multiscale representation of it, I.e each learning step looking at a down-sampled image in a coarse-to-fine approach, starting at the coarsest level to find the internal distribution of the most general features, like background, main shapes, general color variety, and each higher scale looking at a more fine representation of the image, which implies learning of more gentle details such as textures, accurate shapes and e.t.c

For each scale there is a GAN which learns to generate images of that scale, thus the whole generation process involves a pyramidical hierarchy of GANS. The authors have chosen their model architecture as fully convolutional, which implied an important property of the images which are generated by it. The convolutions prevent from the model to entirely memorize the learnable image, I.e optimizing the inner weights in a way, which will imply a generation of an identical image as was given, it also implies the learning of the patches of the image, looking at each scale at the image partially. With a single such network, SinGAN model is applied to many image processing tasks beside image generation such as super-resolution enhancement, style transfer and e.t.c

Brief overview of our work:

In our work we focus on the impact of an added attention mechanism to the model, researching such aspect as the performance measured by SIFID and RMSE scores in image generation and super-resolution tasks, along with examining the total effect of it and address the potential of learning single image internal structure. Hypothetically, the attention mechanism, which also appears in many GAN-networks and improves their performance, can enhance the ability of the model to percept important features of the input and the interaction between them on different scales by integrating the features that we're extracted by the districted perceptive field of the convolutions. In that way we are trying to preserve the ability of the network to extract local features for the task of generating different but

similar images and add an ability of the model to look after connections of the features and to look at them in a more integral way, which is achieved by the attention mechanism.

Another task, that well be presented in our work is an expansion of animation generation. Author's original approach suggests a random walk in z-space, which is the space was learned in the adversial-GAN learning proccess from which the generator can generate images from noise. After the training, the user can generate gif with a hardcoded parameters of 0.95 and 0.05 for the z_opt of the image was trained and the new generated noise, in this fashion implementing the random walk in z-space.

We examine this method from a perspective of the traditional sequence approach, addressing our claim that it is not naturally time-series animation, but an analogue of it with an injection of some randomness , because z-space isn't aware of time except the time it has seen internally in the image. We will try to compare this SinGANs authors approach with the traditional models such as rnn and lstm, training the original model in a end-to-end sequence generation task by adding a sequencer into the model convolutional architecture. This is done for the purpose of both enjoy the benefits of a multiscale patch approach to learn the images internal distribution and the ability of a rnn/lstm to percept times-series features.

Summary of relevant previous work:

Generative Adversarial Network (GAN):

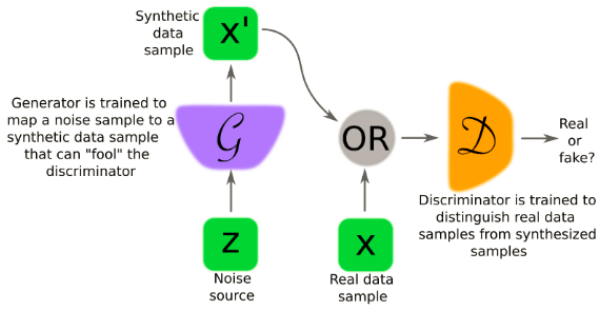


Figure of a GAN from https://arxiv.org/pdf/1710.07035.pdf

A network which is made of a pair of networks which are trained in a semi-supervised or unsupervised fashion, without explicit annotations of the data, in a competitive process in which the Generator network generates data from a random noise and the Discriminator learn to discriminate generated images from real.

The main goal of the generator is to learn the latent space, which models the real data distribution, thus after the learning process, sampling it and decoding it with respect to the optimized parameters in the learning process provides the result of generating new data which has never seen before, which is being predicted as taken from the real distribution of the data.

The optimization process involves the process of the Generator which tries to “confuse” the Discriminator by maximizing its loss of its discrimination of examples that were generated by it and which are real. In this way the Generator never sees real examples but only gets its updates due to the discriminator output, and the process of the Discriminator which tries to maximize its correct classification of real/fake.

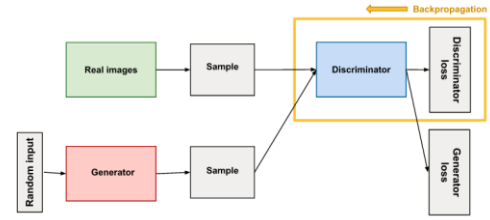


Figure 1: Backpropagation in discriminator training.

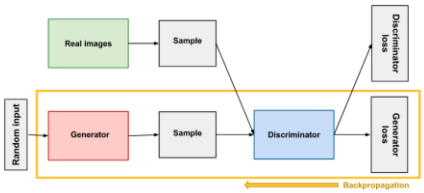


Figure 1: Backpropagation in generator training.

More formally, D and G play the following two-player minimax game with value function V(G,D):

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

(Goodfellow et.al 2014)

Another common loss computation approach is WGAN, which is a an extension of the GAN regular loss, improvement of which is implemented in the SinGAN model. This is the WGAN-GP loss.

In Wasserstein approach (Arjovsky et.al 2017) instead of using a discriminator to determine if the input is real or fake, the WGAN replaces the discriminator with a critic that scores the realness or fakeness of a given input data.

This is motivated by the fact that training the generator should seek a minimization of the distance between the distribution of the data observed in the training dataset and the distribution observed in generated examples.

$$\max_{w\in\mathcal{W}} \mathbb{E}_{x\sim\mathbb{P}_r}[f_w(x)] - \mathbb{E}_{z\sim p(z)}[f_w(g_\theta(z)]$$

(Arjovsky et.al 2017)

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(\boldsymbol{x}^{(i)}\right)+\log\left(1-D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}\log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$
WGAN	$\nabla_w\frac{1}{m}\sum_{i=1}^{m}\left[f\left(x^{(i)}\right)-f\left(G\left(z^{(i)}\right)\right)\right]$	$\nabla_\theta\frac{1}{m}\sum_{i=1}^{m}f\left(G\left(z^{(i)}\right)\right)$

f has to be a 1-Lipschitz function. To enforce the constraint, WGAN applies a very simple clipping to restrict the maximum weight value in ***f***, i.e. the weights of the discriminator must be within a certain range controlled by the hyperparameters

WGAN-GP is very similar to WGAN but it uses gradient penalty instead of the weight clipping to enforce the Lipschitz constraint.

Conditional and unconditional GANs:

Conditional GANs train on a labeled data set and let you specify the label for each generated instance. Instead of modeling the joint probability P(X, Y), conditional GANs model the conditional probability P(X | Y) (Mirza et al, 2014)

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))].$$

Pyramidal GAN architecture:

This idea was firstly introduced in the model of Laplacian Generative Adversarial Network (Denton et al. 2015) as an improvement of a CGAN (Conditional GAN) (Mehdi et al.2014).

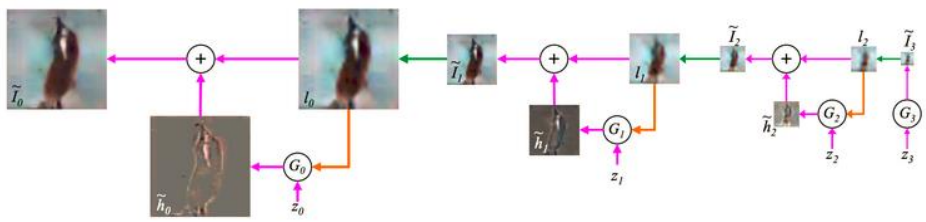
In LapGAN firsttly sampling is done and in the training after it, we have a set of generative convnet models {G_1, G_2, ...}, each of which captures the distribution of coefficients for natural images at a different level of the Laplacian pyramid. Sampling an image is akin to a reconstruction procedure, except that the generative models are used to produce those coefficients.

$$\tilde{I}_k = u\left(\tilde{I}_{k+1}\right) + \underbrace{\tilde{h}_k}_{\text{coef}} = u\left(\tilde{I}_{k+1}\right) + G_k\left(z_k, u\left(\tilde{I}_{k+1}\right)\right)$$

The recurrence starts by setting the final level to be 0 (here the final is the coarsest, I.e the lower comparing to the SinGAN), and using the model at the final level G_k to generate a residual image I_tilda_k using noise vector z_k:

$$\tilde{I}_K = G_K(z_K).$$

Models at all levels except the final are conditional generative models that take an upsampled version of the current image as a conditioning variable, in addition to the noise vector.



Single image training:

While the idea of using single training image has been there for quite some time(mostly being for a specific task like harmonization, style transfer, etc) the proposed procedure takes it one step forward by making the generative model unconditional(generating real-like images sampled from training distribution by passing random noise) and not limited to texture images, as also making this model all-in-one, as being capable of handle many tasks for which GANs traditionally used: Image-to-Image translation as style transfer, Super-Resolution and e.t.c

Attention:

The first introduced method that we apply is the attention mechanism. The attention mechanism is used in a variety of tasks in neural language processing and computer vision and was firstly introduced by Bahdanau (Bahdanau et.al 2015) (Fig 1.) for a neural translation model. The attention mechanism emerged naturally from problems that deal with time-varying data (sequences). It is an architecture-level feature which allows neural networks to *attend* to various parts of and image sequentially and aggregate information over time, that fits well to the perspective of the SinGAN author about the motivation to learn the internal structure of the image, in which same objects are represented in different time (state through the time axe). Hard (Mnih et.al 2015) and soft (Xu et.al 2015, Luong et.al 2015) attention (Fig.2) were introduced for visual tasks:

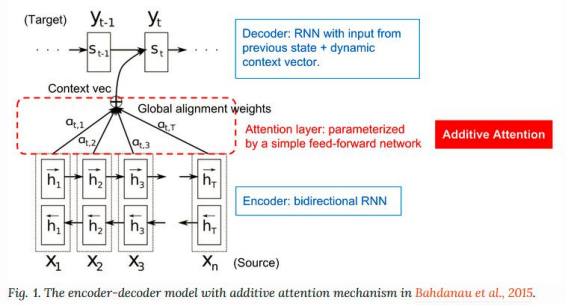
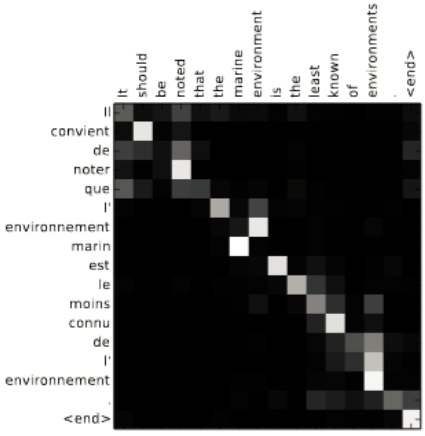
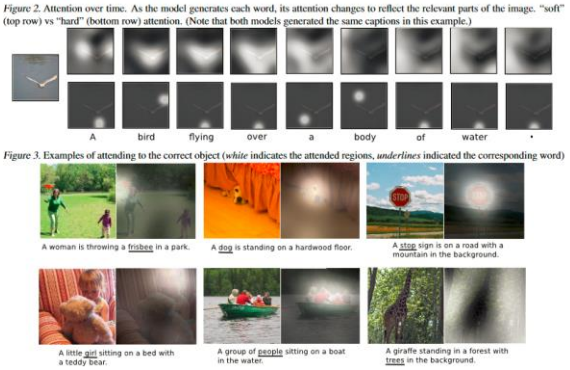


Fig. 1. The encoder-decoder model with additive attention mechanism in Bahdanau et al., 2015.





(from Xu)

After the Additive form of attention introduced by Badhanau (Badhanau et.al 2015), different forms of attention appeared: General, Location-Based and Dot-Product(Luong et.al 2015), Scaled Dot-Product (Vaswani et.al 2017).

Another type of attention mechanism is Self-Attention (referred as “intra-attention” Cheng et.al 2016), which can relate different positions of the same input sequence and theoretically can adopt any of the attentions score functions in different attentions mechanism were mentioned above.

The Transformer was introduced in the article “Attention is all you need” (Vaswani, et al., 2017), this model has a lot of improvements comparing to soft attention and it made possible to do sequence to sequence modeling without recurrent network units. This model is entirely built on the self-attention mechanisms without using sequence-aligned recurrent architecture.

Attention mechanisms used in our experiments heavily based on some architecture concepts introduced by this model and improvements made to the classic attention models and transformer-kind models.

In many of those models appear 3 key-elements: key, value and query.

Given:

n **key-value** pairs: $\{(\boldsymbol{k_i}, \boldsymbol{v_i})\}_{i=1}^{n}$, where $\boldsymbol{k_i} \in \mathbb{R}^{d_k}$, $\boldsymbol{v_i} \in \mathbb{R}^{d_v}$ (n - arbitrary amount of pairs, more is beteer, but mor expensive)
A **query**, $\boldsymbol{q} \in \mathbb{R}^{d_q}$ (encoded input)

The attention layer will compute a weighted sum of the values, where the attention weights computed by some similarity function **e** between the query and the keys:

$$e : \mathbb{R}^{d_k} \times \mathbb{R}^{d_q} \mapsto \mathbb{R}$$

$$\boldsymbol{y} = \sum_{i=1}^{n} \boldsymbol{a_i v_i} \in \mathbb{R}^{d_v}$$

Where:

$$\begin{aligned} \boldsymbol{b_i} &= \boldsymbol{e}(\boldsymbol{k_i}, \boldsymbol{q}) \in \mathbb{R} \\ \boldsymbol{b} &= [b_1, \ldots, b_n]^\top \\ \boldsymbol{a} &= \mathbf{softmax}\,(\boldsymbol{b})\,. \end{aligned}$$

For soft attention mechanism we mentioned above, the attention weights can be computed by the dot-product operator as a similarity function.

For hard attention mechanism values of a could be taken from some fixed distribution.

In a more convenient way, we can stack the keys and the values into matrices:

$$, \boldsymbol{K} \in \mathbb{R}^{n \times d}, \boldsymbol{V} \in \mathbb{R}^{n \times d_v}.$$

Then we can compute the attention for single query by:

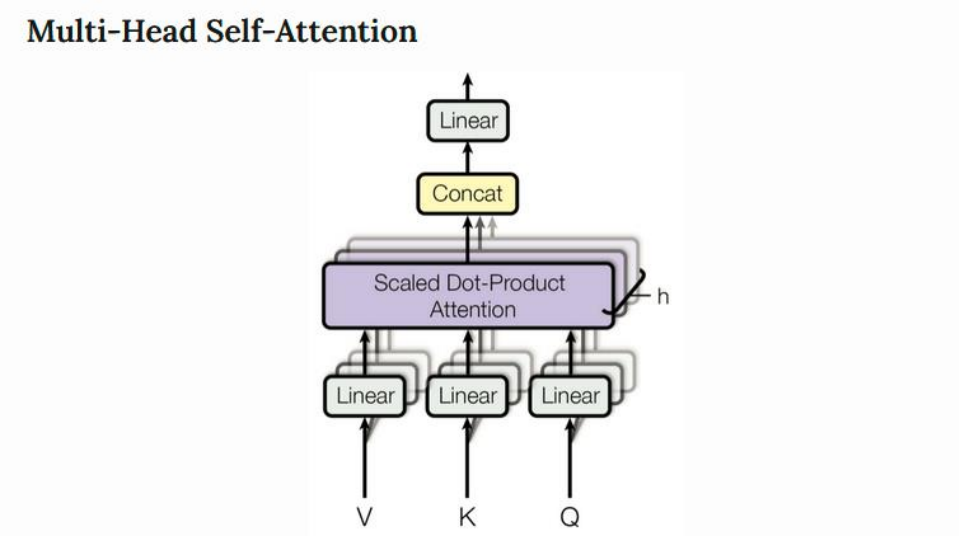
$$\mathbf{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{softmax}(\,\mathbf{q}\,\mathbf{K}^\top)\mathbf{V}$$

The transformer adopts this behavior with a scaled Dot-Pruduct similarity function in the following way:

$$\mathbf{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{n}})\mathbf{V}$$

Where Q is matrix of stacked queries, as was done with the keys and the values and n is the dimension by which the attention is normilized.

One major component of the Transformer attention model is the multi-head self-attention:



which runs through the scaled dot-product attention multiple times in parallel.

Multi-head attention allows the model to jointly attend to information from different representation ***subspaces*** *at different positions. (Vaswani at.el 2017).*

In the previous way of looking, it can be seen as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

GANs with Attention:

Self-Attention GAN (SAGAN) (Zhang et.al 2018) adds self-attention layers into GAN to enable both the generator and the discriminator to better model relationships between spatial regions.

The classic DCGAN (Deep Convolutional GAN) represents both discriminator and generator as multi-layer convolutional networks. However, feature maps which are extracted by the convolutional networks are restrained by the filter size, as the feature of one pixel is limited to a small local region. In order to connect regions far apart, the features must be dilute through layers of convolutional operations and the dependencies are not guaranteed to be maintained.

As the (soft) self-attention in the vision context is designed to explicitly learn the relationship between one pixel and all other positions, even regions far apart, it can easily capture global dependencies. Hence GAN equipped with self-attention is expected to handle details better.

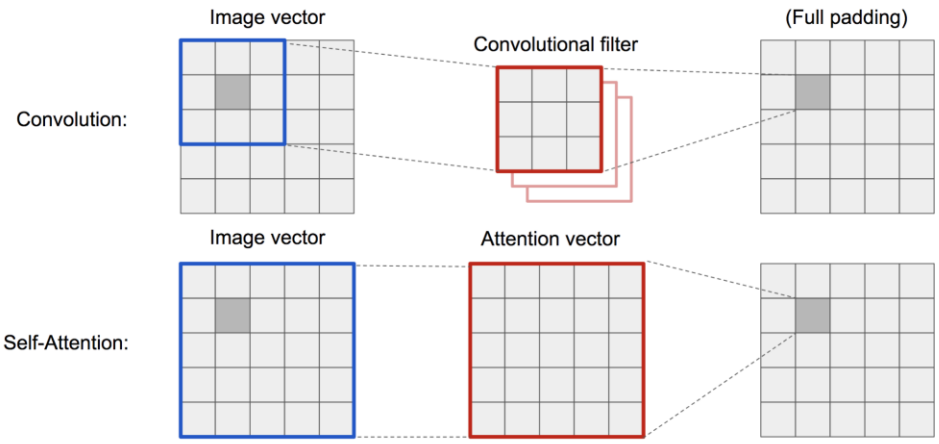


Fig. 19. Convolution operation and self-attention have access to regions of very different sizes.

The SAGAN adopts the non-local neural network to apply the attention computation. The convolutional image feature maps is branched out into three copies **x**, corresponding to the concepts of key, value, and query in the transformer:

- Key: $f(\mathbf{x}) = \mathbf{W}_f \mathbf{x}$
- Query: $g(\mathbf{x}) = \mathbf{W}_g \mathbf{x}$
- Value: $h(\mathbf{x}) = \mathbf{W}_h \mathbf{x}$

Then we apply the dot-product attention to output the self-attention feature maps:

$$\alpha_{i,j} = \text{softmax}(f(\mathbf{x}_i)^\top g(\mathbf{x}_j))$$

$$\mathbf{o}_j = \mathbf{W}_v \left(\sum_{i=1}^{N} \alpha_{i,j} h(\mathbf{x}_i) \right)$$

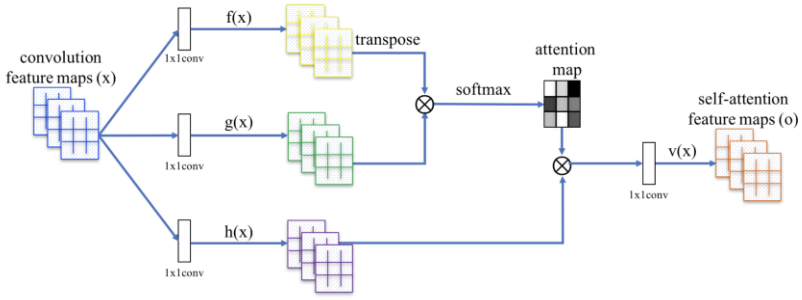


Fig. The self-attention mechanism in SAGAN. (Image source: Fig. 2 in [Zhang et al., 2018](#))

Furthermore, the output of the attention layer is multiplied by a scale parameter and added back to the original input feature map:

$$\mathbf{y} = \mathbf{x}_i + \gamma \mathbf{o}_i$$

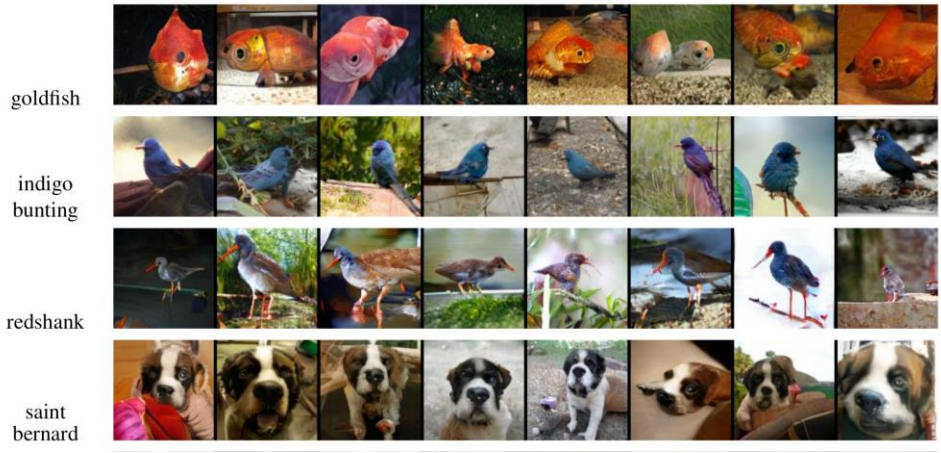
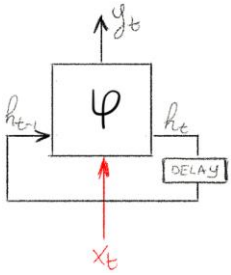


Fig. 21. 128×128 example images generated by SAGAN for different classes. (Image source: Partial Fig. 6 in [Zhang et al., 2018](#))

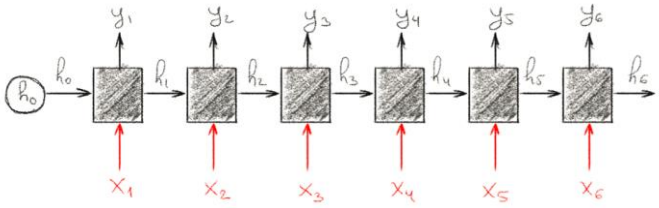
Relevant papers for our second task:

RNN (Recurrent neural network) /LSTM (Long Shortm-Term Memory):

A recurrent unit:



a recurrent unit unrolled in time:



The main difference between a feed-forward single unit composed of two linear layer and an activation function between them and a single recurrent unit is that the recurrent has another input and another learnable parametric matrices, one for producing the state of the unit in concrete time-stamp and one for translating the state to a result output, comparing to the feed-forward unit that doesn't have any memory representation.

This memory is occurring from the recurrent unit processing not only the current input (which is, unlike in feed-forward, serial, I.e has time-stamp) but also a state-output of the previous input which encodes the state of the previous unit aka hidden state, in that way making the recurrent unit communicating through time using this extra outputs of previous unit which are their hidden-states.

More formally:

A basic RNN can be defined as follows.

$$\forall t \geq 0: \\ \boldsymbol{h_t} = \varphi_h\left(\boldsymbol{W_{hh}}\boldsymbol{h_{t-1}} + \boldsymbol{W_{xh}}\boldsymbol{x_t} + \boldsymbol{b_h}\right) \\ \boldsymbol{y_t} = \varphi_y\left(\boldsymbol{W_{hy}}\boldsymbol{h_t} + \boldsymbol{b_y}\right)$$

where,

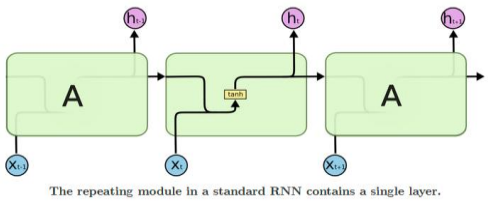
- $\boldsymbol{x_t} \in \mathbb{R}^{d_i}$ is the input at time t.
- $\boldsymbol{h_{t-1}} \in \mathbb{R}^{d_h}$ is the **hidden state** of a fixed dimension.
- $\boldsymbol{y_t} \in \mathbb{R}^{d_o}$ is the output at time t.
- $\boldsymbol{W_{hh}} \in \mathbb{R}^{d_h \times d_h}$, $\boldsymbol{W_{xh}} \in \mathbb{R}^{d_h \times d_i}$, $\boldsymbol{W_{hy}} \in \mathbb{R}^{d_o \times d_h}$, $\boldsymbol{b_h} \in \mathbb{R}^{d_h}$ and $\boldsymbol{b_y} \in \mathbb{R}^{d_o}$ are the model weights and biases.
- φ_h and φ_y are some non-linear functions. In many cases φ_y is not used.

As we expect from the previous intuitive explanation, there is a recursive connection between all the unit through the time-stamp of the inputs and their corresponding hidden-states.

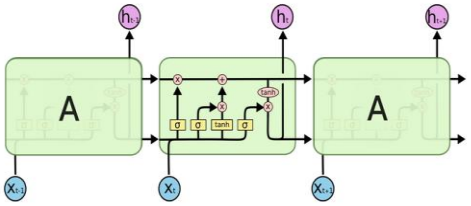
LSTM can be treated as an improved version of RNN with added cell states additionally to hidden states and some additional inner computations relate to it, but conceptionally they can be treated similarly.

The cell state, similarly to hidden state, is affected by the current input and the previous cell state.

RNN Vs LSTM

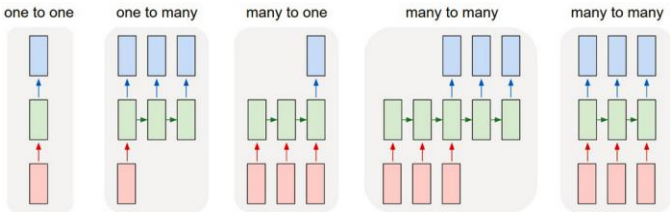


The repeating module in a standard RNN contains a single layer.



$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f) \\ i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i) \\ o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o) \\ c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + b_c) \\ h_t = o_t \circ \sigma_h(c_t)$$

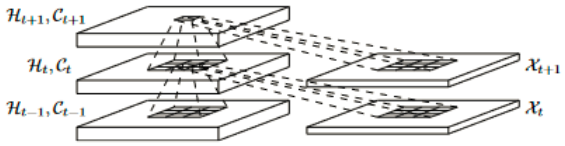
Few popular application structures of RNN/LSTM networks are (as *input* to *output)*:



On which of the following we use we'll expand in the methodology section.

Conv-LSTM:

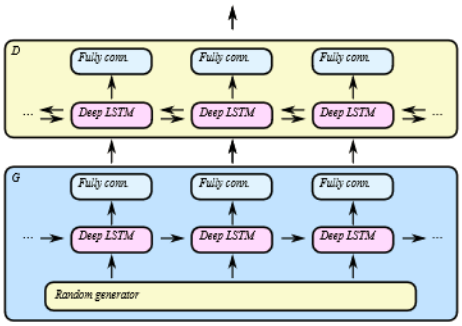
Introduced in the paper "Convolutional LSTM network: a Machine Learning Approach for Precipitation Nowcasting" (et al X-Shi 2015) , it replaces the FC linear layers appear in traditional LSTM by a convolution operator.



$$
\begin{aligned}
i_t &= \sigma(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + W_{ci} \circ \mathcal{C}_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + W_{cf} \circ \mathcal{C}_{t-1} + b_f) \\
\mathcal{C}_t &= f_t \circ \mathcal{C}_{t-1} + i_t \circ \tanh(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c) \\
o_t &= \sigma(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + W_{co} \circ \mathcal{C}_t + b_o) \\
\mathcal{H}_t &= o_t \circ \tanh(\mathcal{C}_t)
\end{aligned}
$$

C-RNN-GAN (Continious Recurrent Neural Network with adversarial training):

This method (et al. Morgen 2016) integrates the method of GAN we saw previously and through the entire original paper and deep learning methods for sequence handling as we just overviewed above.



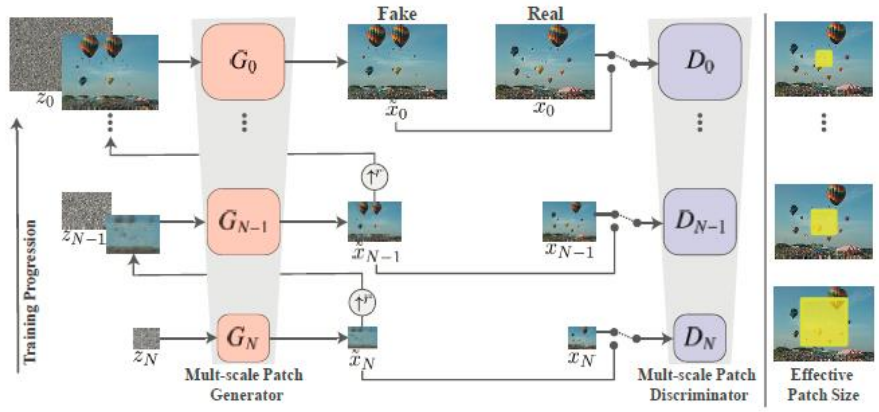
Where the loss is expanded to over all the sequence elements:

$$L_G = \frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(\boldsymbol{z}^{(i)})))$$

$$L_D = \frac{1}{m}\sum_{i=1}^{m}\left[-\log D(\boldsymbol{x}^{(i)}) - (\log(1 - D(G(\boldsymbol{z}^{(i)}))))\right]$$

Methodology:

The model in the SinGAN paperwork built in a hierarchical approach with a set of Generators and Discriminators, a pair for each scale. In a coarse-to-finest fashion, I.e., on level 0 the image is down sampled to the smallest sizes and each scale the image down sampled less. On the first scale the generation is pure generative, and the next scales generate new details from noise and sum the result with the previous output of the previous level Generator. Each epoch is trained for 2000 iterations of 3 steps for each of the Discriminator and the Generator. The WGAN-GP loss is used for the adversarial training as we've seen before. On each epoch the gradients aren't propagated into the previous G-D pairs trained in the previous epochs, they are fixed and only the G-D pair of the current level is trained.

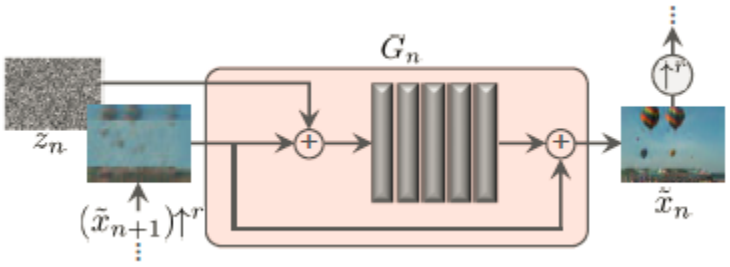


The architecture of the G-D each is the same, which is a regular practice, since the adversarial players should be even opponents to each other, which guarantees a rich development process for both. It consists 5 convolutional layers with small perceptive field, which guarantees that the generator would not just memorize the whole picture but learn the internal distribution of the patches of the image to be capable to generate similar, but not the same image.

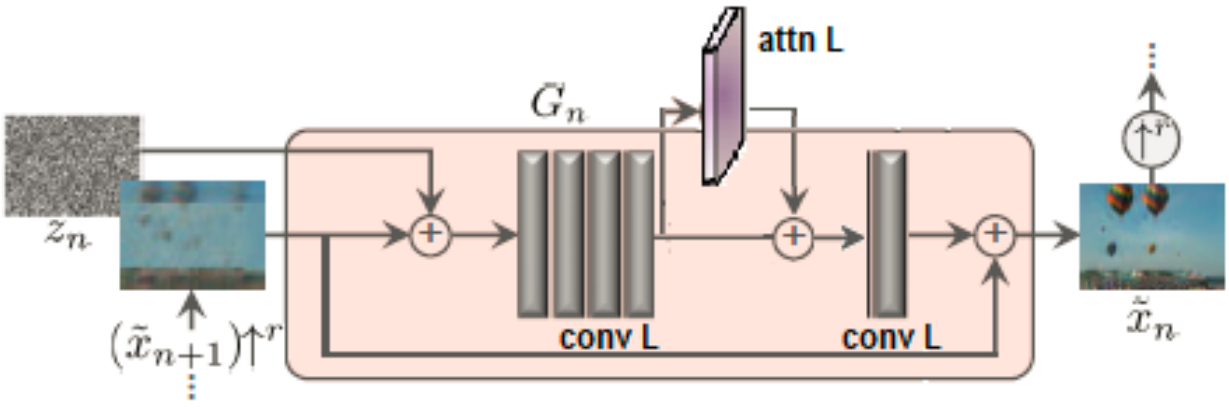
The loss function is a little bit different from what we used to, it contains the regular max-min adversarial loss in a WP-GAN approach with gradient penalty and reconstruction loss from a fixed noise through all of the scales:

$$\min_{G_n} \max_{D_n} \mathcal{L}_{\text{adv}}(G_n, D_n) + \alpha \mathcal{L}_{\text{rec}}(G_n).$$

That loss ensures our original image is in the learned distribution (latent space that represents it to be acurate).



Our improvement is to add attention layer in between the convolutions which extract the feature maps and the final convolution that transforms' the features maps to the generated image. The attention layer added to each pair of the G-Ds, I.e for each scale. The result of the attention is added to the input as it is done in SAGAN and other implementations of GANs with attention.

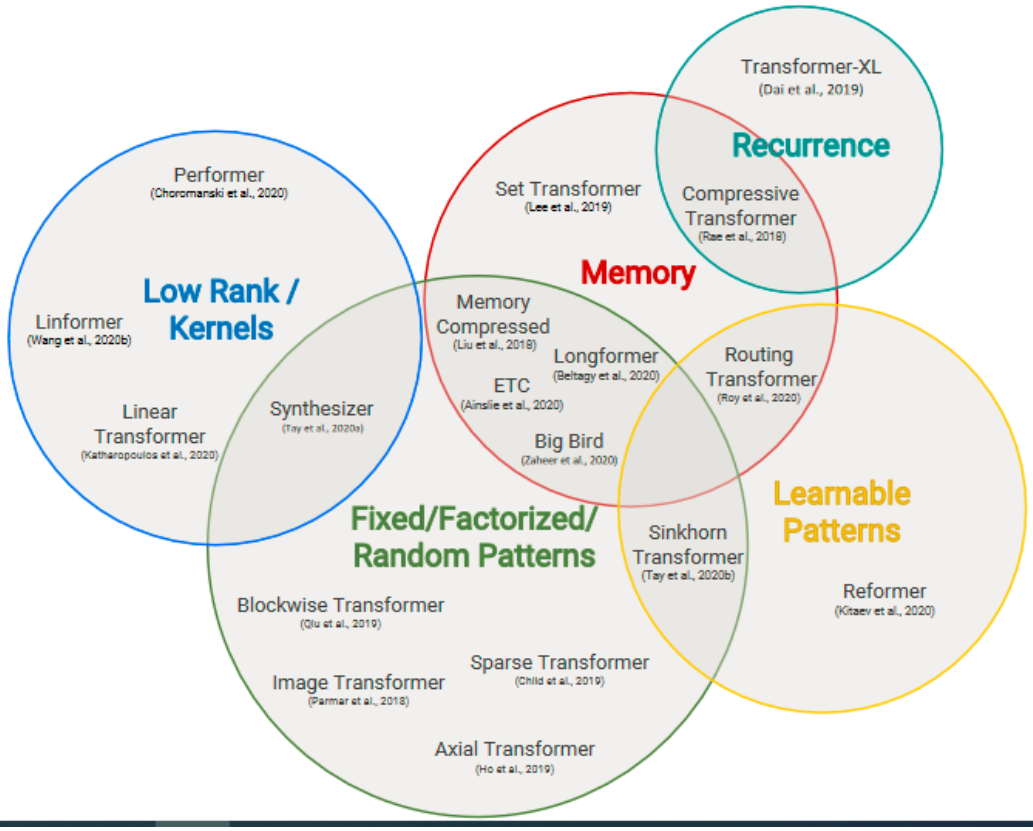


In the experiments we've also tried another known approach to imply the attention, which is used by the Transformer, in it's decoder layer: the attention out will go through additional Feed-Forward unit made of two Linear layers with a non-linear ReLU activation function between them. We've also try the attention without the 1x1 conv on the output, which is another variation of implementing the attention appear in different implementations available on code-share resources.

We've also tried different locations of the attention layer: after the head (first convolution) and after the body (before the last convolution).

One of the major problems and a great drawback of the original method which appear in our experiments is that the model is very memory-consuming due to the fact that the model need all the previous generators in memory each iteration to create the image of the previous scale for up sampling it and adding to it the details generated by the current scale, each such generator approximately weights 500-1000 Mb, thus in the last scale the whole process memory peak can reach about 14 GB in it's original version, and with our addition even more. This prevented from us use the classical attention we've introduced previously, and pushed us to look for and try cheaper versions of attention:

Reducing memory consumption with attention mechanisms less memory-consuming then O(N^2):



Efficient Transformers: A Survey

We've mainly concentrated on the Axial Transofrmer Self-attention implemented in it, but also have tried, not very successfully the Performer attention model and Linear Attention.

(et al. Ya Tay 2020)

For the second part, we've used the CRNN-GAN code from the original paper with our adaptation from music generation to image generation.

We've tried different approaches for the task of animation generation:

Many to Many: the input is a series of frames and the output is the generated series of frames trained in an adversarial fashion as in the original paper only with CRNN architecture totally replacing the original convolutional architecture, this is done for each scale, as in the original paper, in such a way that we learn to generate each frame from coarse to finest scale, and the temporal relations between the frames (which are learned by the RNN).

This approach can be seen as an attempt to make the SinGAN be able to populate GIFs from one given GIF.

Seeing that it is very memory expensive, we replaced the linear layers by convolutions, making our CRNN-GAN being a Conv-LSTM-GAN, which made possible to run training on more then previous 0,1 scales up to 4,5 scales. Then we adjusted the training process to handle OOM exception in a way that we just interrupt the current scale training when the memory is out (it grows for each training iteration), and then continuing to train on the next scale, which made for us possible to see the output of the finest scales, although with bigger reconstruction loss and total quality (SIFID and e.t.c) score for the images, as can be seen visually.

We've also tried one to many approach, where the input size initially is only one frame, and we create from it a sequence of frames, although the Discriminator still gets the whole original sequence of frames for being able to derivate the loss in average for all the generated frames and catch the temporal relations, I.e the sequential movement of the objects in the GIF.

We observe the results in the next section.

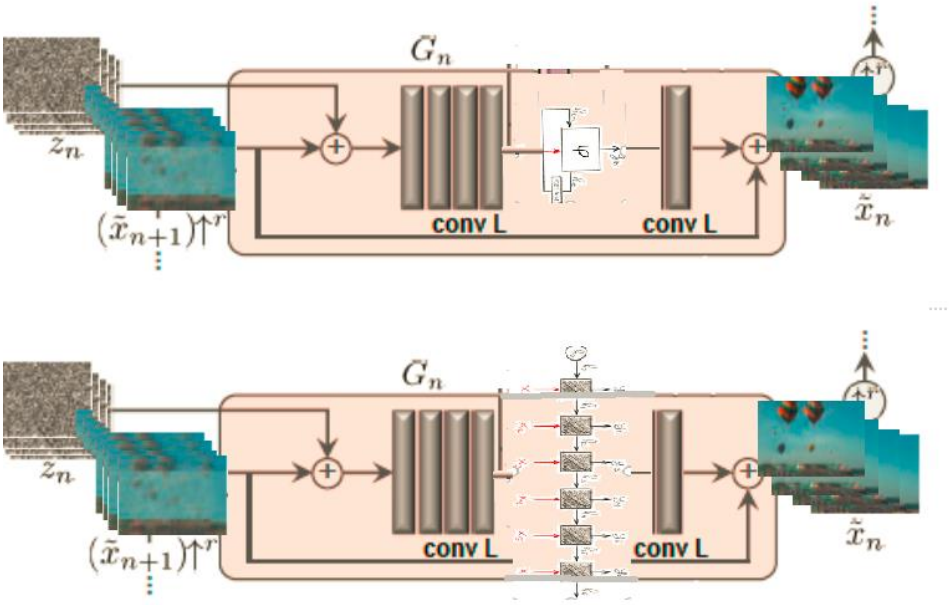
Our main objective is to try to create a high-resolution animation (GIF) from existing GIF or single image as described above, which can capture, unlike the original model some patterns of movement in time, and not just populating random similar states of the same object, which for example in a youtube clip of the performance of this SinGAN ability demonstrate that the leaves of a tree, which are brought to movement through the original method are moving chaotically, and of course if the there is a smal hurricane it can be real, but in that example we will try to make a model which can also make the win blow only into one direction, which implies a sequential movement pattern of the object, which is in this case is a tree. Another example is sea waves, which from one hand could be chaotical, but also could be pretty much structural.

The examples that we use in our experiments are: ice-cream on which red cream flows from up to down with the goal to generate GIFs of icecreams with flowing up to down red cream, but not identical to the original picture.

We use also a more easy to learn animation of a flying plane. This is definitely not something the original method can generate, but it isn't also the most interesting example, because we are more interested in the ability to generate more gentle animation with details, just as the original method takes an image and makes in the SIFID sense. I.e the internal statistics, very similar images.

Finally we show that our model, with the existing memory restrictions in some sense is able to learn the movement of a waterfall and generate similar GIF and also generate a reversed waterfall movement, thus the LSTM architecture injected to the convolutional original architecture produce the models ability to learn sequential movement, although with some restrictions and difficulties we will describe in the following part. We take frame series size to be 15, and for the last experiment 11.

Some illustration of it (with one recurrent unit, which can also be seen in an unrolled mode as in the second image):



- In the second image the input starts from the tom time-stamp and uses the previous hidden state and the next input to generate second output and so on.

We also experiment with the placement of the LSTM architecture. Being between the convolutions, the lstm works with a feature maps from the sampled noise after some deconvolving, placed first it works only with the noise and placed last it works with fully sampled image.

The loss in that case is the mean as for a batch of images, but here it can be thought of as not a batch dimension, but more as a time-series/temporal dimension, and we minimize the loss of each frame averaging over all the frames.

Reproduction of the paper results, experiments, and results:

In our experiments we address with major priority to the SIFID scores to measure how well the image generation with attention will do. SIFID score is the FID score used as it described in the original paper for measuring the GAN wellness for image generation task.

Firstly, lets do the task for the original architecture. The original SIFID scores of 0.05 and 0.09 were done for 50 images when each image is about one hour of training. From our experiments, we've seen there is also some variance in the SIFID scores from train to train, which is expected to due to the adversarial nature of the training process. For that reason in our experiments we've looked at only one image for the SIFID measuring, trying to firstly filter from all the applied attention mechanisms those with the most interesting results.

We've also firstly measured the SIFID between the original size image and the fake, instead of the downsampled and the fake as in the original paper, therefore there is some bias (0.43 approximately) added to the SIFID score, which then we noticed and figured out the relations between SIFID scores of

the same image results, it will be clearer in the results themselves. You can subtract from the scores we've got 0.43 and approximately to have the score in the same scale as in the original paper.

Later we've detailed about it and it is also presented the score directly in the correct scale with the correct SIFID computation.

For the original code with its architecture as is, with the following convolution parameters:

```
GeneratorConcatSkip2CleanAdd(
  (head): ConvBlock(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (body): Sequential(
    (block1): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block2): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block3): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (tail): Sequential(
    (0): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1))
    (1): Tanh()
  )
)
```

```
WDiscriminator(
  (head): ConvBlock(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (body): Sequential(
    (block1): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block2): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block3): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (tail): Conv2d(32, 1, kernel_size=(3, 3), stride=(1, 1))
)
```

From which we can see that the preceptive field is small as was described, then the head convolution expands the input from 3 channels to 32 extracting the feature maps and after that by tail convolution reducing it to an image in the generator and into a True/False value in the discriminator pixel-wise, after which the mean error will be computed and propogated backwards.

For running the training on this image alone, we've got the following SIFID scores, for running them on 1000 images with the original each time:

```
SIFID:  0.6022438
SIFID:  0.5128222
SIFID:  0.6147368
SIFID:  0.54870075
SIFID:  0.63898677
SIFID:  0.5877294
SIFID:  0.5889551
SIFID:  0.6082555
SIFID:  0.6998271
SIFID:  0.5618686
SIFID:  0.5690808
SIFID:  0.60525614
```

The scores were computed between the original jpg and the translated to jpg outputs, which initially were computed as png, but this is an important note, which the authors clarified in project's github issues section.

The scores vary between 0.5 and 0.6 approximately which is 0.07 - 0.17 approximately in the paper terms. For 50 images the SIFID score was 0.09, so generally we did reproduce for one image (comparing to it's fake on 1000 images) the mean of 50 pairs (for each real only 1 fake and not 1000 as we did) comparisons SIFID score.

That is achieved for one of the images taken from the input of the original model "mountains.jpg"

Some of the generated images:

Real downsampled vs real:

```
SIFID:  0.43780917
```

Real downsampled vs reconstructed:

```
SIFID:  0.022671148
```

Fake vs Rec:

```
SIFID:  0.49459746
```

Real Downsampled vs fake:

```
SIFID:  0.11146882
SIFID:  0.13971148
SIFID:  0.16312307
SIFID:  0.13759486
```

Here we can see from the SIFID scores of the original algorithm. As was mentioned, in the paper it is 0.09 comparing to out “real downsampled vs fake’ scores 0.13 approximately in average.

The SIFID score of the real downsampled image vs real is a constant expectedly (just a downsample operation for adjusting the image to its maximal allowed size as in the original paper, which is 250 x 250 with appropriate axes proportions if it isn’t a perfect squre)

Time and memory consumptions:

```
IPython CPU timings (estimated):
  User   :    2991.29 s.
  System :     851.32 s.
Wall time:    3639.17 s.
RAM: Consumed Peaked  Used Total | Exec time 3639.176s
CPU:        35     23     2496 MB |
GPU:        56  14076     1125 MB |
```

As can be seen, a regular run consumes at its peak 14 GB.

One additional run with all the concrete SIFID scores, which helps to see the relations more accurately.

One:

Real vs fake:

```
SIFID:  0.5946847
```

Real vs ds real:

```
SIFID:  0.43780917
```

Ds real vs rec:

```
SIFID:  0.014320088
```

Real vs rec:

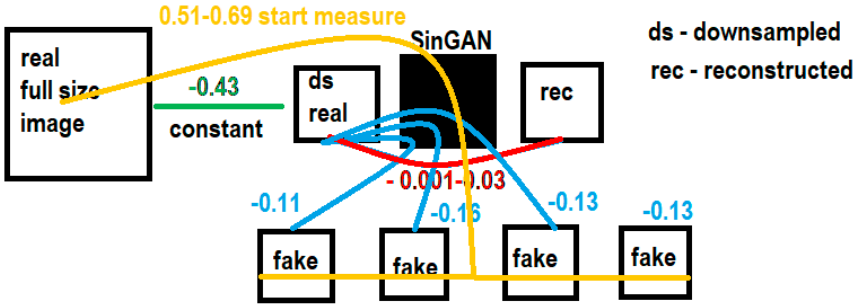
```
SIFID:  0.46832287
```

Rec vs fake:

```
SIFID:  0.09263113
```

Ds real vs fake:

```
SIFID:  0.12306097
```

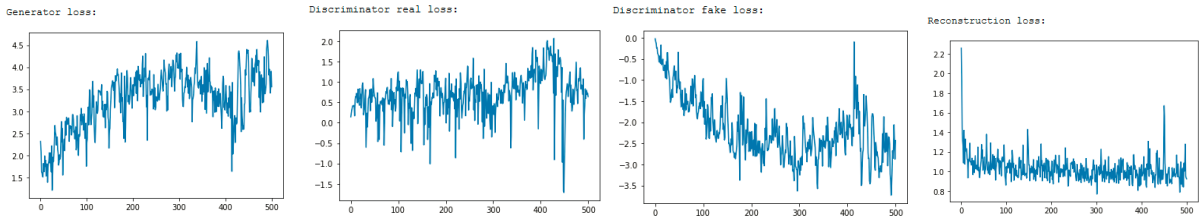


We can examine the SIFID change comparing reconstructed and fake samples, about both we can say they are taken from the same “domain” or distribution, and that SIFID difference is the pure contribution of the latent space which we created to sample an generate images. We can also see that the SIFID change is small between the real down sampled image, which is the input to the reconstructed, which is a desirable result, which tells us that the constraint of reconstruction worked very good preserving the internal distributions of the original image, which means that our latent space contains almost perfectly the original image. We examine those results, unlike the author of the paper, to get more insights and understanding about our suggested architecture improvement attempt.

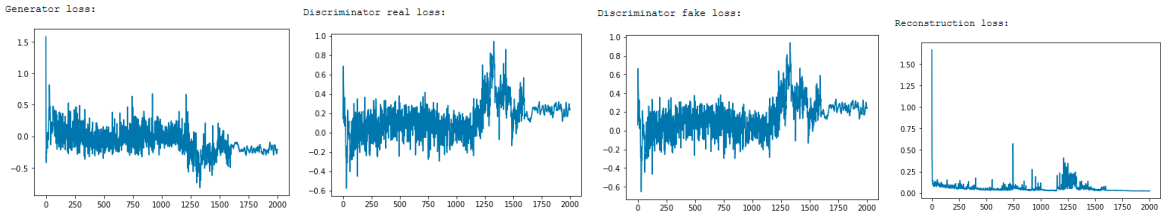
Heavy memory consumption of this model noticed by us after running the following experiment with the full attention mechanism as we’ve seen in the overview costs N x N memory complexity, where N is the number of pixels in an image, therefore even for scale 4 of 8 for an image of maximal size 250 x 250, this matrix of attention weights is enormously huge ((125*125)*(125*125)).

For the further comparisons of the training process, we also present the losses of the generator and discriminator, and the reconstruction loss of the first and the last scale, to see if there are any changes in them and their behavior generally.

0 scale:



8 scale:



One important indicator of the training going well is the mirror reflection relation between of the loss of the generator and the discriminator, which is due to their adversarial learning (when one is succeeding the other is failing), and the total reconstruction loss which for this concrete image at scale 0 is about 1 and at final scale 8 drops beyond 0.1 approximately.

Next experiment we run the model with axial attention mechanism, which is an attention with smaller complexity as described in the methods.

we've got higher SIFID score surprisingly, which means that the internal distribution of the generated images with the attention is less like the generation without the attention mechanism:

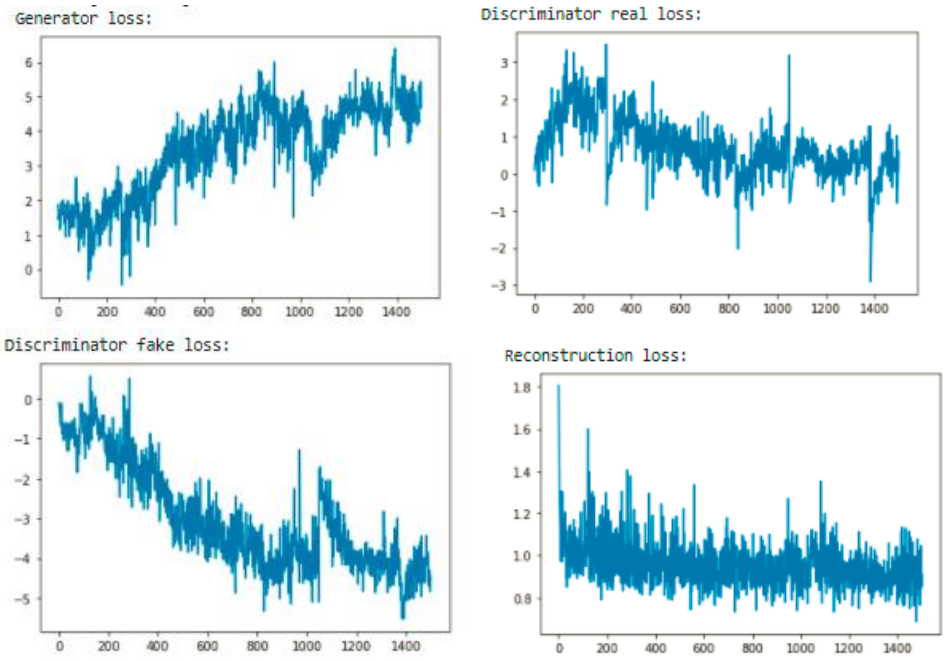
```
SIFID:  0.7525073
```

Here is how the architecture looks, we''l show it once.

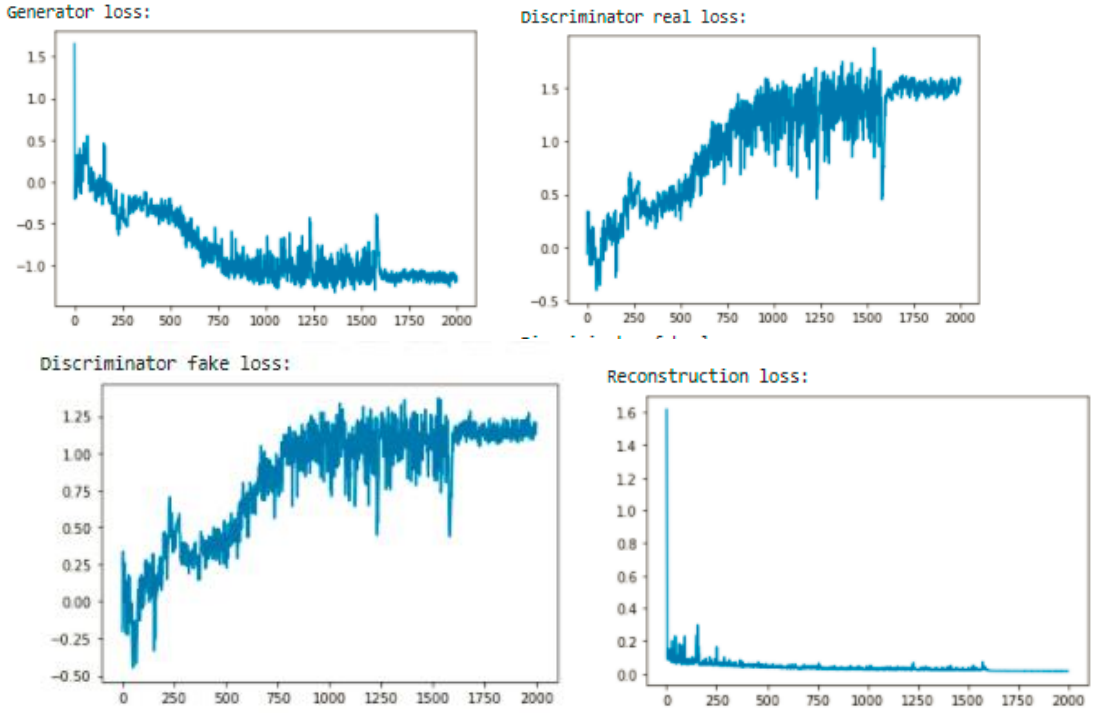
```
AxialGeneratorConcatSkip2CleanAdd(
  (head): ConvBlock(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (body): Sequential(
    (block1): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block2): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block3): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (attn): AxialAttention(
    (axial_attentions): ModuleList(
      (0): PermuteToFrom(
        (fn): SelfAttention(
          (to_q): Linear(in_features=32, out_features=32, bias=False)
          (to_kv): Linear(in_features=32, out_features=64, bias=False)
          (to_out): Linear(in_features=32, out_features=32, bias=True)
        )
      )
      (1): PermuteToFrom(
        (fn): SelfAttention(
          (to_q): Linear(in_features=32, out_features=32, bias=False)
          (to_kv): Linear(in_features=32, out_features=64, bias=False)
          (to_out): Linear(in_features=32, out_features=32, bias=True)
        )
      )
    )
  )
  (tail): Sequential(
    (0): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1))
    (1): Tanh()
  )
)
```

```
AxialWDiscriminator(
  (head): ConvBlock(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (body): Sequential(
    (block1): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block2): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (block3): ConvBlock(
      (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
      (norm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (LeakyRelu): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (attn): AxialAttention(
    (axial_attentions): ModuleList(
      (0): PermuteToFrom(
        (fn): SelfAttention(
          (to_q): Linear(in_features=32, out_features=32, bias=False)
          (to_kv): Linear(in_features=32, out_features=64, bias=False)
          (to_out): Linear(in_features=32, out_features=32, bias=True)
        )
      )
      (1): PermuteToFrom(
        (fn): SelfAttention(
          (to_q): Linear(in_features=32, out_features=32, bias=False)
          (to_kv): Linear(in_features=32, out_features=64, bias=False)
          (to_out): Linear(in_features=32, out_features=32, bias=True)
        )
      )
    )
  )
  (tail): Conv2d(32, 1, kernel_size=(3, 3), stride=(1, 1))
)
```

Scale 0:



Scale 8:



The losses are pretty much the same as without the attention.

```
RAM: Consumed Peaked  Used Total | Exec time 5412.842s
CPU:       42     22     3121 MB |
GPU:      124  14286     1449 MB |
```

The memory consumption remains approximately the same and this attention mechanism made it possible to run the model.

Although SIFID score became bigger, visually it isn't notable.

Fake samples:

Further we will show only the interesting result of the SIFID, the generated images, as the pattern of the losses and the architecture are repeating themselves pretty much similarly or at least analogically.

- Similar results were gotten without Linear layer at the end of attention as in some implementations with SIFID=0.79, thus we didn't researched that architecture choice further.

A much better SIFID comparing to previous, but like the original results are obtained <u>for positioning the axial attention layer after the head (first convolution):</u>

Real vs fake

```
SIFID:  0.5118312
SIFID:  0.4902432
SIFID:  0.5556303
```

SIFID: 0.6

Fake Vs reconstructed:

```
SIFID:  0.45598355
```

```
SIFID:  0.46310917
```

Real ds vs fake:

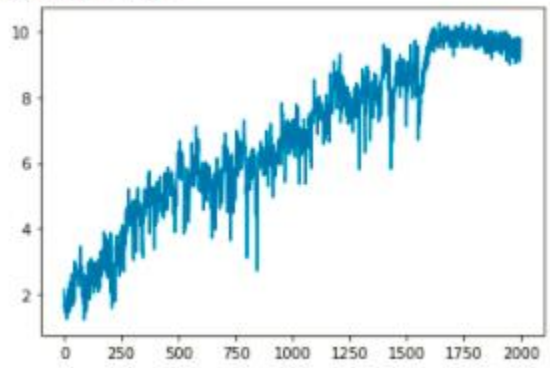
```
SIFID:  0.12205405
```

Downs vs real:

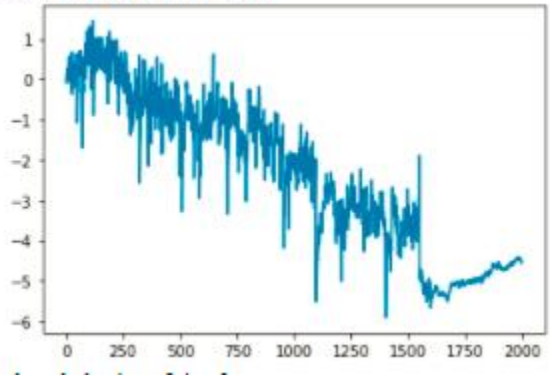
```
SIFID:  0.43780917
```

```
RAM: Consumed Peaked  Used Total | Exec time 8440.169s
CPU:       42     22     2492 MB |
GPU:       98  13438      993 MB |
```

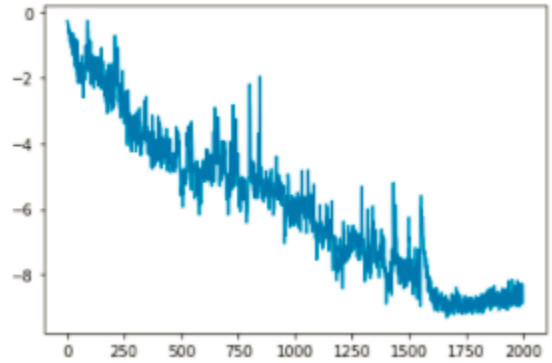
Generator loss:



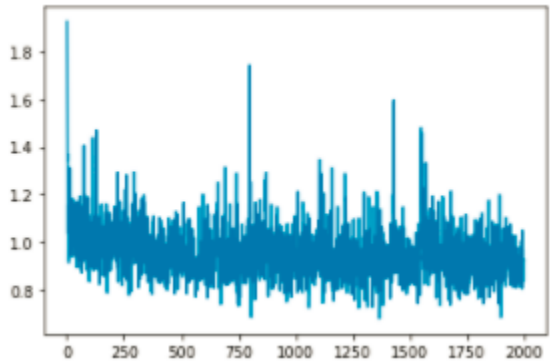
Discriminator real loss:



Discriminator fake loss:

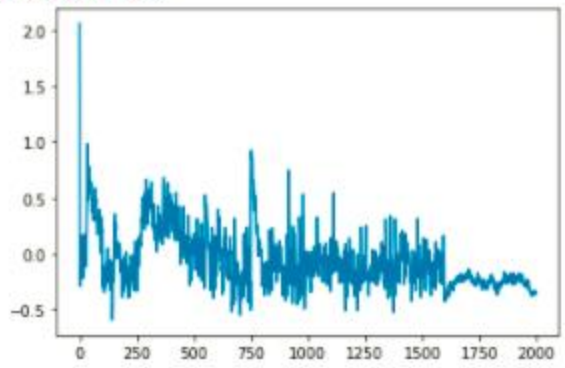


Reconstruction loss:

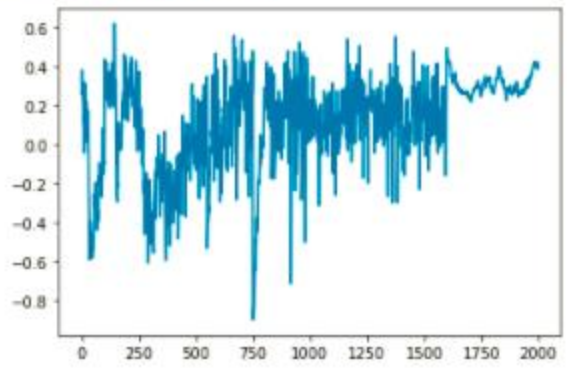


Scale 8:

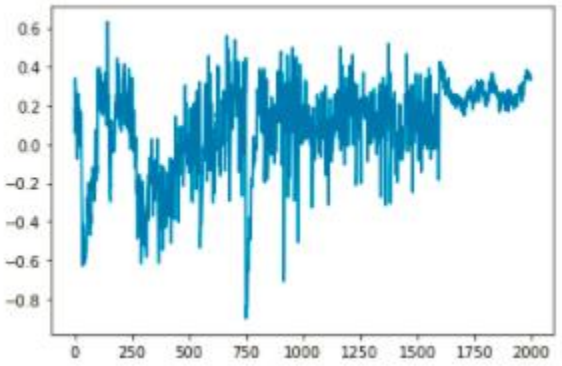
Generator loss:



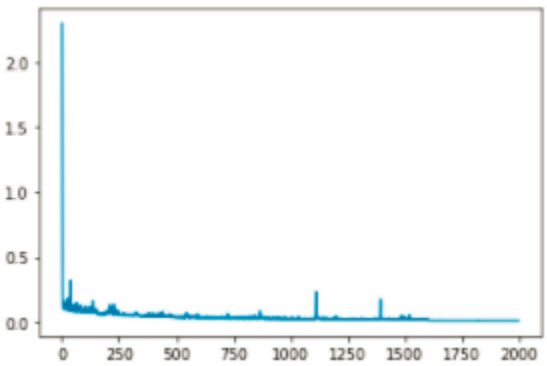
Discriminator real loss:



Discriminator fake loss:



Reconstruction loss:



There are some visual changes in the fake images, the sky became brighter, there is a change in the texture of the ground beneath the water and additional change in the texture of the clouds, although we cannot say if it is for better or for worth.

The SIFID score we did documented in the ds real vs fake (as in the original paper) is about 0.12 is better our initial average of 4 attempts of 0.13 approximately, and we've also gotten 0.49 which is the best result we've ever seen in our experiments, which makes this architecture to be a candidate of improvement of the original results, especially due to visual changes either.

Another last attempt of this architecture with all the SIFID scores also shows another good small result of 0.10 comparing to the 0.13 in average and 0.11 as the best we've seen for the original model.

Real vs fake:

```
SIFID:  0.50021785
```

Ds real vs fake:

```
SIFID:  0.1027337
```

Real vs ds real:

```
SIFID:  0.43780917
```

Ds real vs rec:

```
SIFID:  0.012777567
```

Real vs rec:

```
SIFID:  0.45784637
```

Rec vs fake:

```
SIFID:  0.09755734
```

Adding after another block another attention didn't improve further but made it worse with 0.7 SIFID score (at the beginning), as much as in the end with `SIFID: 0.81400985`

Another variety of using attentions is as in the Transformer, being followed by a feed forward block with normalization and dropout.

And We've got `SIFID: 0.79627615`

And relative (real ds vs fakr, as in the paper): `SIFID: 0.19748399`

Downasampled real vs rec:

```
SIFID:  0.0128432
```

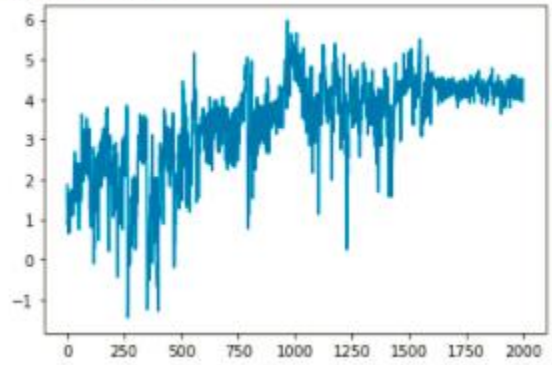
Reconstructed vs fake:

```
SIFID:  0.1196644
```

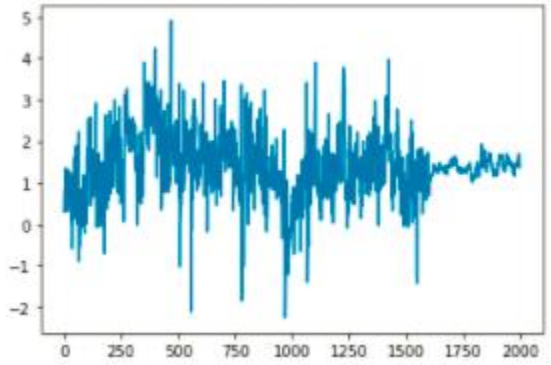
Real vs downsampled:

SIFID: 0.43780917

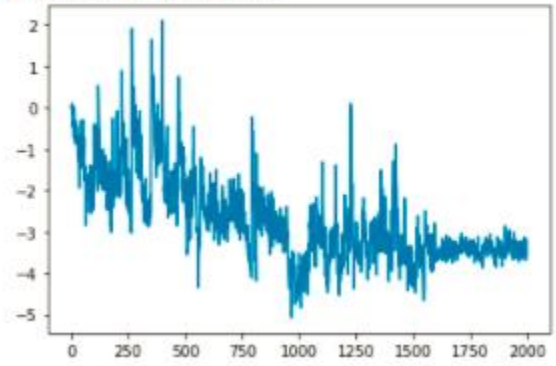
Generator loss:



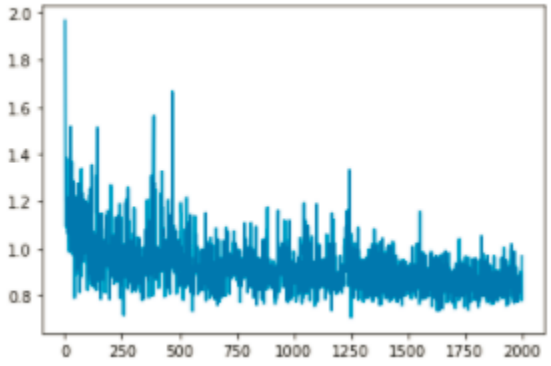
Discriminator real loss:



Discriminator fake loss:

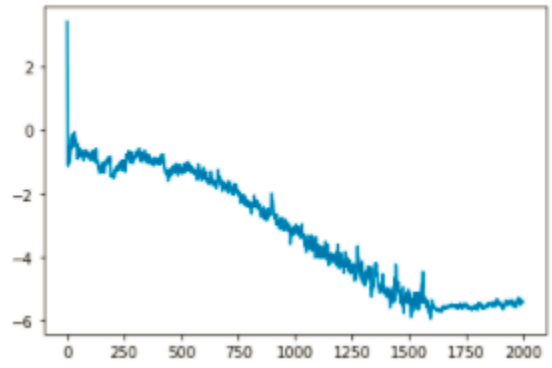


Reconstruction loss:

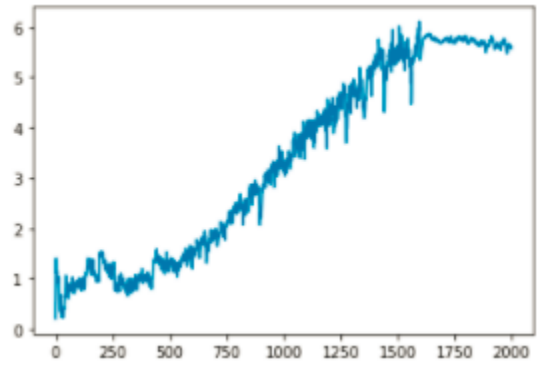


Scale 8:

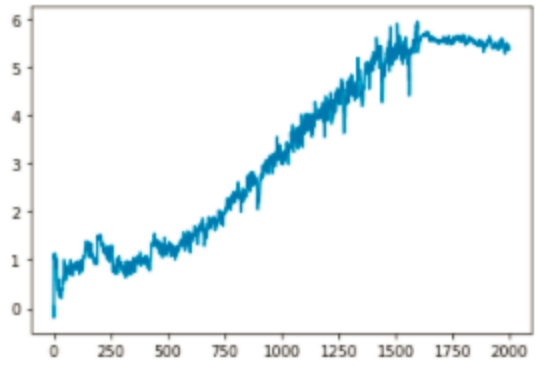
Generator loss:



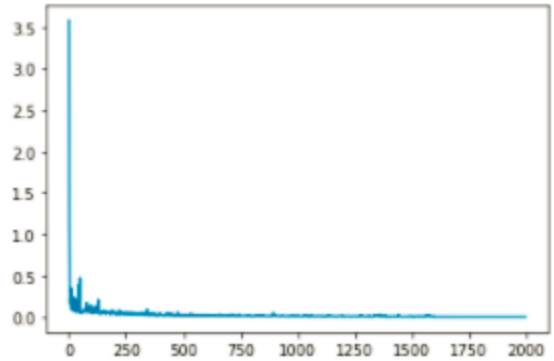
Discriminator real loss:



Discriminator fake loss:



Reconstruction loss:




```
RAM: Consumed Peaked  Used Total | Exec time 7795.123s
CPU:       43     22     2505 MB |
GPU:      112  12322     1007 MB |
```

Which didn’t improve nor the original neither our best so far results, but made us notice an interesting visual result:

As you can notice in the images, the mountain became like all itself variations, the image became much less variant. The woods in the right are pretty much alike.

Only more finest details like the number of mountain peaks remain with a variety. This result prompts us to think that with different modes with local attention we can in some sense control the generator, which is generally in this kind of adversarial training usually is uncontrollable. For example rearranging the pictures to patches and then implying the generation process per patch would result in preserving the main shapes per patch, as in the whole image in our current result, or for example applying different hard or soft masks on some objects, giving the image different weights and e.t.c can maybe help us to control more the process of generation, in this sense supervising the Generator in it generation process.

We describe very shortly an experiment we run with such rearrangement, as it is done in Vision Transformer model:

We rearranged the image into patches and tried to run in different modes the above attention architecture with this change. In most of the cases we've gotten fully memorized image in the output, maybe because small patches are too much simple for convolutions to learn. Although the unsuccessfully attempt, it could be interesting to research this direction more, but we've stopped at this point leaving it out of our scope.

Good SIFID scores were obtained placing this architecture after the head convolution, as we did with the previous attention (axial attention without the feed-forward after it).

We can see from the following visual results, that the previous effect of "memorizing" became weaker, there is similar visual variety as with just with the attention, but the colors of the sky aren't brighter or more contrast, same as previously some textures on the brown ground beneath the water near the shore appeared. Relying on the fact that both attention and attention-feedforward gave better results from all other experiments, they are potentially may improve the scores in the original 50 images test.

It is also interesting to notice that in the end of the training there is an almost perfect equilibrium between the generator loss and discriminator loss, both around 0 loss, a result which didn't appeared in other experiments.

(0) Full decoder before body:

```
SIFID:  0.5141851
```

(1) ds real vs fake:

```
SIFID:  0.110174455
```

(2) Real vs ds_real:

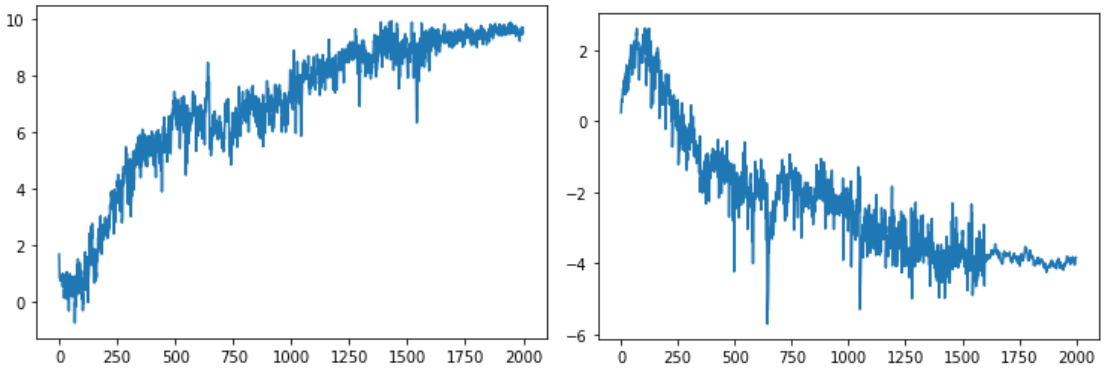
```
SIFID:  0.43780917
```

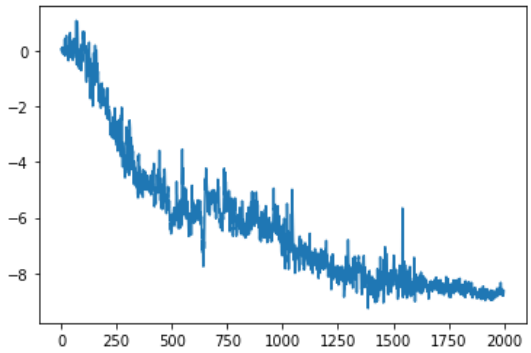
(3) Ds_real vs rec:

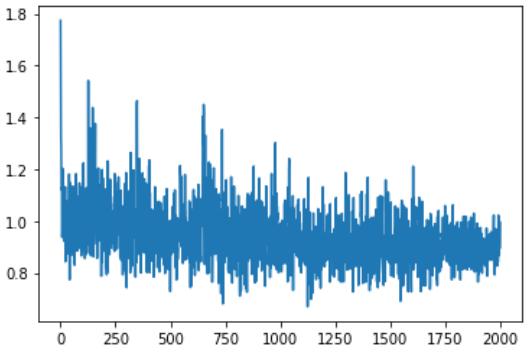
```
SIFID:  0.009493371
```

(4) Real vs rec:

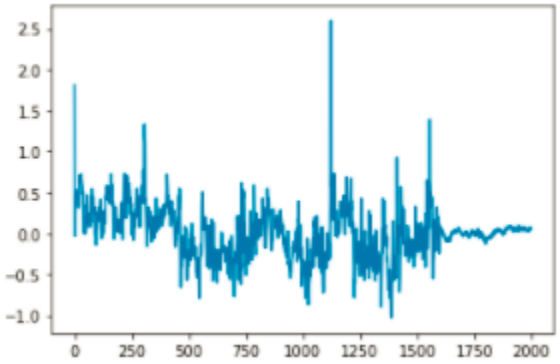
```
SIFID:  0.44847354
```

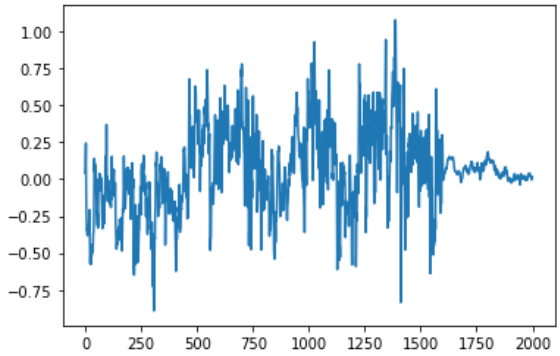


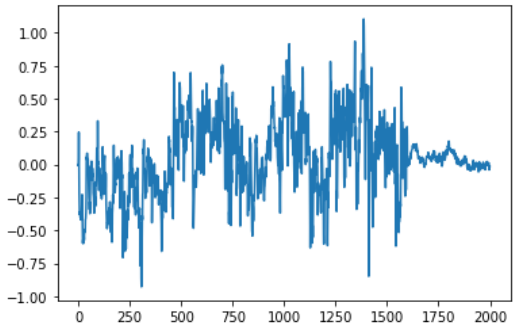


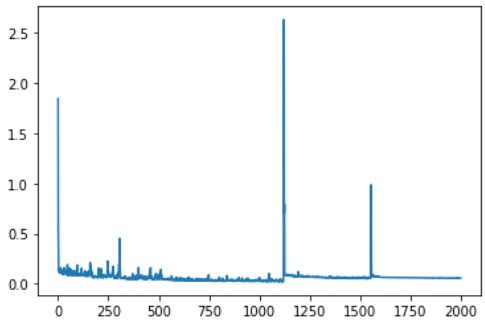


Generator loss:










```
RAM: Consumed Peaked  Used Total | Exec time 6146.966s
CPU:        43     21     3122 MB |
GPU:       132  12432     1457 MB |
```

We've also check only FF to see if it maybe contributes to the last results of "memorizing" or even improving the model SIFID, but just one FF layer instead of the attention made SIFID score much worse and visually the shadows in the water became less accurate:

Real vs fake

```
SIFID:  0.83970845
```

Real vs ds real:

```
SIFID:  0.43780917
```

Ds real vs rec

```
SIFID:  0.03471085
```

Ds real vs fake

```
SIFID:  0.2732511
```

Rec vs fake:

```
SIFID:  0.14566441
```

Real vs rec:

```
SIFID:  0.5292287
```

In this series of experiments with placing the attention layer in different places (I.e activating it in different places at the forward path), we try another two options:

Attention after the last convolution layer, I.e after generating the image in the original architecture:

```
SIFID:  0.104621276

 SIFID:  0.57480407
```

Which gave us a result similar to the best result seen so far.

Attn last and before last with after head and before head:

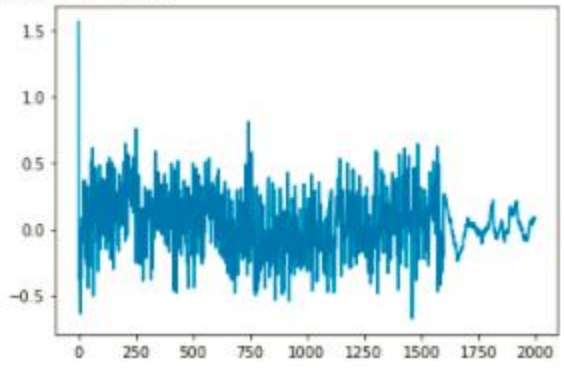
Experiment with two attention layers, in the Generator there is one layer before the last convolution and one after it, and a mirror placement in the Discriminator, one before the first convolution and one after it.

The scores doesn't indicate any anomality, but the images are significantly more distorted comparing to the original and other methods we've tried.

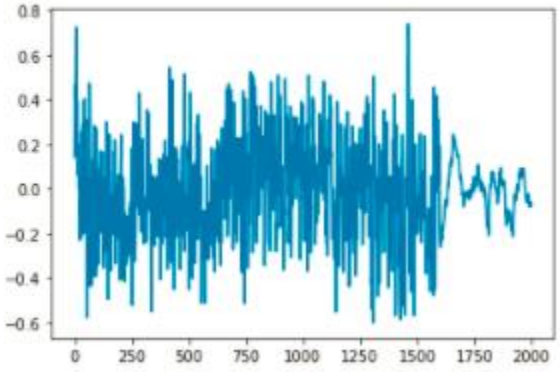
```
SIFID:  0.13840498

 SIFID:  0.5809623
```

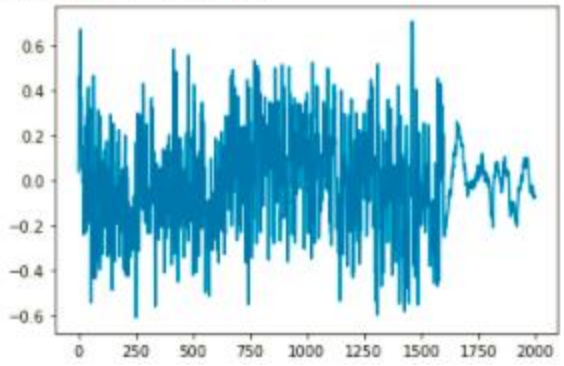
Generator loss:



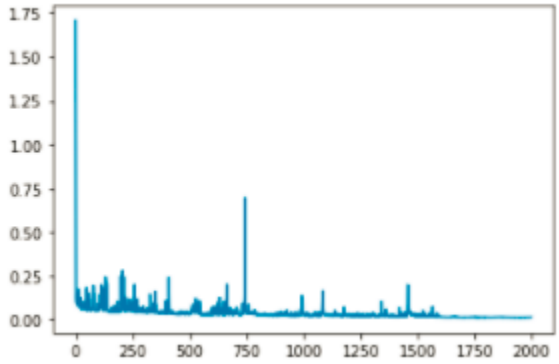
Discriminator real loss:



Discriminator fake loss:



Reconstruction loss:



Decoder (attention followed by feed-forward layer) after tail (last convolution):

```
SIFID:  0.5951977
```

```
SIFID:  0.12844104
```

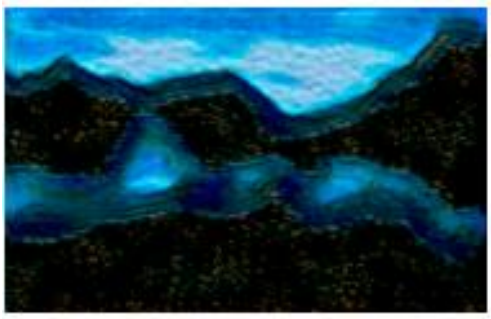
Also gave us good SIFID score, therefore another reasonable choice, although needed to be confirmed in greater number of experiments seems to be the placement of the attention after the last convolution, which means we are attending the formulated output image (comparing to the original model).

Linear Image attention didn't work, although the SIFID scores were not the worst:

```
SIFID:  13.824705
```

Reconstructed:

Generated:



Same strange results were obtained by the Performer.

Super-Resolution benchmark:

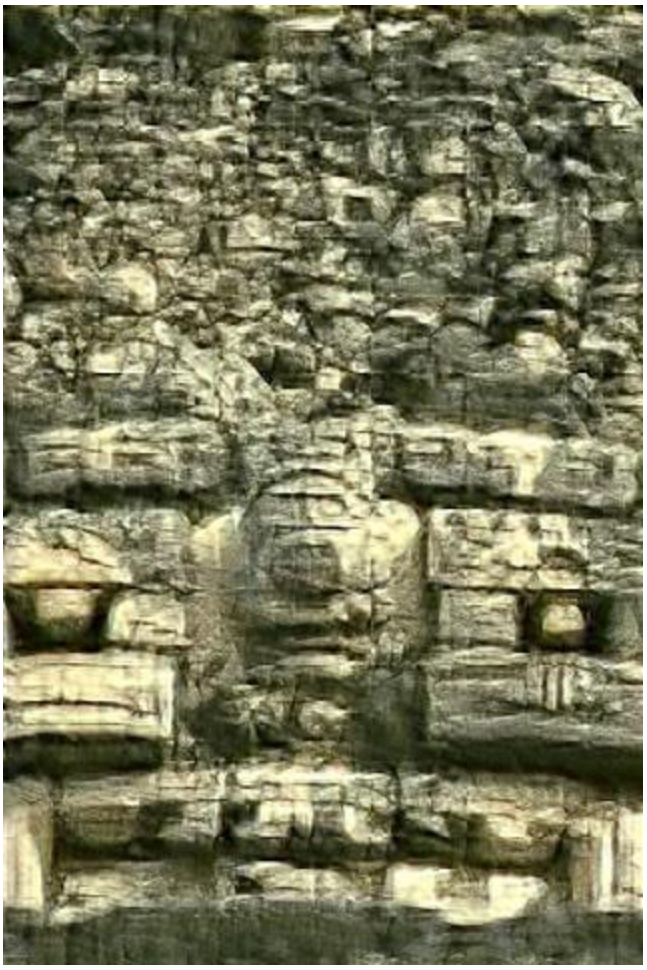
Original (in its Low-Resolution size 120 x80):

A few words about our benchmark computations:

Although in the original paper the author redirect us to another paper which widely uses a whole Matlab framework that computes RMSE and NIQE, we use only NIQE as it was the easiest and fastest to test.

In Matlab it just one line of code, although it requires 36 images for the statistics comparison of that measurement (NIQE), so we've used the same image 36 times, and checked that the NIQE score of the original image comparing to those statistics the NIQE model computed is 0. Thus, we would want to look for the closest SR image to 0 of this collection of duplicates f this image.

Super-Resolution with the original architecture:



NIQE – Naturalness Image Quality Evaluator.

A few experiments of the most interesting results of SIFID score improvement:

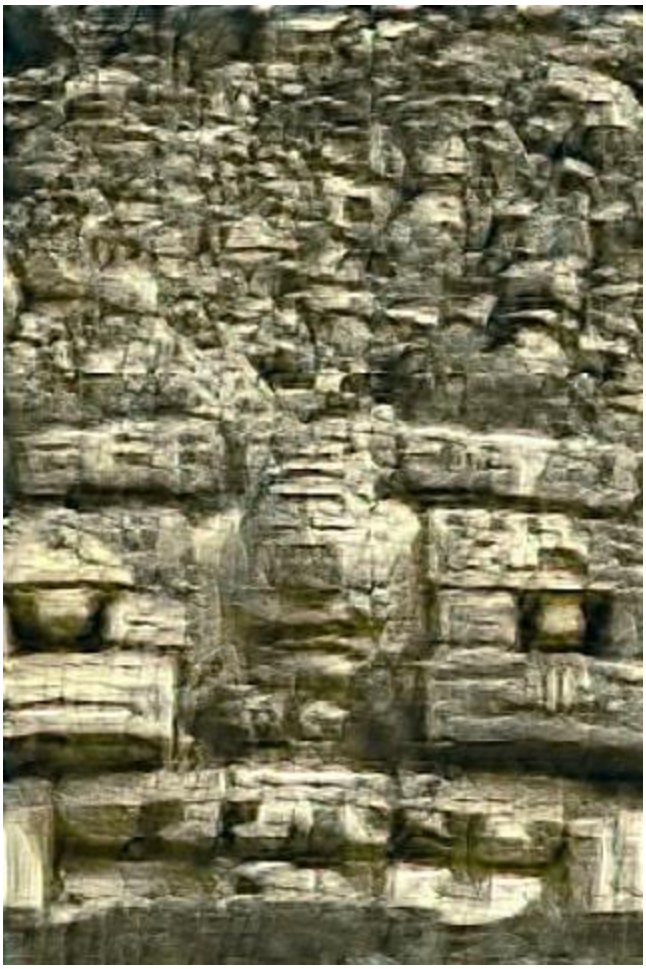
NIQE:

15.3049

14.9033

25

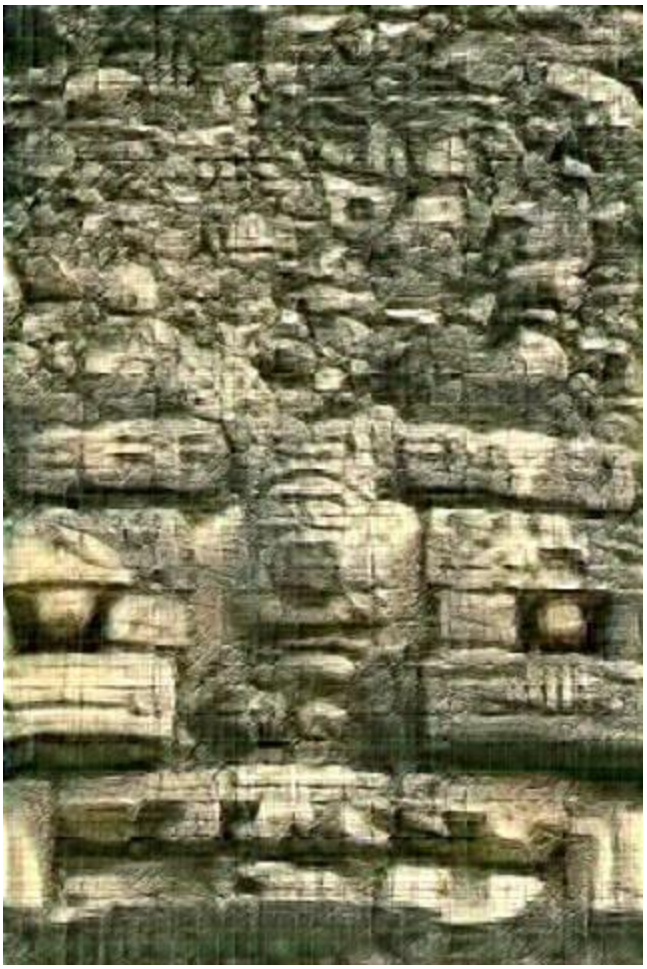
<u>With axial attention after last convolution:</u>



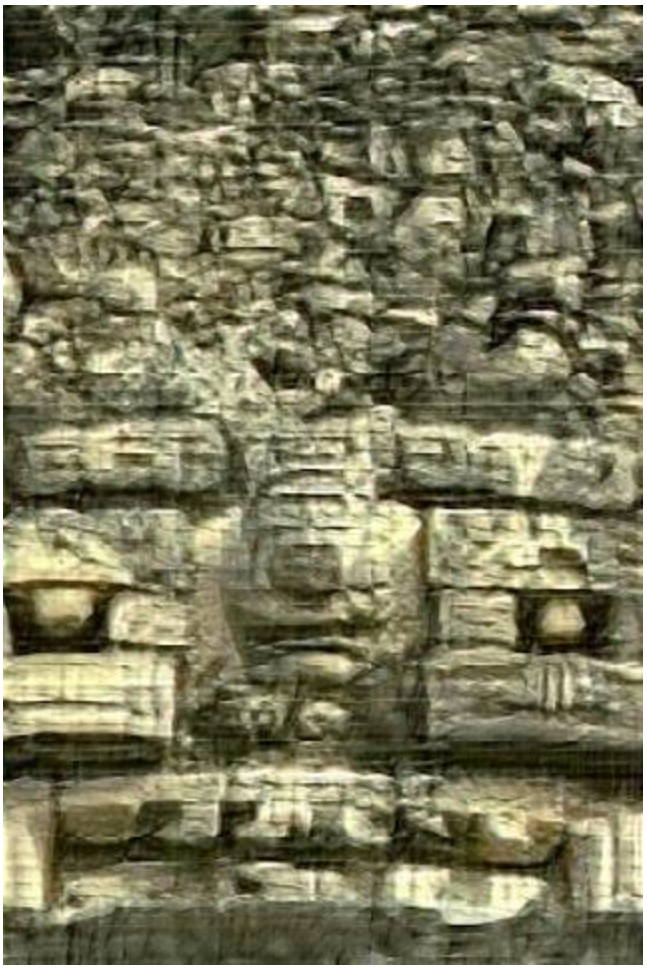
18

After head:

```
9.1649
```



After body:



34.1039

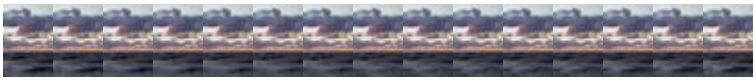
The best result was obtained for the architecture where the attention is placed after the first (head) convolution. Visually it hard to tell which is better.

Experiments with animation improvement:

1.CRNN

A CRNN-GAN adapted from music generation task for this task was too memory-consuming because of the Linear modules, it consumed all the memory even after training only one scale and lacking the convolutional architecture resulted in full memorizing of the sequence of frames, length of which is taken arbitrarily to be 15 (as the length of a simple gif of sea waves):

Real:



Fake generated:

The reconstructed are exactly the same as the real and the generated.

2. ConvLSTM:

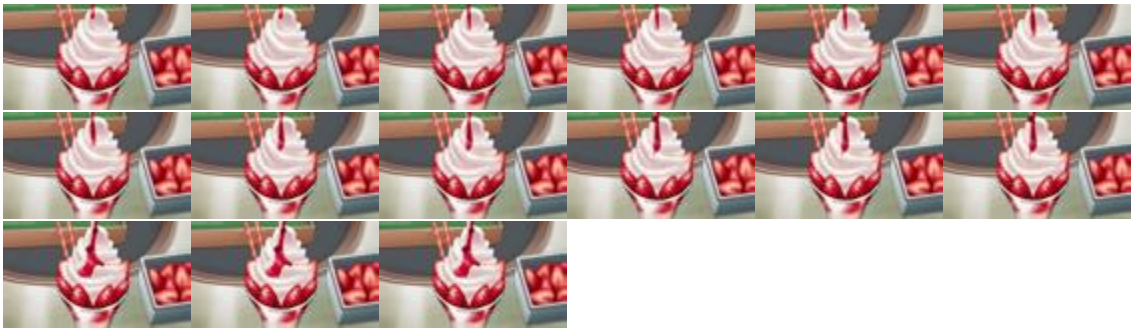
First as a baseline we run the code without any sequencers and treat the multiple frames as a batch of multiple single images that we want to learn. We except good reconstruction and SIFID scores, as we saw for single image, with more memory consumption which can cause out of memory, for that reason we handle it in such a way that the scale training interrupts for the concrete scale and iteration after an OOM exception, continuing to train on the next scale repeatedly. In such a way, if OOM happens, the similarity of the fake and the real in SIFID score can suffer by some factor.

We also expect that as we saw with one image, although the frames have high similarity, the generated images can have a great diversity, which will cause our fake frames not to be a sequence anymore, but just a batch of fake images, I.e there is no sequence memory even if the frames themselves being a sequence and thus being very similar and differ only by some pixels which are the expression of the animation of some movement.

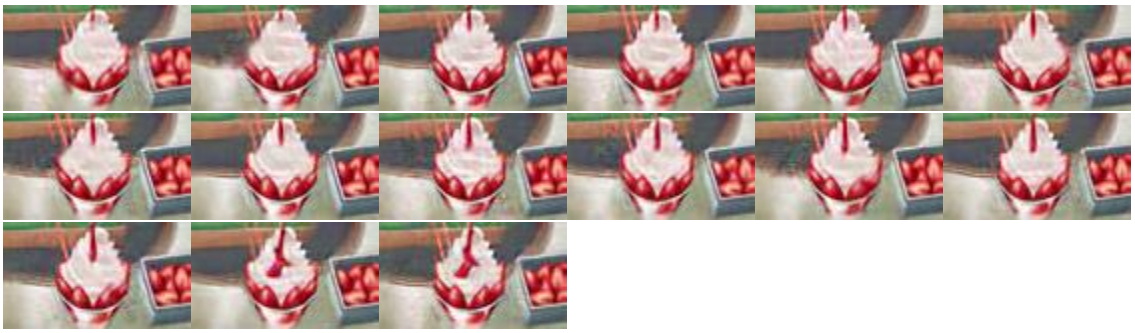
Training 0-3 scales takes about 2 hours and the memory consumption of 14 GB approximately.

The results are exactly as expected above:

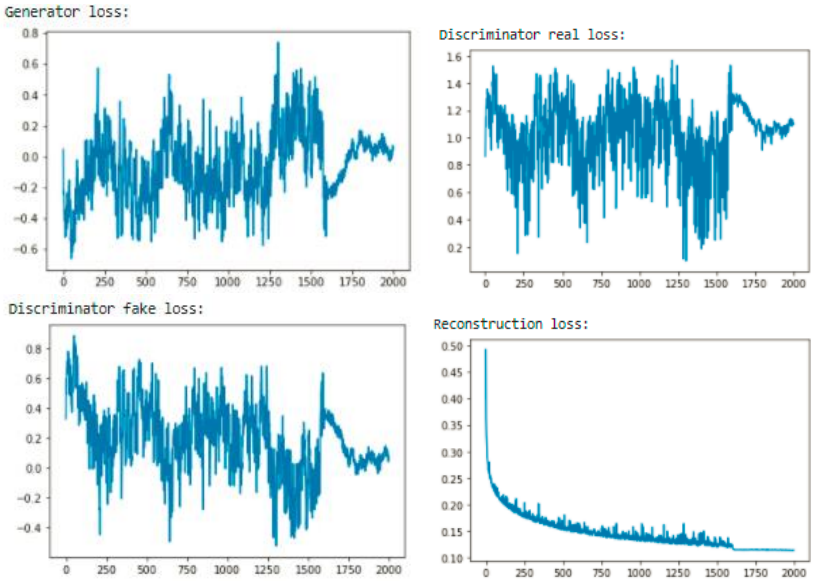
Real:



Reconstructed:



Generated:


Generator loss:
Discriminator real loss:
Discriminator fake loss:
Reconstruction loss:

2 approaches:

Many to many (gif to gif) (one lstm cell)

Real:

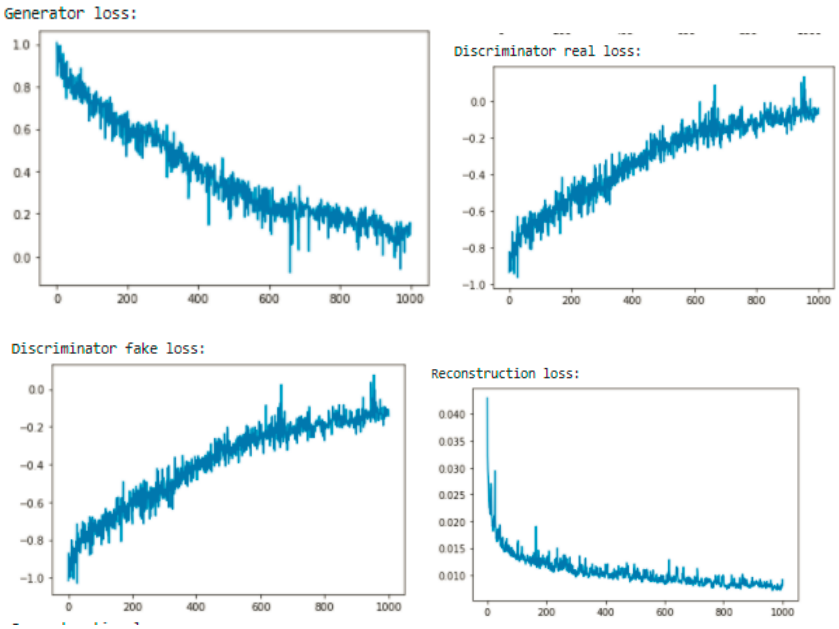
Fake:

Although here we used convolutional architecture, including the convolutions which were in the original architecture:

We've took the head and the tail as is, and between them placed the LSTM instead of the 3 convolutions of the body block were there originally.

This experiment resulted as can be seen in memorizing, although there isn't any FC layers as before.

The reason for that behavior is not clear.



Another image experiment which approved the memorizing behavior.

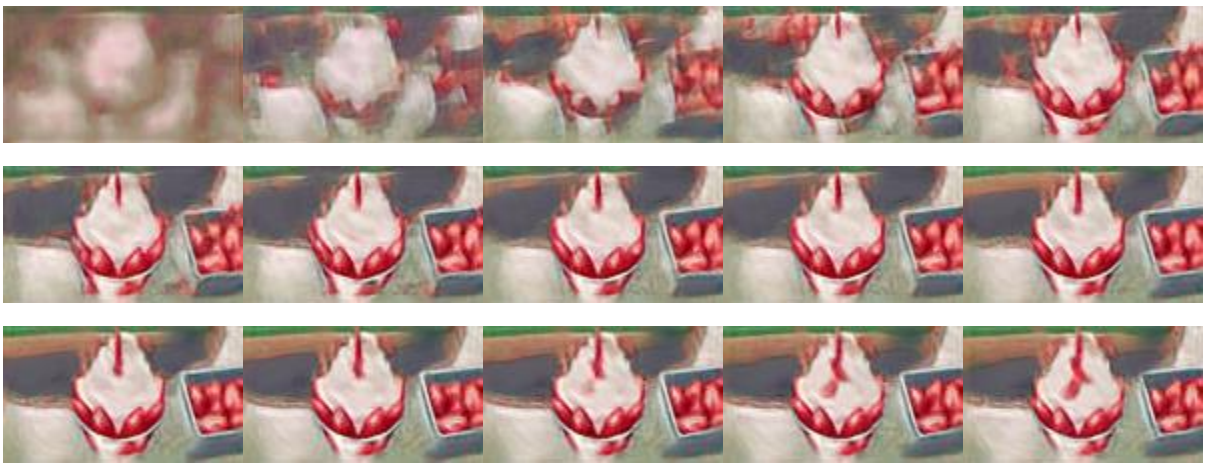
Real:

Fake:

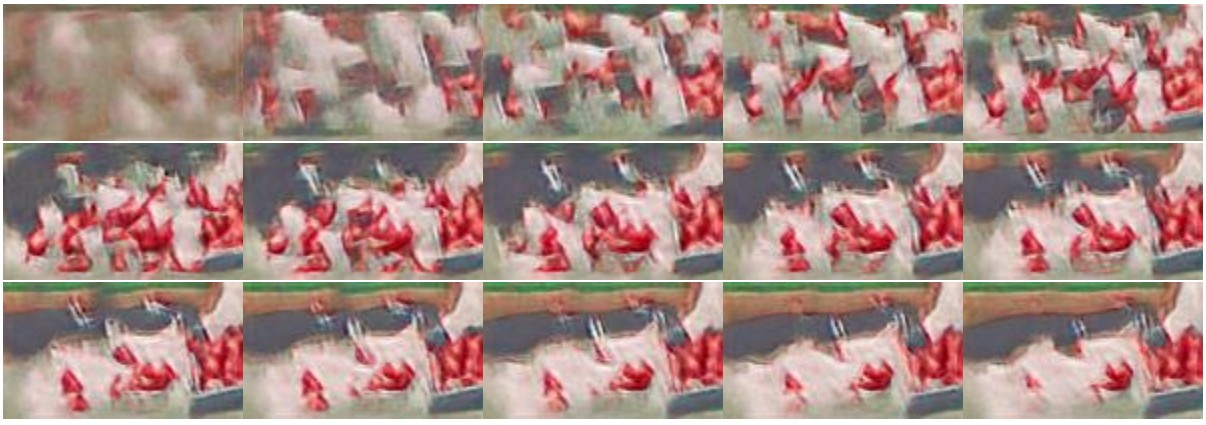
Another approach:

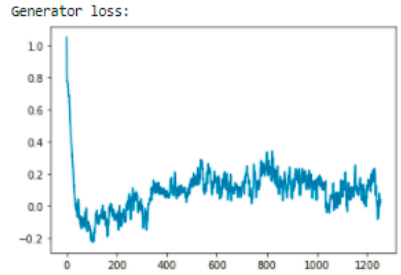
We placed LSTM before the head convolution, thus enforcing the LSTM sequence learning first on the sampled noise (I.e can be thought as giving the noise a sequence behavior through the LSTM or in another words we are creating a latent space of sequences with maximal similarity to the original sequence).

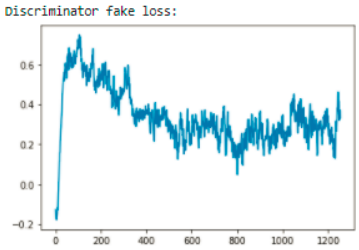
Reconstructed:

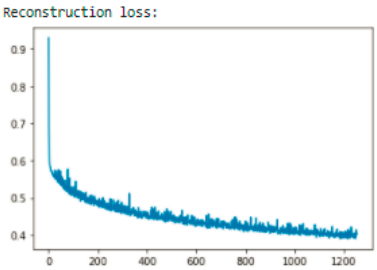


Fake:









In that approach we handled the previous difficulties of memorization and generation of totally different frames (as without the LSTM), but there is some delay in the frames quality in the fine details generation.

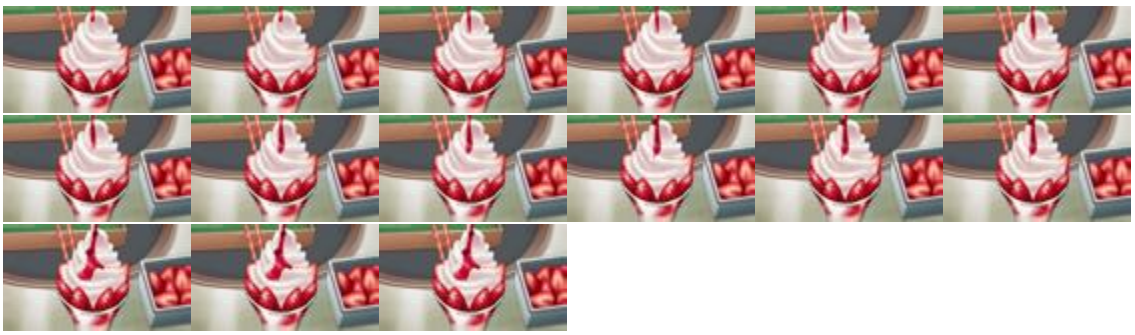
The first frames of each scale are very blurred. To overcome this effect, our intuition was that it caused because we operate on the noise, when the images are still have no sharp shapes or even any features, thus in the next attempt we try to firstly generate the image and then pass it through the LSTM network using a different approach of one to many.

<u>One to many approach:</u>

Here the input is only the first frame. The LSTM component uses the first input and an initialized first-time hidden space, and then repeatedly takes the output as the current input and the previous hidden state as the current (each time creating the next). The Discriminator discriminates between the generated t frames and the ground-truth of same length of series.

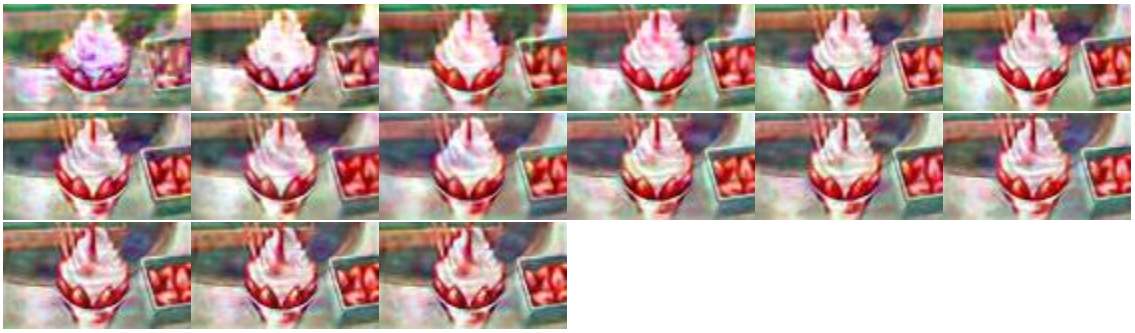
Scale 3:

Real:

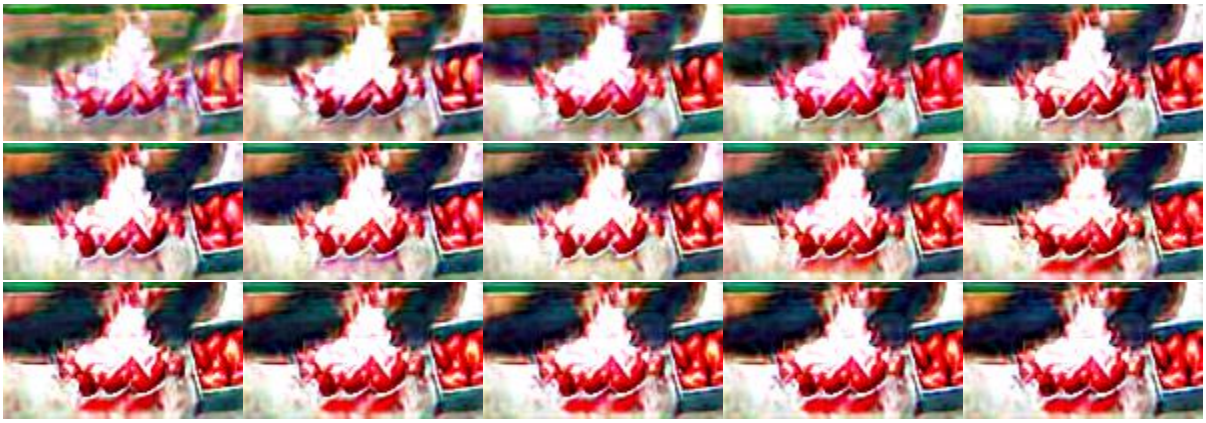


Fake:

Reconstructed:



Fake:

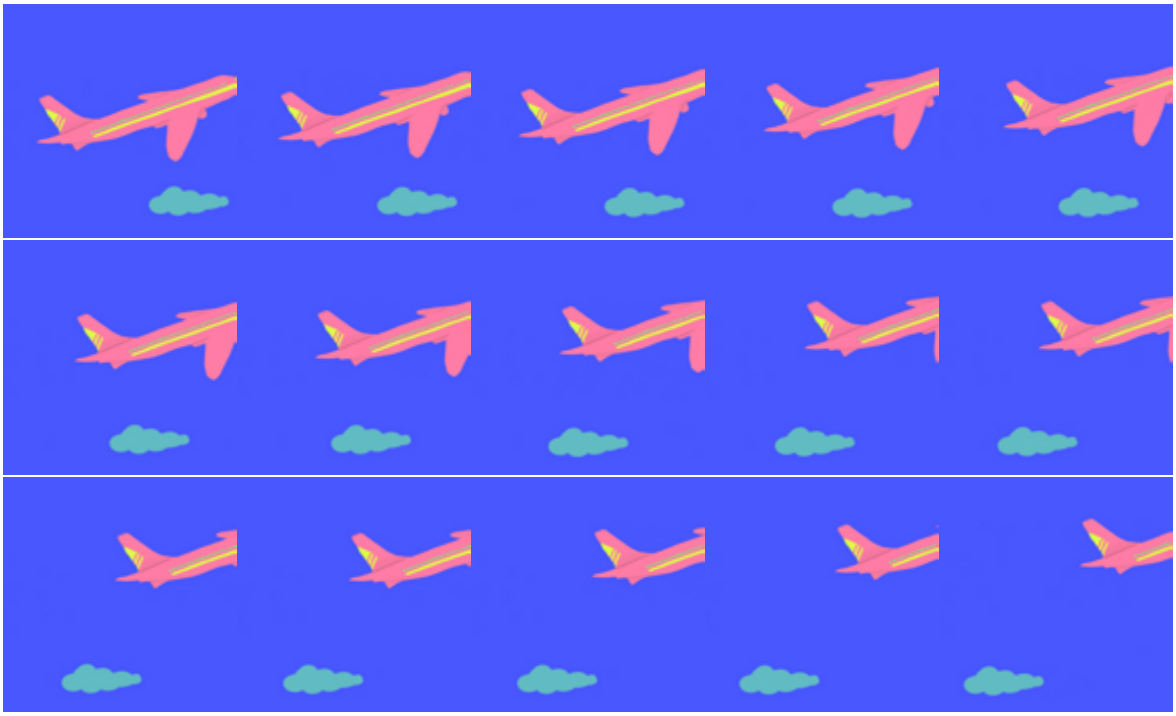


Here is the first time, although it's not visually straightforward noticeable, but if you pay attention, you can see that the architecture learned that over a time-series there is some red blur that moves from in a vertical up-to-down direction. Notice how appear some red streams and blurs on top of the distorted ice-cream and more important on its bottom, in both examples of fake frames. We can see also that in the reconstructed frames the frames seemed to be reconstructed in a good manner.

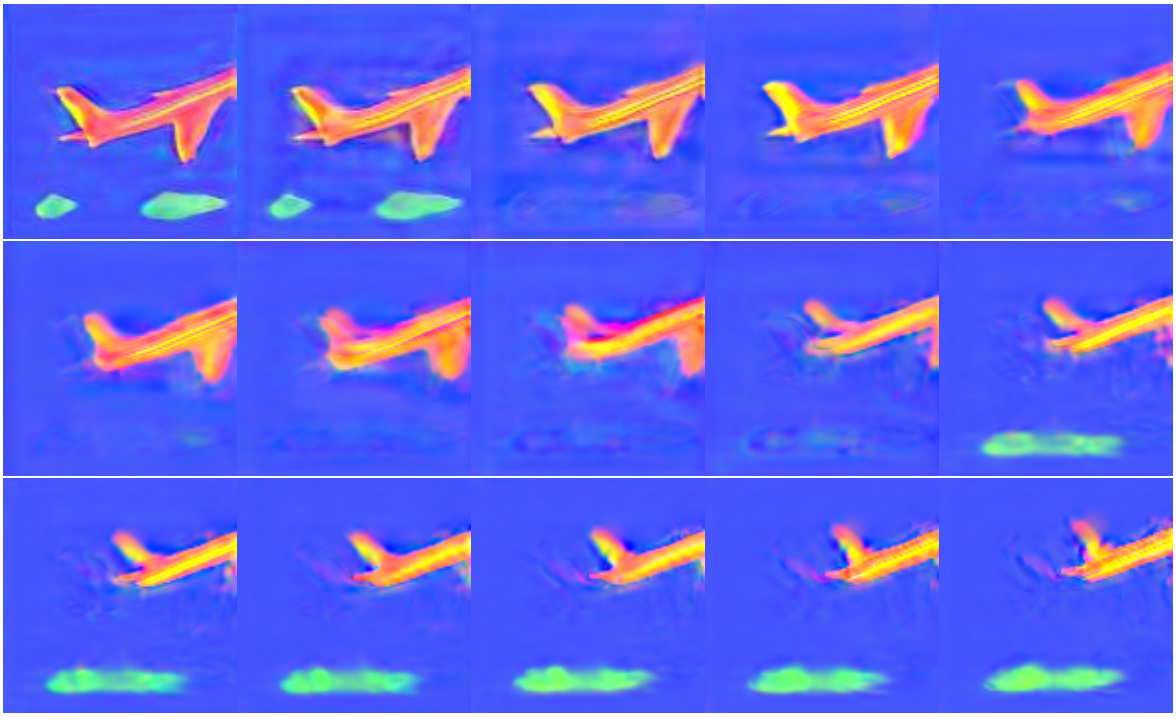
After seeing this for 3-rd scale, we try it on another frame sequence of a flying airplane with more simplistic details

Scale 6 with not full training (as was described previously, the memory utilized until it is exhausted for a concrete scale and the training process proceeds to the next scale) :

Real:



Fake:



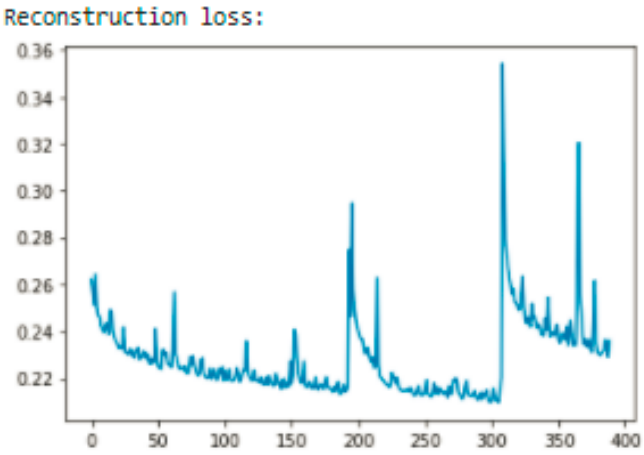
Here we see a little randomness at the cloud structure and in the shape of the plane.

Test case:

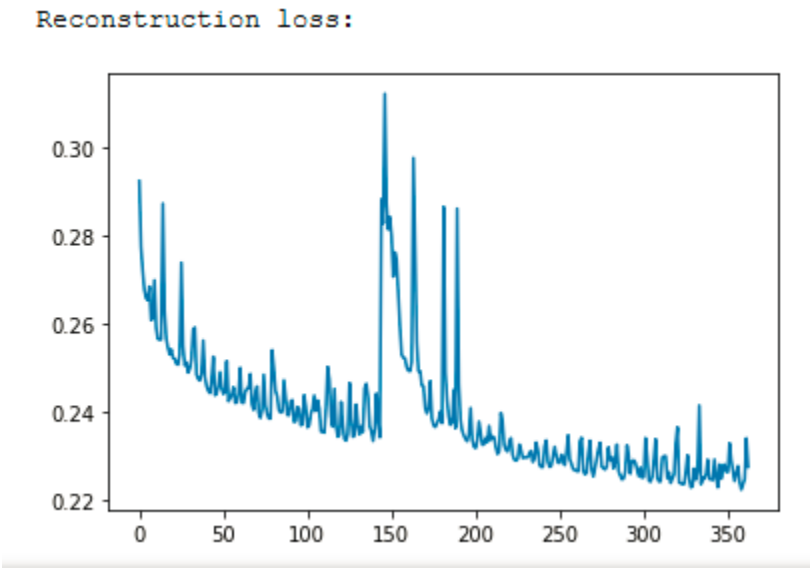
Waterfall movement vs reversed waterfall movement:

Please look at the final animation result in the attached files.

Waterfall reconstruction loss:



Reverse waterfall loss:



As we can see from the loss graphs and the animations themselves, it seems that our main problem is lack of enough memory to run the whole 2000 iterations each scale training. It can be solved by multi-GPU parallelism, where all the needed previously trained generators for creating the image which will be up-sampled and added with new scale generated details, could be loaded into device:0 and all the other network will be loaded into the other device, device:1, as those previously trained generators are used for creating from coarse to current scaled “background” each iteration.

In this setting, guided by the last results, maybe we could expect even better detailed top-down and bottom-up water movement with the ability to generate some different waterfalls.

Although it should be noticed that in the waterfall example, we didn't see to many randomness as in the ice-cream generation, but we assume it is due to the bad reconstruction loss due to the disability to train for enough iterations.

References:

Original paper: https://arxiv.org/abs/1905.01164

https://arxiv.org/abs/1409.0473

https://arxiv.org/pdf/1406.6247

https://arxiv.org/pdf/1502.03044

https://arxiv.org/pdf/1710.07035

https://arxiv.org/abs/1912.12180

https://arxiv.org/abs/2009.06732

https://arxiv.org/abs/2009.14794

https://arxiv.org/pdf/2010.11929

https://arxiv.org/pdf/1805.08318

https://arxiv.org/pdf/1508.04025

https://arxiv.org/pdf/1406.2661

https://arxiv.org/pdf/1701.07875

https://arxiv.org/abs/1506.05751

https://arxiv.org/pdf/2102.07074

https://arxiv.org/pdf/1611.09904

https://arxiv.org/pdf/1506.04214v2

Code:

First task:

https://github.com/ilyak93/SinGanF

https://github.com/ilyak93/SinGanF2

Second task:

https://github.com/ilyak93/SinGan