

Denoising Diffusion Probabilistic Models


Article authors: Jonathan Ho, Ajay Jain, Pieter Abbeel

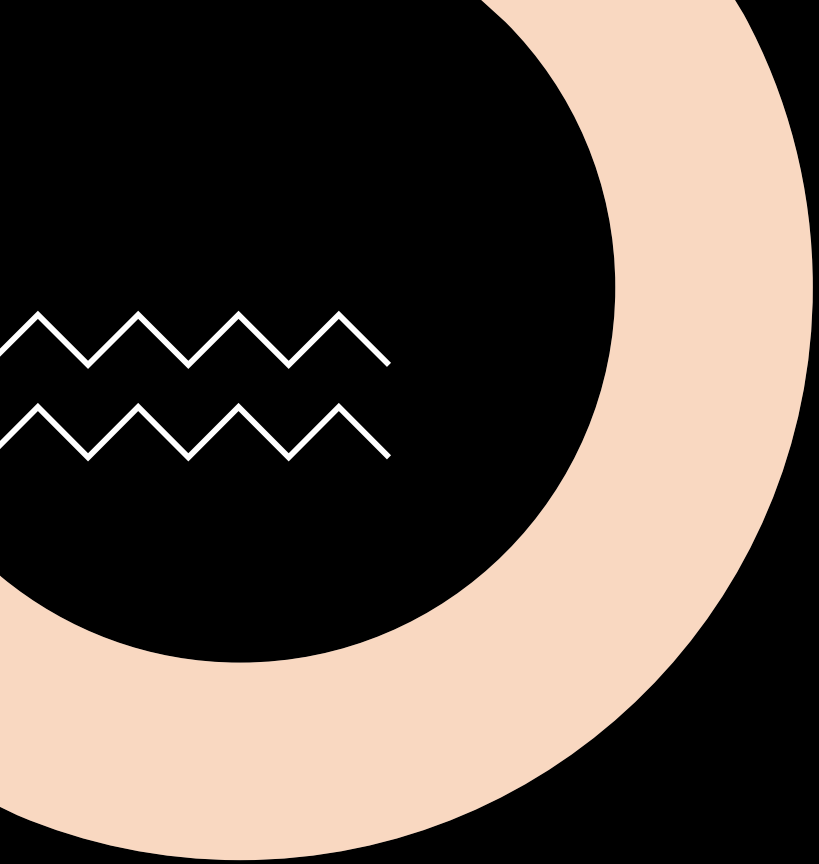
Students:
Vincenzo Pluchino
Philip Tambè



Introduction


The article presents a different structure for the diffusion probabilistic models (called only diffusion models). The authors show that diffusion models actually are capable of generating high quality samples, sometimes better than the published results on other types of generative models (such as Normalizing Flows, GANs or VAEs). Prior to this work, these models were not used to generate images of a certain quality. The only goal was to generate images after introducing noise into them

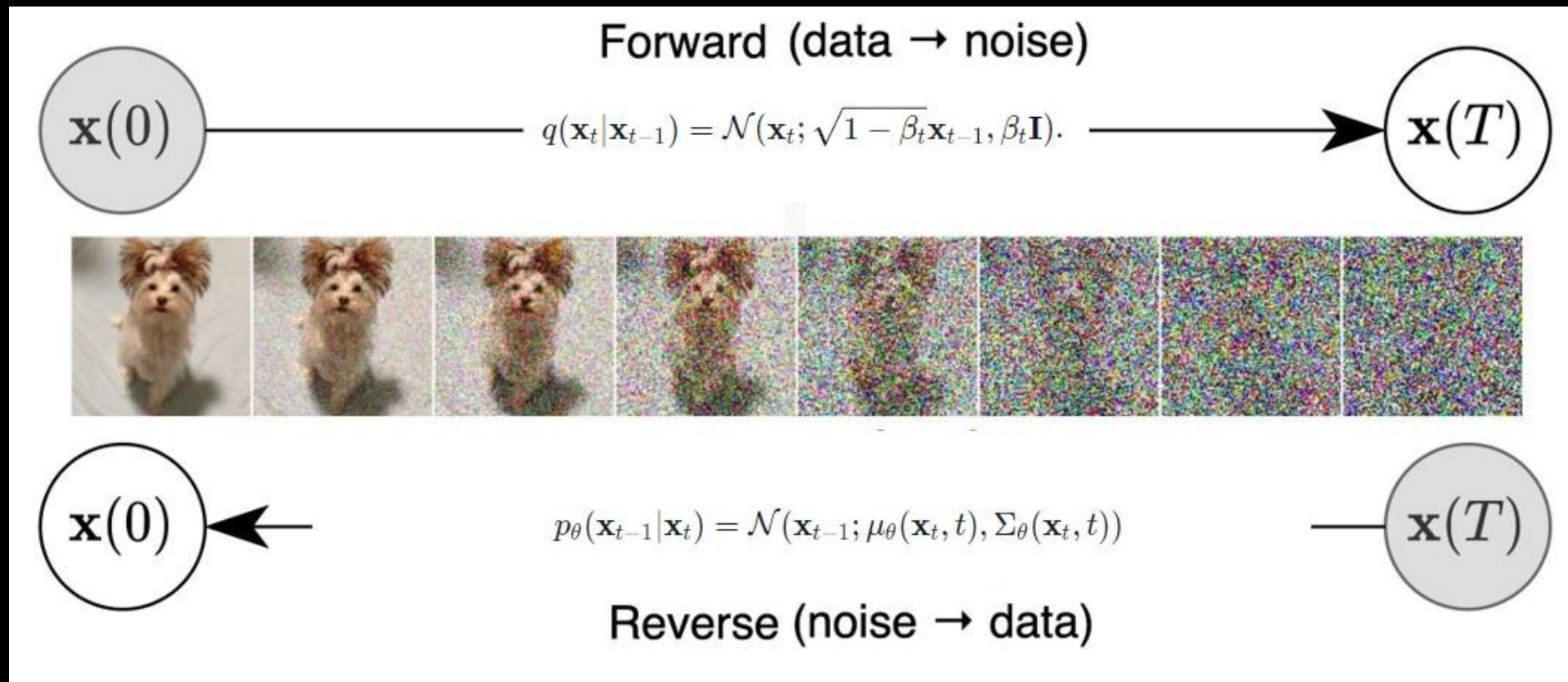




What is a diffusion model?

A denoising diffusion model convert noise from some simple distribution to a data sample. This type of model is a parameterized Markov chain trained using variational inference to produce samples matching the data after finite time. The set-up consists of 2 processes:

- a fixed (or predefined) **forward diffusion process q** of our choosing, that gradually adds Gaussian noise to an image, until you end up with pure noise.
 - a learned **reverse denoising diffusion process p_{θ}** , where a neural network is trained to gradually denoise an image starting from pure noise, until you end up with an actual image.
- 



Both the forward and reverse process indexed by t happen for some number of finite time steps T ($T=1000$). You start with $t=0$ where you sample a real image \mathbf{x}_0 from your data distribution, and the forward process samples some noise from a Gaussian distribution at each time step t , which is added to the image of the previous time step.

- The authors defined the forward diffusion process $q(\mathbf{x}_t | \mathbf{x}_{t-1})$ which adds Gaussian noise at each time step t , according to a known variance schedule $0 < \beta_1 < \beta_2 < \dots < \beta_T < 1$ as

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}).$$

- β_t aren't constant at each time step; t in fact defines a so-called "**variance schedule**", which can be linear, quadratic, cosine, etc. β_t is considered an hyperparameter
- So starting from \mathbf{x}_0 , they end up with $\mathbf{x}_1, \dots, \mathbf{x}_t, \dots, \mathbf{x}_T$, where \mathbf{x}_T is pure Gaussian noise if we set the schedule appropriately.
- So there's the need to know the conditional distribution $p(\mathbf{x}_{t-1} | \mathbf{x}_t)$, then it's possible to run the process in reverse:
by sampling some random Gaussian noise \mathbf{x}_T and then gradually "denoise" it, the result is a sample from the real distribution \mathbf{x}_0

$$p_{\theta}(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_{\theta}(\mathbf{x}_t, t), \Sigma_{\theta}(\mathbf{x}_t, t))$$

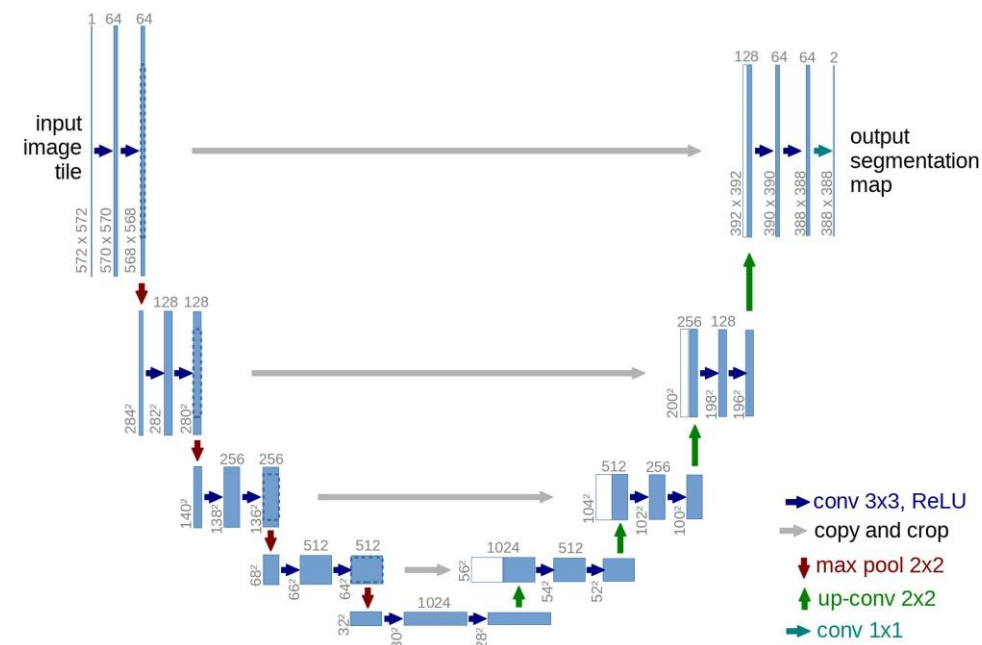
In short:

- Take a random sample \mathbf{x}_0 from the real unknown and possibly complex data distribution $q(\mathbf{x}_0)$
- Sample a noise level t uniformly between 1 and T
- Sample some noise from a Gaussian distribution and corrupt the input by this noise at level t
- the neural network is trained to predict this noise based on the corrupted image \mathbf{x}_t (i.e. noise applied on \mathbf{x}_0 based on known schedule β_t)

In reality, all of this is done on batches of data, as one uses stochastic gradient descent to optimize neural networks.

U-Net Network

In the model proposed in the paper there's a **U-Net** network, it is like any autoencoder, consists of a bottleneck in the middle that makes sure the network learns only the most important information. Importantly, it introduced residual connections between the encoder and decoder, greatly improving gradient flow.









How the model was build up?

- first, a **convolutional layer** is applied on the batch of noisy images, and position embeddings are computed for the noise levels
- next, a sequence of **downsampling stages** are applied. Each downsampling stage consists of **2 ResNet/ConvNeXT blocks + groupnorm + attention + residual connection + a downsample operation**
- at the middle of the network, again **ResNet or ConvNeXT blocks** are applied, interleaved with attention
- next, a sequence of **upsampling stages** are applied. Each upsampling stage consists of **2 ResNet/ConvNeXT blocks + groupnorm + attention + residual connection + an upsample operation**
- finally, a **ResNet/ConvNeXT block** followed by a convolutional layer is applied.

Code

The code used for this project is the [Denoising Diffusion Pytorch](#), which is a «translated» version (from Tensorflow to Pytorch) of the code used by the authors of the article and it's available on a repository on Github.

There are six files on the repo, including the init, but we only focus on «denoising_diffusion_pytorch.py» which contains all the elements for training and testing the model.

-  `_init_.py`
-  `continuous_time_gaussian_diffusion.py`
-  `denoising_diffusion_pytorch.py`
-  `elucidated_diffusion.py`
-  `learned_gaussian_diffusion.py`
-  `weighted_objective_gaussian_diffusion.py`

Code implementation of U-NET

Here it's defined the Unet class with all the necessary parameters. In the following code (not showing in the image) there are also the definitions of all the layers and the forward function

```
class Unet(nn.Module):
    def __init__(
        self,
        dim,
        init_dim = None,
        out_dim = None,
        dim_mults=(1, 2, 4, 8),
        channels = 3,
        resnet_block_groups = 8,
        learned_variance = False,
        sinusoidal_cond_mlp = True
    ):
        super().__init__()

        # determine dimensions

        self.channels = channels

        init_dim = default(init_dim, dim // 3 * 2)
        self.init_conv = nn.Conv2d(channels, init_dim, 7, padding = 3)

        dims = [init_dim, *map(lambda m: dim * m, dim_mults)]
        in_out = list(zip(dims[:-1], dims[1:]))

        block_klass = partial(ResnetBlock, groups = resnet_block_groups)

        # time embeddings

        time_dim = dim * 4

        self.sinusoidal_cond_mlp = sinusoidal_cond_mlp

        if sinusoidal_cond_mlp:
            self.time_mlp = nn.Sequential(
                SinusoidalPosEmb(dim),
                nn.Linear(dim, time_dim),
                nn.GELU(),
```

Code implementation of Training

- The Trainer class is one of the most important class in the code, it enables to specify the input parameters for the training execution.

We have overwritten some default values to set this parameters with the same options that are reported in the paper, in particular we indicates the batch size = 128, the Learning Rate = $2e-4$ and ema decay = 0.9999.

- Trainer class also define some logic to periodically save parameters of model and generated images. Later the parameters of model can be reloaded to continue the training.
- We really appreciated these features that allowed us to train the project on Google Colab several times.

```
class Trainer(object):
    def __init__(
        self,
        diffusion_model,
        folder,
        *,
        ema_decay = 0.995,
        image_size = 128,
        train_batch_size = 32,
        train_lr = 1e-4,
        train_num_steps = 100000,
        gradient_accumulate_every = 2,
        amp = False,
        step_start_ema = 2000,
        update_ema_every = 10,
        save_and_sample_every = 1000,
        results_folder = './results',
        augment_horizontal_flip = True
    ):
        ...
```

```

class GaussianDiffusion(nn.Module):
    def __init__(
        self,
        denoise_fn,
        *,
        image_size,
        channels = 3,
        timesteps = 1000,
        loss_type = 'l1',
        objective = 'pred_noise',
        beta_schedule = 'cosine',
        p2_loss_weight_gamma = 0., # p2 loss weight,
        p2_loss_weight_k = 1
    ):
        super().__init__()
        assert not (type(self) == GaussianDiffusion and

        self.channels = channels
        self.image_size = image_size
        self.denoise_fn = denoise_fn
        self.objective = objective

        if beta_schedule == 'linear':
            betas = linear_beta_schedule(timesteps)
        elif beta_schedule == 'cosine':
            betas = cosine_beta_schedule(timesteps)
        else:
            raise ValueError(f'unknown beta schedule

```

Code implementation of Gaussian Diffusion

Most of the parameters of the GaussianDiffusion class are already mentioned in the previous slides (betat, timestep...). The «denoise_fn» parameter is the model that you want to use (in our case is the U-NET).

Other parts of the code for this class that are not showed in the image are just the functions for implemeting the various formulas for the calculations (see funct. like q_sample, q_posterior, p_sample)

Experiment parameters & details

The authors used a neural network architecture similar to another work (PixelCNN) which is a U-net based on another study.

The dataset is CIFAR10, which model in this case has 35.7 million parameters. About the hardware, they used TPU v3-8 (similar to 8 V100 GPUs) for all experiments and thanks to this they were capable of doing 800k steps in 10 hours.

They also performed the majority of the hyperparameter search to optimize for CIFAR10 sample quality.

About the values of other parameters:

- T was setted = 1000 without a sweep, and they chose a linear schedule from $\beta_1 = 10^{-4}$ to $\beta_T = 0,02$
- They used random horizontal flips during training and Adam as optimizer
- The learning rate was setted to 2×10^{-4} , batch size=128 and EMA decay= 0.9999

Table 1: CIFAR10 results.

Model	IS	FID
Conditional		
EBM [11]	8.30	37.9
JEM [17]	8.76	38.4
BigGAN [3]	9.22	14.73
StyleGAN2 + ADA (v1) [29]	10.06	2.67
Unconditional		
Diffusion (original) [53]		
Gated PixelCNN [59]	4.60	65.93
Sparse Transformer [7]		
PixelQNN [43]	5.29	49.46
EBM [11]	6.78	38.2
NCSNv2 [56]		31.75
NCSN [55]	8.87 ± 0.12	25.32
SNGAN [39]	8.22 ± 0.05	21.7
SNGAN-DDLS [4]	9.09 ± 0.10	15.42
StyleGAN2 + ADA (v1) [29]	9.74 ± 0.05	3.26
Ours (L , fixed isotropic Σ)	7.67 ± 0.13	13.51
Ours (L_{simple})	9.46 ± 0.11	3.17

Experiment Results

The metrics used for evaluate the model are **Inception Score IS** and **Fréchet inception distance FID**:

- The **Inception Score** is an objective metric for evaluating the quality of generated images, specifically synthetic images output by generative adversarial network models. The score seeks to capture two properties of a collection of generated images: **Image Quality**. Do images look like a specific object?
Image Diversity. Is a wide range of objects generated?
- The **Fréchet inception distance (FID)** is used to assess the quality of images created by a generative model. Unlike the earlier IS, which evaluates only the distribution of generated images, the FID compares the distribution of generated images with the distribution of real images that were used to train the generator.

Our project assignment

- Since the dataset to be replaced (CIFAR10) coincides with the one used in the article to test their model, the principle aim of our project was to replicate the results of the evaluated metrics
- However, the hardware difference was very significantly because we haven't got any valid GPU so we trained using Colab. For this reason, doing 800k steps like the paper would have involved more than a week of execution 24/24 (which is not possible on Colab); so, we reached the maximum of 100k steps
We have relied on the fact that learning process it's not a linear process, so we are confident that in the initial timesteps the model learned the most part of what it would learn even if it was trained for 800k

Changes to the code and our contributions

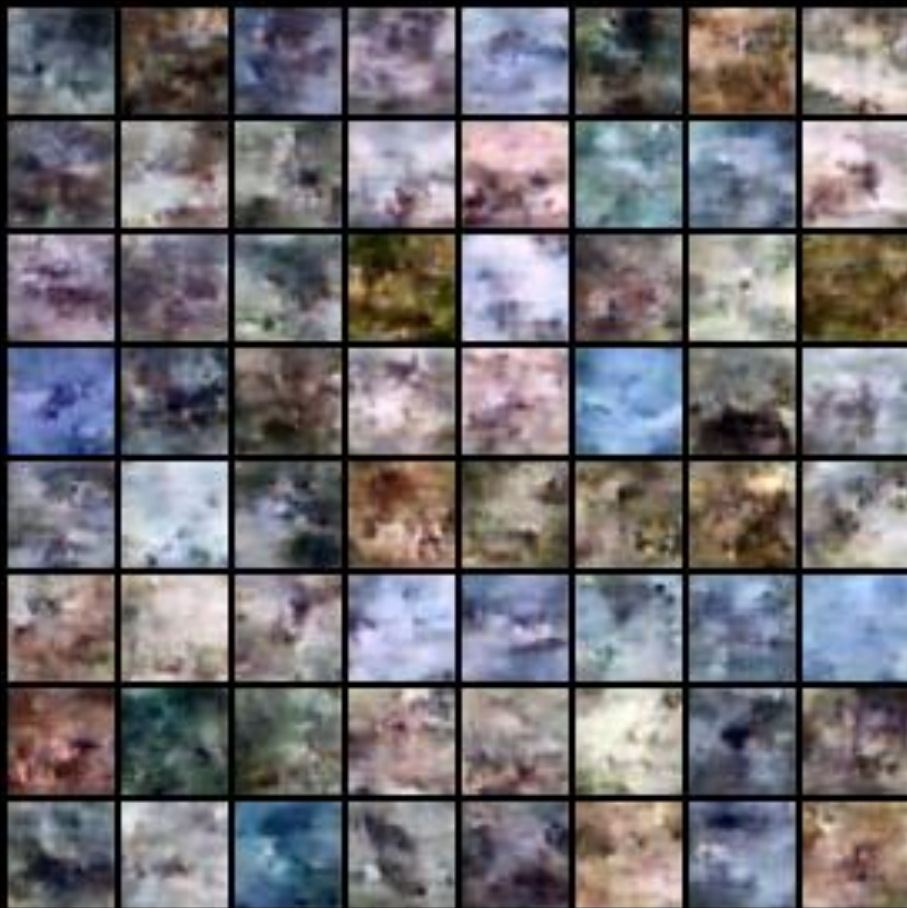
We have modified only few parts of the code provided to us. For example, we changed the save method for taking the output images during the training and saving them in a separate folder for calculate the metrics.

We upload the entire code on a [Github repo](#) for using it instead of the original one.

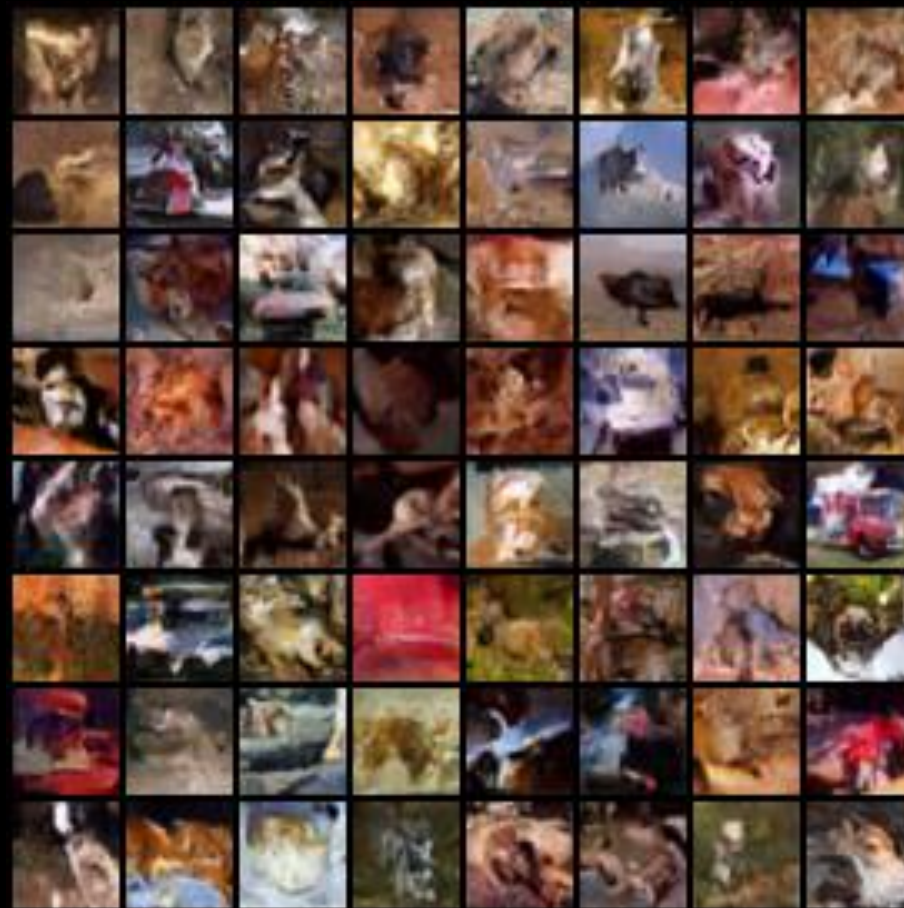
We created a [notebook Colab](#) for doing all the necessary stuff for testing and training the CIFAR10 dataset using the diffusion model of the paper. So, all operations like load dataset, creating the Unet model, the trainer class, setting the correct parameters, etc.

A particular note concerns the calculation of metrics: the original code did not provide any method to calculate IS and FID, for this reason we used external modules and functions

The images generated during our training...



With 3000 steps...

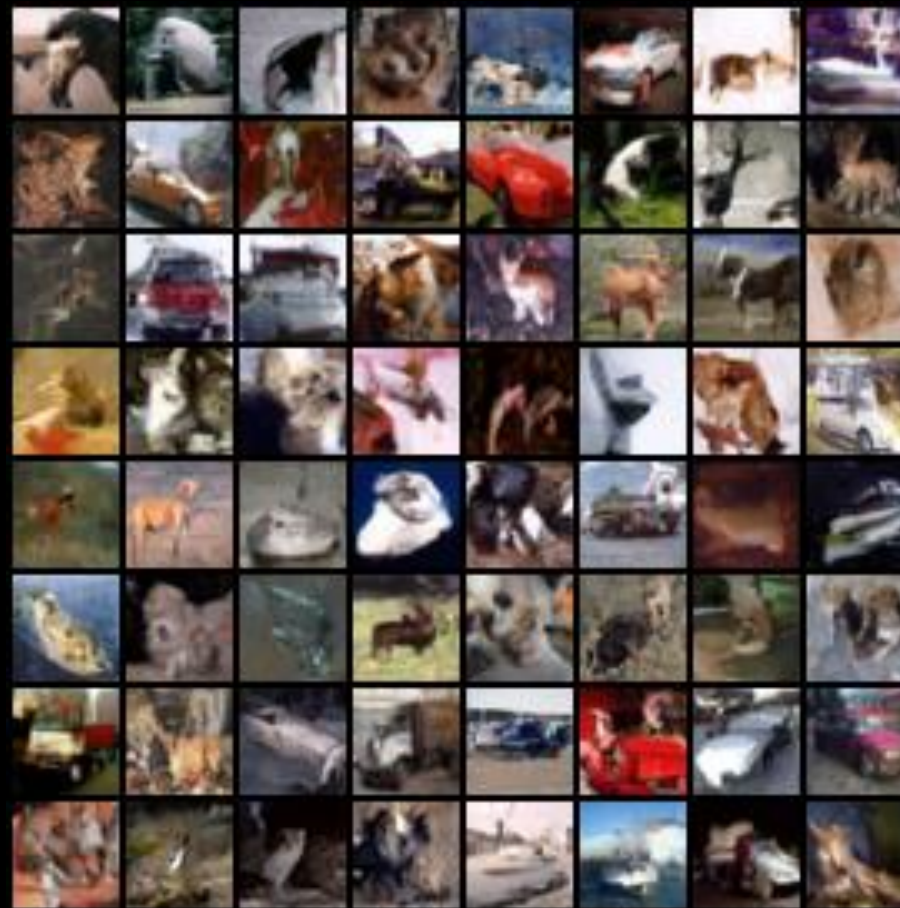


With 15000 steps...

The images generated during our training...



With 30000 steps...



With 45000 steps...

The images generated during our training...



With 60000 steps...



With 75000 steps...

The images generated during out training...




With 90000 steps...



With 100000 steps...

Our Results

```
# start training function
trainer.train()


load model
loss: 0.0301: 12%  100000/800000 [1:35:05<118:14:14, 1.64it/s]
```

With our limited resources we have running the training model just for 100k steps, approximately the 12% of the study training. So we obtain about 1000 images, with these images we have calculated the Inception Score and FID.

the **Inception Score** that we obtained is 5.75 ± 0.40 while the result of the paper is 9.46 ± 0.11

```
Calculating Inception Score...
Computing KL-Div Mean...
10 / 10
(5.759524471083759, 0.40455183883212237)
```

the **FID** found by us is 44.47 compared to that of the study which is 3.17

```
Found 1000 images in the folder /content/results/images
FID images : 100%|| 32/32 [00:27<00:00, 1.16it/s]
44.47033408922039
```

The results that we obtain are far from those obtained in the study. Surely this is due to the fact that we did the training for just 100,000 steps

What are our opinions about this work?

Pros: ✓

- the denoising diffusion model is very advanced and interesting, during the work we realized that the code files on github were daily maintained and updated.
- Despite being the state of the art there is so much excitement around this model, this has allowed us to find useful information in forums and sites.

Cons: ✗

- On the other hand, we found the paper extremely complex. The authors of the study report many mathematical formulas and take for granted a lot of knowledge.
- the model code was initially difficult to approach because it was complex and divided into several files. One of the main problem is the absence of comments, so for example there are a lot of functions where parameters have generic names (like x, t, a, etc) without any explanation