Diffusion Model

Introduction

We have read about two types of generative models, i.e. GAN and VAE. Both models have shown great success in generating high quality samples, but each has some limitations of its own. The output generated by GANs has less diversity due to adversarial and unstable training nature. However, VAE relies on surrogate loss. Unlike both, diffusion models are inspired by physical phenomenon of "non-equilibrium thermodynamics". They define a Markov chain of diffusion process to first add random noise gradually and then learn how to remove the noise to construct desired data samples. With diffusion models, we are in the domain of Generative Deep Learning that indicates the learning of the distribution of data to generate new data.

It outperforms the GAN especially in image generation. In generative modelling, diffusion models have attracted a lot of attention particularly in image generation. It has shown impressive performance in many conditional settings such as converting text description to images. Let's take a closer look to it

General Idea

Imagine if you add a Gaussian noise to an image and repeat the process enough times to get a unrecognizable static image of noise sample. How we can undo this? i.e., start from noise sample and end up with the coherent image. This is the general idea behind the diffusion models. The figure 1 explains it.

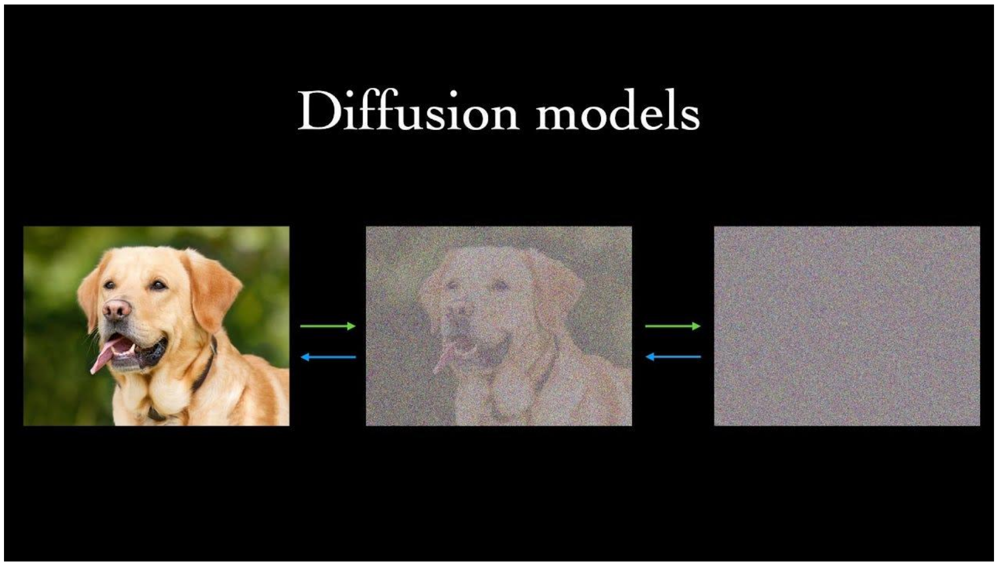


Figure 1

In short, we apply lot of noise to the image and then use a Neural Network for removing the noise. When the neural network has learned how to remove the noise, we apply some random noise sample and model remove the noise until an image is found.

More formally, the structure of the data is destroyed systematically and slowly through an iterative process (**Forward Diffusion Process**). Then, learn how to restore the structure of the data using an iterative process (**Reverse Diffusion Process**), yielding highly flexible and tractable generative model of data.

How it works?

For better understanding, the whole process is divided into two sub-processes as illustrated in the figure 2.

1) **Forward Diffusion process:** This process involves how to apply noise to the original image in an iterative manner to destroy the information slowly. So, start with the original image, we add small amount of Gaussian Noise. This step is repeated for sufficient number of times and all the information is destroyed, i.e., pure noise is obtained. This process seems to be very simple and fixed.

2) **Reverse Diffusion process:** Unlike forward process, the reverse process involves how to restore the image from the noise. It also involves the neural network which is provided with a noise sample and it repeatedly remove the noise step by step until a coherent image is obtained. Why we are doing it in step-by-step fashion? The reason is that going from noise to image directly leads to worse outcome and isn't tractable. So, a Neural network is involved to learn the removal of noise in a stepwise manner.

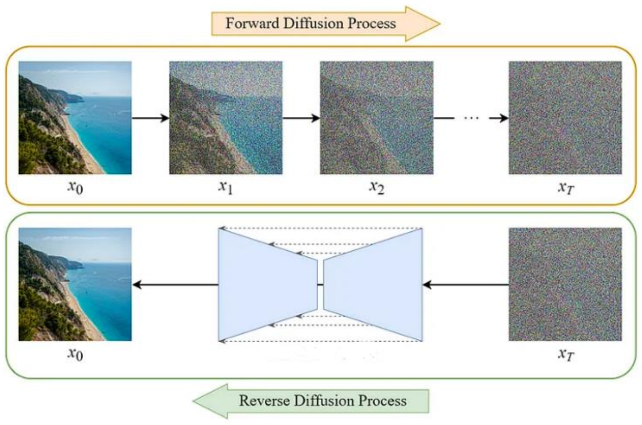


Figure 2(Forward and Reverse Diffusion Process)

The neural network involved takes the noise sample and produces the actual image. However, what actually the neural network is predicting? The neural network is actually predicting the noise in the image directly at step t and remove that amount of noise from the image and repeat this process until got clear image.

Note: In forward process, we never add same amount of noise at each time step. This thing is regulated by a schedule which scales the mean and variance to ensure that the variance isn't explode as we add more noise. The first paper "Denoising Diffusion Probabilistic Model (DDPM)" employ the **Linear schedule** for this purpose. However, later in 2021, OPENAI authors found this approach to be suboptimal and introduced the **Cosine Schedule.** The Linear schedule is found to be redundant at later stages, i.e., in the last couple of time steps, the image already becomes the noise, i.e., information was destroying too fast. So, applying noise further is uninformative. The above two issues are resolved by cosine schedule. This is illustrated in the figure 3.

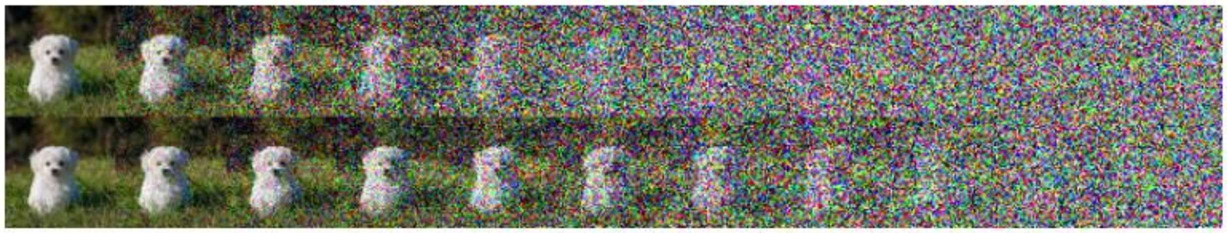


Figure 3(Linear vs Cosine schedule)

Architecture

The architecture of the neural network involve in reverse diffusion process is the **U-Net** architecture. This architecture takes image as an input and using down sample and Resnet blocks, it projects the image to a smaller resolution. After this, there's a bottleneck in the middle. After the bottleneck, it projects the image back to the original image dimensions using up sample blocks. Also, the attention blocks and skip connections are also used between the layers of same spatial resolution. This architecture is represented in the figure 4.

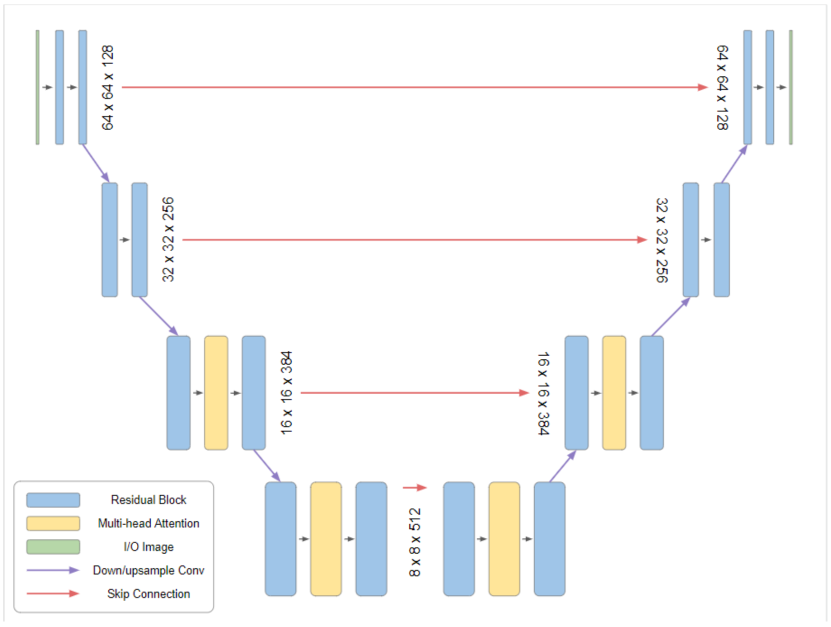


Figure 4(U-Net Architecture)

Note: The way of telling the model at which time step it is, is done via **Sinusoidal Embeddings** as represented in the image 5 as time embeddings. These embeddings is projected into each residual block. Since forward diffusion process is using a schedule that scales the mean and variance to control the amount of noise applied at each time step t. Similar to this, the reverse process uses these sinusoidal embedding to take care of removing different amount of noise at different time steps.

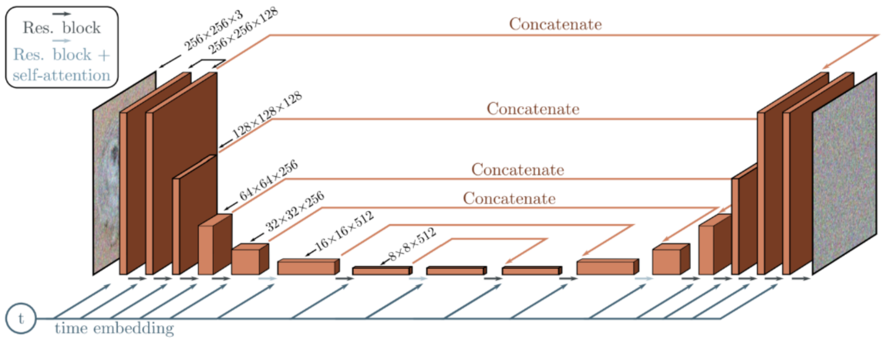


Figure 5(Representing time embeddings)

The OPENAI authors improved the architecture in 2021 paper as:

- Increase the depth and decrease the width of the network
- Add more attention layers and increase the attention blocks.
- Use the residual blocks from BigGAN and use this for up sampling and down sampling blocks.

- They introduce the idea of incorporating time step and class label slightly differently using **Adaptive Group Normalization**

$$AdaGN(h, y) = y_s \; groupNorm(h) + y_b$$

- It introduces the **Classifier Guidance** method which uses sperate classifier to help the diffusion model to generate certain class.

Mathematical Definition

The mathematical derivation of the loss function is defined once the processes are defined clearly.

1) **Define Notation:**
   - ➢ $\boldsymbol{x_t} \; = \boldsymbol{Image\ at\ time\ step\ t}$
   - ➢ $\boldsymbol{x_0} \; = \boldsymbol{Original\ Image / \ Image\ at\ timestep\ 0}$
   - ➢ $\boldsymbol{x_{42}} = \boldsymbol{Image\ after\ applying\ 42\ iterations\ of\ Noise}$
   - ➢ $\boldsymbol{x_T} \; = \boldsymbol{Final\ Image\,.}$ It follows isotropic gaussian (looks same in every direction)
   - ➢ $\boldsymbol{q(x_t|x_{t-1})} = \boldsymbol{Forward\ Process.}$ x_t is the most noised version of image while x_{t-1} is the less noised version of image. This indicates that it takes x_{t-1} and after adding a bit of noise, it produces x_t which has more noise than x_{t-1}.
   - ➢ $\boldsymbol{p(x_{t-1}|x_t)} = \boldsymbol{Reverse\ Process.}$ This indicates that it takes image x_t and after removing noise from the image, it produces x_{t-1} using a neural network.

2) **Define Forward diffusion process:**

   One time step of forward process is defined as:

$$\boldsymbol{q(x_t|x_{t-1}) = N(x_t\,, \sqrt{1-\beta_t}\, x_{t-1}\,, \beta_t I)}$$

   Which is a normal distribution having mean $\boldsymbol{\sqrt{1-\beta_t}\, x_{t-1}}$ and variance $\boldsymbol{\beta_t I}$**.** It's a conditional gaussian distribution with mean that depends on the previous image and a specific variance. The sequence of $\boldsymbol{\beta}$ refers to the variance schedule. It describes how much noise we want to add in each of the time step. All betas are ranging from 0 to 1 and increases with time, i.e.

$$\boldsymbol{\beta_1 \; < \; \beta_2 \; < \; \ldots\ldots < \beta_t <. \ldots\ldots < \beta_T}$$

$$\boldsymbol{\beta_t \; \in [0, 1]}$$

Let’s get some intuition about β. Imagine an image with lot of pixels. Each pixel in the image has three channels, red, green and blue denoted as (R, G, B). Each pixel has value in range (0,255). After normalization, every pixel now lies in range (-1,1). Let’s consider an Image A as represented in the figure 6. The color code for the selected pixel is [1,-1,-1]. It indicates the red pixel with no green and blue added. The distribution of the next image is described by the mean and variance. Values of the red pixel multiplied by the $\sqrt{1-\beta}$ is the exact mean of that pixel.

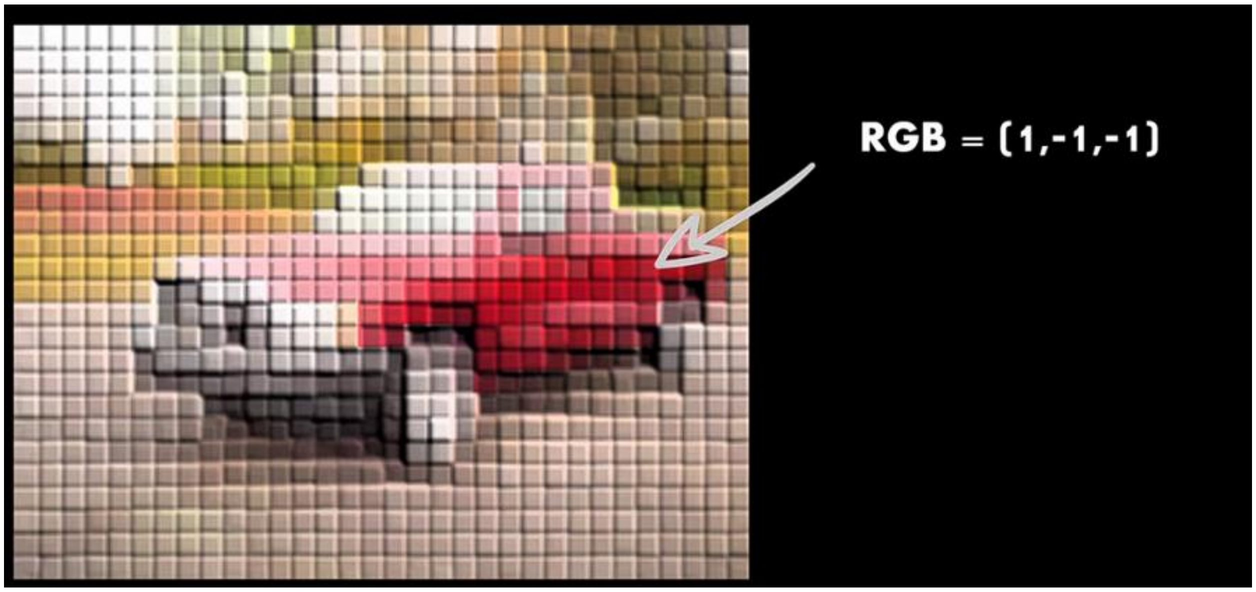


Figure 6(Image A)

Depending on the noise level β, it could be 0.99 for first image. Since variance is fixed, if we choose large beta, that indicates the pixel distribution is not just wider but also shifted which results in the more corruption in image. Sampling from such distribution, we consequently end up with more noise. Eventually, beta controls how fast we converge towards a mean of zero which correspond to standard Gaussian distribution. This is explained in the figure 7.

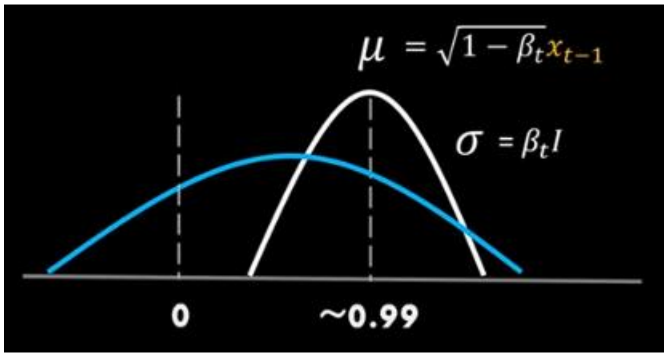


Figure 7(Effect of beta)

The important thing here is to add right amount of noise such that we arrive at isotropic gaussian distribution with mean zero and have fixed variance in all directions. Otherwise, sampling won't work. In other words, we don't want to add too much noise and vice versa. There are different strategies for this scheduling, we will be using Linear Scheduling where we add noise linearly. Other strategies include quadratic, cosine, and sigmoidal.

Coming back to the above formula of forward diffusion, we have to repeat this process T times to get output $\boldsymbol{x_t}$. The above relation can be written after applying the reparameterization trick ($N(\mu\,,\sigma^2) = \mu + \sigma \odot \epsilon$) as:

$$N\left(\sqrt{1-\beta_t}\, x_{t-1}, \sqrt{\beta_t}\,\right) = \sqrt{1-\beta_t}\, x_{t-1} + \sqrt{\beta_t} \odot \epsilon$$

Where $\epsilon \sim N(0,1)$

However, researchers found better solution instead of sequential approach. The solution represents the direct calculation of the noisy version of the image for a specific timestep t. Let's define $\boldsymbol{\alpha_t = 1 - \beta_t}$ and $\boldsymbol{\bar{\alpha} = \prod_{s=1}^{t} \alpha_s}$**.** The accumulative alphas are the multiplication of all alphas.

$$N\left(\sqrt{\alpha_t}\, x_{t-1}, \sqrt{1-\alpha_t}\right) = \sqrt{\alpha_t}\, x_{t-1} + \sqrt{1-\alpha_t} \odot \epsilon$$

Extend the R.H.S to earlier time steps and go from t-2 timestep to t. It can be done easily,

$$= \sqrt{\alpha_t\ \alpha_{t-1}}\, x_{t-2} + \sqrt{1-\alpha_t\ \alpha_{t-1}} \odot \epsilon$$

From time step t-3 to t, it would be:

$$= \sqrt{\alpha_t\ \alpha_{t-1}\, \alpha_{t-2}}\, x_{t-3} + \sqrt{1-\alpha_t\ \alpha_{t-1}\ \alpha_{t-2}} \odot \epsilon$$

Thus, from x_0 to x_t, the relation would be:

$$= \sqrt{\alpha_t\ \alpha_{t-1}\ \ldots\ldots\ldots\alpha_0}\, x_0 + \sqrt{1-\alpha_t\ \alpha_{t-1}\ \ldots\ldots\ldots\alpha_0} \odot \epsilon$$

Applying Accumulative alphas:

$$= \sqrt{\boldsymbol{\bar{\alpha}}}\, \boldsymbol{x_0} + \sqrt{\boldsymbol{1-\bar{\alpha}}} \odot \boldsymbol{\epsilon}$$

Thus, the forward process is simplified as:

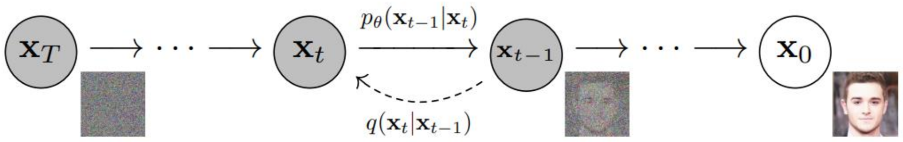
$$\boxed{\boldsymbol{q(x_t|x_{t-1}) = N(x_t, \sqrt{\bar{\alpha}}\, x_0\,, \sqrt{1-\bar{\alpha}}\ I)}}$$

3) Define Reverse diffusion process:

One time step of reverse process that involves the neural network to learn the removal of noise from the image is represented as:

$$\boldsymbol{p(x_{t-1}\,|x_t) = N\left(x_{t-1}\,, \mu_\vartheta(x_t, t), \sum_\theta(x_t, t)\right)}$$

Where $\boldsymbol{\mu_\vartheta(x_t, t)}$ represents the mean at particular time step and $\boldsymbol{\sum_\theta(x_t, t)}$ represents the variance. The variance is fixed and the neural network predicts the mean or noise in the image. The reverse (from x_T to x_0) and forward process (from x_0 to x_T) is represented in the figure 6.



$\mathbf{x}_T$
$\mathbf{x}_t$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$\mathbf{x}_{t-1}$
$\mathbf{x}_0$

Figure 8(Forward and Reverse diffusion process)

If we apply reverse formula for all time steps, we can go from x_t to the original data distribution as:

$$p_\theta\,(x_{0:T}) = p_\theta(x_T) + \prod_{t=1}^{T} p_\theta(x_{t-1}\mid x_t)$$

Objective/Loss Function:

One way to define the objective function is to simply define the negative likelihood of the probability of original image i.e. $\boldsymbol{-\log(p_\theta(x_0)}$. To compute this, we need to keep track of all the previous time steps which is the main issue. The above expression is completely dependent on all the other time steps, i.e. $x_1, x_2, \dots \dots . \dots ., x_T$. Thus, we have to keep track of the t-1 random variables which is just can't possible.

Another way of defining the loss function is to compute the **variational lower bound** just like we did in the VAE. The overall setup of diffusion process is pretty much similar to the VAE, i.e. the forward diffusion process is analogous to the encoder part of VAE (producing latency from data) while the reverse diffusion process is analogous to the decoder part (producing data from latency). However, the forward process is fixed unlike encoder in VAE. So, therefore, only one network is used for learning the reverse process. The training objective is similar to VAE as:

$$\log\big(p_\theta(x)\big) \;\geq \mathbb{E}_{q(Z|x)}(\log(p_\theta(x|z))) - \; D_{KL}(q_\varphi(z|x)\;||\; p_\theta(z))$$

The log likelihood term encourages model to maximize the expected density function assign to the data. The KL divergence ensures that the posterior probability is similar to the prior distribution. So, mapping the objective function in diffusion models when observed variable is x_0, latent variable is $x_{(1:T)}$ and forward diffusion process to be fixed via schedule, the above relation becomes:

$$\log\big(p_\theta(x_0)\big) \;\; \geq \mathbb{E}_{q(x_{1:T}|x_0)}(\log(p_\theta(x))) - \;\; D_{KL}(q(x_{1:T}|x_0) \;||\; p_\theta(x_{1:T}))$$

The idea of the lower bound comes when we can't compute a function directly. For instance, f(x) can't be computed directly because it is dependent on others (-ve log likelihood in our case). In that case, we can easily prove that using another function say g(x) which is computable and smaller than f(x), the maximum of g(x) will ensure that f(x) will always increase.

This above case is ensured by subtracting the KL divergence that defines the similarity between two distribution and always non-negative. Since, subtracting something non-negative from a function will always result in something less than original. Here, we want to maximize the negative of the lower bound, that means we should minimize the negative of the training objective as:

$-\log\big(p_\theta(x_0)\big) \;\; \leq -\mathbb{E}_{q(x_{1:T}|x_0)}(\log(p_\theta(x))) + \;\; D_{KL}(q(x_{1:T}|x_0) \;||\; p_\theta(x_{1:T}))$ -------**(A)**

Note: For brevity, write $-\mathbb{E}_{x_{1:T} \sim q(x_{1:T}|x_0)}$ as $\mathbb{E}_q$

Derivation

The KL divergence term in EQ(A) is expanded as:

$$D_{KL}\big(q(x_{1:T}|x_0) \,||\, p_\theta(x_{1:T})\big) = \mathbb{E}_q\{\log\left(\frac{\big(q(x_{1:T}|x_0)\big)}{p_\theta(x_{1:T}|x_0)}\right)\}$$

Consider denominator, apply Bayes Rule to it and write it as joint probability:

$$\big(p_\theta(x_{1:T}|x_0)\big) = \frac{p_\theta(x_0|x_{1:T})p_\theta(x_{1:T})}{p_\theta(x_0)} = \frac{p_\theta(x_0\,,\; x_{1:T})}{p_\theta(x_0)} = \frac{p_\theta(x_{0:T}\,)}{p_\theta(x_0)}$$

The above KL divergence term now becomes:

$$D_{KL}\big(q(x_{1:T}|x_0) \,||\, p_\theta(x_{1:T})\big) = \mathbb{E}_q\{\log\left(\frac{\big(q(x_{1:T}|x_0)\big)}{\frac{p_\theta(x_{0:T}\,)}{p_\theta(x_0)})}\right)\}$$

Apply logarithmic properties:

$$D_{KL}\big(q(x_{1:T}|x_0) \;||\; p_\theta(x_{1:T})\big) = \mathbb{E}_q\left\{\log\left(\frac{\big(q(x_{1:T}|x_0)\big)}{p_\theta(x_{0:T})}\right) * p_\theta(x_0)\right\}$$

$$= \mathbb{E}_q\{\log\left(\frac{\big(q(x_{1:T}|x_0)\big)}{p_\theta(x_{0:T})}\right)\} + \mathbb{E}_q(\log\big(p_\theta(x_0)\big))$$

Put the simplified form of KL divergence to EQ(A),

$$-\log\big(p_\theta(x_0)\big) \;\leq -\mathbb{E}_q(\log(p_\theta(x_0)) + \;\mathbb{E}_q(\log\left(\frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right)) \;+ \mathbb{E}_q(\log\big(p_\theta(x_0)\big))$$

$$-\log\big(p_\theta(x_0)\big) \le \mathbb{E}_q(\log\left(\frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right))$$

This is our variational lower bound that will be minimize where the numerator refers to the forward process starting from original image of the data while the denominator is still need to simplify more. Since, reverse process is a Markov chain of all the previous time steps as defined above, so it can be written as:

$$p_\theta\,(x_{0:T}) = p_\theta(x_T) + \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t)$$

The forward process is also written as a Markov chain. Let's simplify the log term:

$$\frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})} = \log\left(\frac{\prod_{t=1}^{T} \mathrm{q}(x_t \mid x_{t-1})}{p_\theta(x_T) + \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t)}\right)$$

$$= -\log\big(p_\theta(x_T)\big) + \log\left(\frac{\prod_{t=1}^{T} \mathrm{q}(\,x_t \mid x_{t-1}\,)}{\prod_{t=1}^{T} p_\theta(\,x_{t-1} \mid x_t\,)}\right)$$

$$= -\log\big(p_\theta(x_T)\big) + \sum_{t=1}^{T} \log\left(\frac{q(\,x_t \mid x_{t-1}\,)}{p_\theta(\,x_{t-1} \mid x_t\,)}\right)$$

Expand the summation:

$$= -\log\big(p_\theta(x_T)\big) + \sum_{t=2}^{T} \log\left(\frac{q(\,x_t \mid x_{t-1}\,)}{p_\theta(\,x_{t-1} \mid x_t\,)}\right) + \log\left(\frac{q(x_1|x_0}{p_\theta(x_0|x_1)}\right)$$

Consider, $q(\,x_t \mid x_{t-1}\,)$, Apply Bayes rule,

$$q(\,x_t \mid x_{t-1}\,) = \frac{q(x_{t-1}|x_t)q(x_t)}{q(x_{t-1})}$$

All of the terms in above relation have high variance since we don't know what we started. If original image is provided with it, the variance is reduced as:

$$q(\,x_t \mid x_{t-1}, x_0\,) = \frac{q\big(x_{t-1}|x_{t,}x_0\big)q(x_t|x_0)}{q(x_{t-1}\,|x_0)}$$

Put the above relation back in formula as:

$$= -\log\big(p_\theta(x_T)\big) + \sum_{t=2}^{T} \log\left(\frac{q\big(x_{t-1}|x_{t,}x_0\big)q(x_t|x_0)}{q(x_{t-1}\,|x_0)p_\theta(\,x_{t-1} \mid x_t\,)}\right) + \log\left(\frac{q(x_1|x_0}{p_\theta(x_0|x_1)}\right)$$

$$= -\log\big(p_\theta(x_T)\big) + \sum_{t=2}^{T} \log\left(\frac{q\big(x_{t-1}|x_{t,}x_0\big)}{p_\theta(\,x_{t-1} \mid x_t\,)}\right) + \sum_{t=2}^{T} log\left(\frac{q(x_t|x_0)}{q(x_{t-1}\,|x_0)}\right) + \log\left(\frac{q(x_1|x_0}{p_\theta(x_0|x_1)}\right)$$

Consider $\sum_{t=2}^{T} log\left(\frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}\right)$ when T=4, it is expanded as:

$$\sum_{t=2}^{4} log\left(\frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}\right) = \log\left(\prod_{t=1}^{4} \frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}\right) = \log\left(\frac{q(x_2|x_0)q(x_3|x_0)q(x_4|x_0)}{q(x_1|x_0)q(x_2|x_0)q(x_3|x_0)}\right) = \log\left(\frac{q(x_4|x_0)}{q(x_1|x_0)}\right)$$

The last and the first term survive only, so we can write it as $\log\left(\frac{q(x_T|x_0}{q(x_1|x_0}\right)$. Replace it:

$$= -\log(p_\theta(x_T)) + \sum_{t=2}^{T} \log\left(\frac{q(x_{t-1}|x_{t,}x_0)}{p_\theta(x_{t-1} \mid x_t)}\right) + \log\left(\frac{q(x_T|x_0}{q(x_1|x_0}\right) + \log\left(\frac{q(x_1|x_0}{p_\theta(x_0|x_1)}\right)$$

After applying the logarithmic rules on last two terms, we got only two terms,

$$= -\log(p_\theta(x_T)) + \sum_{t=2}^{T} \log\left(\frac{q(x_{t-1}|x_{t,}x_0)}{p_\theta(x_{t-1} \mid x_t)}\right) + \log(q(x_T|x_0) - \log(p_\theta(x_0|x_1)$$

Combine first and third term and fuse them in one logarithm term as:

$$= \sum_{t=2}^{T} \log\left(\frac{q(x_{t-1}|x_{t,}x_0)}{p_\theta(x_{t-1} \mid x_t)}\right) + \log\left(\frac{q(x_T|x_0}{p_\theta(x_T)}\right) - \log(p_\theta(x_0|x_1)$$

$$= D_{KL}(q(x_T|x_0 \,||\, p_\theta(x_T)) + \sum_{t=2}^{T} \mathrm{D_{KL}}\left(q(x_{t-1}|x_{t,}x_0) \,||\, p_\theta(x_{t-1} \mid x_t)\right) - \log(p_\theta(x_0|x_1)$$

The first term is ignored completely since it has no learnable parameters. It consists of simple forward process which is adding noise over and over and $p_\theta(x_T)$ is just the noise sample. Therefore, we are left with

$$= \sum_{t=2}^{T} \mathrm{D_{KL}}\left(q(x_{t-1}|x_{t,}x_0) \,||\, p_\theta(x_{t-1} \mid x_t)\right) - \log(p_\theta(x_0|x_1)$$

Considering the KL divergence, both forward and reverse process is written as $N(x_{t-1}, \mu_\vartheta(x_t, t), \beta I)$ and $N(x_{t-1}, \tilde{\mu}(x_t, x_0), \tilde{\beta} I)$ respectively. Consider, forward process where the $\tilde{\beta}$ is fixed, the $\tilde{\mu}(x_t, x_0)$ is equivalent to:

$$\tilde{\mu}(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t + \frac{\sqrt{\bar{\alpha}_t - 1}\beta_t}{1-\bar{\alpha}_t} x_0$$

Since, $x_t = \sqrt{\bar{\alpha}}\, x_0 + \sqrt{1-\bar{\alpha}} \odot \epsilon$, x_0 can be derived from this as:

$$x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon\right)$$

Put the value of x_0 in the mean expression,

$$\tilde{\mu}(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t + \frac{\sqrt{\bar{\alpha}_t - 1}\beta_t}{1-\bar{\alpha}_t} \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \sqrt{1-\bar{\alpha}_t}\ \epsilon\right)$$

$$= \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)$$

This $\tilde{\mu}(x_t, x_0)$ is predicted by the neural network and we have to calculate simple means square error between actual $\tilde{\mu}(x_t, x_0)$ and predicted $\mu_\vartheta(x_t, t)$ mean.

$$\boldsymbol{L_t = \frac{1}{2\sigma^2} \mid \tilde{\mu}(x_t, x_0) - \mu_\vartheta(x_t, t)|^2}$$

$$L_t = \frac{1}{2\sigma^2}\left|\frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right) - \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right)\right|^2$$

After further simplifying the above expression, we end up with the following expression:

$$L_t = \frac{\beta_t^2}{2\sigma^2 \alpha_t (1-\bar{\alpha}_t)} \, |\epsilon - \epsilon_\theta(x_t, t)|^2$$

This is the mean squared error between the actual noise ϵ and predicted noise $\epsilon(x_t, t)$ at time t. Recent research shows that the scaling factor can be ignored since it results in better sampling quality. So we end up with:

$$\boldsymbol{L_t = |\epsilon - \epsilon_\theta(x_t, t)|^2}$$

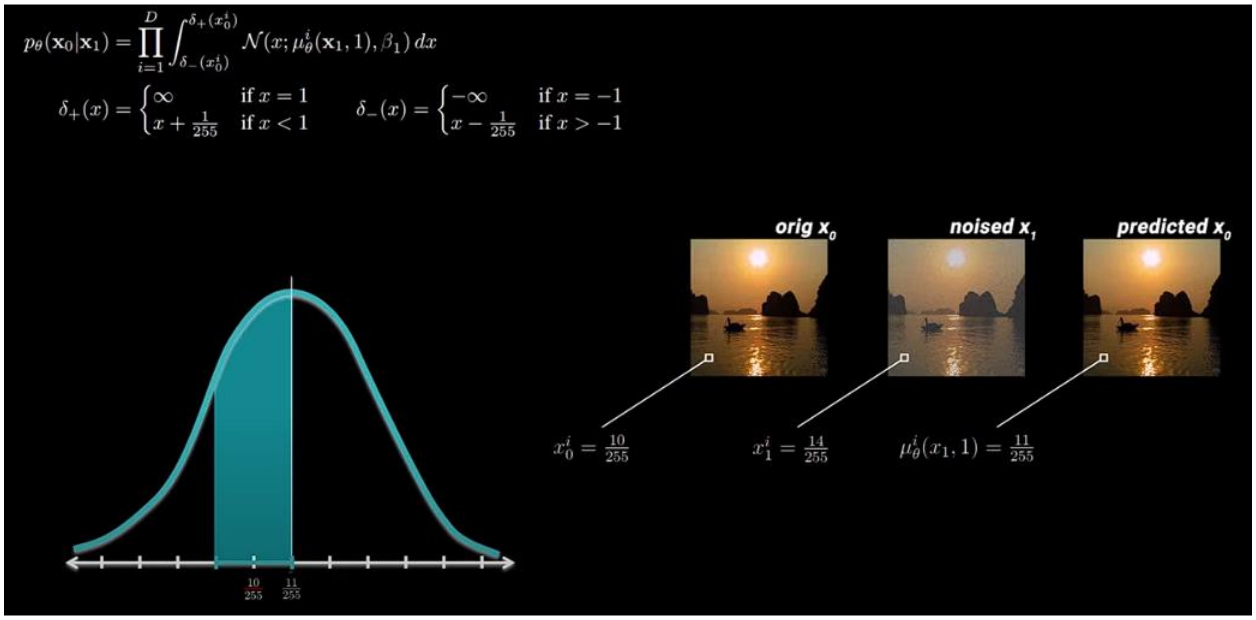
The last term in the objective function is taken care as:

$$p_\theta(x_0|x_1) = \prod_{i=1}^{D} \int_{\delta_-(x_0^i)}^{\delta_+(x_0^i)} N(x, \mu_\theta^i(x_1, 1), \beta_1) dx$$

Where

$$\delta_+(x) = \begin{cases} \infty & if\ x = 1 \\ x + \frac{1}{255} & if\ x < 1 \end{cases}, \quad \delta_-(x) = \begin{cases} -\infty & if\ x = -1 \\ x - \frac{1}{255} & if\ x > -1 \end{cases}$$

The D is the data dimensionality (no of pixels in our case). if computing the mean is around the true pixel area, the probability will be high, otherwise, probability will be low. The figure 7 explains the above concept. If the neural network predicts the mean 20/255, we got no probability mass since true value is around 10/255.

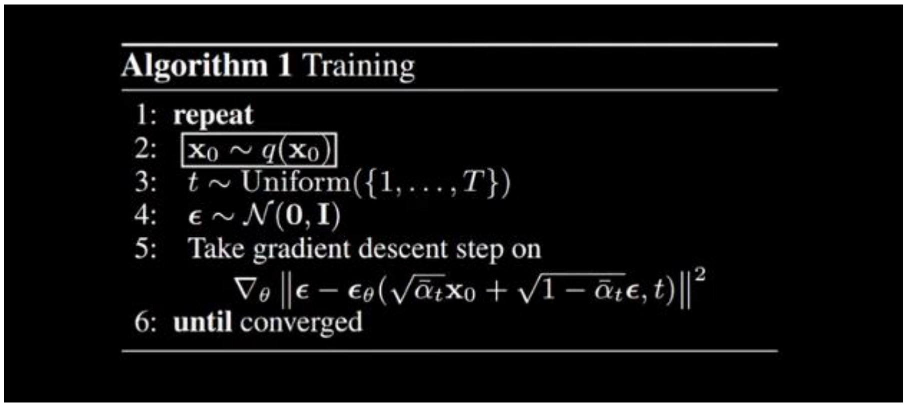


$$p_\theta(\mathbf{x}_0|\mathbf{x}_1) = \prod_{i=1}^{D} \int_{\delta_-(x_0^i)}^{\delta_+(x_0^i)} \mathcal{N}(x; \mu_\theta^i(\mathbf{x}_1, 1), \beta_1)\, dx$$

$$\delta_+(x) = \begin{cases} \infty & \text{if } x = 1 \\ x + \frac{1}{255} & \text{if } x < 1 \end{cases} \qquad \delta_-(x) = \begin{cases} -\infty & \text{if } x = -1 \\ x - \frac{1}{255} & \text{if } x > -1 \end{cases}$$

Figure 9

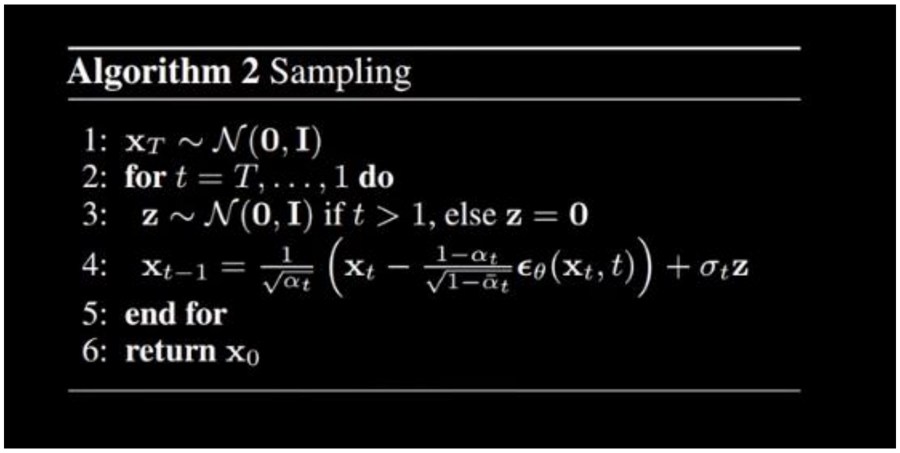
The training algorithm is represented in the figure 8 and sampling method is explained in the figure 9.



Algorithm 1 Training

```
1: repeat
2:   $\boxed{\mathbf{x}_0 \sim q(\mathbf{x}_0)}$
3:   $t \sim \text{Uniform}(\{1, \ldots, T\})$
4:   $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:   Take gradient descent step on
         $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: until converged
```

Figure 10(Training)



```
Algorithm 2 Sampling
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: for $t = T, \ldots, 1$ do
3:   $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4:   $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: end for
6: return $\mathbf{x}_0$
```

Figure 11(Sampling method)

Implementation of Diffusion Model

The architecture and design of the model is based on two research papers, “Denoising Diffusion Probabilistic Model” from researchers of Berkely University and its follow up paper from OpenAI researchers “Diffusion Model beats GANs on Image Synthesis”. Here we implement just the base model.

As discussed, Diffusion model works by destroying input from image by gradually adding noise to the image and then recover that image from the noise via backward process (also called Denoising). This is also called **Markov Chain** because it’s a sequence of stochastic events where each time step depends on previous time step. The specialty of diffusion model is that the latent space has the same dimensions as that of the input. The task that the model can be described as predicting the noise that was added in each of the image. That’s why backward process is said to be parameterized. So, how to implement this type of model, we’ll mainly need three things:

1) Implement Schedular that sequentially add noise
2) Implement a neural model that predicts noise
3) Implement a way to encode the current time step.

First let’s discuss about the dataset we have used. Its **Stanford cars** included in pytorch. It consists of total of around16,000 images (8,000 train and 8,000 test). We are using all of them as building a generative model. Let’s analyze the images that has been crop to same size and reduced in size so that learning can done faster. The images shown in the figure 10 are in variety of poses and backgrounds. This represents the diversity in the data set.

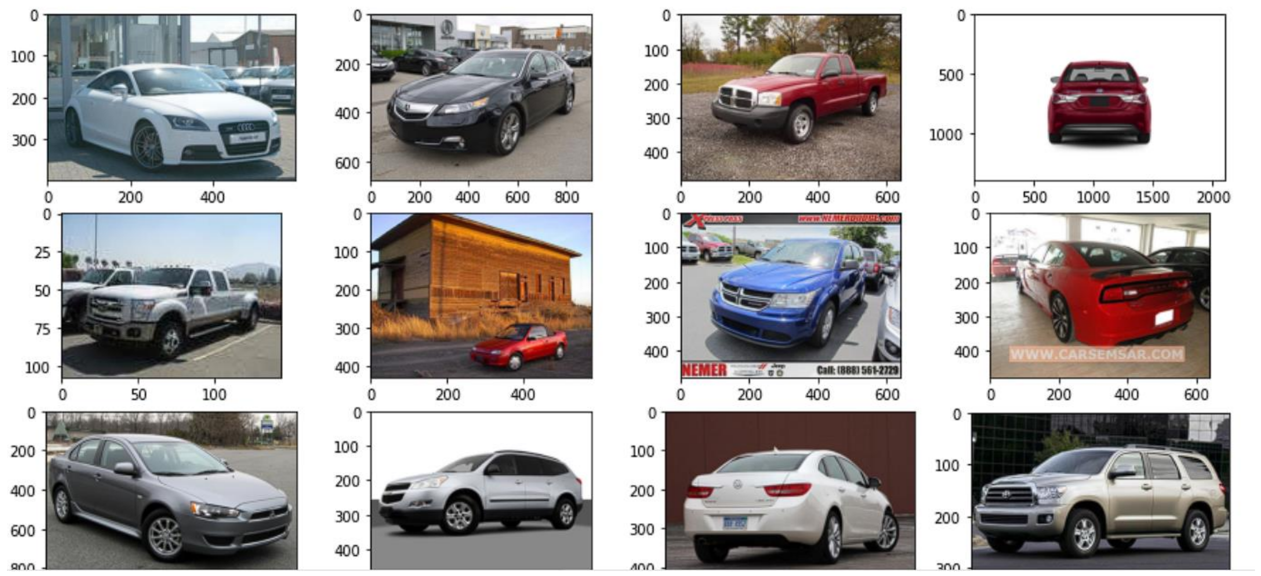
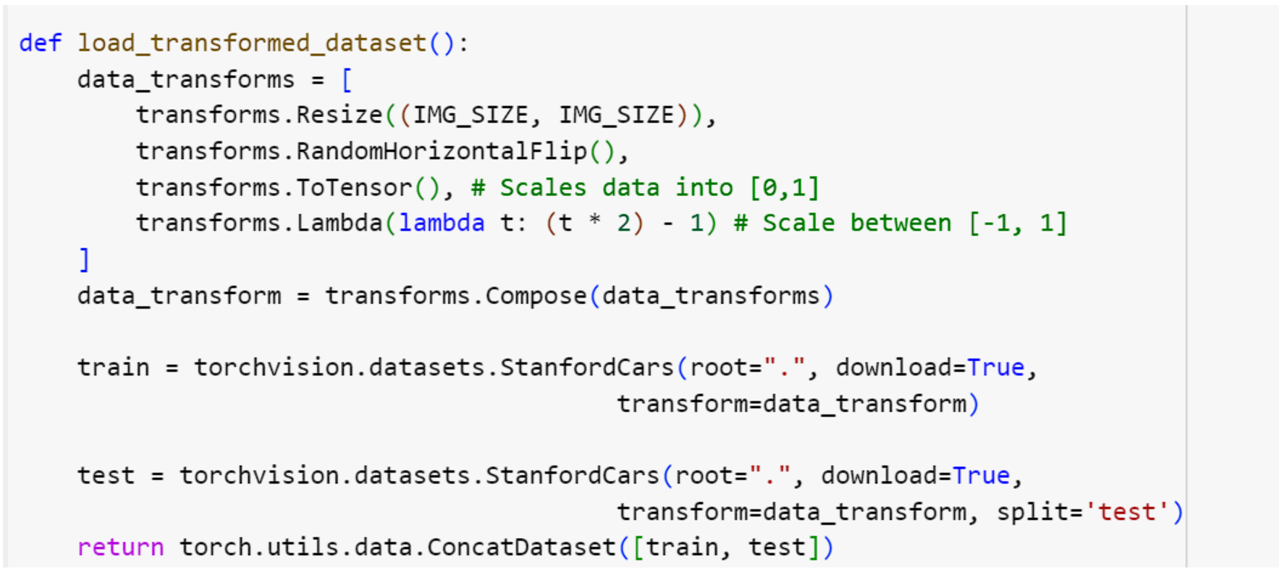


Figure 12(Standford Cars dataset)

So, here we are done up with downloading the dataset, converting into Tensors, perform cropping and resizing and convert the image pixel in range [-1,1] as explained in the figure 11.



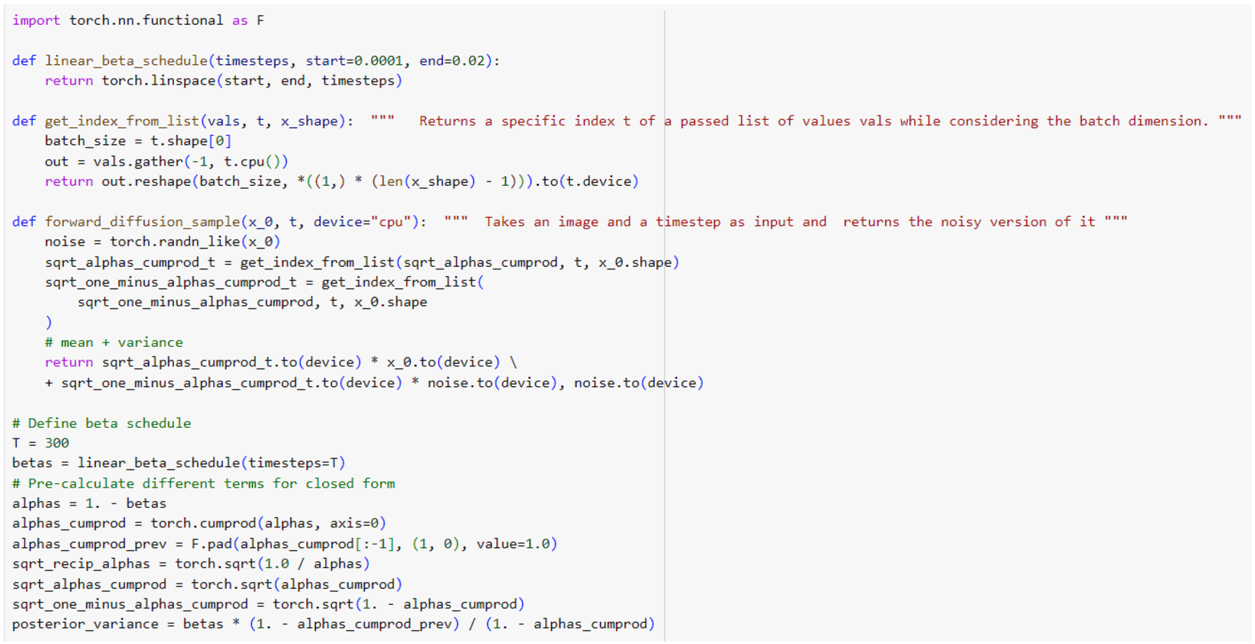
```
def load_transformed_dataset():
    data_transforms = [
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(), # Scales data into [0,1]
        transforms.Lambda(lambda t: (t * 2) - 1) # Scale between [-1, 1]
    ]
    data_transform = transforms.Compose(data_transforms)

    train = torchvision.datasets.StanfordCars(root=".", download=True,
                                         transform=data_transform)

    test = torchvision.datasets.StanfordCars(root=".", download=True,
                                         transform=data_transform, split='test')
    return torch.utils.data.ConcatDataset([train, test])
```

Figure 13(Dataset Transformation)

1) **Implement noise Schedular that sequentially add noise:** We have been following some existing implementations of diffusion model. As discussed earlier, this step involves adding noise to the image using Markov chain as:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t \mid x_{t-1})$$

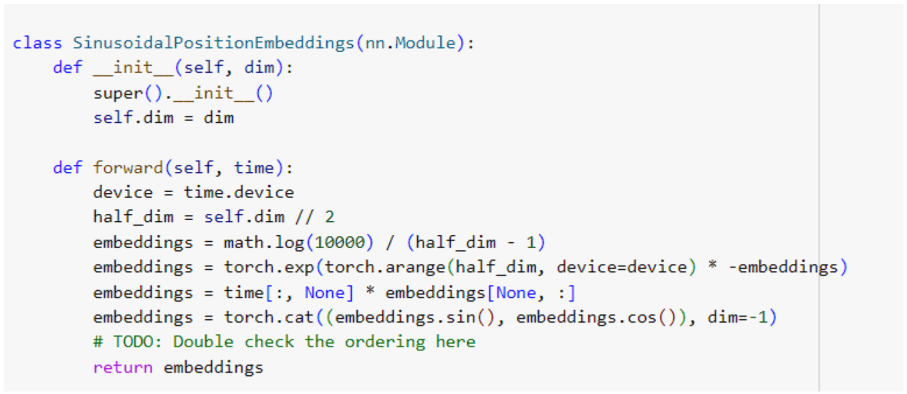
As discussed, this step is also known as Forward Diffusion Process. This shows that the noise that is added depends on the previous image. The way how noise is sampled is described as $q(x_t|x_{t-1}) = N(x_t\,, \sqrt{1-\beta_t}\, x_{t-1}\,, \beta_t I)$ and the closed form is written as $q(x_t|x_{t-1}\,) = N(x_t, \sqrt{\bar{\alpha}}\, x_0\,, \sqrt{1-\bar{\alpha}}\ I)$. We simply sample a time step t and add noise to the image as shown in the figure 12.



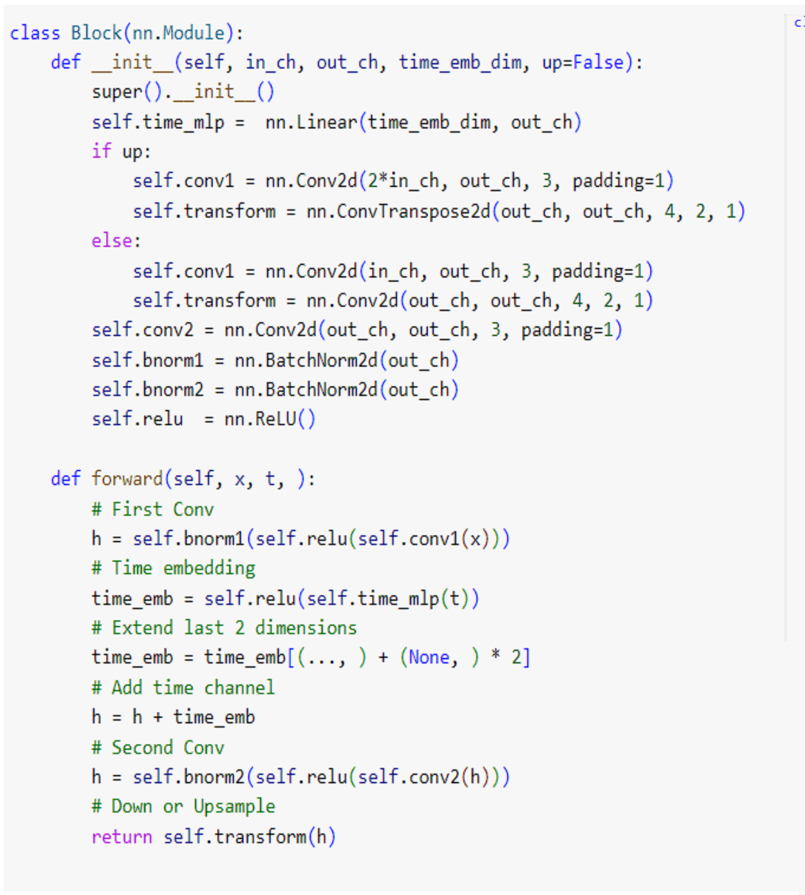
```python
import torch.nn.functional as F

def linear_beta_schedule(timesteps, start=0.0001, end=0.02):
    return torch.linspace(start, end, timesteps)

def get_index_from_list(vals, t, x_shape):  """  Returns a specific index t of a passed list of values vals while considering the batch dimension. """
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())
    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

def forward_diffusion_sample(x_0, t, device="cpu"):  """  Takes an image and a timestep as input and  returns the noisy version of it """
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )
    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.to(device)

# Define beta schedule
T = 300
betas = linear_beta_schedule(timesteps=T)
# Pre-calculate different terms for closed form
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

Figure 14(Forward diffusion)

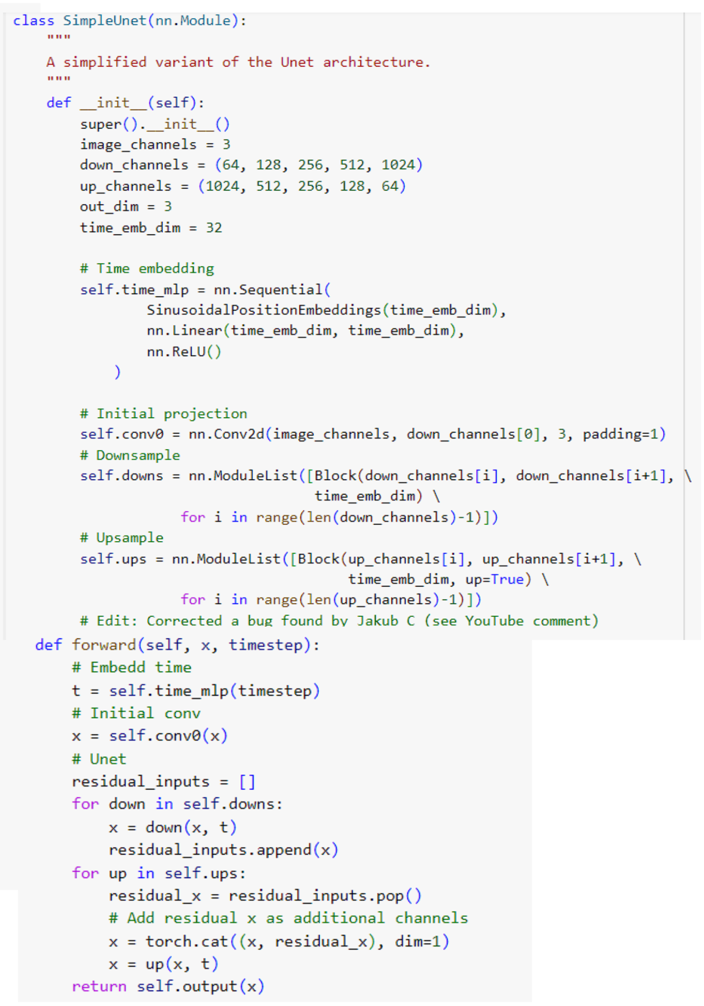
2) **Implement a neural model that predicts noise:** The architecture used by the researchers to implement the neural network is known as **U-Net. U-Nets** are famous for image segmentation. This architecture produces the output having the same dimension as that of input. Briefly, input is passed through a series of convolutional and down sampling layers until a bottle neck is reached. After this, the image tensors are then up sampled and pass through more convolutional layers. The architecture of the neural network is explained above. It is implemented as explained in the figure which takes the noisy image as input and predict the noise in the image. Because the variance is fixed, it only outputs one value which is the mean of the gaussian distribution of the image. This is also called denoising score matching.



```python
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        # TODO: Double check the ordering here
        return embeddings
```



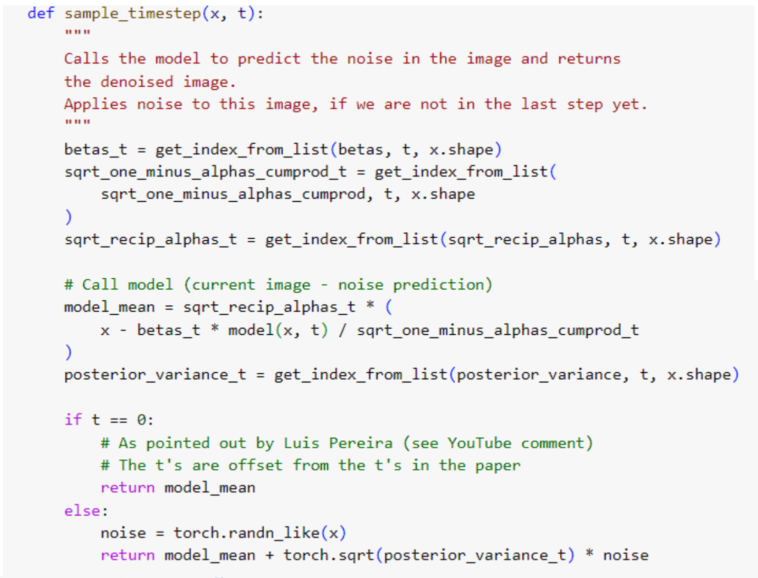
```python
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
        super().__init__()
        self.time_mlp =  nn.Linear(time_emb_dim, out_ch)
        if up:
            self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
        self.bnorm1 = nn.BatchNorm2d(out_ch)
        self.bnorm2 = nn.BatchNorm2d(out_ch)
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        return self.transform(h)
```



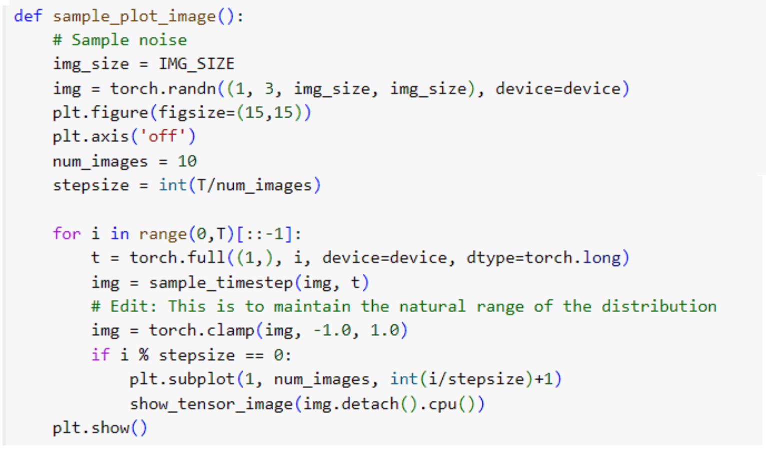
```python
class SimpleUnet(nn.Module):
    """
    A simplified variant of the Unet architecture.
    """
    def __init__(self):
        super().__init__()
        image_channels = 3
        down_channels = (64, 128, 256, 512, 1024)
        up_channels = (1024, 512, 256, 128, 64)
        out_dim = 3
        time_emb_dim = 32

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)
        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1], \
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])
        # Edit: Corrected a bug found by Jakub C (see YouTube comment)
    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)
```

3) **Implement time Step encoding:** Another important thing for the neural model is to identify which time step we are in because the model uses same shared weights for each input image. As discussed, the neural network has shareable weights across the time steps. That means, it can't distinguish between different time steps. For this, the researcher uses positional encodings as use in the Transformer paper. This is a clever way to encode discrete positional information. Positional embeddings are added as additional input besides the noisy image.

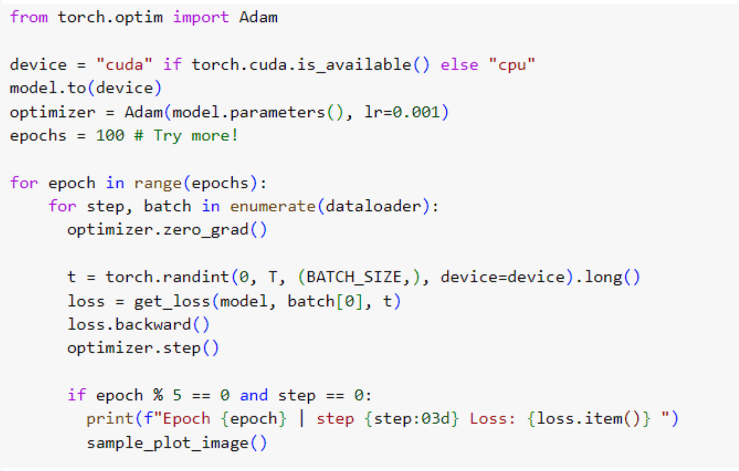
The overall training process is implemented as shown in the image below



```python
def sample_timestep(x, t):
    """
    Calls the model to predict the noise in the image and returns
    the denoised image.
    Applies noise to this image, if we are not in the last step yet.
    """
    betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x.shape
    )
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

    # Call model (current image - noise prediction)
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )
    posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)

    if t == 0:
        # As pointed out by Luis Pereira (see YouTube comment)
        # The t's are offset from the t's in the paper
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise
```



```python
def sample_plot_image():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
        img = sample_timestep(img, t)
        # Edit: This is to maintain the natural range of the distribution
        img = torch.clamp(img, -1.0, 1.0)
        if i % stepsize == 0:
            plt.subplot(1, num_images, int(i/stepsize)+1)
            show_tensor_image(img.detach().cpu())
    plt.show()
```



```python
from torch.optim import Adam

device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
optimizer = Adam(model.parameters(), lr=0.001)
epochs = 100 # Try more!

for epoch in range(epochs):
    for step, batch in enumerate(dataloader):
      optimizer.zero_grad()

      t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
      loss = get_loss(model, batch[0], t)
      loss.backward()
      optimizer.step()

      if epoch % 5 == 0 and step == 0:
        print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
        sample_plot_image()
```

Comparison with other Generative Models

Till now, we have studied two different architectures for generative modelling of data, i.e. GAN and VAE.

GAN	VAE
1) Produce high quality image 2) Difficult to train due to adversarial nature of training the model which causes problems like vanishing gradients and mode collapse.	1) Produce diverse samples quickly but quality of image is compromised. 2) Very easy to train as compared to GAN but output can be blurry.

Diffusion models is rather a new generative model that has shown to produce high quality image samples with diversity. This model has shown great success in text guided image generation. It works by destroying the input until noise is left over and then use the noise to recover the image back. However, diffusion model has its own downside, i.e. the sampling speed is much slower than GAN or VAE due to sequential reverse diffusion process.