Diffusion Transformer (DiT)

Zhe Li

In this note, we will explain how the Diffusion Transformer (DiT) works in this seminal paper[1]. We intend to explain it in two parts: the training stage and the inference stage.

1 The Inference Stage

Firstly we discuss how to generate images in inference stage, which consists the three components: diffusion process, transformer model, and VAE decoder. In the Fig 1, we present the entire process of how to generate 8 images given 8 labels. Firstly we generate noise tensor with size $8 * 4 * 32 * 32$, feed that into diffusion transformer with 8 labels, repeated this operation a number of times (so called number of diffusion steps). Top line is the stretch of diffusion process and we omit the details of how diffusion works in this overview diagram. After diffusion and transformer, we have $16 * 4 * 32 * 32$ tensors, and feed the first chunk of tensor $8 * 4 * 32 * 32$ to VAE decoder. Finally we obtain the 8 RGB images with size $8 * 3 * 256 * 256$. In the notes, we are talking about generating 8 images. Note that the following explanation and diagram is mainly based on the implementation [2].

Figure 1: The Process to Generate Images with DiT Given The Class Labels

1.1 Latent Diffusion Transformer

In the Fig 2, we present the overall structure of diffusion Transformer model. There are three inputs to transformer: noise with size $16 * 4 * 32 * 32$, labels with size 8 and time step t. For the noise part, note that in the implementation, generate noise $8 * 4 * 32 * 32$ tensor, and then concatenate those two noise tensor to have tensor $16 * 4 * 32 * 32$. Just

[1] https://arxiv.org/pdf/2212.09748
[2] https://github.com/facebookresearch/DiT?tab=readme-ov-file

make it clear this subtle difference with above section. Labels is a vector, for example $[207, 360, 387, 974, 88, 979, 417, 279]$, where 207 refers dog label. t refer to diffusion step, which is also feed into transformer.

For the noise tensor $16 * 4 * 32 * 32$ input, the first step is called patch embedding, which essentially is 2D-convolutional layer with kernel size $(2, 2)$ and stride $(2, 2)$, thus each channel is converted to $\frac{32}{2} * \frac{32}{2} = 256$ numbers. To encode positional information, there is the positional embedding table with size $256 * 1152$, in which each row 1152-dim vector is representing each position. we add patch embedding with positional embedding to generate $16 * 256 * 1152$ tensor as one source of inputs to DiT Block.

For label $[207, 360, 387, 974, 88, 979, 417, 279]$, in the implementation, this 8-dim vector is padded with another null labels to generate 16-dim vector as $[207, 360, 387, 974, 88, 979, 417, 279, 1000, 1000, \cdots, 1000]$. Convert this label vector to label embedding $16 * 1152$ by extracting 16 row from label embedding table with size $1001 * 1152$, here 1001 is the number of class (Imagenet 1000 class, one extra is null class, I guess) and 1152 is the dimension of label embedding.

For time step t, which is 16-dim vector, essentially is $[5, 5, \cdots, 5]$ if we set the total number of time step as 6 in the diffusion process. Convert this 16-dim t vector to time embedding $16 * 1152$ matrix. In the time embedding block, there is time embedding, fully connected layer, SiLu normalization layers.

Add the label embedding matrix and the time embedding matrix to obtain the matrix with size $16 * 1152$ as another source of inputs to feed into DiT Block.

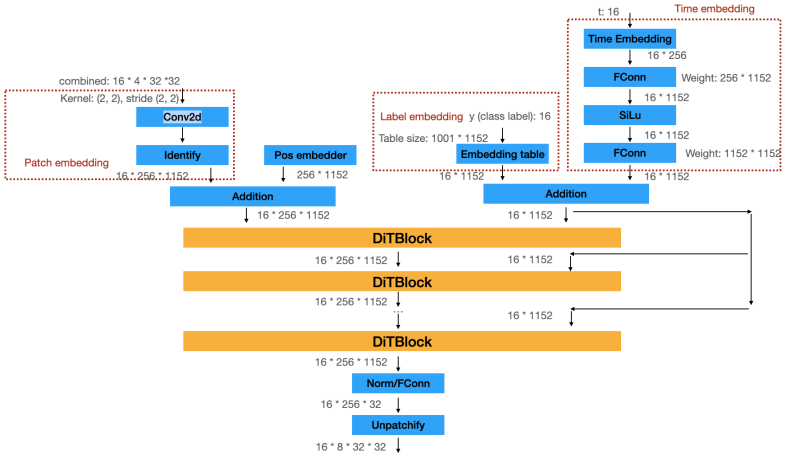

Figure 2: Latent Diffusion Transformer

In the Fig 3, we present what is insider of the DiT block and how those two sources of inputs are used. The top of the Fig 3 shows the inputs and outputs of DiT Block. The inputs to DiT Block are the addition of noise embedding and positional embedding with size of $16*256*1152$ and the addition of the label embedding and time embedding with size of $16*1152$. The output is a tensor with size of $16*256*1152$. To facilitate the following discussion, we denoted the addition of the noise embedding and position embedding as $\mathbf{x}$ and the addition of the label embedding and the time embedding as $\mathbf{o}$.

Inside of DiT block there are normalization layer, traditional multi-head attention layer, fully connected layer, GELU normalization layer. One novelty inside of this block is so called adaptive layer normalization. Firstly apply SiLu normalization and fully connected on embedding $\mathbf{o}$ to obtain the large matrix with size $16*6912$, then chunk this large matrix to six parts and each part with size $16*1152$, divided those six parts into two groups msa and mlp. Inside each group three small matrix with size $16*1152$ are corresponding shift, scale and gate. The following roughly capture how shift, scale and gate are used.

$$\mathbf{x} = \mathbf{x} + \text{gate} * \text{attn}(\text{modulate}(\text{scale} * \mathbf{x} + \text{shift})) \tag{1}$$

Here modulate is just some function.

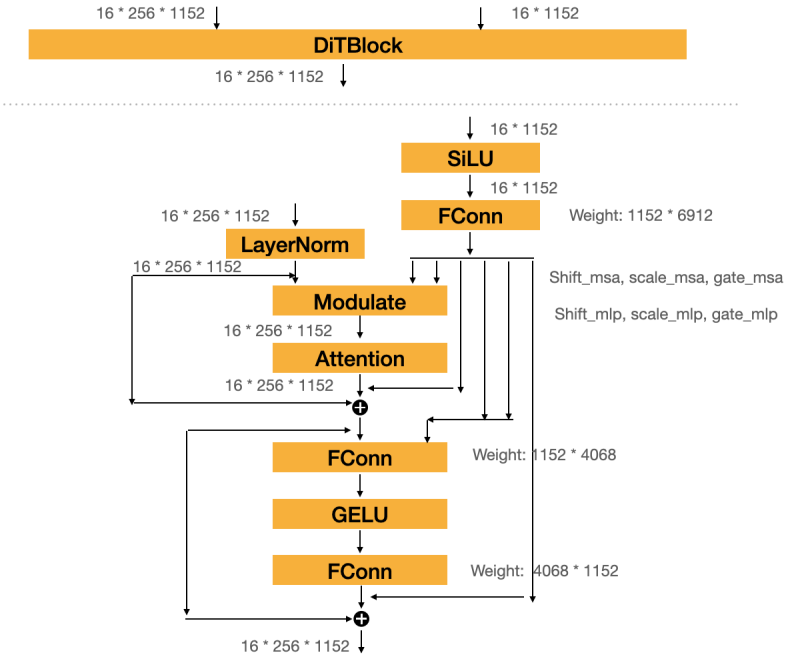

Figure 3: DiT Block

After the 28 repeated DiT block, there are normalization layer, fully connected layer, and unpatchify layer. The final output of diffusion transformer is $16 * 32 * 32$ size.

1.2 Backward Diffusion Process

Following the previous section, we extract one transformer block as the Fig 4

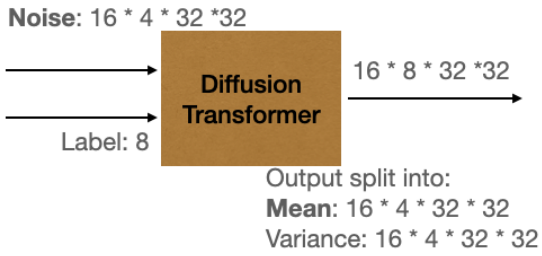

Figure 4: Transformer Block

The input are noise with size $16 * 4 * 32 * 32$ and label with size 8 and the output is the size $16 * 8 * 32 * 32$, which consists of two parts: the mean $8 * 4 * 32 * 32$ and the variance $8 * 4 * 32 * 32$. At this point, let's denote the noise as $\mathbf{x}_t$ and the mean as ϵ.

- Compute $\mathbf{x}_0$, based on $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$, we have

$$\mathbf{x}_0 = \sqrt{\frac{1}{\bar{\alpha}_t}}\mathbf{x}_t - \sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}} * \epsilon \tag{2}$$

- Compute $\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0)$ and $\tilde{\beta}_t$ based on the $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t I)$ where

$$\begin{aligned} \tilde{\beta}_t &= \beta_t * \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \\ \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) &= \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t}\mathbf{x}_t \end{aligned} \tag{3}$$

  Actually if we plug $\mathbf{x}_0 = \sqrt{\frac{1}{\bar{\alpha}_t}}\mathbf{x}_t - \sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}} * \epsilon$ to the $\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0)$, which yields

$$\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{1}{\bar{\alpha}_t}(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon) \tag{4}$$

  In the implementation, the authors computed $\mathbf{x}_0$ first and then compute $\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0)$, maybe because they would like to re-use some functions.

- Output sample based on t value

$$
\begin{aligned}
t = 0, \text{sample} &= \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) \\
t \neq 0, \text{sample} &= \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) + \tilde{\beta}_t * \text{noise}
\end{aligned} \tag{5}
$$

For the variance part, there are different strategies on how to compute variance either fixed or learned from models[3].

So far we roughly present the structure of how to generate an image based on diffusion process if we can decode the sample $16 * 4 * 32 * 32$ through VAE decode, which will be discussed in the next section.

We include the Fig 5 to detail understand how the diffusion process is exactly implemented in the code base[4].

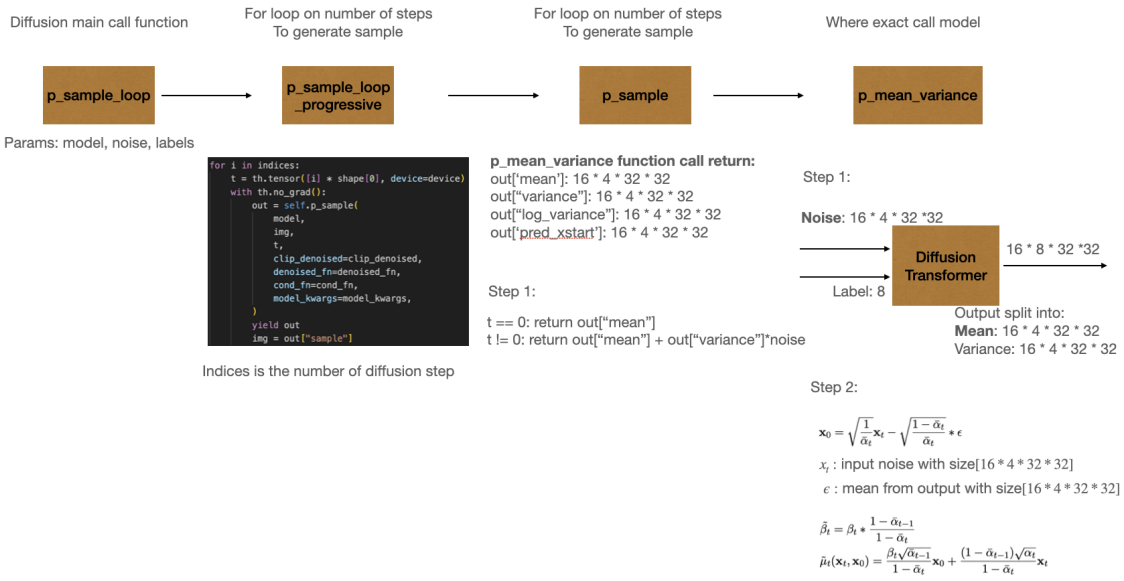

Figure 5: Diffusion Process in Implementation

We include some code explanation related the above process in the Appendix section 3.

[3] Improved Denoising Diffusion Probabilistic Models
[4] https://github.com/facebookresearch/DiT?tab=readme-ov-file

1.3 VAE Decoder

In the Fig 6, we present the entire network structure of VAE decoder model. After diffusion transformer stage, we have the de-noised (terminology in diffusion model) tensor with size of $16 * 8 * 32 * 32$. We ignore the half part of that matrix and feed the another half of matrix $8 * 4 * 32 * 32$ to VAE decoder model [5]. The first step is Post quantization convolutional layer, which I guess conducts operation on value level. After that, there is 2D convolutional layer, Unet Middle block, four Up Decoder Block, group normalization layer, SiLu, and convolutional layer. The final output of this network is 8 RGB images, that is, a tensor with size $8 * 3 * 256 * 256$.

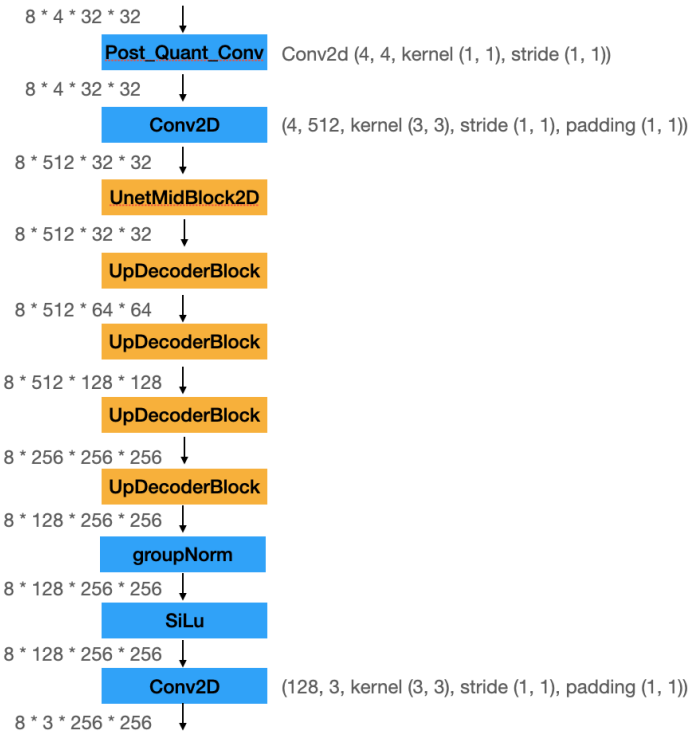

Figure 6: VAE Decoder Model

The Fig 7 shows the block structure for Unet Middle Block, which consists of some conventional layers such as group normalization layer, convolutional layer, attention layer and others. In this block, as seen the dimension of the tensor remains the same.

[5] https://huggingface.co/stabilityai/sd-vae-ft-mse

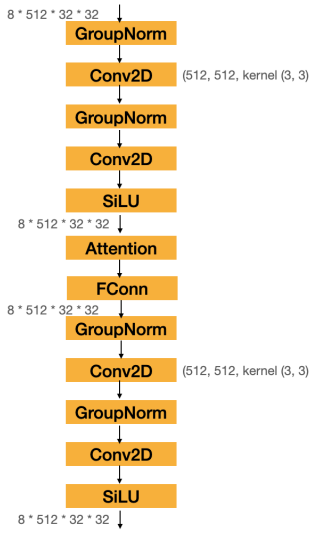

Figure 7: Unet Middle Block

The Fig 8 shows the block structure of Up Decoder Block, similarly which consists of some conventional layers and up sampling layer. One thing to note is that in this block spatial dimension is two time scaled up. In the implementation, up sampling layer is essentially using torch interpolate function with mode as nearest. The following is one of up sampling examples on the input $2 * 2$ matrix with the scale factor 2 and the mode being nearest.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \tag{6}$$

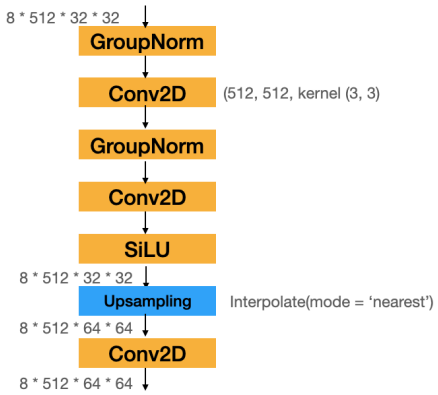

Figure 8: Up Decoder Block

2 The Training Stage

Assume batch size as 16, sample 16 images and their labels and conduct some preprocessing operation, we have images tensor with size $(16 * 3 * 256 * 256)$ and labels tensor with size (16). Firstly we encode those images through VAE Encode model to get encoded tensor with size $(16 * 4 * 32 * 32)$ as shown in the Fig **??**, we denoted this encoded tensor as $\mathbf{x}_0$ and label tensor as y. At each training iteration, we randomly draw timesteps t from $[0, \text{num\_timestep}]$, where num_timesteps is hype-parameter of diffusion model and t is size of 16 corresponding to 16 images. Note that the loss function is mean square error (MSE) type and variance is learned through model.

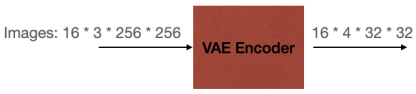

Figure 9: Encode Image through VAE Encode Model

- Step 1: Compute $\mathbf{x}_t$ from $\mathbf{x}_0$, t and noise based on q_sample$(\mathbf{x}_0, t)$, which is

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} * \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} * \text{noise} \tag{7}$$

- Step 2: Feed $\mathbf{x}_t, t, y$ to Transformer model to get model_output tensor with size $(16 * 8 * 32 * 32)$, which consists two parts the mean (also known as ϵ_t) and the variance.

- Step 3: Compute loss, the loss could have two parts L_{mse}

$$L_{mse} = (\epsilon_t - \text{noise})^2 \tag{8}$$

where noise is as same as used in the Step 1 and the Eq. 7. ϵ_t and noise both are size of $(16 * 4 * 32 * 32)$. We can compute mean of L_{mse} for each image, then finally mse loss will be 16-dim vector. The other part of loss is based on variational lower bound, which will be updated soon.

The Fig 10 shows the first two steps to compute loss in the training stage.

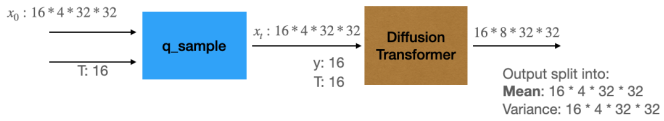

Figure 10: The First Two Steps To Compute Loss In Training

3 Appendix

3.1 Implementation notes for diffusion process

From the previous notes, we have

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} * \epsilon \tag{9}$$

that is,

$$\mathbf{x}_0 = \sqrt{\frac{1}{\bar{\alpha}_t}}\mathbf{x}_t - \sqrt{\frac{1 - \bar{\alpha}_t}{\bar{\alpha}_t}} * \epsilon \tag{10}$$

```
def _predict_xstart_from_eps(self, x_t, t, eps):
    assert x_t.shape == eps.shape
    return (
        _extract_into_tensor(self.sqrt_recip_alphas_cumprod, t, x_t.
                                                shape) * x_t
        - _extract_into_tensor(self.sqrt_recipm1_alphas_cumprod, t, x_t
                                                .shape) * eps
    )
```

Given $\mathbf{x}_t$ and $\mathbf{x}_0$, we know $\mathbf{x}_{t-1}$ is from the following posterior distribution

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t I) \tag{11}$$

where $\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0)$ and $\tilde{\beta}_t$ can be computed as

$$\begin{aligned} \tilde{\beta}_t &= \beta_t * \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \\ \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) &= \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t}\mathbf{x}_t \end{aligned} \tag{12}$$

```
# calculations for posterior q(x_{t-1} | x_t, x_0)
self.posterior_variance = (
            betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.
                                                    alphas_cumprod)
       )

self.posterior_mean_coef1 = (
            betas * np.sqrt(self.alphas_cumprod_prev) / (1.0 - self.
                                                    alphas_cumprod)
        )
self.posterior_mean_coef2 = (
            (1.0 - self.alphas_cumprod_prev) * np.sqrt(alphas) / (1.0 -
                                                    self.alphas_cumprod)
        )
 def q_posterior_mean_variance(self, x_start, x_t, t):
        """
        Compute the mean and variance of the diffusion posterior:
            q(x_{t-1} | x_t, x_0)
        """
        assert x_start.shape == x_t.shape
        posterior_mean = (
            _extract_into_tensor(self.posterior_mean_coef1, t, x_t.shape) *
                                                     x_start
            + _extract_into_tensor(self.posterior_mean_coef2, t, x_t.shape)
                                                     * x_t
        )
        posterior_variance = _extract_into_tensor(self.posterior_variance,
                                                t, x_t.shape)
        posterior_log_variance_clipped = _extract_into_tensor(
            self.posterior_log_variance_clipped, t, x_t.shape
        )
        assert (
            posterior_mean.shape[0]
            == posterior_variance.shape[0]
            == posterior_log_variance_clipped.shape[0]
            == x_start.shape[0]
        )
        return posterior_mean, posterior_variance,
                                                posterior_log_variance_clipped
```

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \tag{13}$$

```
    def q_mean_variance(self, x_start, t):
        """
        Get the distribution q(x_t | x_0).
        :param x_start: the [N x C x ...] tensor of noiseless inputs.
        :param t: the number of diffusion steps (minus 1). Here, 0 means
                                                one step.
        :return: A tuple (mean, variance, log_variance), all of x_start's
                                                shape.
        """
        mean = _extract_into_tensor(self.sqrt_alphas_cumprod, t, x_start.
                                                shape) * x_start
        variance = _extract_into_tensor(1.0 - self.alphas_cumprod, t,
                                                x_start.shape)
        log_variance = _extract_into_tensor(self.
                                                log_one_minus_alphas_cumprod,
                                                 t, x_start.shape)
        return mean, variance, log_variance
```

The above code returns the mean, variance and log_variance of $\mathbf{x}_t$. The following q_sample function returns $\mathbf{x}_t$.

```
    def q_sample(self, x_start, t, noise=None):
        """
        Diffuse the data for a given number of diffusion steps.
        In other words, sample from q(x_t | x_0).
        :param x_start: the initial data batch.
        :param t: the number of diffusion steps (minus 1). Here, 0 means
                                                one step.
        :param noise: if specified, the split-out normal noise.
        :return: A noisy version of x_start.
        """
        if noise is None:
            noise = th.randn_like(x_start)
        assert noise.shape == x_start.shape
        return (
            _extract_into_tensor(self.sqrt_alphas_cumprod, t, x_start.shape
                                                    ) * x_start
            + _extract_into_tensor(self.sqrt_one_minus_alphas_cumprod, t,
                                                    x_start.shape) * noise
        )
```

In the above code, x_start is $\mathbf{x}_0$ in the Eq.13. _extract_into_tensor function extract t^{th} element from self.sqrt_alphas_cumprod and broadcast as the same shape as x_start, then element-wise product with x_start.

KL divergence: assume two normal distribution P and Q where

$$\begin{aligned} P : x &\sim \mathcal{N}(\mu_1, \sigma_1^2) \\ Q : x &\sim \mathcal{N}(\mu_2, \sigma_2^2) \end{aligned} \tag{14}$$

then [6].

$$KL[P||Q] = \frac{1}{2}(-1 + \log(\frac{\sigma_2^2}{\sigma_1^2}) + \frac{\sigma_1^2}{\sigma_1^2} + \frac{(\mu_2 - \mu_1)^2}{\sigma_2^2}) \tag{15}$$

See the following code to compute KL divergence

```
def normal_kl(mean1, logvar1, mean2, logvar2):
    """
    Compute the KL divergence between two gaussians.
    Shapes are automatically broadcasted, so batches can be compared to
    scalars, among other use cases.
    """
    return 0.5 * (
        -1.0
        + logvar2
        - logvar1
        + th.exp(logvar1 - logvar2)
        + ((mean1 - mean2) ** 2) * th.exp(-logvar2)
    )
```

[6] https://statproofbook.github.io/P/norm-kl.html