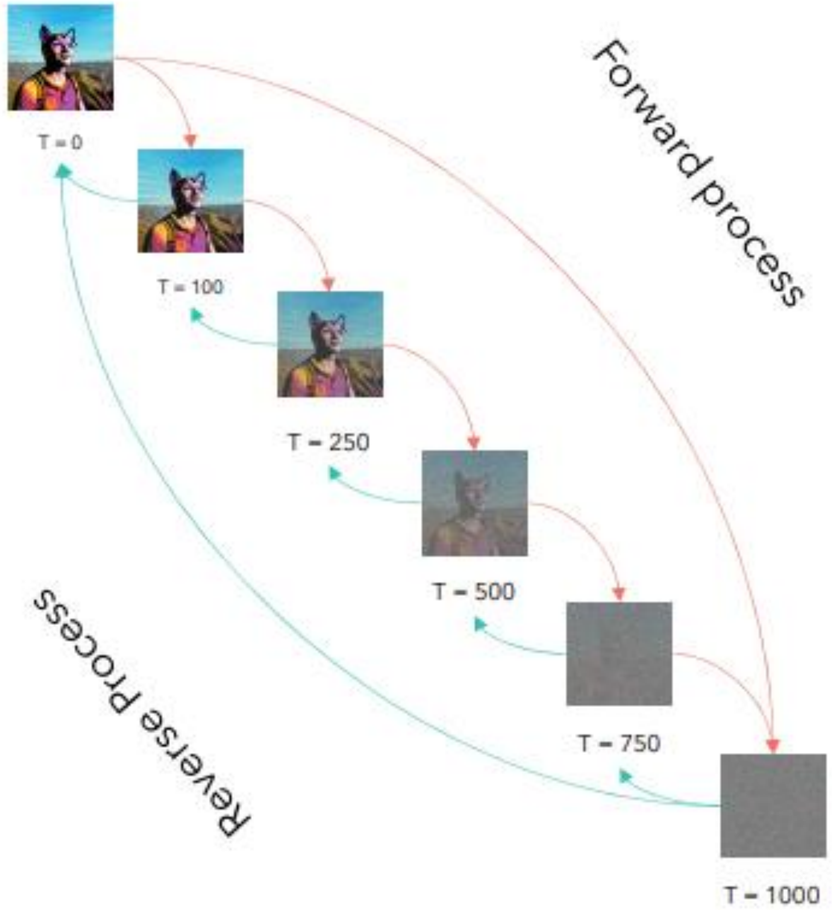
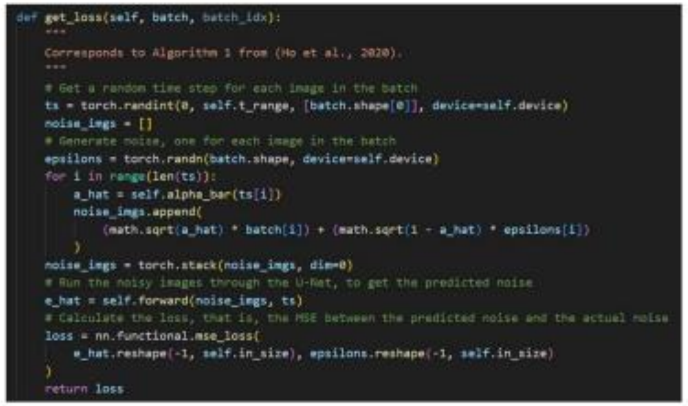
UNIVERSIDAD EAFIT

Escuela de
Ciencias Aplicadas
e Ingeniería



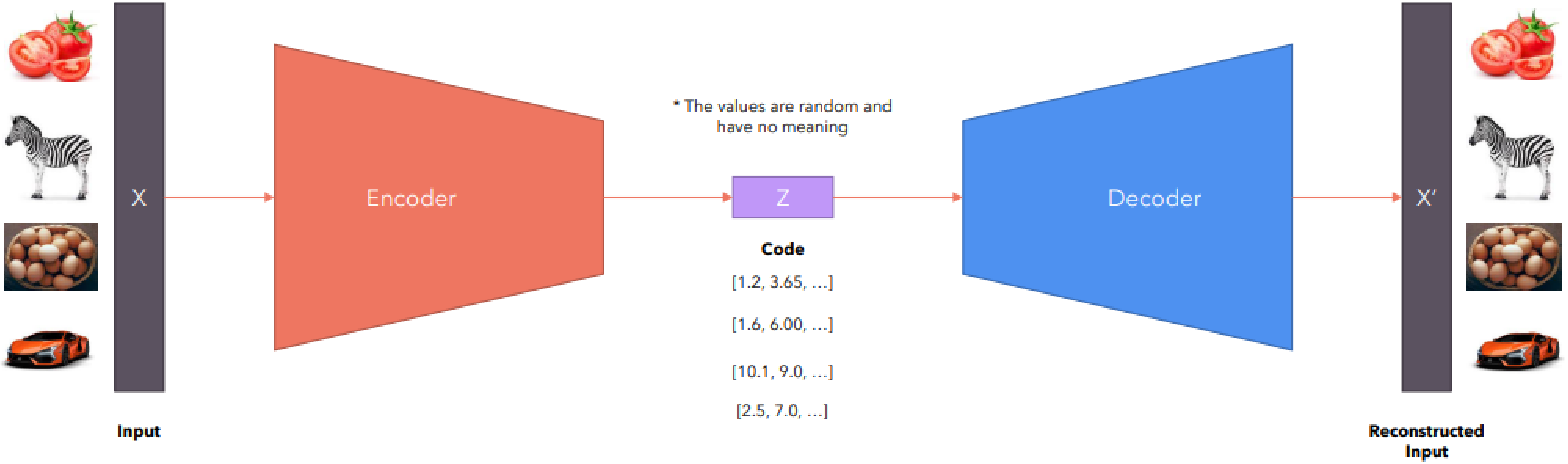


```python
def get_loss(self, batch, batch_idx):
    """
    Corresponds to Algorithm 1 from (Ho et al., 2020).
    """
    # Get a random time step for each image in the batch
    ts = torch.randint(0, self.t_range, [batch.shape[0]], device=self.device)
    noise_imgs = []
    # Generate noise, one for each image in the batch
    epsilons = torch.randn(batch.shape, device=self.device)
    for i in range(len(ts)):
        a_hat = self.alpha_bar(ts[i])
        noise_imgs.append(
            (math.sqrt(a_hat) * batch[i]) + (math.sqrt(1 - a_hat) * epsilons[i])
        )
    noise_imgs = torch.stack(noise_imgs, dim=0)
    # Run the noisy images through the U-Net, to get the predicted noise
    e_hat = self.forward(noise_imgs, ts)
    # Calculate the loss, that is, the MSE between the predicted noise and the actual noise
    loss = nn.functional.mse_loss(
        e_hat.reshape(-1, self.in_size), epsilons.reshape(-1, self.in_size)
    )
    return loss
```

Denoising Diffusion Probabilistic Models (DDPM)

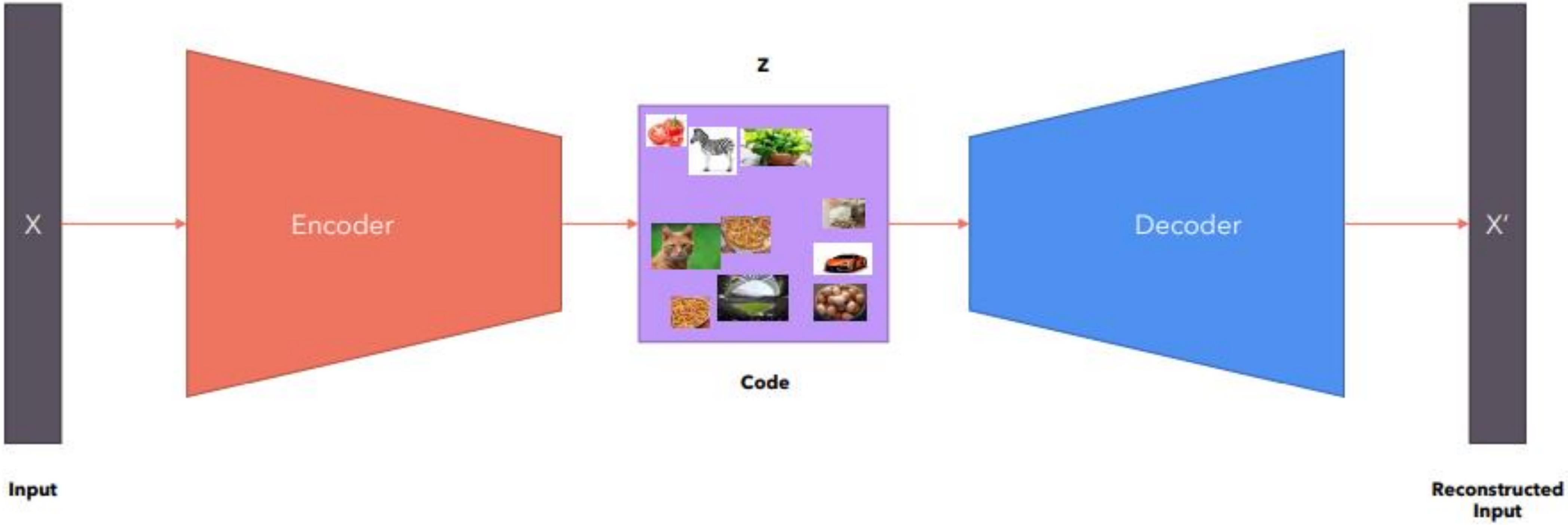
What is an Autoencoder?

UNIVERSIDAD EAFIT
Escuela de Ciencias Aplicadas e Ingeniería



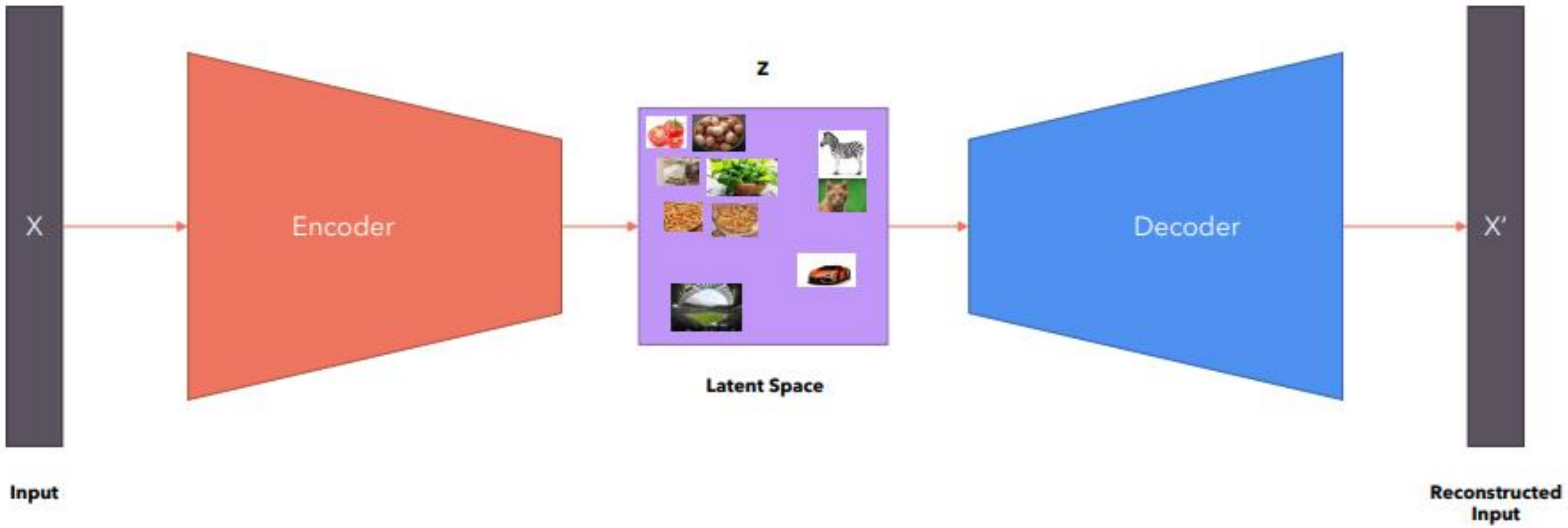
What’s the problem with Autoencoders??

The code learned by the model **makes no sense**. That is, the model can just assign any vector to the inputs without the numbers in the vector representing any pattern. The model doesn't capture any **semantic relationship** between the data.

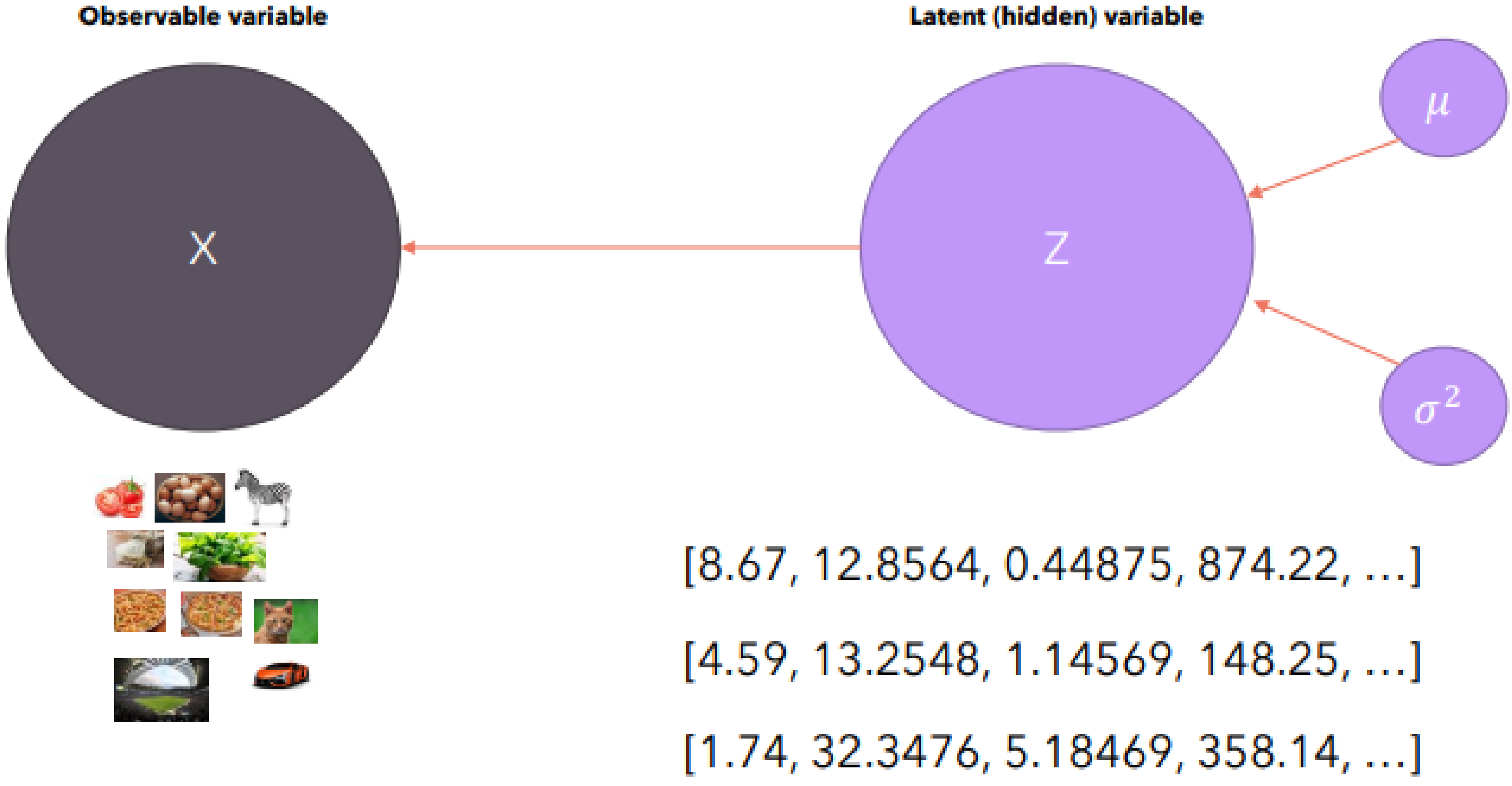


Introducing the Variational Autoencoder

The variational autoencoder, instead of learning a code, learns a "**latent space**". The latent space represents the parameters of a (multivariate) distribution.



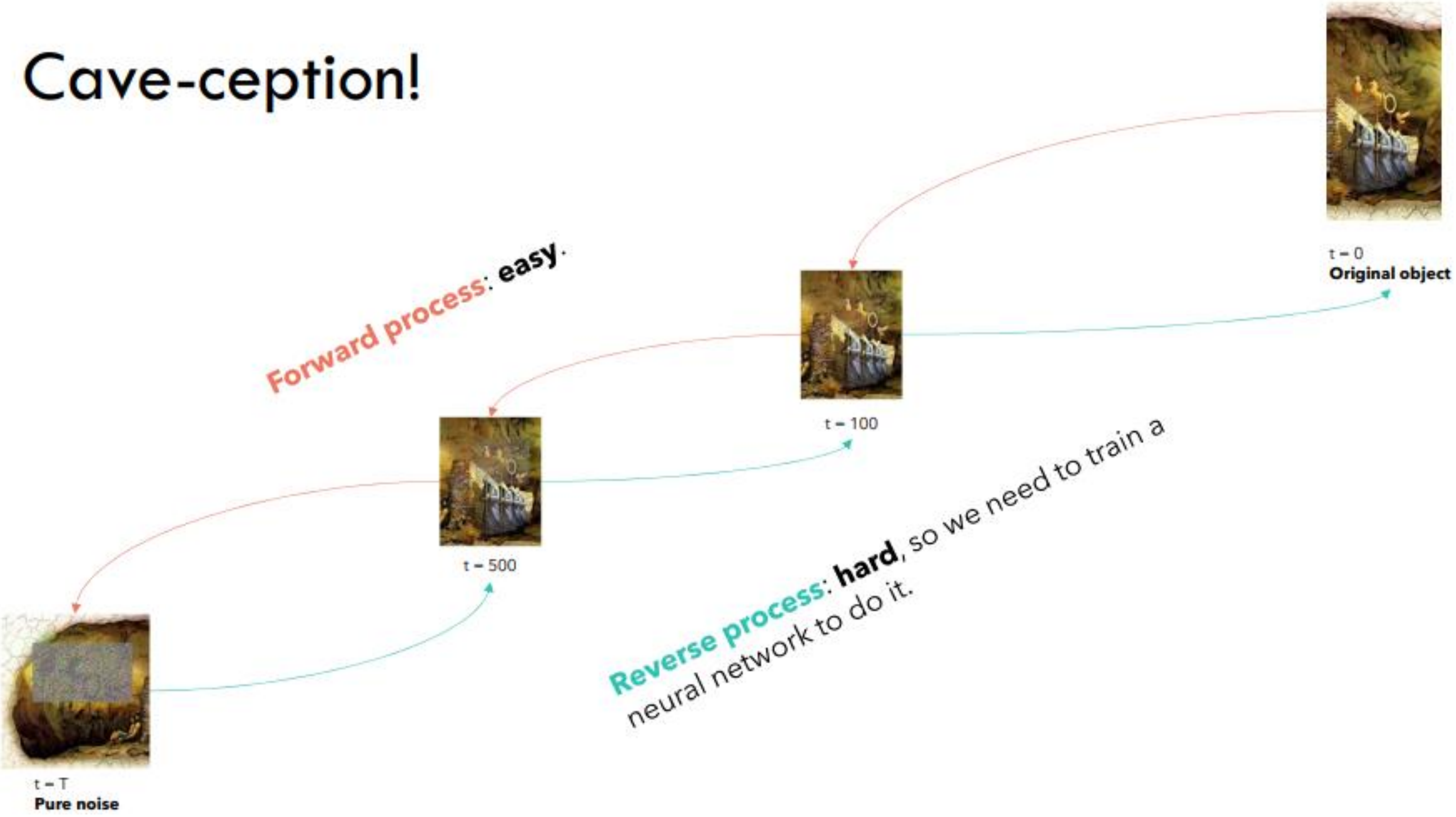
Why is it called latent space?

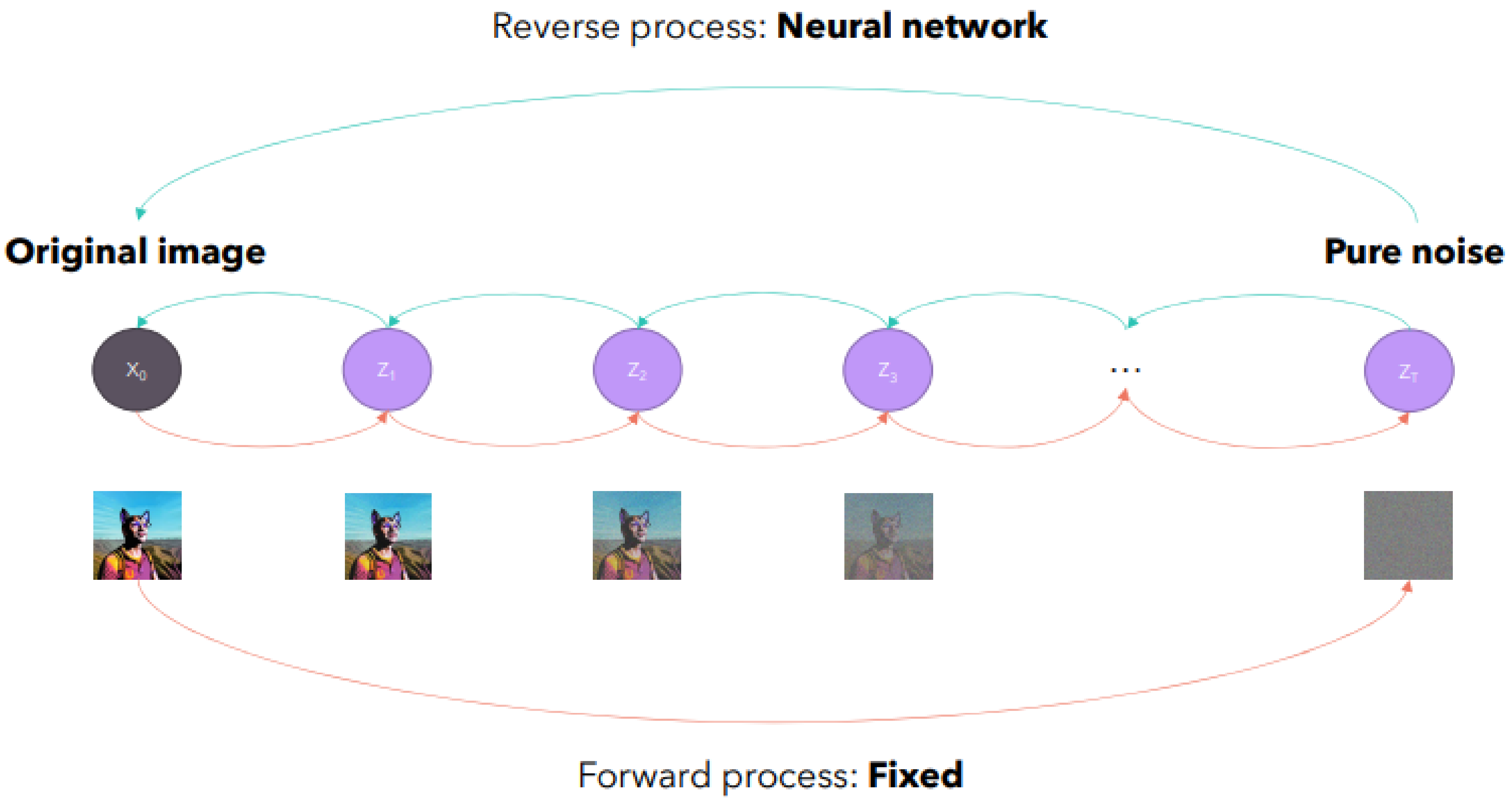


Plato’s allegory of the cave


WE NEED TO GO
DEEPER

Cave-ception!




Reverse process: **Neural network**
Original image
Pure noise
x_0
z_1
z_2
z_3
...
z_T
Forward process: **Fixed**

Let’s have fun with… math!

UNIVERSIDAD EAFIT
Escuela de Ciencias Aplicadas e Ingeniería

Just like with a VAE, we want to learn the parameters of the latent space

Reverse process **p**

2 Background

Diffusion models [53] are latent variable models of the form $p_\theta(\mathbf{x}_0) := \int p_\theta(\mathbf{x}_{0:T})\, d\mathbf{x}_{1:T}$, where $\mathbf{x}_1, \ldots, \mathbf{x}_T$ are latents of the same dimensionality as the data $\mathbf{x}_0 \sim q(\mathbf{x}_0)$. The joint distribution $p_\theta(\mathbf{x}_{0:T})$ is called the *reverse process*, and it is defined as a Markov chain with learned Gaussian transitions starting at $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$:

$$p_\theta(\mathbf{x}_{0:T}) := p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t), \qquad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) := \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)) \tag{1}$$

What distinguishes diffusion models from other types of latent variable models is that the approximate posterior $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$, called the *forward process* or *diffusion process*, is fixed to a Markov chain that gradually adds Gaussian noise to the data according to a variance schedule $\beta_1, \ldots, \beta_T$:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) := \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}), \qquad q(\mathbf{x}_t|\mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{2}$$

Evidence Lower Bound **(ELBO)**

Training is performed by optimizing the usual variational bound on negative log likelihood:

$$\mathbb{E}\left[-\log p_\theta(\mathbf{x}_0)\right] \le \mathbb{E}_q\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] = \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t\ge1} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right] =: L \tag{3}$$

The forward process variances β_t can be learned by reparameterization [33] or held constant as hyperparameters, and expressiveness of the reverse process is ensured in part by the choice of Gaussian conditionals in $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$, because both processes have the same functional form when β_t are small [53]. A notable property of the forward process is that it admits sampling $\mathbf{x}_t$ at an arbitrary timestep t in closed form: using the notation $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$, we have

Forward process **q**

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{4}$$

Ho, J., Jain, A. and Abbeel, P., 2020. Denoising diffusion probabilistic models. Advances in Neural Information Processing Systems, 33, pp.6840-6851.

How to derive the loss function?

1. We start by writing our objective: we want to maximize the log likelihood of our data, $\log(p_\theta(x_0))$, marginalizing over all other latent variables.
2. We find a lower bound for the log likelihood, that is, $\log\left(p_\theta(x_0)\right) \geq ELBO$
3. We maximize the $ELBO$ (or minimize the negated term).

Algorithm 1 Training

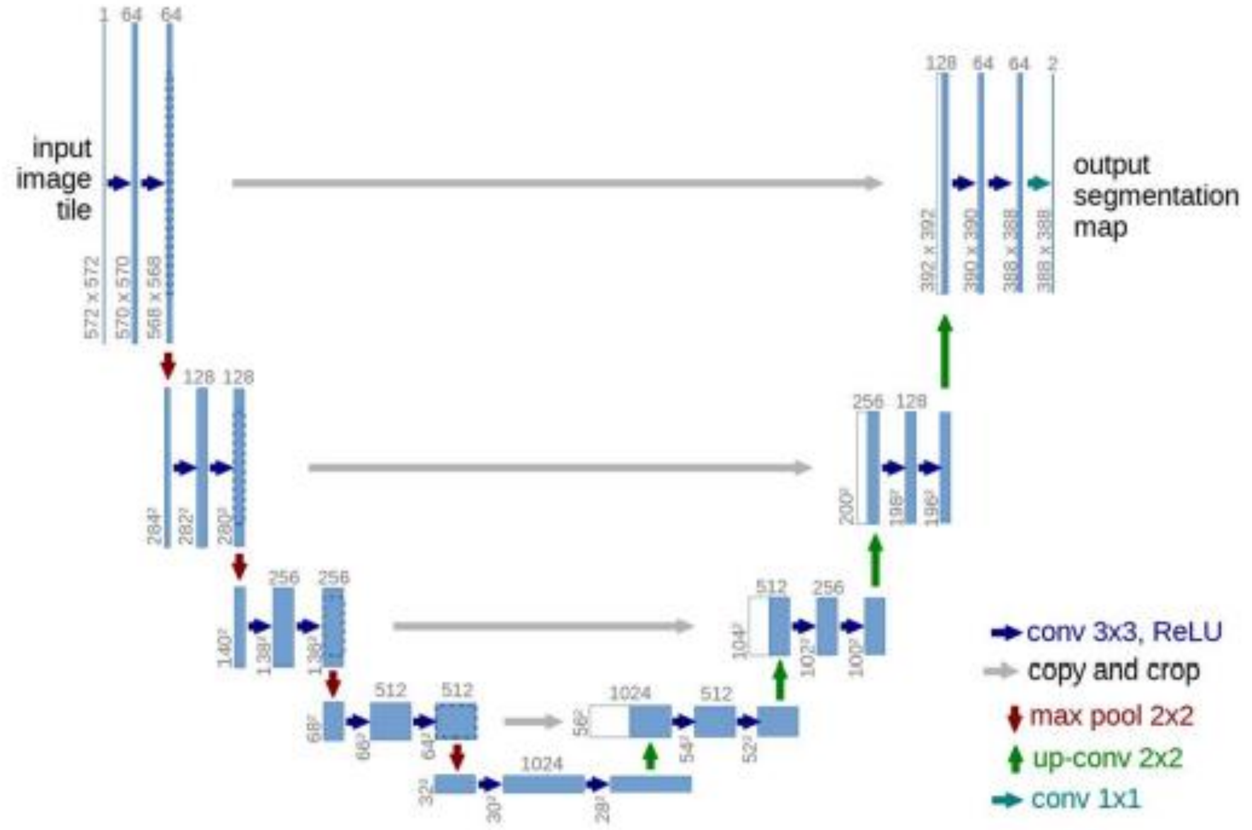
repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ We take a sample from our dataset
  $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$ We generate a random number t, between 1 and T
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ We sample some noise
  Take gradient descent step on
  $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$ We add noise to our image, and we train the model to learn to predict the amount of noise present in it.
until converged

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ We sample some noise
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

We keep denoising the image progressively for T steps.

U-Net



Ronneberger, O., Fischer, P. and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18* (pp. 234-241). Springer International Publishing.

Training code

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
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

```python
def get_loss(self, batch, batch_idx):
    """
    Corresponds to Algorithm 1 from (Ho et al., 2020).
    """
    # Get a random time step for each image in the batch
    ts = torch.randint(0, self.t_range, [batch.shape[0]], device=self.device)
    noise_imgs = []
    # Generate noise, one for each image in the batch
    epsilons = torch.randn(batch.shape, device=self.device)
    for i in range(len(ts)):
        a_hat = self.alpha_bar(ts[i])
        noise_imgs.append(
            (math.sqrt(a_hat) * batch[i]) + (math.sqrt(1 - a_hat) * epsilons[i])
        )
    noise_imgs = torch.stack(noise_imgs, dim=0)
    # Run the noisy images through the U-Net, to get the predicted noise
    e_hat = self.forward(noise_imgs, ts)
    # Calculate the loss, that is, the MSE between the predicted noise and the actual noise
    loss = nn.functional.mse_loss(
        e_hat.reshape(-1, self.in_size), epsilons.reshape(-1, self.in_size)
    )
    return loss
```

Sampling code

Algorithm 2 Sampling

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

```python
def denoise_sample(self, x, t):
    """
    Corresponds to the inner loop of Algorithm 2 from (Ho et al., 2020).
    """
    with torch.no_grad():
        if t > 1:
            z = torch.randn(x.shape)
        else:
            z = 0
        # Get the predicted noise from the U-Net
        e_hat = self.forward(x, t.view(1).repeat(x.shape[0]))
        # Perform the denoising step to take the image from t to t-1
        pre_scale = 1 / math.sqrt(self.alpha(t))
        e_scale = (1 - self.alpha(t)) / math.sqrt(1 - self.alpha_bar(t))
        post_sigma = math.sqrt(self.beta(t)) * z
        x = pre_scale * (x - e_scale * e_hat) + post_sigma
        return x
```

UNIVERSIDAD EAFIT | Escuela de Ciencias Aplicadas e Ingeniería

The full code is available on GitHub!

Full code: https://github.com/hkproj/pytorch-ddpm

Thanks!