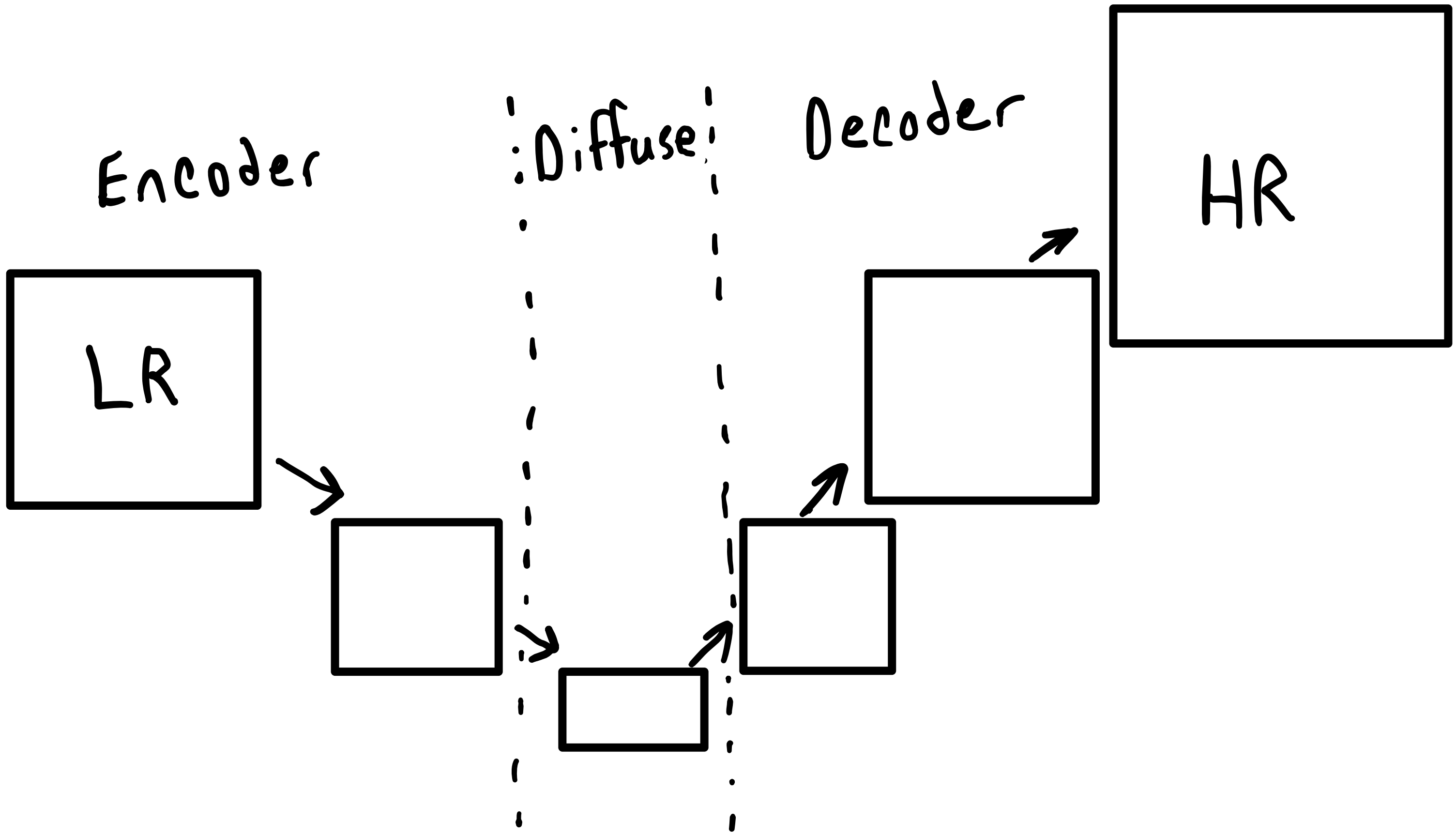
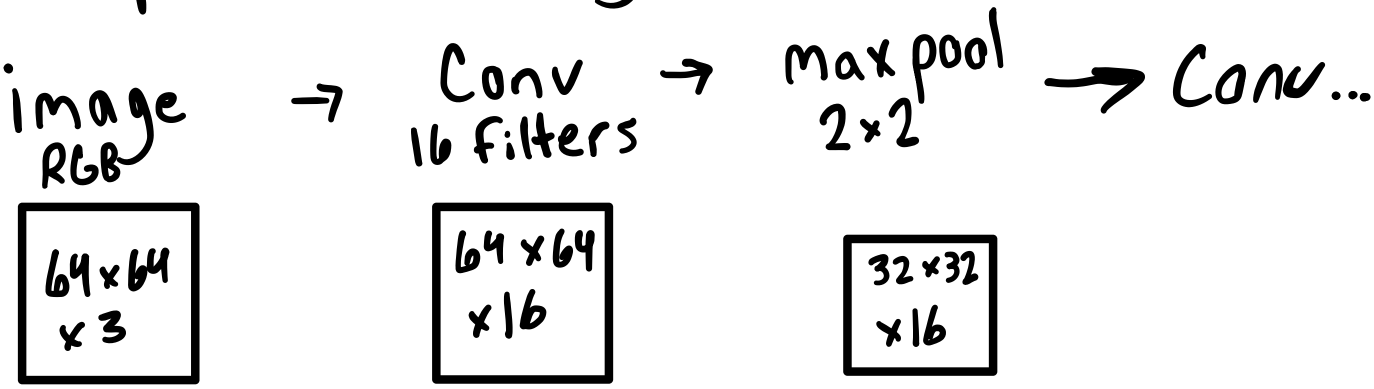
Latent Diffusion



Applies noise to a lower dimensional latent space instead of directly to the image. The latent space will capture the essential features and structure of the image while allowing the computationally intensive diffusion process to occur in a smaller space - yielding higher efficiency.

Encoder

A series of Convolutional and pooling layers that Compress the image.



Diffusion

Gradually applies noise to the latent space and then attempts to learn how to reverse the process. When training, the HR image Will be Converted into a latent Space and noise Will be added to make it resemble its LR Counterpart. We Will encode the LR image to the Same latent space and use it as the guiding Condition for noise addition.

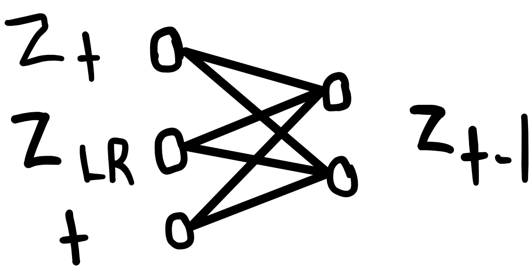
Forward Diffusion

High Resolution $\xrightarrow{\text{Encode}}$ Latent space $\boxed{X_0}$ · Noise = $\boxed{X_t}$ · Noise ...

Low Res $\xrightarrow{\text{Encode}}$ $\boxed{X_1}$ = $\boxed{X_1}$

$$\boxed{(X_t \mid X_0 X_1) = \mathcal{N}(X_t; \mu(X_0 + X_1); \Sigma_t)}$$

Reverse Diffusion

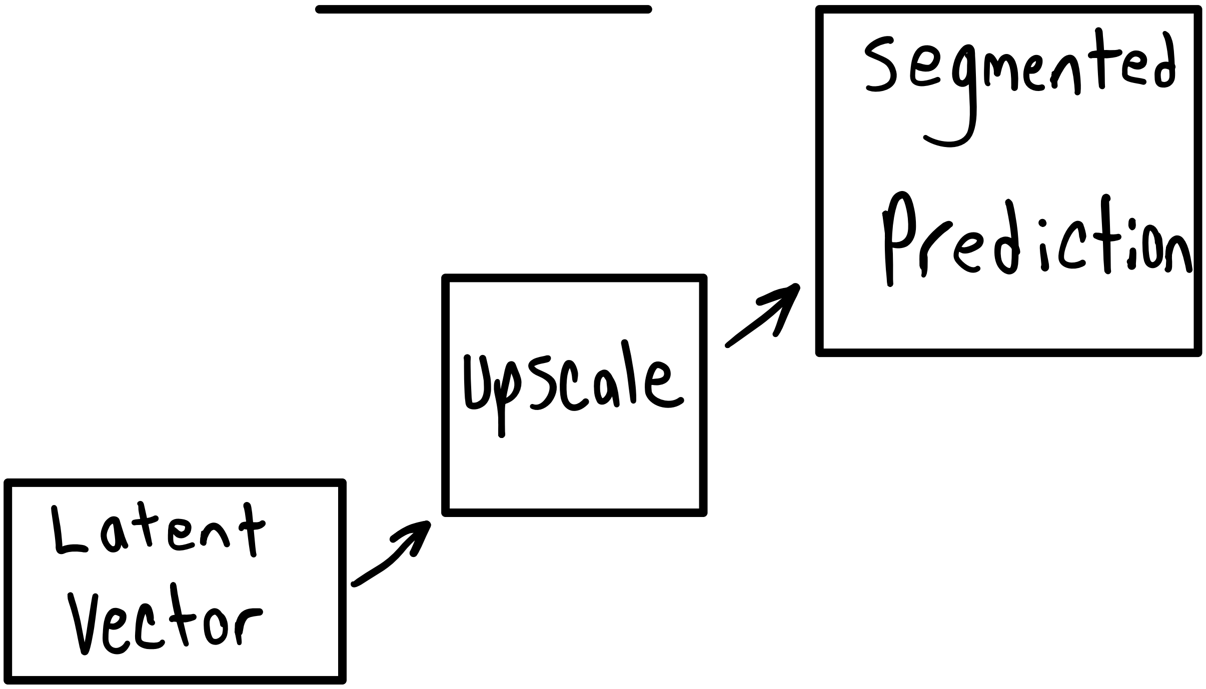


$$\boxed{Z_{t-1} = NN(Z_t; Z_{LR}; t)}$$

Decoder

Instead of generating an image from the latent vector, we can decode it directly into a semantic segmentation prediction. Our Classifier has a similar decoder that is trained on an extracted Feature map that is equivalent to the latent vector. It uses a Series of Transposed Convolution layers with upsampling to decompress the latent vector / Feature map into the desired resolution.

Decode



The encoder and decoder can be taken directly from the UNet architecture of our Classifier. As such, our diffusion model is essentially a UNet with diffusion applied at the center.

I^2SB

Forward Diffusion / q_sample:

In order to create a bridge between the initial state X_0 and final state X_1, intermediate states must be calculated.

For any timestep t, X_t can be calculated as a function of X_0 and X_1 $(X_t | X_0, X_1)$ if we know the accumulated variance between X_0, X_t (σ_t^2) and X_t, X_1 $(\bar{\sigma}_t^2)$. Where

$$\sigma_t^2 = \int_0^t \beta_\tau \, d\tau \quad , \quad \bar{\sigma}_t^2 = \int_t^1 \beta_\tau \, d\tau .$$

So, if we know the noise schedule (β), we can find how much noise was added between each timestep. Then we can find X_t:

The general form of the forward process:

$$q(x_t | x_0) = \mathcal{N}(x_t; \mu_t \cdot x_0, \Sigma_t)$$

Where $q(x_t | x_0)$ is a normal ($\mathcal{N}$) probability distribution whose mean (μ_t) is $\sqrt{1-\beta_t} \cdot x_0$ and covariance (Σ_t) is $\sigma_t^2 \cdot$ Identity matrix (I).

q_sample is simply sampling from the $q(x_t)$ probability distribution.

I^2SB calculates the mean and covariance as:

$$\mu_t = \frac{\bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} x_0 + \frac{\sigma_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} x_1, \quad \Sigma_t = \frac{\sigma_t^2 \cdot \bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} \cdot \text{Identity Matrix}$$

(mean) (Covariance)

$$\sigma_t = \int_0^t \beta_T \, dT \quad \bar{\sigma}_t = \int_t^1 \beta_T \, dT$$

The mean represents the expected value of the distribution.

The covariance is a measure of how much the changes applied to x_0 relate to the changes applied to x_1. We want this to be positive, because that means the changes are related and the process is upscaling to high resolution efficiently.

In the case of I^2SB, the general form is altered due to the reliance on both the initial and final state (X_0, X_1) of the diffusion process instead of only the initial (X_0), giving us:

$$q(X_t | X_0, X_1) = \mathcal{N}(X_t; \mu_{t0} X_0 + \mu_{t1} X_1; \Sigma_t)$$

$$= \mathcal{N}\left(X_t; \frac{\bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} X_0 + \frac{\sigma_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} X_1; \frac{\sigma_t^2 \cdot \bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} \cdot I\right)$$

To perform the forward process, we first need to calculate σ and $\bar{\sigma}$ for every timestep by using our β schedule.

$$\sigma_t = \int_0^t \beta_\tau \, d\tau$$

```
std_fwd = np.sqrt(np.cumsum(betas))
```

$$\bar{\sigma}_t = \int_t^1 \beta_\tau \, d\tau$$

```
std_bwd = np.sqrt(np.flip(np.cumsum(np.flip(betas))))
```

Then we can compute:

$$\frac{\bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2}; \quad \frac{\sigma_t^2}{\bar{\sigma}_t^2 + \sigma_t^2}; \quad \frac{\sigma_t^2 \cdot \bar{\sigma}_t^2}{\bar{\sigma}_t^2 + \sigma_t^2} \quad \text{given } \sigma, \bar{\sigma}$$

```python
def compute_gaussian_product_coef(sigma1, sigma2):
    """ Given p1 = N(x_t|x_0, sigma_1**2) and p2 = N(x_t|x_1, sigma_2**2)
        return p1 * p2 = N(x_t| coef1 * x0 + coef2 * x1, var) """

    denom = sigma1**2 + sigma2**2
    coef1 = sigma2**2 / denom
    coef2 = sigma1**2 / denom
    var = (sigma1**2 * sigma2**2) / denom
    return coef1, coef2, var
```

Sampling from probability distribution of given timestep using precalculated μs and Σ

```python
def q_sample(self, step, x0, x1, ot_ode=False):
    """ Sample q(x_t | x_0, x_1), i.e. eq 11 """

    assert x0.shape == x1.shape
    batch, *xdim = x0.shape

    mu_x0  = unsqueeze_xdim(self.mu_x0[step],  xdim)
    mu_x1  = unsqueeze_xdim(self.mu_x1[step],  xdim)
    std_sb = unsqueeze_xdim(self.std_sb[step], xdim)

    xt = mu_x0 * x0 + mu_x1 * x1
    if not ot_ode:
        xt = xt + std_sb * torch.randn_like(xt)
    return xt.detach()
```

Noise / β Schedules

Noise is added based on the preset schedule for the forward diffusion process. In the case of I^2SB, this schedule yields β_T, and β_T is used to calculate the accumulated variances σ_t and $\bar{\sigma}_t$. $\left(\int \beta_T \, dT\right)$

I^2SB uses what they call a symmetric beta schedule

Mathematically it is a linear schedule between two points:

$$\beta_t = \left(\sqrt{\beta_0} + t\left(\frac{\sqrt{\beta_T} - \sqrt{\beta_0}}{T}\right)\right)^2$$

" (we) consider a symmetric scaling of β_t where the diffusion shrinks at both boundaries... This is suggested by prior SB models (De Bortoli / Chen 2021)"

```python
def make_beta_schedule(n_timestep=1000, linear_start=1e-4, linear_end=2e-2):
    # return np.linspace(linear_start, linear_end, n_timestep)
    betas = (
        torch.linspace(linear_start ** 0.5, linear_end ** 0.5, n_timestep, dtype=torch.float64) ** 2
    )
    return betas.numpy()
```

Noise Shrinking at both ends of the diffusion process has been found to outperform the original DDPM by Ho et al that used a linear noise schedule, where noise increases by a constant amount. Noise at any given timestep can be found by: $\beta_t = \beta_0 + t\left(\frac{\beta_T - \beta_0}{T}\right)$.

A later paper, Improved DDPMs by Alex Nichols, shows that models generating lowres images from pure gaussian noise can benefit from a cosine β schedule:

$$\beta_t = 1 - \cos\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)$$

This lowers the amount of noise added at the beginning and end of the forward process. Lower noise at the beginning helps preserve overall structure of data and stablizes the learning process

Reverse Diffusion / p-posterior

During reverse diffusion, we need to reconstruct the clean image from the noisy image. Finding the previous timestep can be done by using the same variances from the forward process. $(X_{t-1} \mid X_t, X_0)$

$$\sigma_\Delta = \sqrt{\sigma_t^2 - \sigma_{t-1}^2}$$

$$X_t = \frac{\sigma_\Delta^2}{\sigma_t^2 + \sigma_\Delta^2} \cdot X_0 + \frac{\sigma_{t-1}^2}{\sigma_{t-1}^2 + \sigma_\Delta^2} X_{t-1} + \sqrt{\frac{\sigma_t^2 \cdot \sigma_\Delta^2}{\sigma_t^2 + \sigma_\Delta^2}} \cdot \epsilon$$

```python
def p_posterior(self, nprev, n, x_n, x0, ot_ode=False):
    """ Sample p(x_{nprev} | x_n, x_0), i.e. eq 4"""

    assert nprev < n
    std_n     = self.std_fwd[n]
    std_nprev = self.std_fwd[nprev]
    std_delta = (std_n**2 - std_nprev**2).sqrt()

    mu_x0, mu_xn, var = compute_gaussian_product_coef(std_nprev, std_delta)

    xt_prev = mu_x0 * x0 + mu_xn * x_n
    if not ot_ode and nprev > 0:
        xt_prev = xt_prev + var.sqrt() * torch.randn_like(xt_prev)

    return xt_prev
```

Loss

We train a neural network to predict p-posterior without knowing σ, because we would not be able to calculate it without x_0 / initial high resolution image.