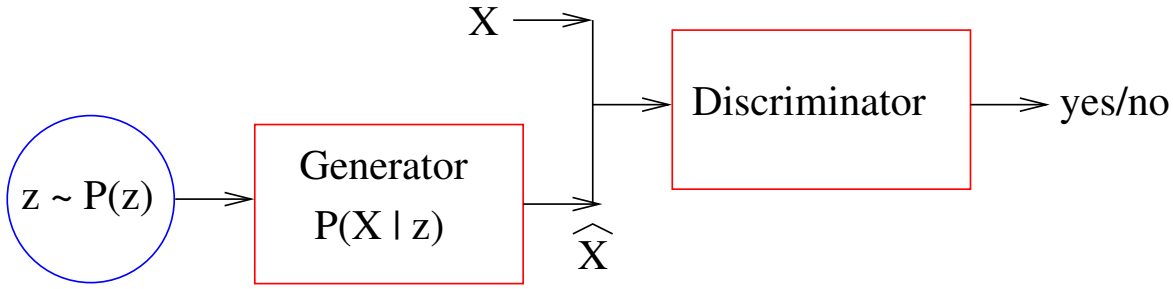
# Generative Adversarial Networks

Suggested reading:

NIPS 2016 Tutorial: Generative Adversarial Networks

# GAN

In a Generative Adversarial Network, the generator is coupled with a **discriminator** trying to tell apart **real** data from **fake** data produced by the generator.




Discriminator and Generator are trained together.

The loss function aims to:

- instruct the detector to spot the generator
- instruct the generator to fool the detector

# A Min Max game

$$Min_G Max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[log\ D(x)] + \mathbb{E}_{z \sim p_z(z)}[log\ (1 - D(G(z)))]$$

- $\mathbb{E}_{x \sim p_{data}(x)}[log\ D(x)]$ = negative cross entropy of the discriminator w.r.t the true data distribution
- $\mathbb{E}_{z \sim p_z(z)}[log\ (1 - D(G(z)))]$ = negative cross entropy of the "false" discriminator w.r.t the fake generator

# Non saturating loss for the generator

In principle, the generator G could be trained to minimize

$$log(1 - D(G(Z)))$$

In practice, it is better to train G to maximize

$$log(D(G(Z)))$$

# Training

Alternately train the discriminator, freezing the generator, and the generator freezing the discriminator:

**Algorithm 1** Training

1: **repeat**
2: freeze Generator's weights $\Theta_G$ ▷ train the discriminator D
3: sample a batch of real data $x \sim P_{\text{data}}$
4: sample a batch of noise $z \sim N(0,1)$
5: update the Discriminator's weights $\Theta_D$ by stochastic **ascent** on:

$$\nabla_{\Theta_D}[\mathbb{E}_{x\sim P_{\text{data}}} logD(x) + \mathbb{E}_{z\sim N(0,1)} log(1 - D(G(z))]$$

6: freeze Discriminator's weights $\Theta_D$ ▷ train the generator G
7: sample a batch of noise $z \sim N(0,1)$
8: update the Generator's weights $\Theta_G$ by stochastic **ascent** on:

$$\nabla_{\Theta_G}[\mathbb{E}_{z\sim N(0,1)} logD(G(z))]$$

9: **until** converged

# An example

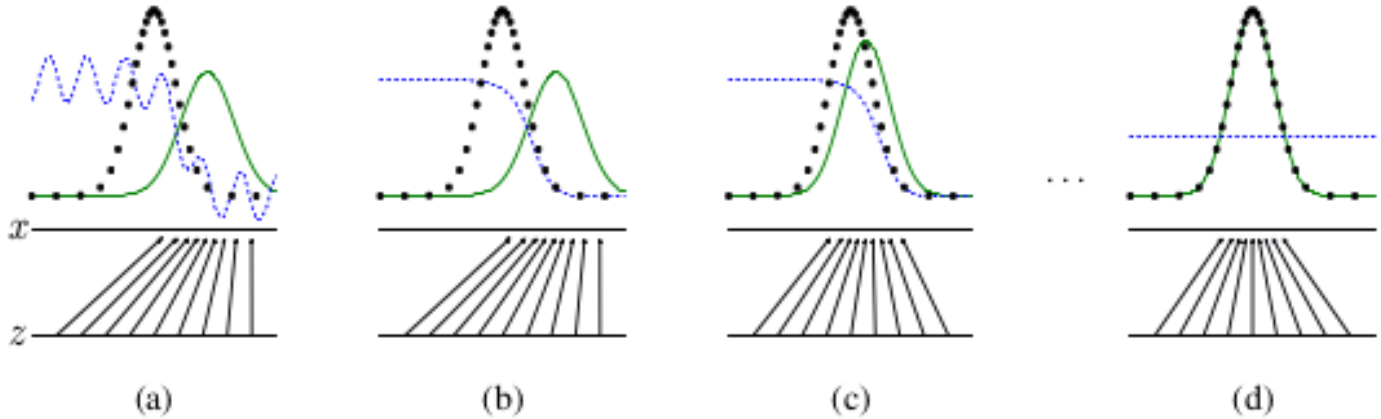


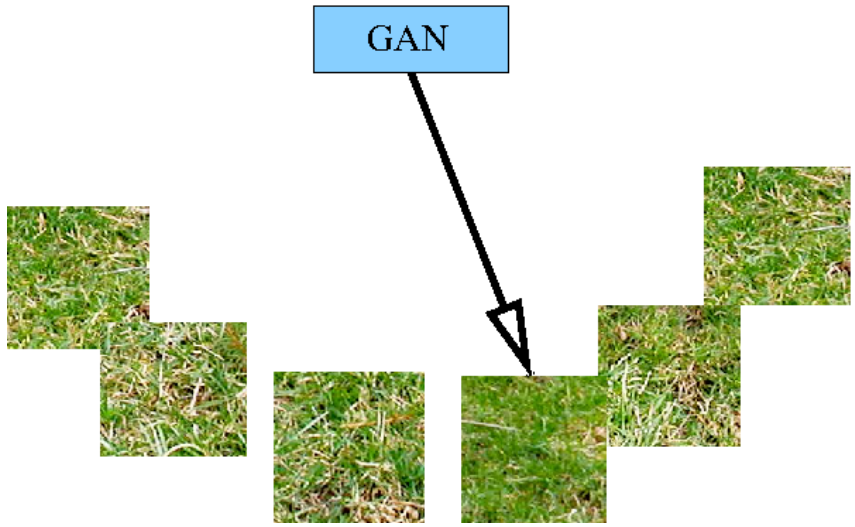

Figure 1: Generative adversarial nets are trained by simultaneously updating the **d**iscriminative distribution ($D$, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the **g**enerative distribution $p_g$ (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution $p_g$ on transformed samples. $G$ contracts in regions of high density and expands in regions of low density of $p_g$. (a) Consider an adversarial pair near convergence: $p_g$ is similar to $p_{\text{data}}$ and $D$ is a partially accurate classifier. (b) In the inner loop of the algorithm $D$ is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$. (c) After an update to $G$, gradient of $D$ has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if $G$ and $D$ have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

# A simple demo

Demo!

# The Gan solution to the patch of grass problem

- The only goal of the generator is to fool the discriminator
- No commitment to reconstruct all training data
- Just generate plausible data



# Gan evolution over time

2014 GAN

2015 DCGAN

2016 CoupleGAN

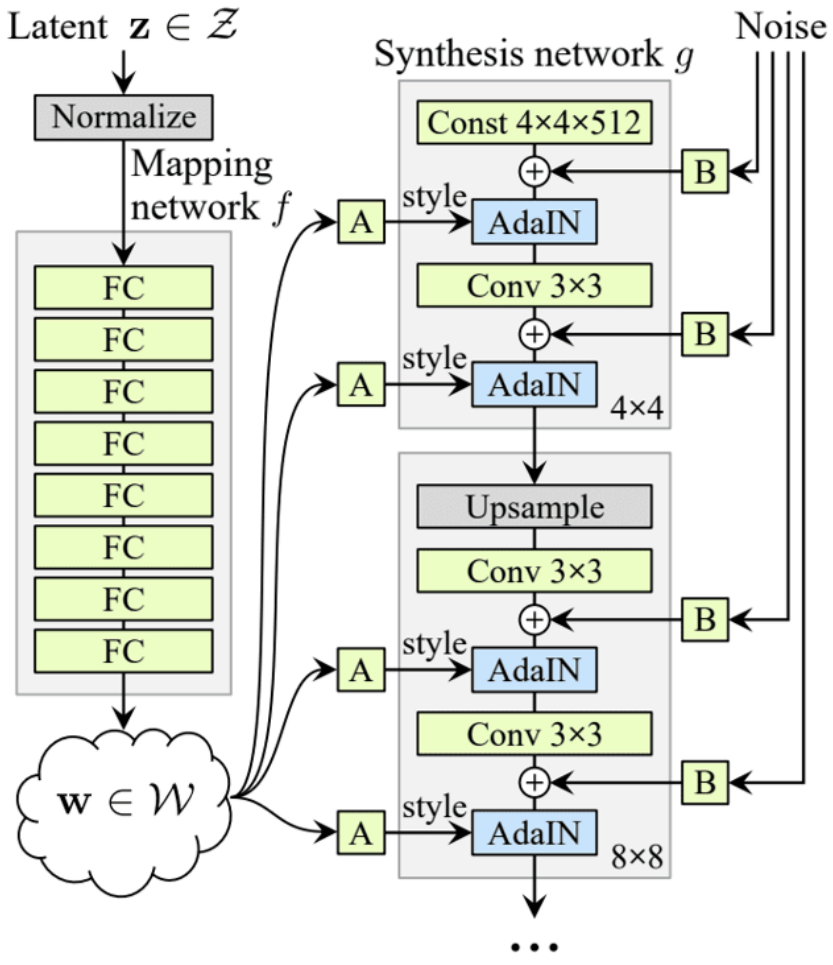
2017 ProGAN

2018 StyleGAN

2019 Style GAN2

# This person does not exist

# StyleGan


Latent $\mathbf{z} \in \mathcal{Z}$
Normalize
Mapping network $f$
FC
FC
FC
FC
FC
FC
FC
FC
$\mathbf{w} \in \mathcal{W}$
Synthesis network $g$
Noise
Const 4×4×512
B
style
A
AdaIN
Conv 3×3
4×4
Upsample
8×8
...

# AdaIN: Adaptive Instance Normalization

Adaptive Instance Normalization is a normalization method that aligns the mean and variance of the content features with those of a different sample (style features).

In AdaIN, we receive a "content" input $x$ and a "style" input $y$ and align the channel-wise mean and variance of $x$ to match those of $y$:

$$AdaIN(x, y) = \sigma(y) + \frac{x - \mu(x)}{\sigma(x)} + \mu(y)$$

where the shape of $x$ is $(B, C, W, H)$ and the shape of $\mu(x), \sigma(x), \mu(y), \sigma(y)$ is $(B, C)$ (intance normalization!)

AdaIN has no learnable affine parameters. Instead, it adaptively computes the affine parameters from the style input.

# Another typical GAN application: super-resolution

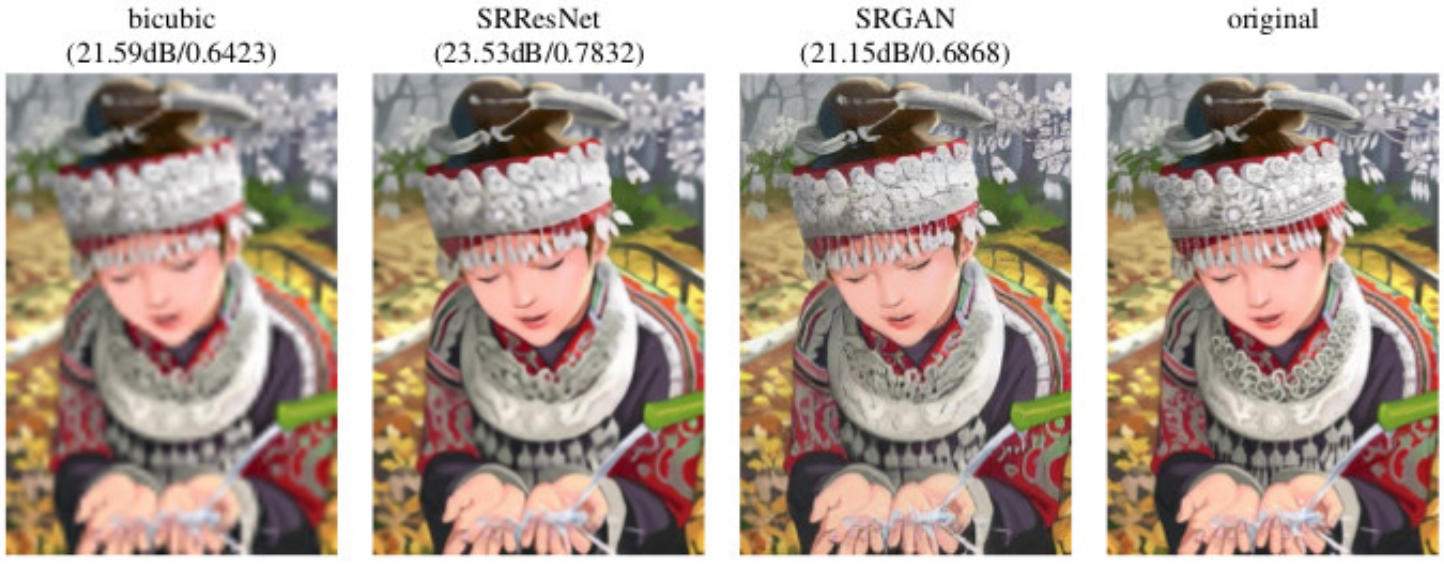



Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

Forcing to operate a choice - instead of mediating - could result in sharper images

Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. C.Ledig et al., 2016.

# Problems with Gans

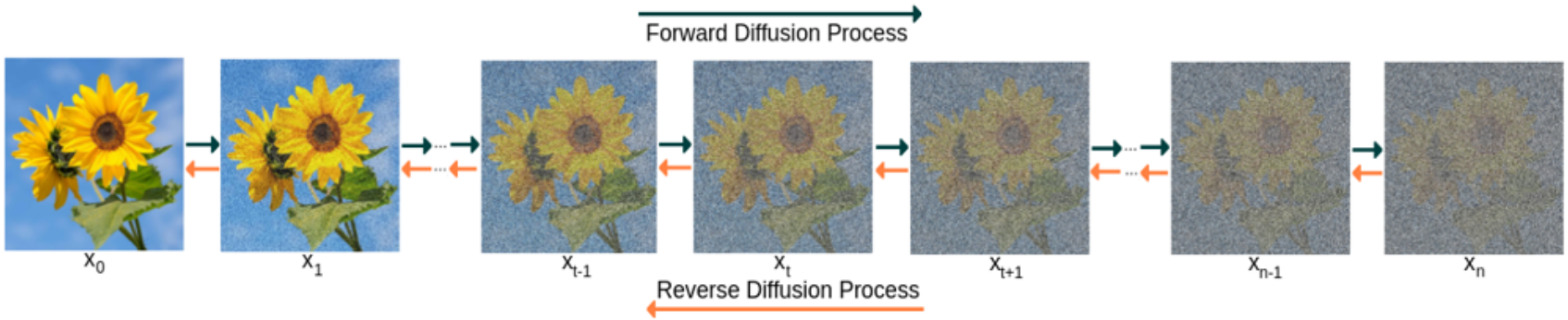
**mode collapse** The generator fails to produce diverse, output, repeatedly generating similar or identical smaples (e.g. generate a single face)
More generally, the generator fails to produce a sufficient diversity, focusing of the most likely cases (e.g. never generate persons with hats)

**weak feedback to the generator** When the discirminator is overly confident, the gradient passed to the generator is low, causing its training to stagnate.

**training instability** : the adversarial nature of GANs lead to oscillations or divergence during training

# Next argument

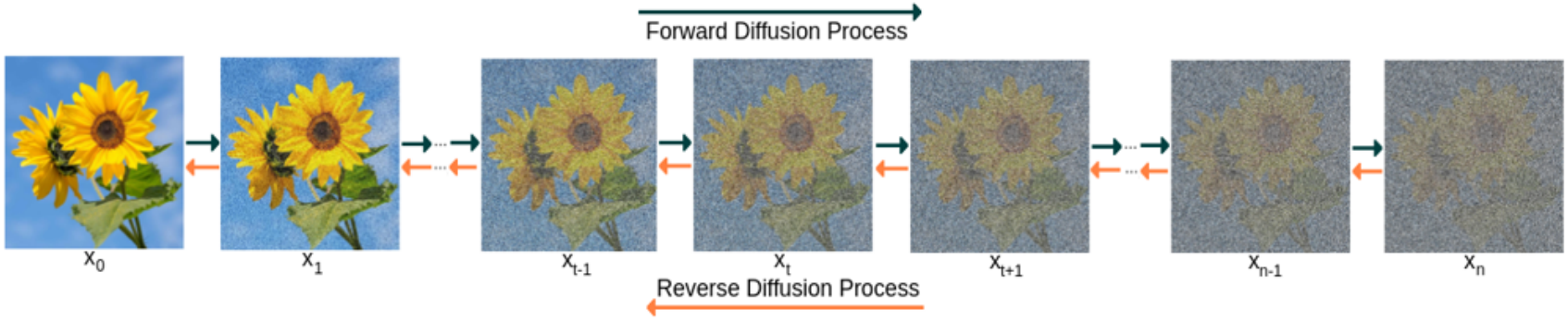
# Diffusion models




Suggested reading: What are diffusion models

# Forward and Backward diffusion

Conceptually, denoising models must be understood in terms of two basic processes: forward and backward diffusion.



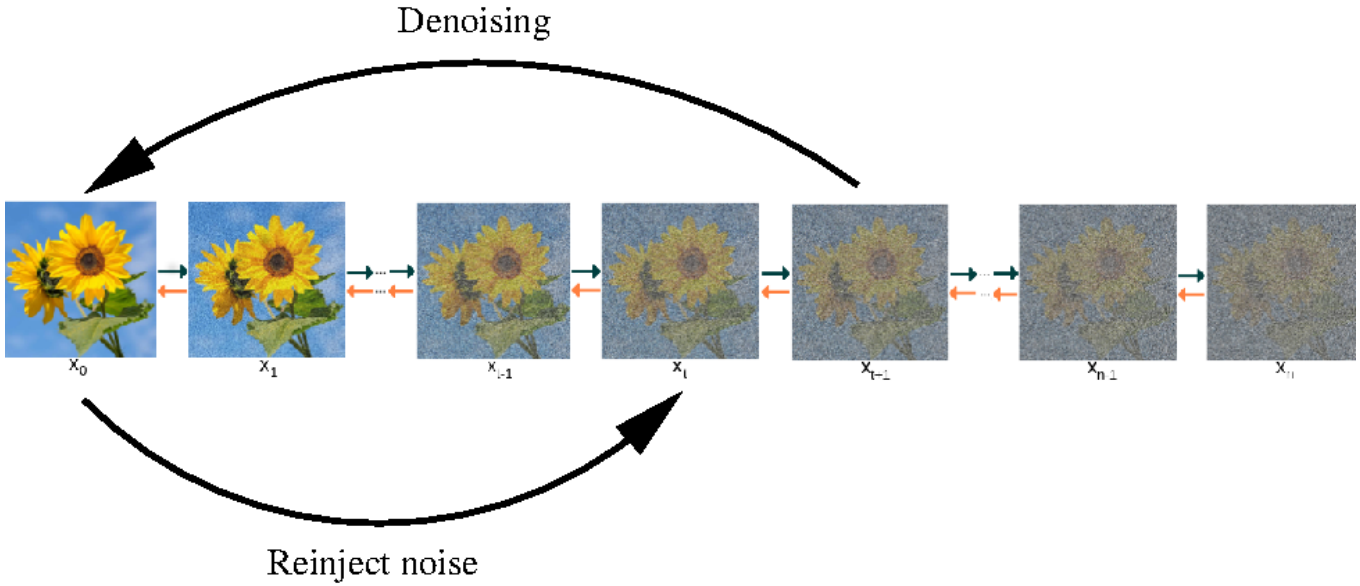

- From left to right, we see **forward diffusion**, where structured data, such as an image, is progressively corrupted by adding noise at each step.
- From right to left, we have **backward diffusion**, the reverse process, where the model learns to reconstruct the original data step by step by denoising it.

# The bakward step

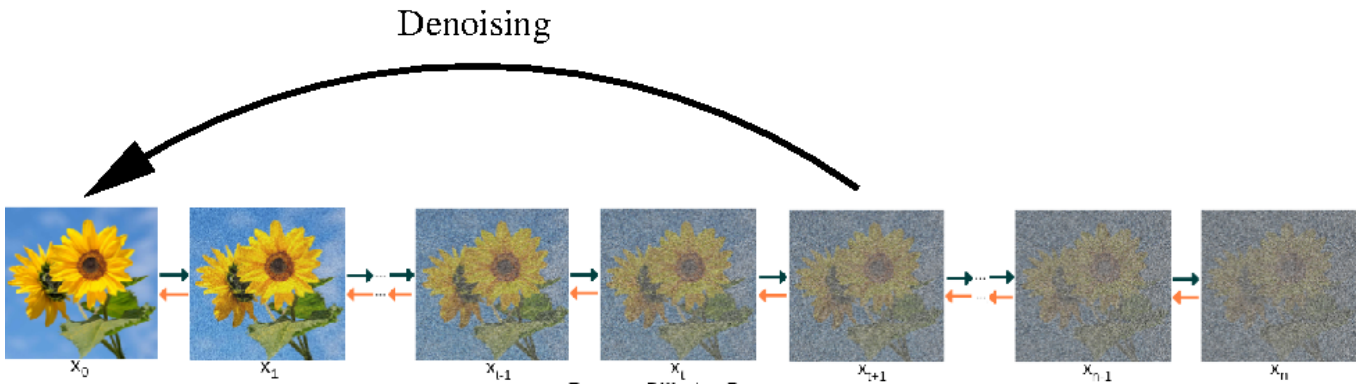
How can we remove a "small" amount of noise?

Each backward step is decomposed into two basic operations:

- **denoising**: trying to remove **all** the noise in the image
- **noise reinjection**: adding back **a smaller** amount of noise



# Denoising

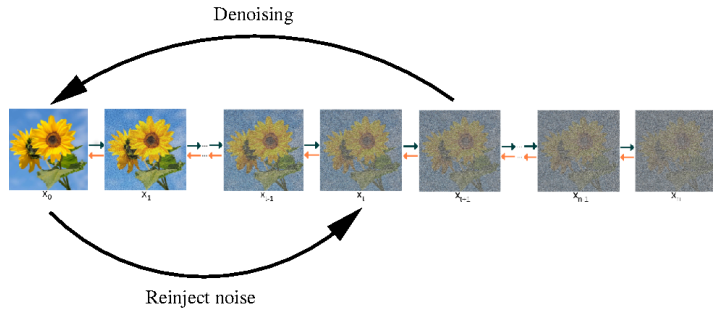



- The main component of the generation process is the **denoising network**, trained to predict the noise contained in an image.
- The network is unique for alla steps $t$, and parametric in $t$. This means that the networks**knows the amount of noise** it is supposed to remove.
- the denoising network is the **only trainable component of** the model.

# Probabilistic (DDPM) vs Deterministic (DDIM) Diffusion

The two versions differ in the way the noise is reinjected during the sampling phase.

- In DDMP we use a new random amount of noise
- In DDIM we reinject the same noise that was removed, but at a weaker extent. In this way, generation in DDIM becomes **deterministic** in terms of the starting noise.



# The denoising network

The denoising network implements the inverse of the operation of adding a given amount of noise to an image (direct diffusion).

The denoising network takes in input:

1. a noisy image $x_t$
2. a signal rate $\alpha_t$ expressing the amount of the original signal remaining in the noisy image

and try to predict the noise in it:

$$\epsilon_\theta(x_t, \alpha_t)$$

The predicted image is:

$$\hat{x}_0^t = (x_t - \sqrt{1-\alpha_t} \cdot \epsilon_\theta(x_t, \alpha_t))/\sqrt{\alpha_t}$$

# Training pseudocode

Assume a progressive signal rate $\alpha_t$ with $t \in [T, \ldots, 0]$, with values in the range [0,1]:

$$\alpha_T < \alpha_{T-1} < \cdots < \alpha_t < \alpha_{t-1} < \cdots < \alpha_0$$

**Algorithm 1** Training

1: **repeat**
2: $x_0 \sim q(x_0)$ ▷ take a sample
3: $t \sim$Uniform(1,..,T) ▷ choose a timestep
4: $\epsilon \sim \mathcal{N}(0; I)$ ▷ create random gaussian noise
5: $x_t = \sqrt{\alpha_t} x_0 + \sqrt{1-\alpha_t}\epsilon$ ▷ corrupt the sample with signal rate $\alpha_t$
6: compute $\epsilon_\theta(x_t, \alpha_t)$ ▷ predict the error
7: Take a gradient descent step on $||\epsilon - \epsilon_\theta(x_t, \alpha_t)||$ ▷ backpropagation
8: **until** converged

# Sampling pseducode

Assume a progressive signal rate $\alpha_t$ with $t \in [T, \ldots, 0]$, with values in the range [0,1]:

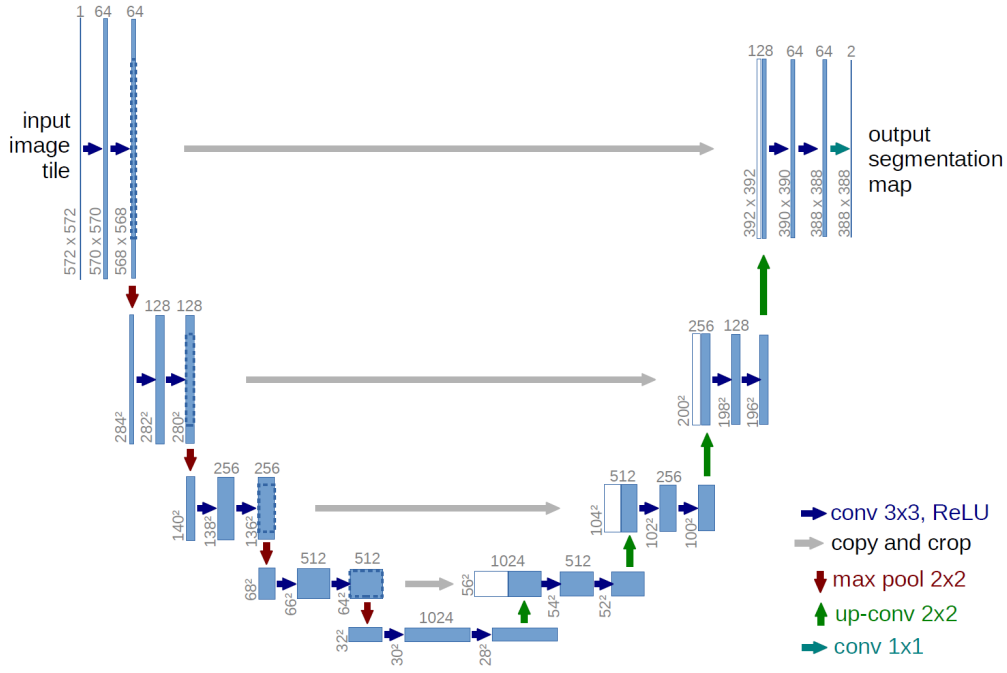
$$\alpha_T < \alpha_{T-1} < \cdots < \alpha_t < \alpha_{t-1} < \cdots < \alpha_0$$

**Algorithm 2** Sampling

1: $x_T \sim \mathcal{N}(0, I)$
2: **for** $t = T, ..., 1$ **do**
3: $\quad \epsilon = \epsilon_\theta(x_t, \alpha_t)$ ▷ predict noise
4: $\quad \tilde{x}_0 = \frac{1}{\sqrt{\alpha_t}}(x_t - \sqrt{1-\alpha_t}\epsilon)$ ▷ compute denoised result
5: $\quad x_{t-1} = \sqrt{\alpha_{t-1}}\tilde{x}_0 + \sqrt{1-\alpha_{t-1}}\epsilon$ ▷ re-inject noise at rate $\alpha_{t-1}$
6: **end for**

# Network architecture

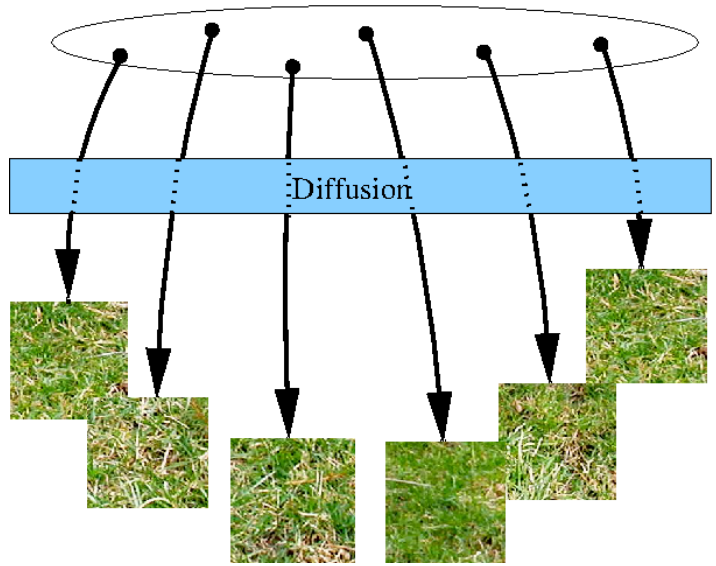
Use a (conditional) Unet!



# Diffusion and the patch of grass problem

Unlike latent-space models, diffusion models operate in the full visible space, avoiding any compression that might oversimplify these complex textures.

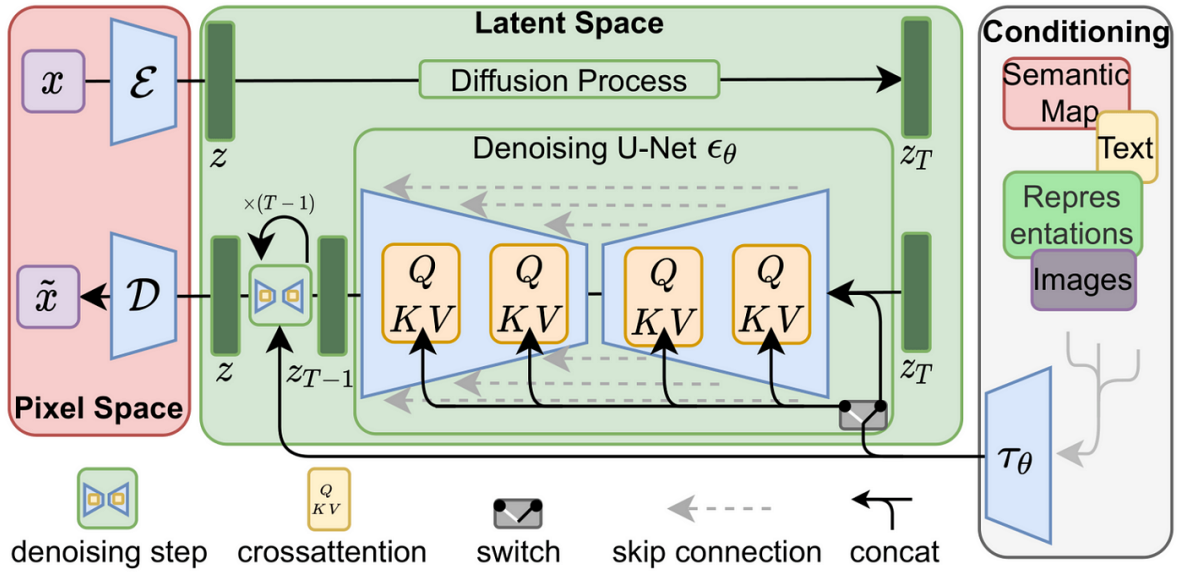
- Each point in the latent space naturally maps to a distinct, well-defined point in the visible space.
- This approach allows diffusion models to faithfully generate detailed and realistic textures without the compromises introduced by latent compression or adversarial training.



# Stable Diffusion

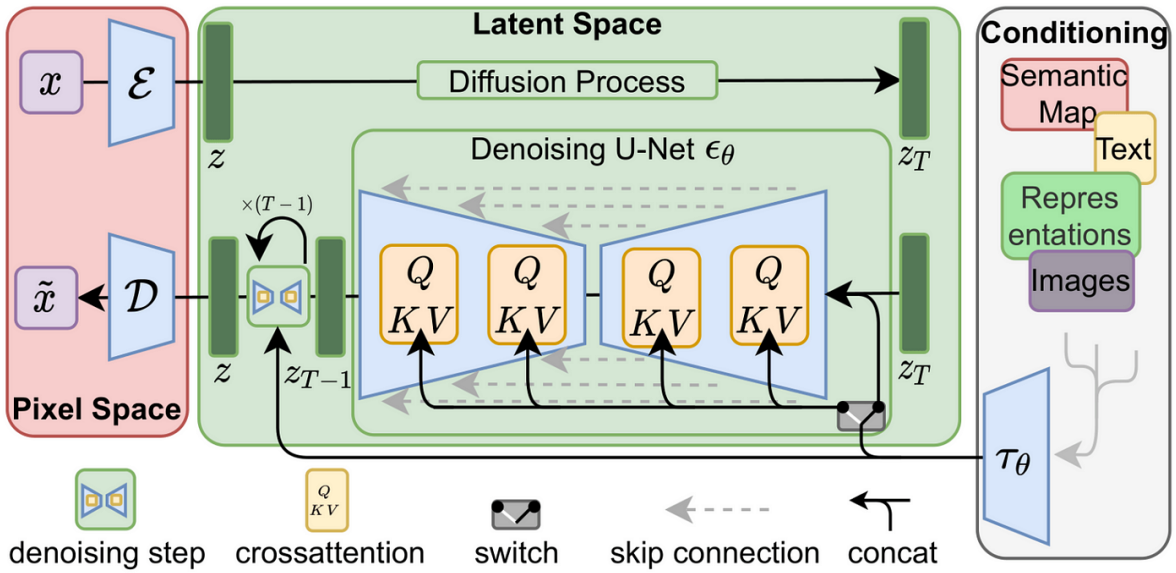
# Stable Diffusion

- Pixel-space diffusion is expensive: Operating directly on high-resolution images (e.g., $512 \times 512$) is slow and memory-intensive.
- **Key idea**: Move the diffusion process to a lower-dimensional latent space using a pretrained VAE.



# Stable Diffusion components

- **VAE** (Encoder & Decoder): Compresses image to latent space and reconstructs it
- **Denoiser**: Learns to remove noise in latent space
- **Text Encoder** (e.g., CLIP): Converts prompt to conditioning vector



# Training

- The VAE is **pretrained and frozen**
- The Denoiser is trained to **sample points** in the latent space
- The text encoder is also **pretrained and frozen**

The quality of the VAE must be high: the decoder limits the sharpness of final outputs.

How is VAE blurriness addressed?

# Avoiding the Usual VAE Blurriness

- Compression is not aggressive
- KL-regularization quite light
- Use of Perceptual losses (e.g., LPIPS) instead of MSE
- possible Use of adversarial losses for photorealism

The VAE inside SD is practically an autoencoder.

Since the denoiser learns the actual distribution of the latent encodings in the latent space, no point to make it similar to a known prior.