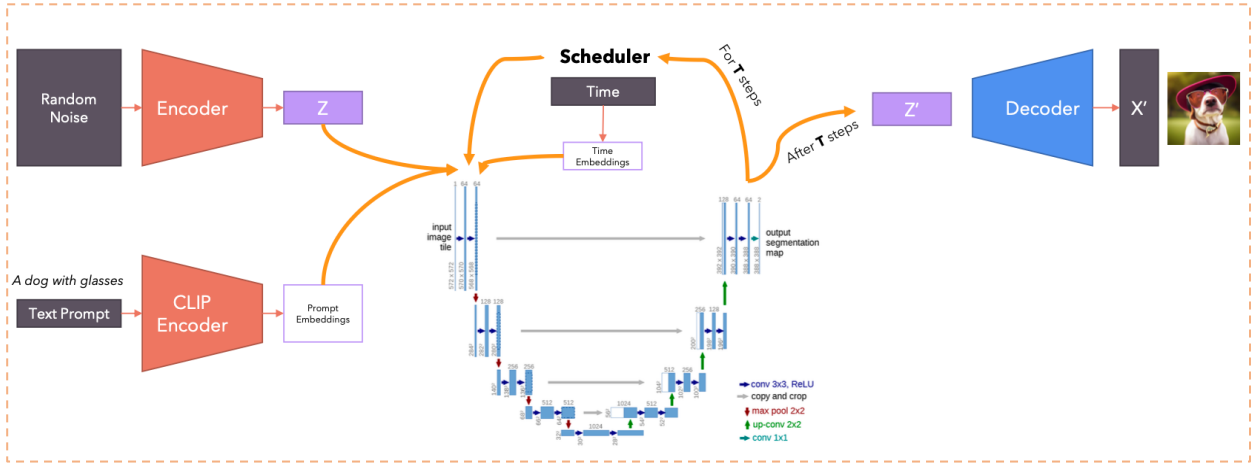
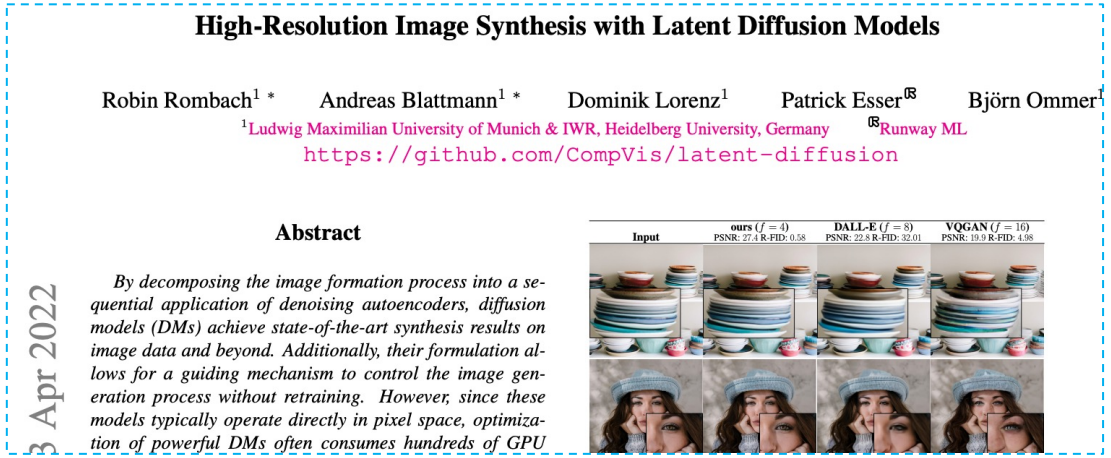
AI VIETNAM
All-in-One Course

Stable Diffusion: Clearly Explained





High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

Abstract

By decomposing the image formation process into a sequential application of denoising autoencoders, diffusion models (DMs) achieve state-of-the-art synthesis results on image data and beyond. Additionally, their formulation allows for a guiding mechanism to control the image generation process without retraining. However, since these models typically operate directly in pixel space, optimization of powerful DMs often consumes hundreds of GPU

Vinh Dinh Nguyen
PhD in Computer Science

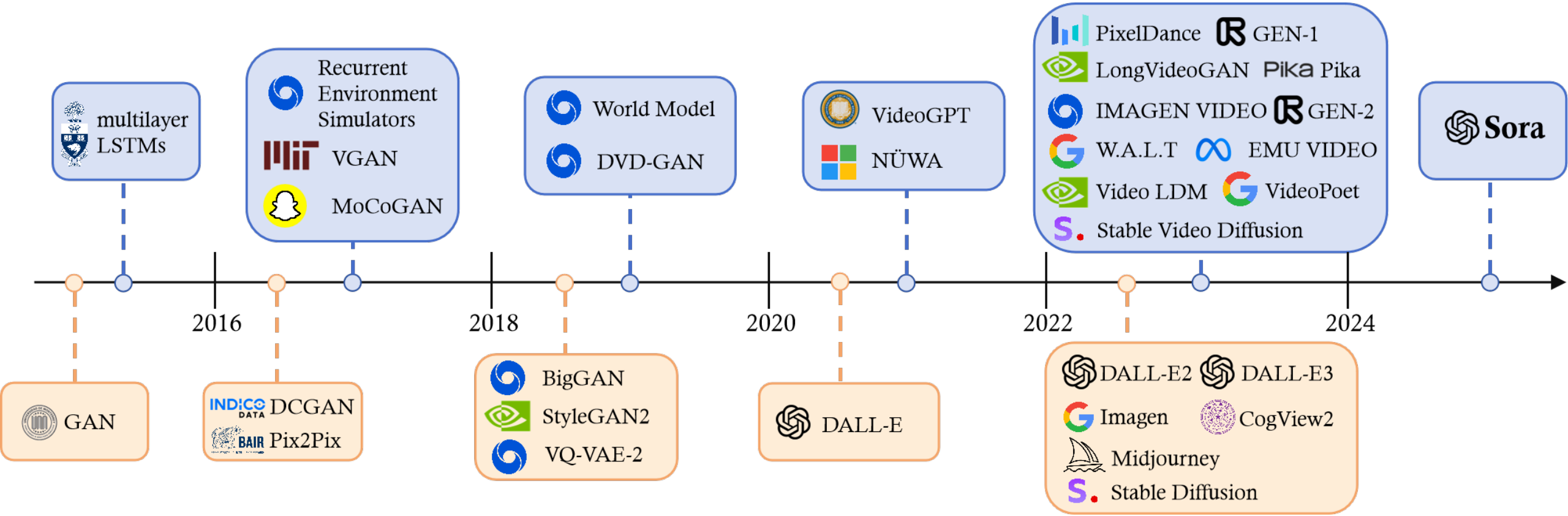
Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

History of Generative AI in Vision Domain



Objective

High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

Abstract

By decomposing the image formation process into a sequential application of denoising autoencoders, diffusion models (DMs) achieve state-of-the-art synthesis results on image data and beyond. Additionally, their formulation allows for a guiding mechanism to control the image generation process without retraining. However, since these models typically operate directly in pixel space, optimization of powerful DMs often consumes hundreds of GPU days and inference is expensive due to sequential evaluations. To enable DM training on limited computational resources while retaining their quality and flexibility, we apply them in the latent space of powerful pretrained autoencoders. In contrast to previous work, training diffusion models on such a representation allows for the first time to reach a near-optimal point between complexity reduction and detail preservation, greatly boosting visual fidelity. By introducing cross-attention layers into the model architecture, we turn diffusion models into powerful and flexible generators for general conditioning inputs such as text or bounding boxes and high-resolution synthesis becomes possible in a convolutional manner. Our latent diffusion models (LDMs) achieve new state-of-the-art scores for image inpainting and class-conditional image synthesis and highly competitive performance on various tasks, including text-to-image synthesis, unconditional image generation and super-resolution, while significantly reducing computational requirements compared to pixel-based DMs.

1. Introduction

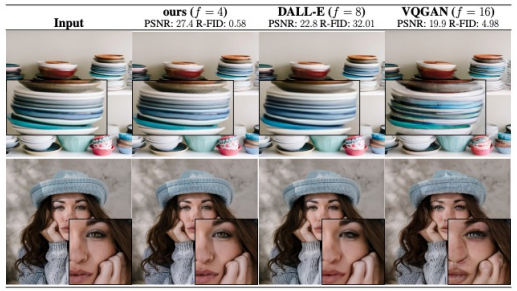


Figure 1. Boosting the upper bound on achievable quality with less agressive downsampling. Since diffusion models offer excellent inductive biases for spatial data, we do not need the heavy spatial downsampling of related generative models in latent space, but can still greatly reduce the dimensionality of the data via suitable autoencoding models, see Sec. 3. Images are from the DIV2K [1] validation set, evaluated at 512^2 px. We denote the spatial downsampling factor by f. Reconstruction FIDs [29] and PSNR are calculated on ImageNet-val. [12]; see also Tab. 8.

results in image synthesis [30,85] and beyond [7,45,48,57], and define the state-of-the-art in class-conditional image synthesis [15,31] and super-resolution [72]. Moreover, even unconditional DMs can readily be applied to tasks such as inpainting and colorization [85] or stroke-based synthesis [53], in contrast to other types of generative models [19,46,69]. Being likelihood-based models, they do not exhibit mode-collapse and training instabilities as GANs and, by heavily exploiting parameter sharing, they can model highly complex distributions of natural images with-

Topics discussed

1. Principle of Diffusion models
2. Model score function of images with UNet model
3. Understanding prompt through contextualized word embedding
4. Let text influence image through cross attention
5. Improve efficiency by adding an autoencoder
6. Latent Diffusion Models in Pytorch

Prerequisites

1. Basics of probability and statistic (multivariate, gaussian, conditional probability, likelihood, Bayers' rule)
2. Basic of Pytorch and Neural Network
3. How the attention mechanism work
4. How convolution layers work

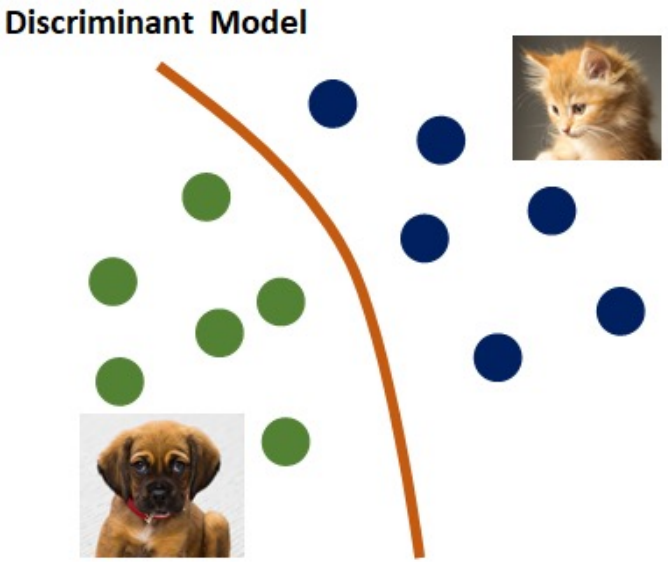
Outline

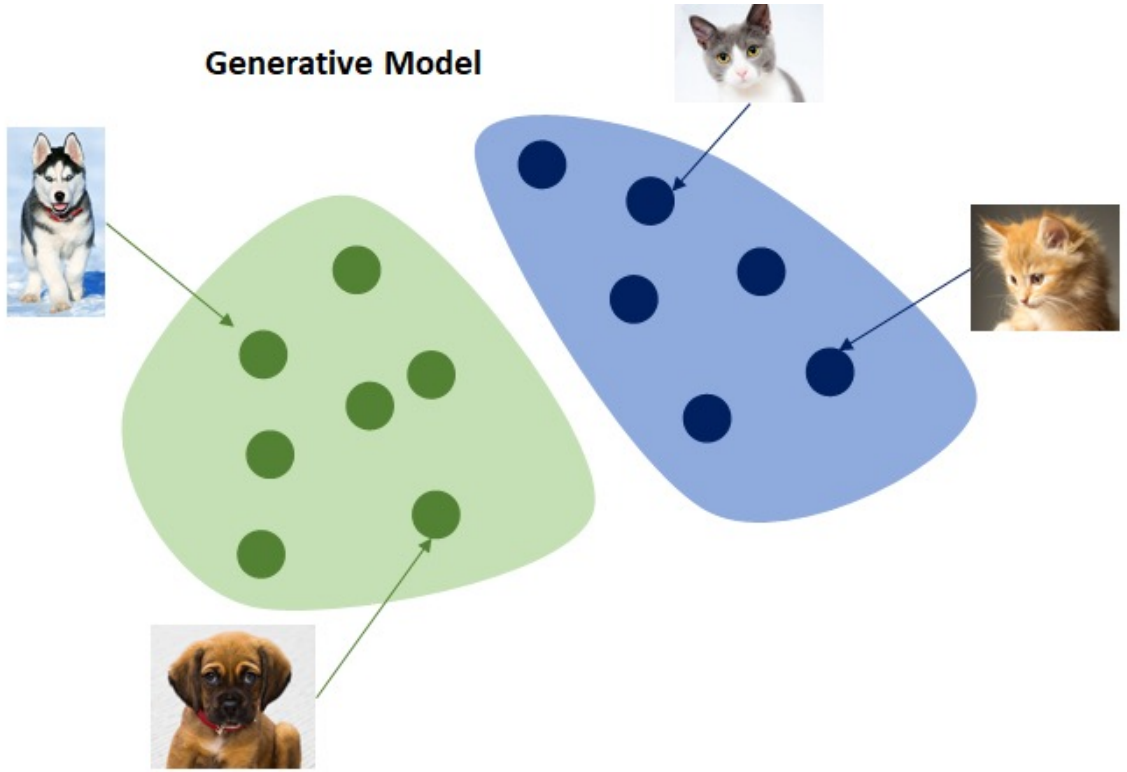
- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

What is a generative model?

A generative model learns a probability distribution of the data set such that we can then sample from the distribution to create new instances of data.

For example, if we have many pictures of cats and we train a generative model on it, we then sample from this distribution to create new images of cats

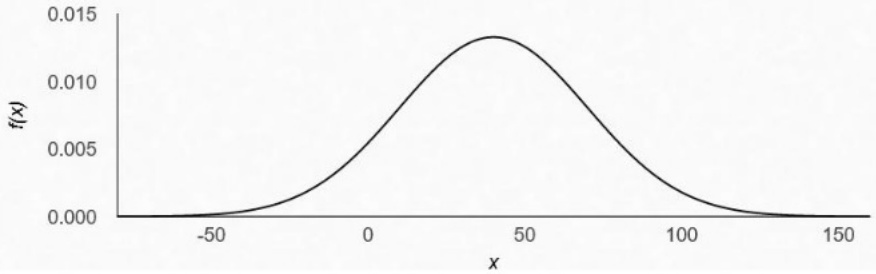




What is a generative model?

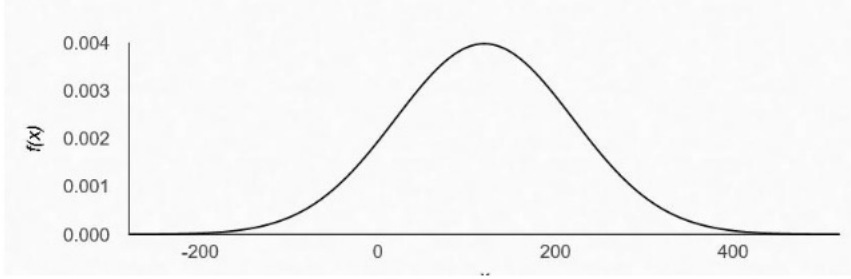
Imagine you're a researcher, and you want to generate thousands of fake identities. Each fake identity, is made up of variables, representing the characteristics of a person (Age, Height).

You can ask the Statistics Department of the Government to give you statistics about the age and the height of the population and then sample from these distributions.



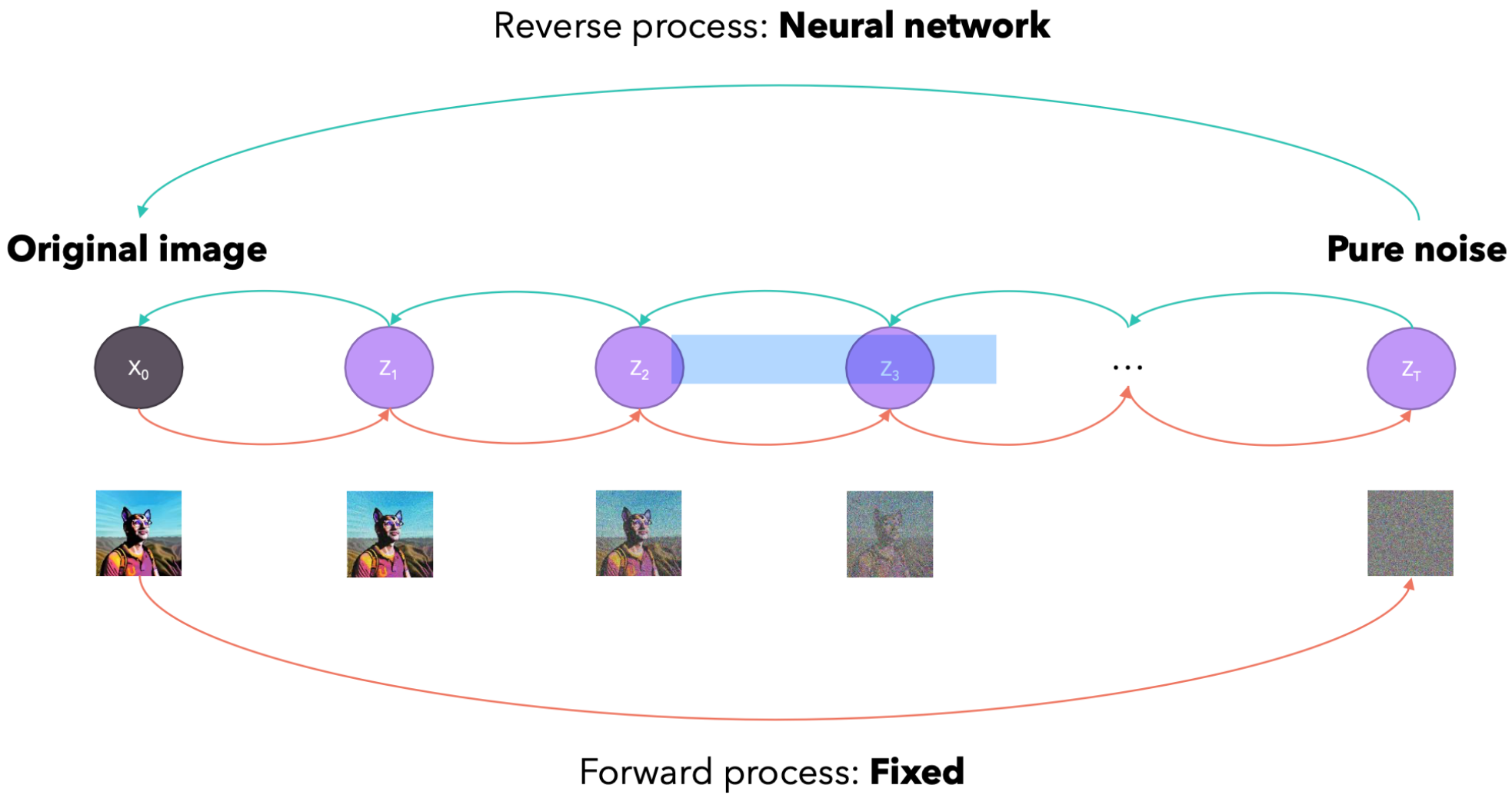
Age: **N**$(40, 30^2)$

To generate fake identities that make sense, you need the joint distribution, otherwise you may end up with an unreasonable pair of (Age, Height)



Height: **N**$(120, 100^2)$

What is a Diffusion Model?



What is a Diffusion Model?

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 $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$:

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

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 $\beta_1, \ldots, \beta_T$:

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

Evidence Lower Bound **(ELBO)**

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

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

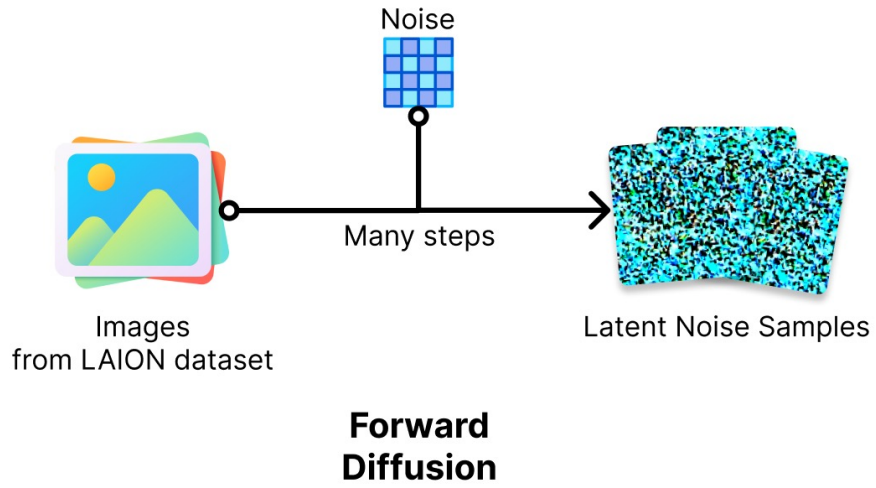
Forward process **q**

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

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

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

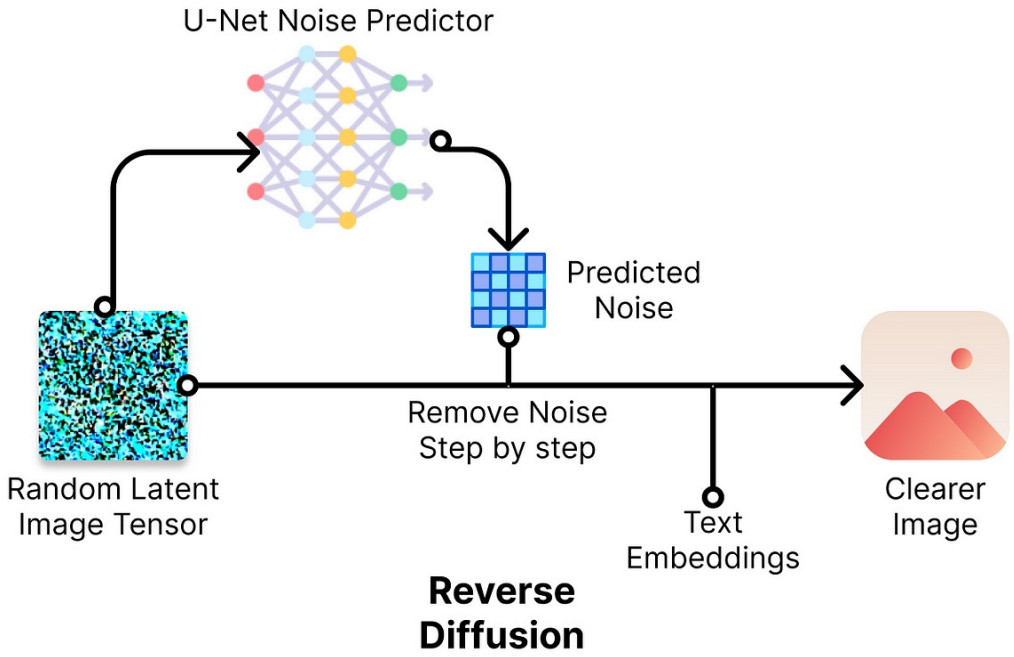
The diffusion model : Forward Process



A forward diffusion process to prepare training samples and a reverse diffusion process to generate the images

As the name “diffusion” suggests, this process is just like dropping a drop of ink into the water — the ink droplet gradually diffuses in water, until you can no longer see that it was previously a drop of ink. The noise patterns added to the images are random, just like ink particles diffuse randomly into water particles, but the amount of noise can be controlled

The diffusion model : Reverse Process

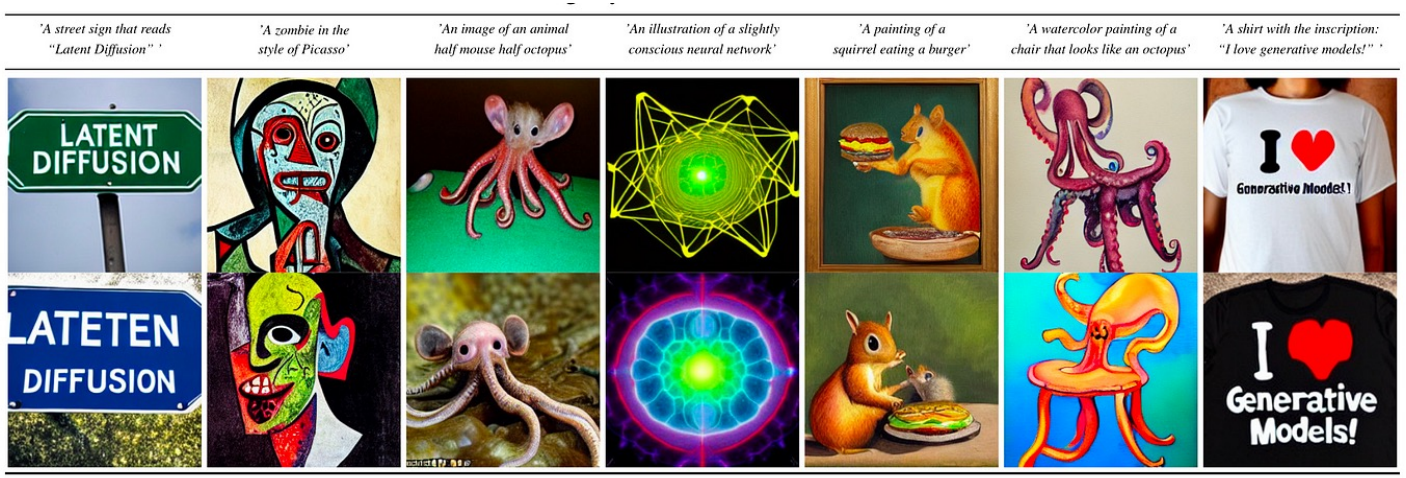


This reverse diffusion process is done gradually through multiple steps to remove the noise. The more denoising steps are done, the clearer the image becomes.

During this reverse diffusion process for getting a clearer image, researchers wanted to control how the image looks through a process called conditioning

In the reverse diffusion process, a noise predictor is trained to predict the noise added to the original image so that the model can remove the predicted noise from the noisy image to get a clearer image (in the latent space). You can think of this process as looking at a partially diffused ink droplet in water and trying to predict the location it dropped earlier.

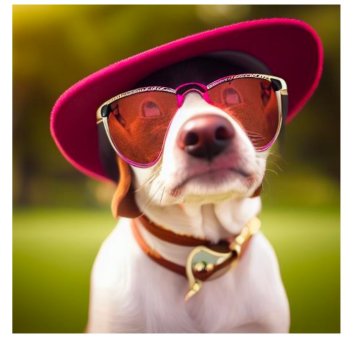
Introduction to Stable Diffusion



Stable Diffusion is a text-to-image deep learning model, based on diffusion models.
Introduced in 2022, developed by the CompViz Group at LMU Munich. https://github.com/Stability-AI/stablediffusion

Prompt
Picture of a dog with glasses →

Output



High-Resolution Image Synthesis with Latent Diffusion Models

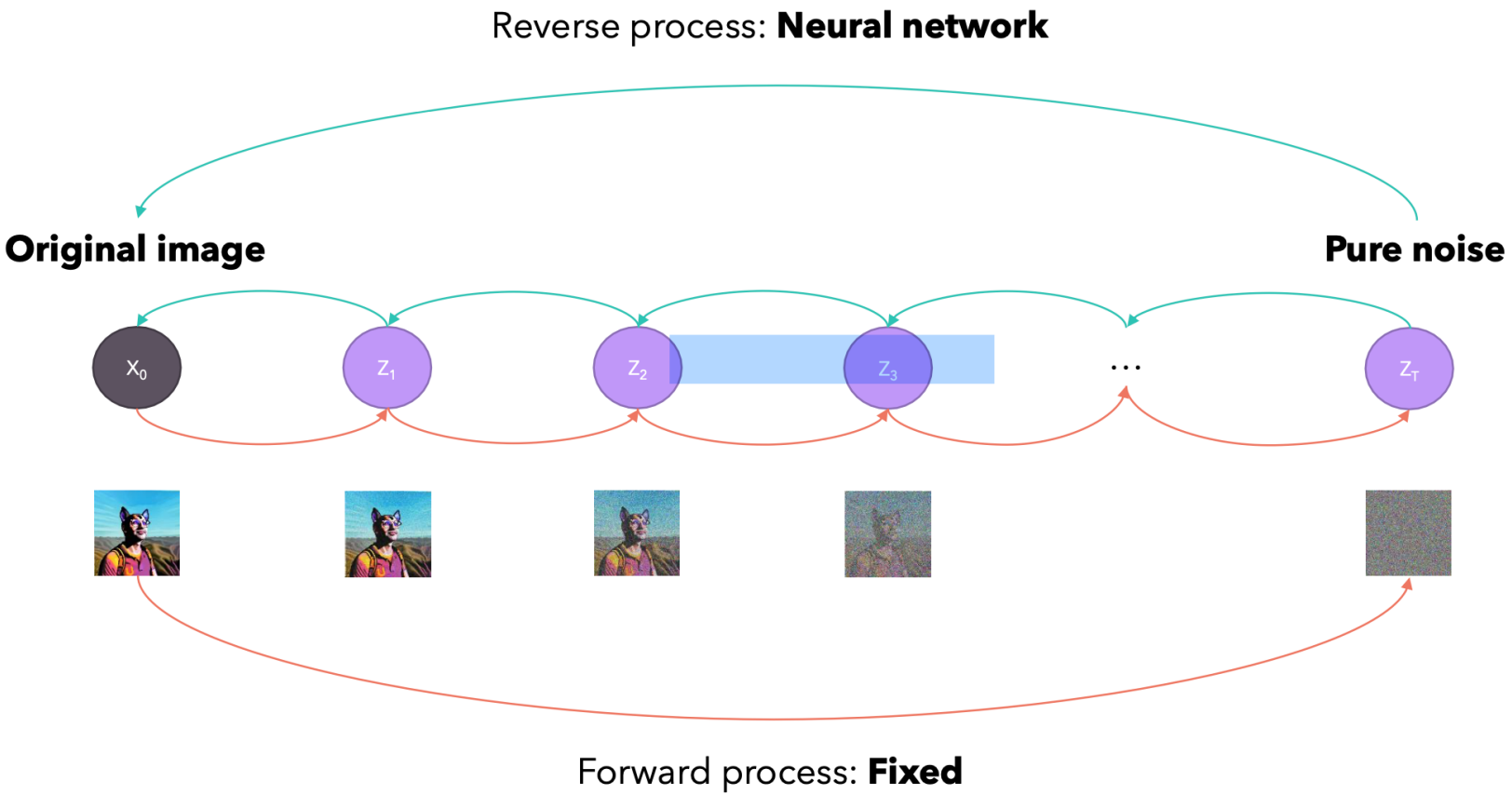
Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

Diffusion Model: Limitations

Since the latent variables have the same dimension (size of the vector) as the original data, if we want to perform many steps to denoise an image, that would result in a lot of steps through the Unet, which can be very slow if the matrix representing our data/latent is large. What if we could "compress" our data before running it through the forward/reverse process (UNet)?



Stable Diffusion: Motivation

This is emphasized when dealing with large image sizes and a high number of diffusion steps (T). The time required for denoising the image from Gaussian noise during sampling can become prohibitively long. To address this issue, a group of researchers proposed a novel approach called **Stable Diffusion**, originally known as Latent Diffusion Model (LDM).

High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

Abstract

By decomposing the image formation process into a sequential application of denoising autoencoders, diffusion models (DMs) achieve state-of-the-art synthesis results on image data and beyond. Additionally, their formulation allows for a guiding mechanism to control the image generation process without retraining. However, since these models typically operate directly in pixel space, optimization of powerful DMs often consumes hundreds of GPU days and inference is expensive due to sequential evaluations. To enable DM training on limited computational resources while retaining their quality and flexibility, we apply them in the latent space of powerful pretrained autoencoders. In contrast to previous work, training diffusion models on such a representation allows for the first time to reach a near-optimal point between complexity reduction and detail preservation, greatly boosting visual fidelity. By introducing cross-attention layers into the model architecture, we turn diffusion models into powerful and flexible generators for general conditioning inputs such as text or bounding boxes and high-resolution synthesis becomes possible in a convolutional manner. Our latent diffusion models (LDMs) achieve new state-of-the-art scores for image inpainting and class-conditional image synthesis and highly competitive performance on various tasks, including text-to-image synthesis, unconditional image generation and super-resolution, while significantly reducing computational requirements compared to pixel-based DMs.

1. Introduction

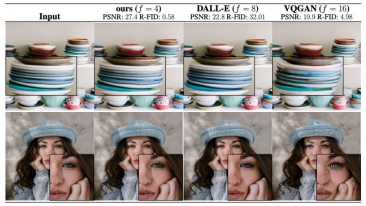
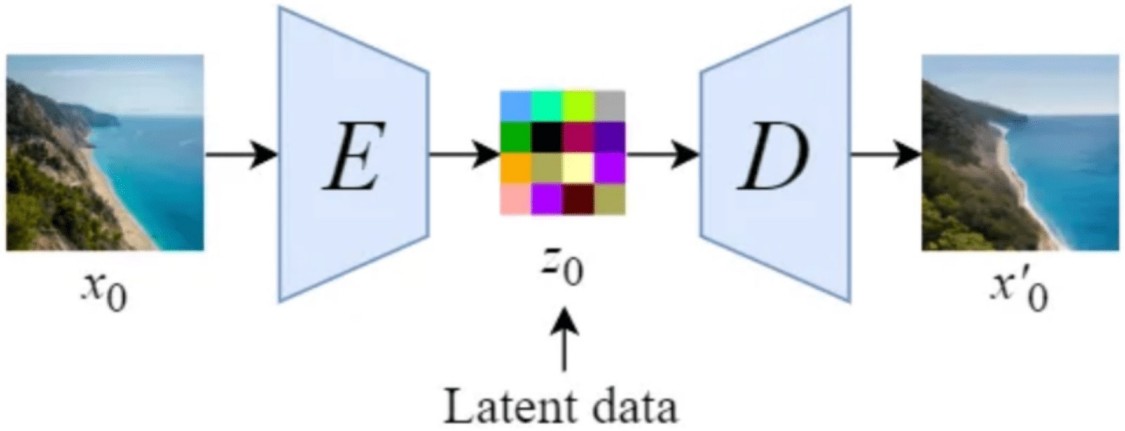


Figure 1. Boosting the upper bound on achievable quality with less agressive downsampling. Since diffusion models offer excellent inductive biases for spatial data, we do not need the heavy spatial downsampling of related generative models in latent space, but can still greatly reduce the dimensionality of the data via suitable autoencoding models, see Sec. 3. Images are from the DIV2K [1] validation set, evaluated at 512^2 px. We denote the spatial downsampling factor by f. Reconstruction FIDs [29] and PSNR are calculated on ImageNet-val. [12]; see also Tab. 8.

results in image synthesis [30,85] and beyond [7,45,48,57], and define the state-of-the-art in class-conditional image synthesis [15,31] and super-resolution [72]. Moreover, even unconditional DMs can readily be applied to tasks such as inpainting and colorization [85] or stroke-based synthesis [53], in contrast to other types of generative models [19,46,69]. Being likelihood-based models, they do not exhibit mode-collapse and training instabilities as GANs and, by heavily exploiting parameter sharing, they can model highly complex distributions of natural images with-



Stable Diffusion: Motivation

This is emphasized when dealing with large image sizes and a high number of diffusion steps (T). The time required for denoising the image from Gaussian noise during sampling can become prohibitively long. To address this issue, a group of researchers proposed a novel approach called **Stable Diffusion**, originally known as Latent Diffusion Model (LDM).

High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

Abstract

By decomposing the image formation process into a sequential application of denoising autoencoders, diffusion models (DMs) achieve state-of-the-art synthesis results on image data and beyond. Additionally, their formulation allows for a guiding mechanism to control the image generation process without retraining. However, since these models typically operate directly in pixel space, optimization of powerful DMs often consumes hundreds of GPU days and inference is expensive due to sequential evaluations. To enable DM training on limited computational resources while retaining their quality and flexibility, we apply them in the latent space of powerful pretrained autoencoders. In contrast to previous work, training diffusion models on such a representation allows for the first time to reach a near-optimal point between complexity reduction and detail preservation, greatly boosting visual fidelity. By introducing cross-attention layers into the model architecture, we turn diffusion models into powerful and flexible generators for general conditioning inputs such as text or bounding boxes and high-resolution synthesis becomes possible in a convolutional manner. Our latent diffusion models (LDMs) achieve new state-of-the-art scores for image inpainting and class-conditional image synthesis and highly competitive performance on various tasks, including text-to-image synthesis, unconditional image generation and super-resolution, while significantly reducing computational requirements compared to pixel-based DMs.

1. Introduction

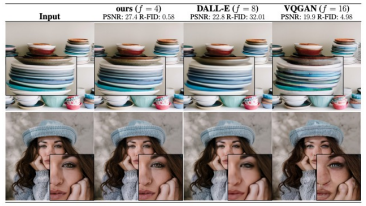
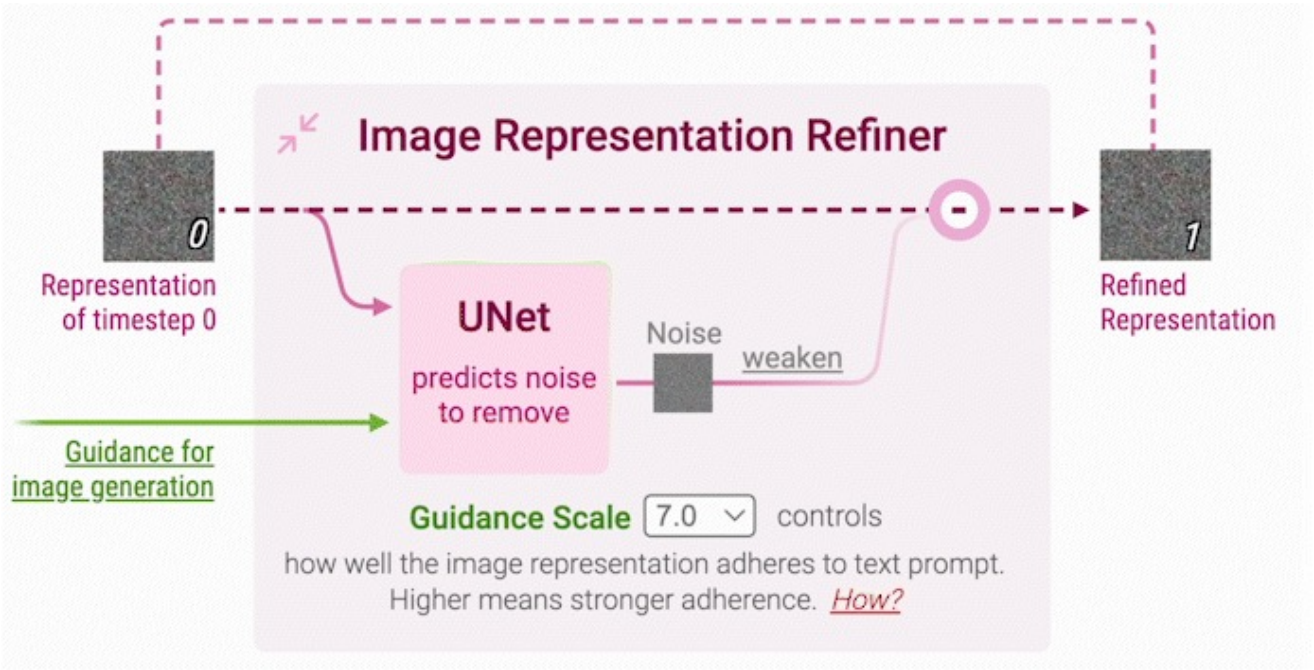
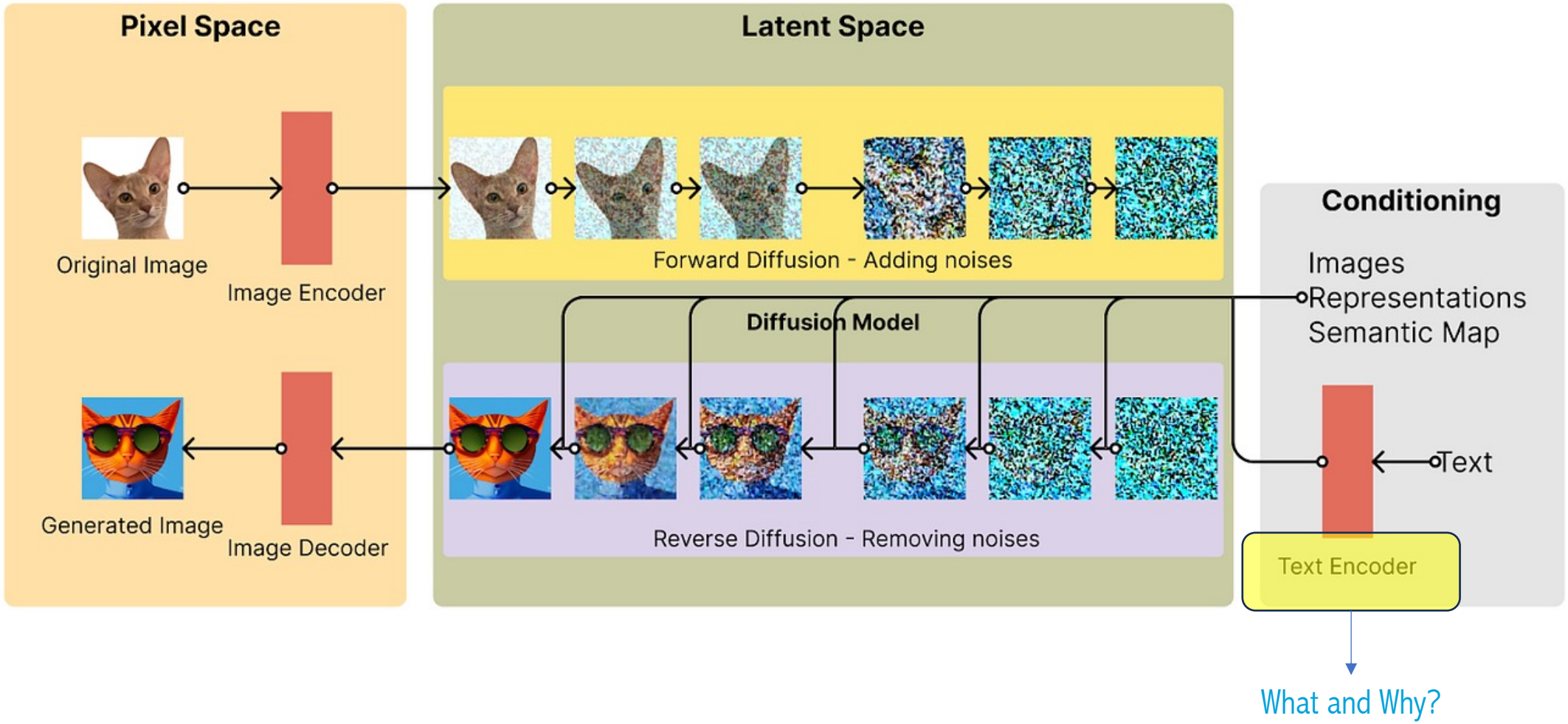


Figure 1. Boosting the upper bound on achievable quality with less agressive downsampling. Since diffusion models offer excellent inductive biases for spatial data, we do not need the heavy spatial downsampling of related generative models in latent space, but can still greatly reduce the dimensionality of the data via suitable autoencoding models, see Sec. 3. Images are from the DIV2K [1] validation set, evaluated at 512^2 px. We denote the spatial downsampling factor by f. Reconstruction FIDs [29] and PSNR are calculated on ImageNet-val. [12]; see also Tab. 8.

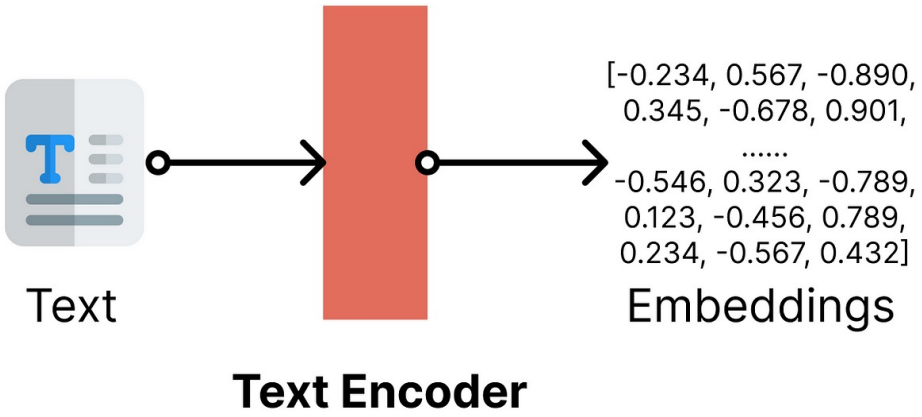
results in image synthesis [30,85] and beyond [7,45,48,57], and define the state-of-the-art in class-conditional image synthesis [15,31] and super-resolution [72]. Moreover, even unconditional DMs can readily be applied to tasks such as inpainting and colorization [85] or stroke-based synthesis [53], in contrast to other types of generative models [19,46,69]. Being likelihood-based models, they do not exhibit mode-collapse and training instabilities as GANs and, by heavily exploiting parameter sharing, they can model highly complex distributions of natural images with-



Stable Diffusion: Motivation



Text encoder: text input to embeddings



Imagine you want to have a foreign artist draw a painting for you but you don't speak their language, you'll probably use Google Translate or a human translator to translate what you want to them. It's the same with image-generation models — machine learning models don't understand text directly, so they need a text encoder to translate your text instructions into numbers that they can understand

These numbers are not just random numbers — they're called text embeddings, which are high-dimensional vectors that can capture the semantic meaning of the texts (i.e. the relationship between words and their context).

Stable diffusion v1 uses CLIP, which is a GPT-based model by OpenAI. (The original stable diffusion research paper uses BERT, another transformer model; Stable diffusion V2 uses OpenClip, a larger version of CLIP)

What and Why?

CLIP (Contrastive Language–Image Pre-training)

A picture of a dog wearing a fabulous jacket

CLIP

Match: 2%

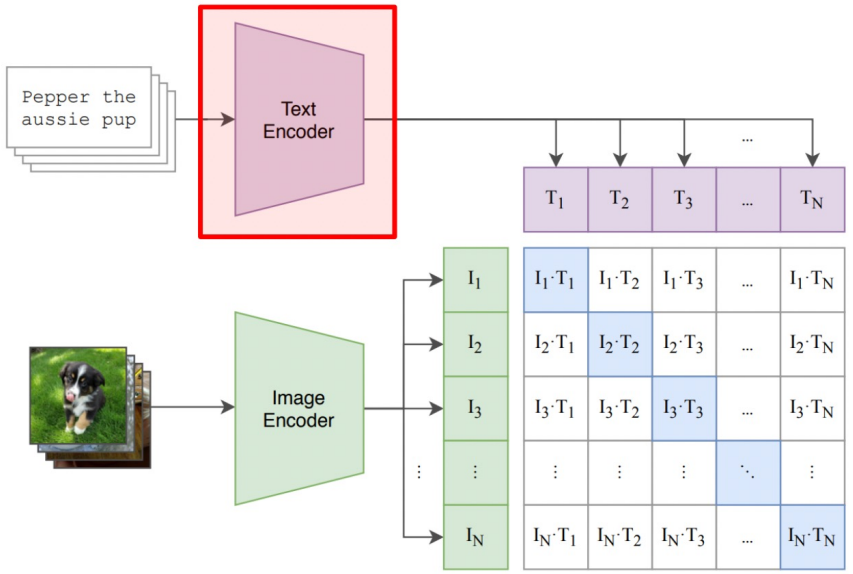
No Match: 98%

A picture of a dog wearing a fabulous jacket

CLIP

Match: 86%

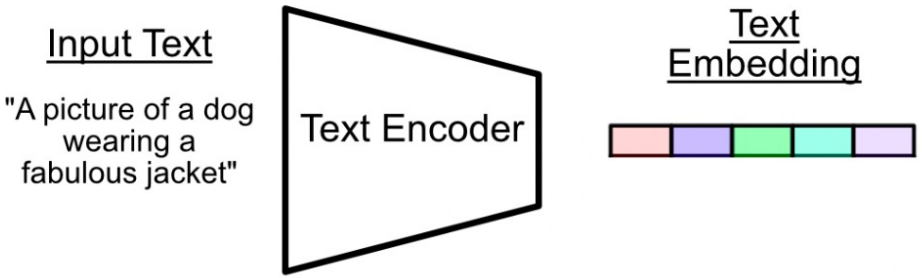
No Match: 14%



ClipText for text encoding.

Input: text.

Output: 77 token embeddings vectors, each in 768 dimensions.



Stable Diffusion: Motivation

Stable Diffusion is a **latent diffusion model**, in which we don't learn the distribution p(x) of our data set of images, but rather, the distribution of a latent representation of our data by using a Variational Autoencoder.

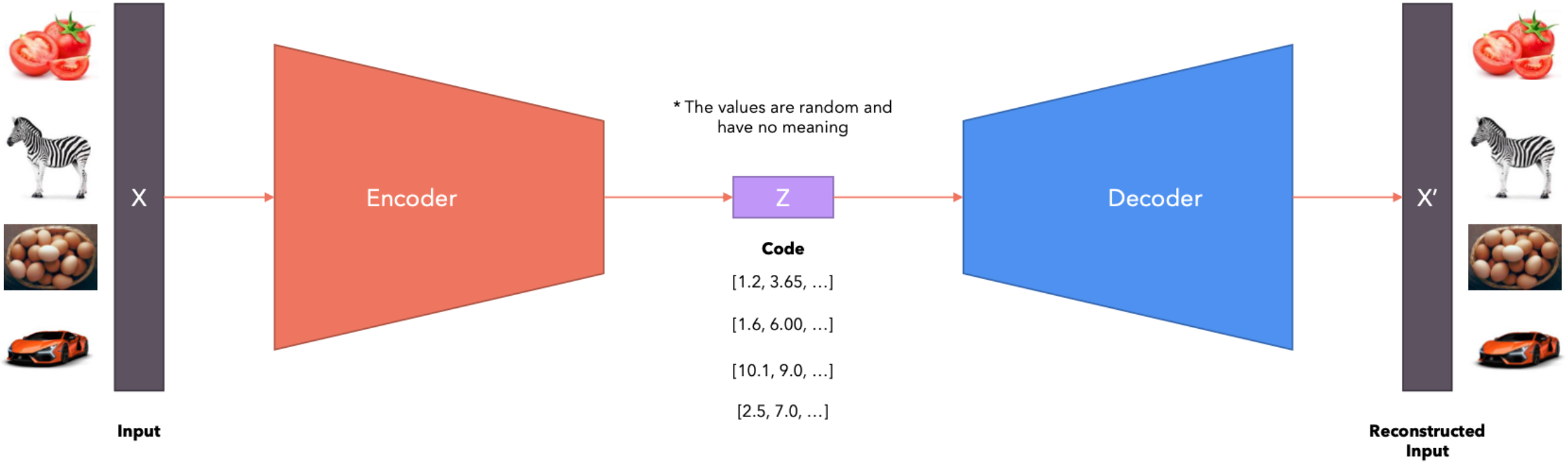
What and Why?

This allows us to reduce the computation we need to perform the steps needed to generate a sample, because each data will not be represented by a 512x512 image, but its latent representation, which is 64x64

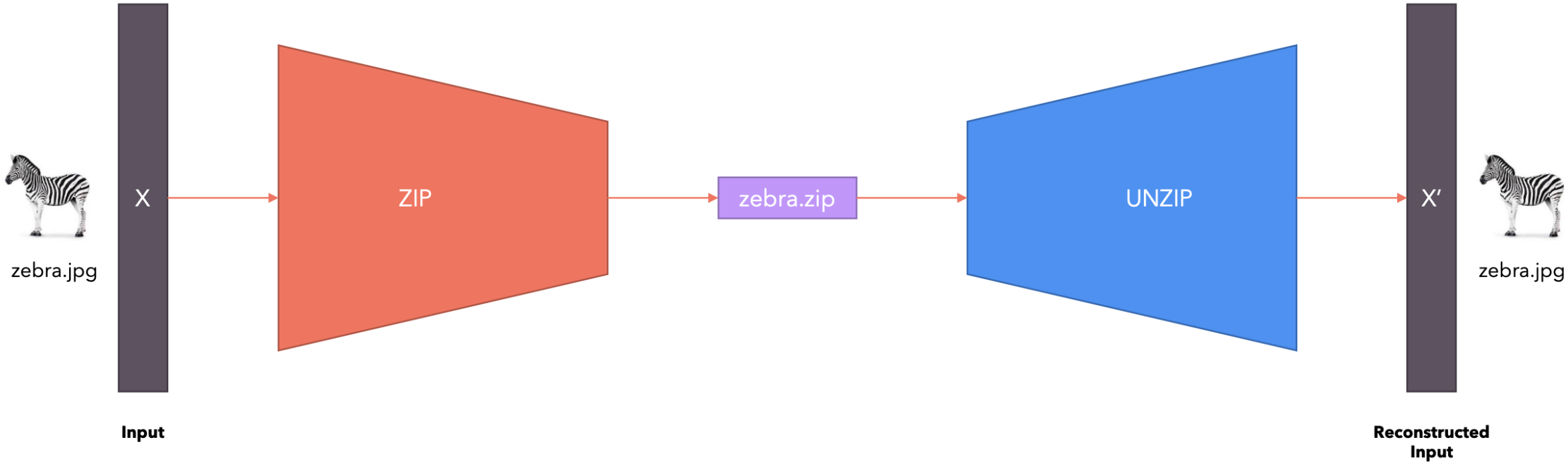
High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

What is Auto-encoder

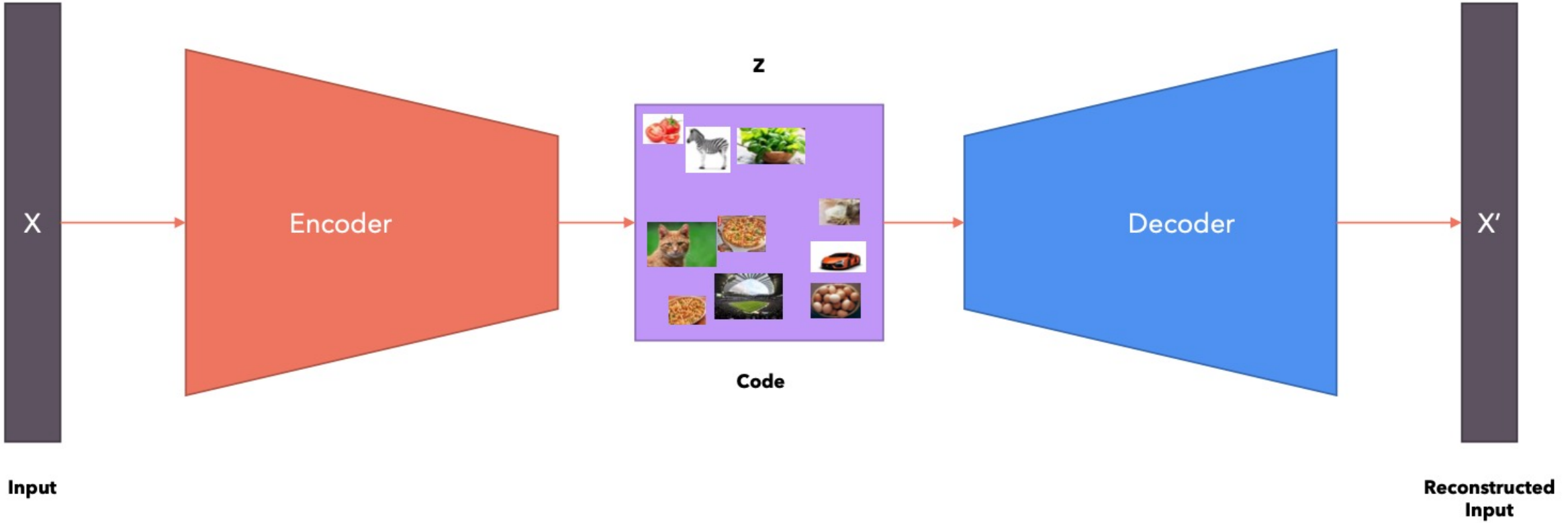


What is Auto-encoder



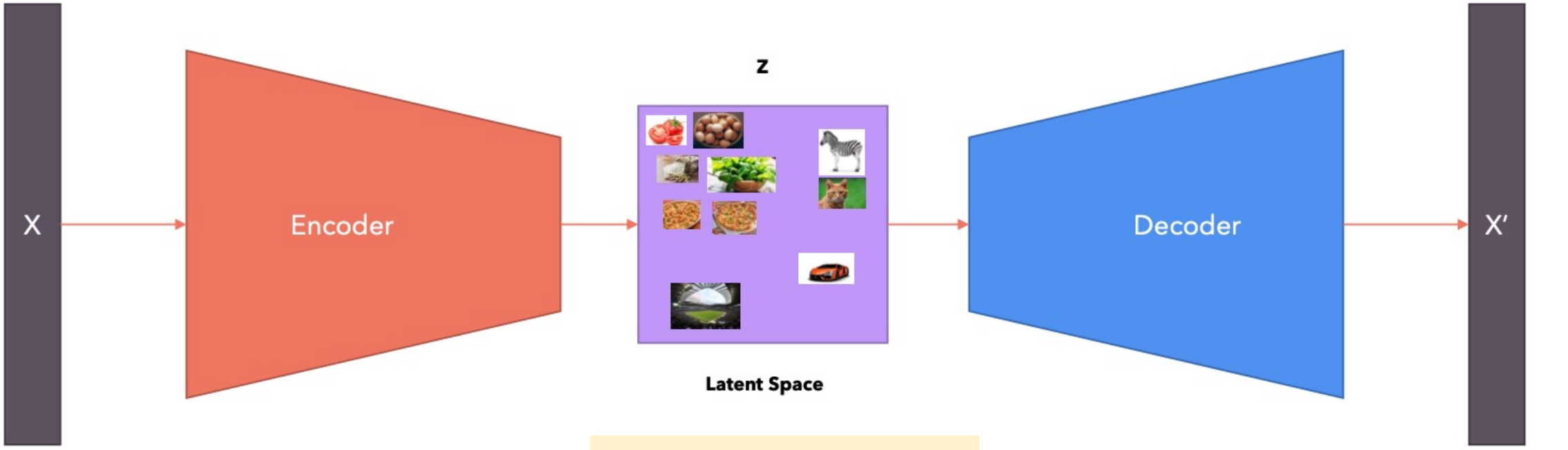
Anology with file compression

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

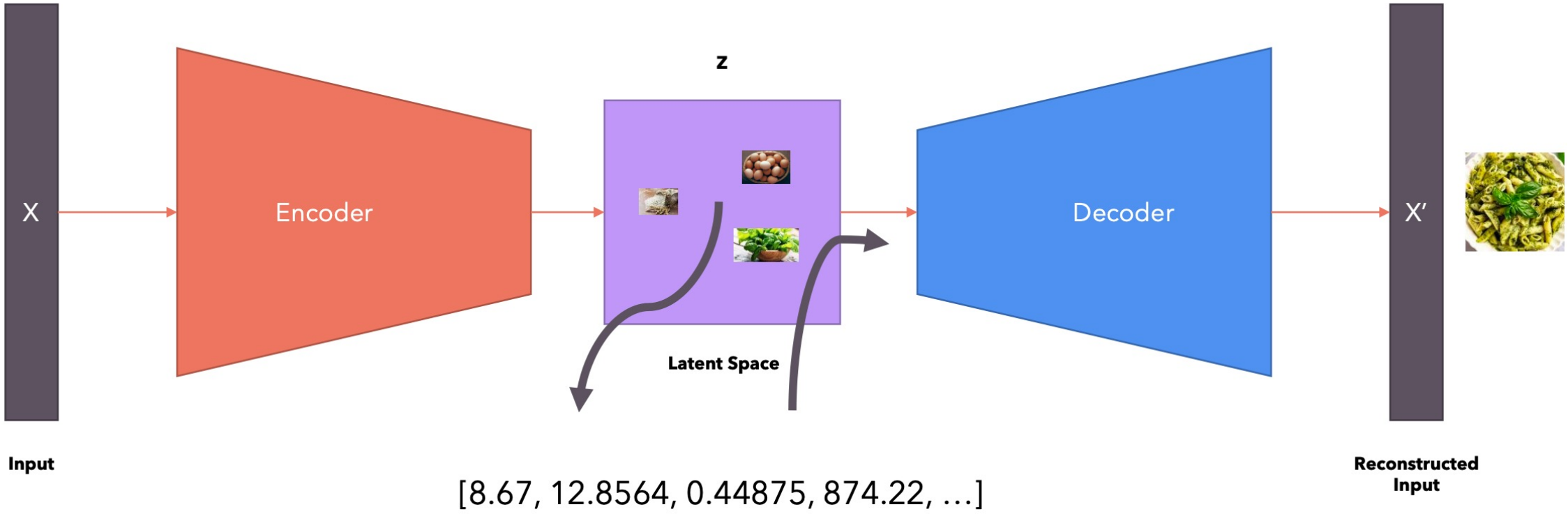


Sampling the latent space. Why?

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

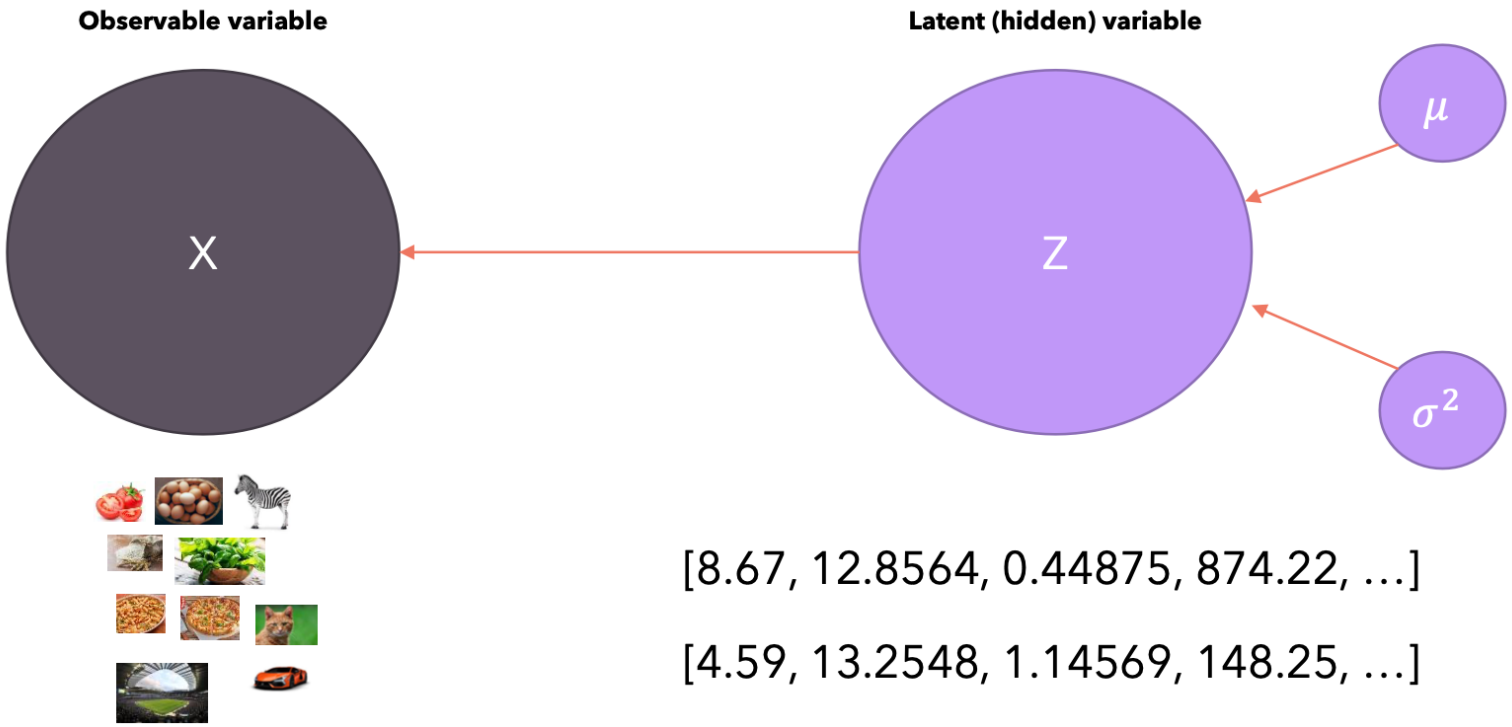
Umar Jamil - https://github.com/hkproj/vae-from-scratch-notes

Sampling the latent space



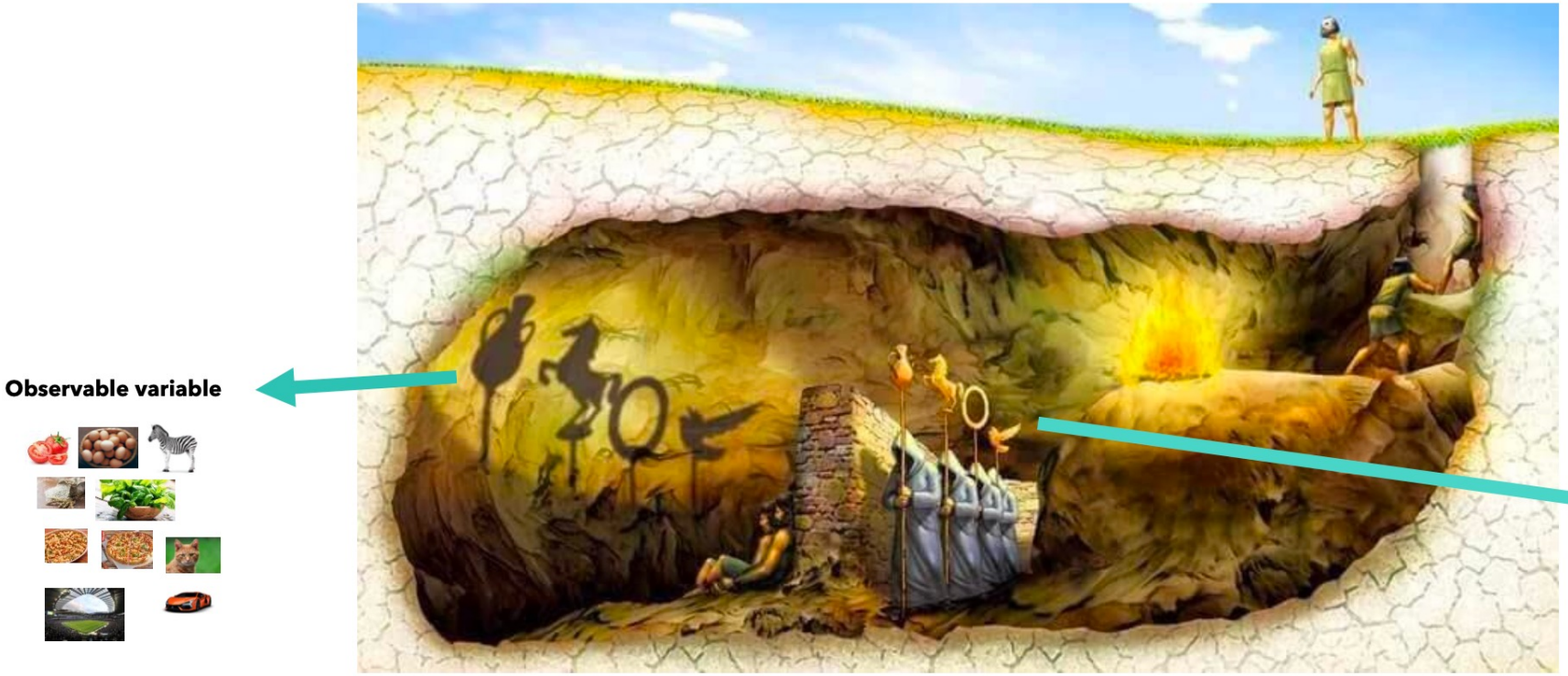
Just like when you use Python to generate a random number between 1 and 100, you're sampling from a uniform (pseudo)random distribution between 1 and 100. In the same way, we can sample from the latent space in order to generate a random vector, give it to the decoder and generate new data.

Why is it called latent space?



Why is it called latent space?

Why is it called latent space?



Plato's allegory of the cave

Basic Math Concepts

Expectation of a random variable

$$E_x[f(x)] = \int xf(x)dx$$

Chain rule of probability

$$P(x, y) = P(x|y)P(y)$$

Bayes' Theorem

$$P(x \mid y) = \frac{P(y|x)P(x)}{P(y)}$$

Math is fun

Kullback-Leibler Divergence

$$D_{KL}(P\|Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

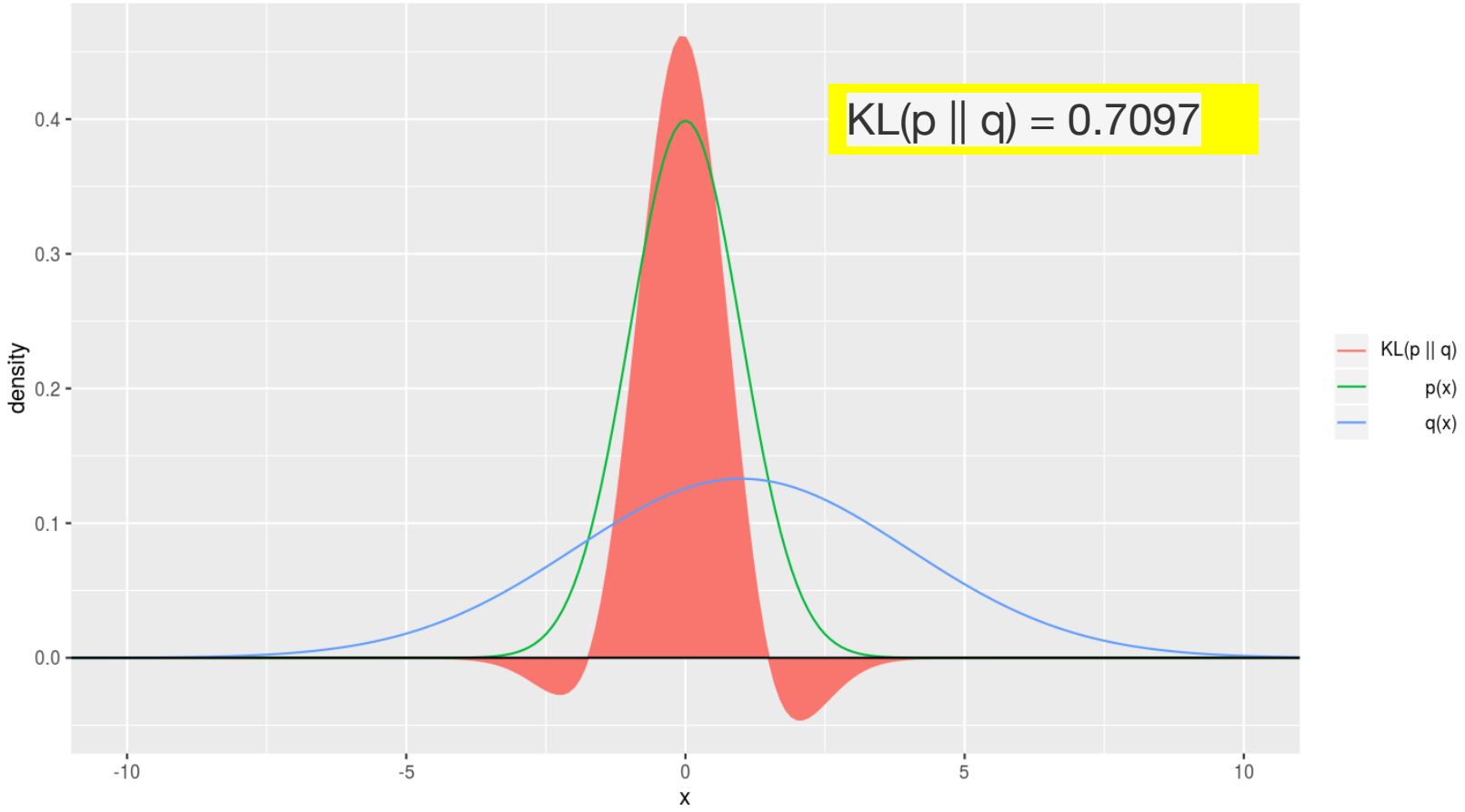
Properties:

- Not symmetric.
- Always ≥ 0
- It is equal to 0 if and only if $P = Q$

KL allows you to measure the distance between two probability distributions

p(x): mean = 0 & std = 1

q(x): mean = 1 & std = 3



Kullback-Leibler Divergence

$$D_{KL}(P\|Q) = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

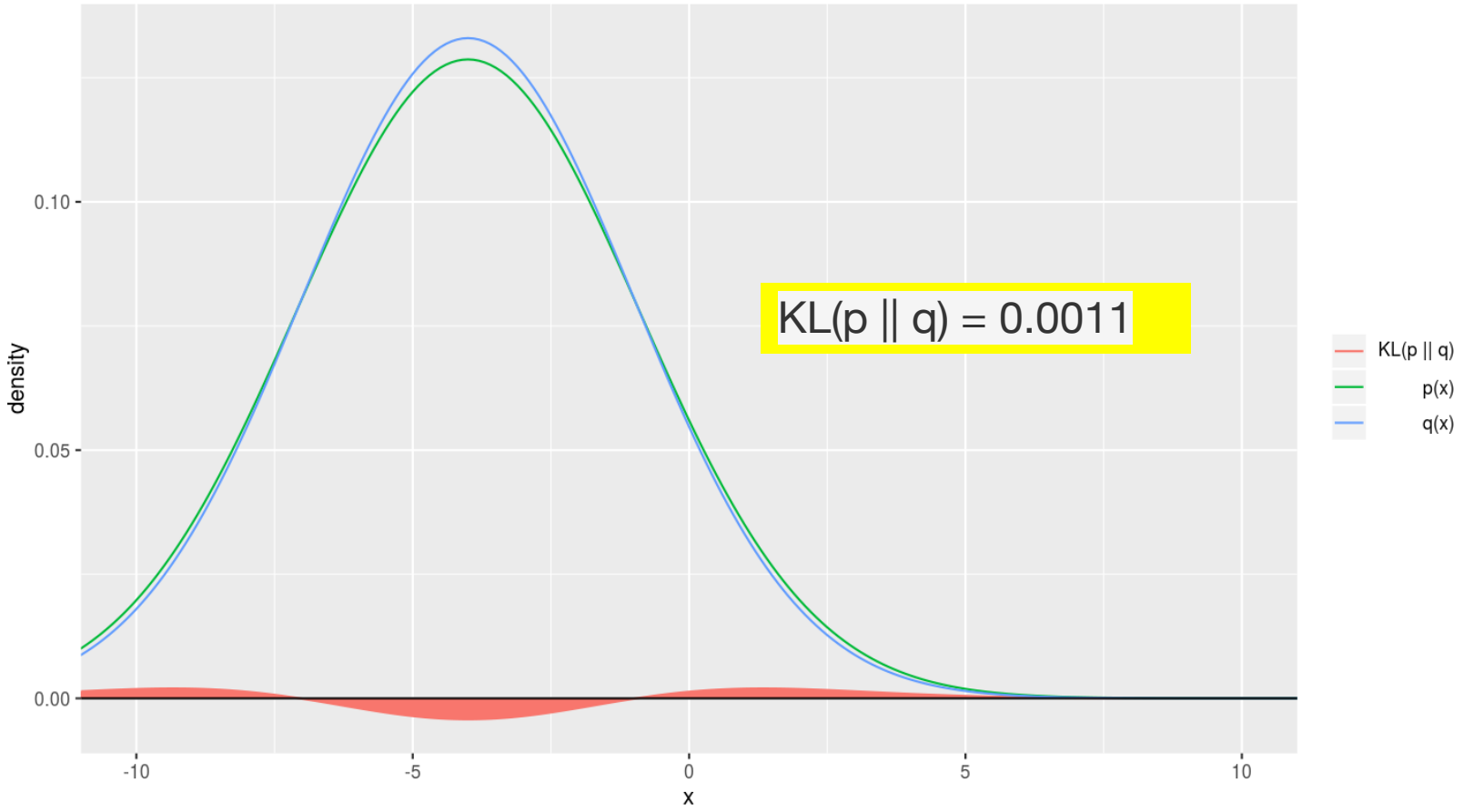
Properties:

- Not symmetric.
- Always ≥ 0
- It is equal to 0 if and only if $P = Q$

KL allows you to measure the distance between two probability distributions

p(x): mean = -4 & std = 3.1

q(x): mean = -4 & std = 3



Overview of the Model

We can define the likelihood of our data as the marginalization over the joint probability with respect to the latent variable

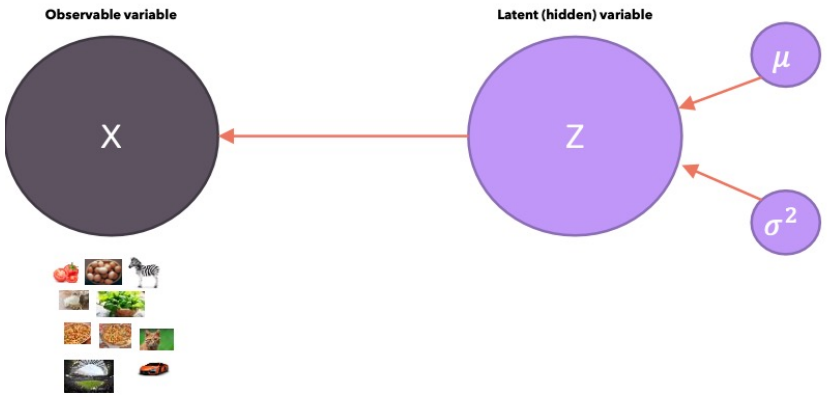
$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z}$$

Is intractable because we would need to evaluate this integral over all latent variables Z.

we can use the Chain rule of probability

$$p(\boldsymbol{x}) = \frac{p(\boldsymbol{x}, \boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})}$$

We don't have a ground truth $p(\boldsymbol{z} \mid \boldsymbol{x})$ … which is also what we're trying to find!



Intractable problem = a problem that can be solved in theory (e.g. given large but finite resources, especially time), but for which in practice any solution takes too many resources to be useful, is known as an intractable problem.

A chicken and egg problem

In order to have a tractable $p(\boldsymbol{x})$ we need a tractable $p(\boldsymbol{z}|\boldsymbol{x})$

$$p(\boldsymbol{x}) = \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})}$$

$$p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{x})}$$

In order to have a tractable $p(\boldsymbol{z}|\boldsymbol{x})$ we need a tractable $p(\boldsymbol{x})$

Question: When you can not find what you want?
Answer: We try to approximate it

Any Solution?

An **approximate** posterior.
Parametrized by φ.

$$p_\theta(\boldsymbol{z}|\boldsymbol{x}) \approx q_\varphi(\boldsymbol{z}|\boldsymbol{x})$$

Our **true** posterior (that we can't evaluate due to its intractability)
Parametrized by θ.

Math is fun

$$\log p_\theta(\boldsymbol{x}) = \log p_\theta(\boldsymbol{x})$$

$$= \log p_\theta(\boldsymbol{x}) \int q_\varphi(\boldsymbol{z}|\boldsymbol{x}) d\boldsymbol{z}$$

Multiply by 1

This is the integral over the domain of a probability distribution which is always equal to 1

$$= \int \log p_\theta(\boldsymbol{x})\, q_\varphi(\boldsymbol{z}|\boldsymbol{x}) d\boldsymbol{z}$$

Bring inside the integral

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x})]$$

Definition of expectation

$$E_x[f(x)] = \int x f(x) dx$$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right]$$

Apply the equation $p_\theta(\boldsymbol{x}) = \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}$

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z}) q_\varphi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x}) q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]$$

Multiply by 1

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] + E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}{p_\theta(\boldsymbol{z}|\boldsymbol{x})}\right]$$

Split the expectation

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] + D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x}) \| p_\theta(\boldsymbol{z}|\boldsymbol{x})\right)$$

Definition of KL divergence

≥ 0

Math is fun

Total Compensation = Base Salary + Bonus

≥ 0

employee

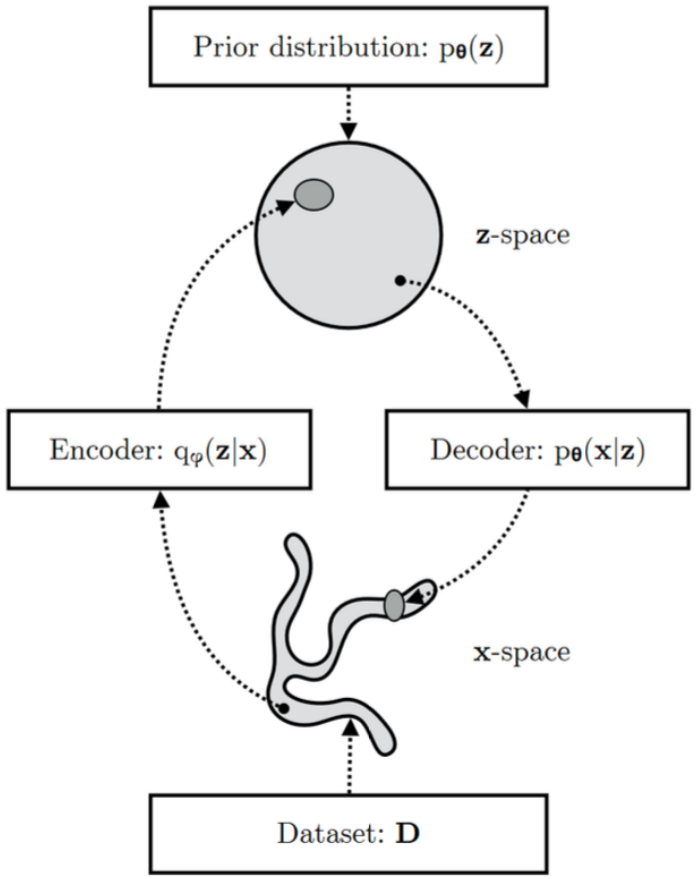
Total Compensation $\geq$ Base Salary

$$\log p_\theta(\boldsymbol{x}) = E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] + D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x}) \| p_\theta(\boldsymbol{z}|\boldsymbol{x})\right)$$

ELBO = Evidence Lower Bound **ELBO** ≥ 0

$$\log p_\theta(\boldsymbol{x}) \geq E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]$$

Math is fun



$$\log p_\theta(\boldsymbol{x}) \geq E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x},\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] = E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]$$ Chain rule of probability

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] + E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]$$ Split the expectation

$$= E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x})\|p_\theta(\boldsymbol{z})\right)$$ Definition of KL divergence

Profit = Revenue - Costs

Maximizing the ELBO means:

1. Maximizing the first term: maximizing the reconstruction likelihood of the decoder
2. Minimizing the second term: minimizing the distance between the learned distribution and the prior belief we have over the latent variable.

Maximizing the ELBO

When we have a function we want to maximize, we usually take the gradient and adjust the weights of the model so that they move along the gradient direction.

When we have a function we want to minimize, we usually take the gradient, and adjust the weights of the model so that they move against the gradient direction.

Stochastic Gradient Descent

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations:

$$w := w - \eta \nabla Q(w) = w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(w),$$

where η is a step size (sometimes called the *learning rate* in machine learning).

$$L(\theta, \varphi, \boldsymbol{x}) = E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] = E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - D_{KL}\left(q_\varphi(\boldsymbol{z}|\boldsymbol{x}) \| p_\theta(\boldsymbol{z})\right)$$

Maximizing the ELBO

ELBO

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) = -D_{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})||p_{\boldsymbol{\theta}}(\mathbf{z})) + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})\right] \quad (3)$$

We want to differentiate and optimize the lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)})$ w.r.t. both the variational parameters $\boldsymbol{\phi}$ and generative parameters $\boldsymbol{\theta}$. However, the gradient of the lower bound w.r.t. $\boldsymbol{\phi}$ is a bit problematic. The usual (naïve) Monte Carlo gradient estimator for this type of problem is: $\nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}\left[f(\mathbf{z})\right] = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}\left[f(\mathbf{z})\nabla_{q_{\boldsymbol{\phi}}(\mathbf{z})}\log q_{\boldsymbol{\phi}}(\mathbf{z})\right] \simeq \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z})\nabla_{q_{\boldsymbol{\phi}}(\mathbf{z}^{(l)})}\log q_{\boldsymbol{\phi}}(\mathbf{z}^{(l)})$ where $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})$. This gradient estimator exhibits exhibits very high variance (see e.g. [BJP12]) and is impractical for our purposes.

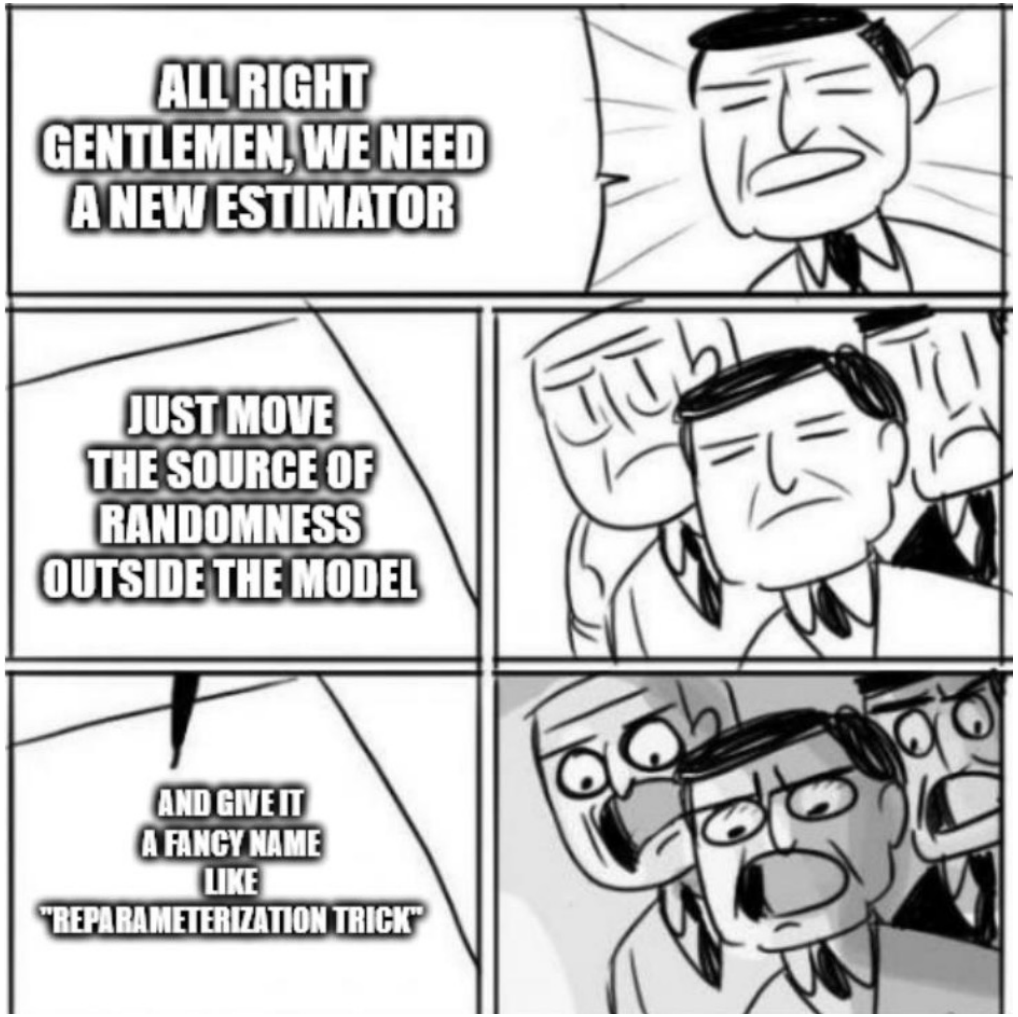
SCORE estimator

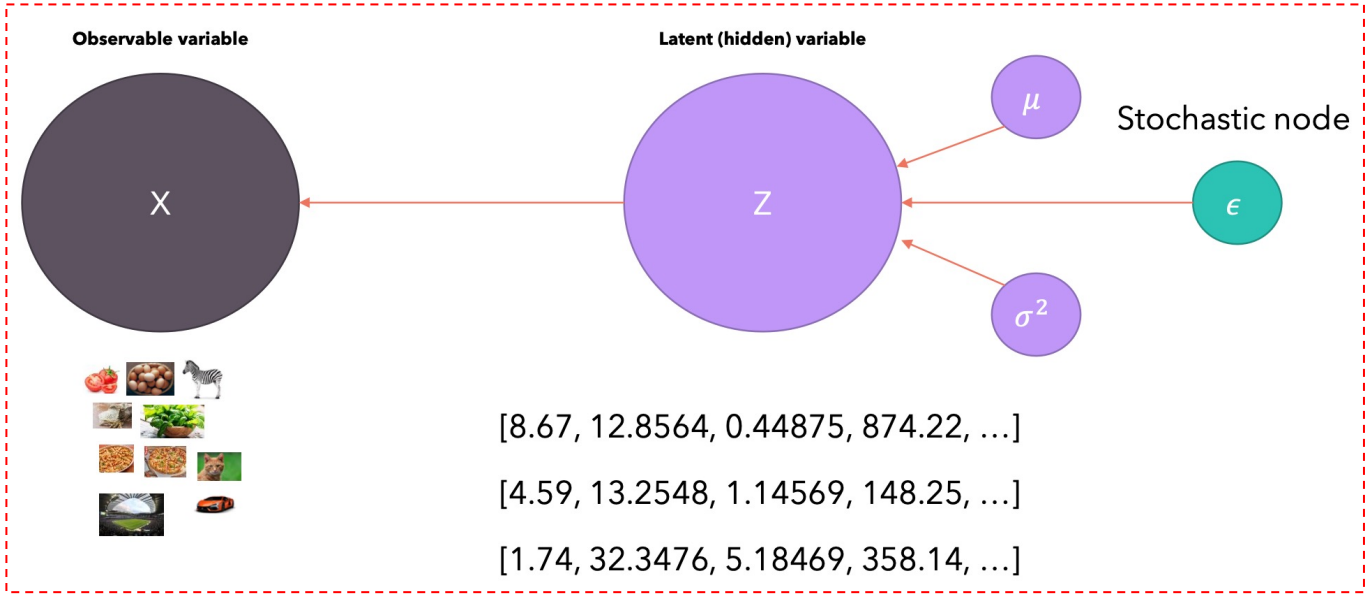
Kingma, D.P. and Welling, M., 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

This estimator is **unbiased**, meaning that even if at every step it may not be equal to the true expectation, on average it will converge to it, but as it is stochastic, it also has a variance and it happens to be high for practical use. Plus, we can't run backpropagation through it!

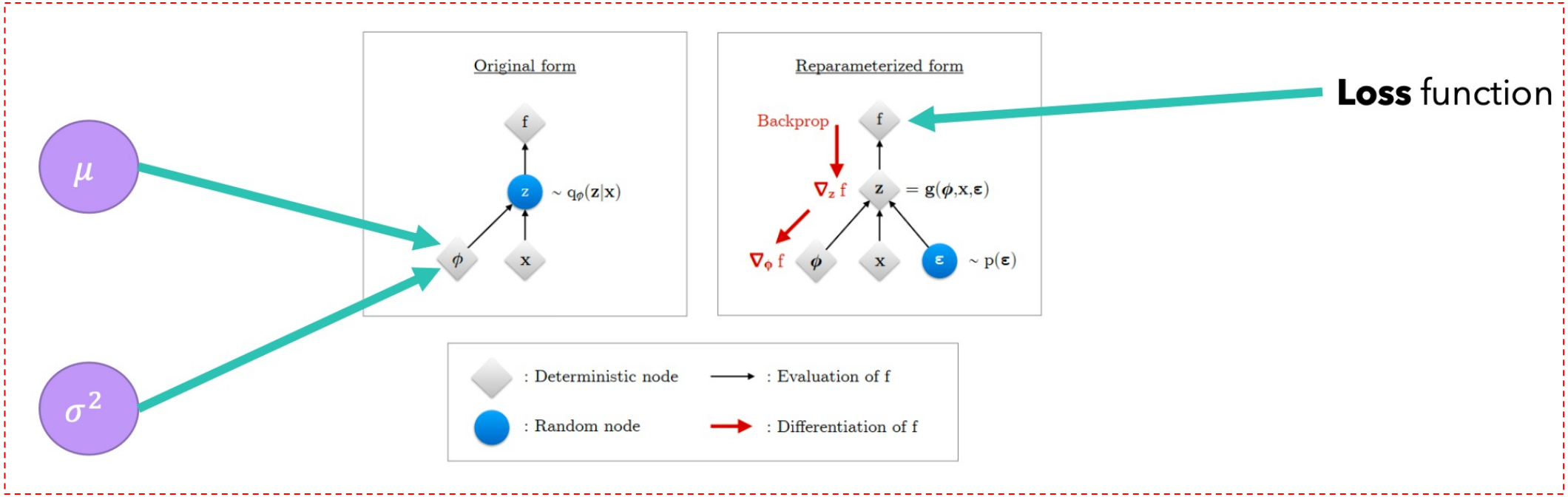
We need a new estimator

The reparameterization trick





The reparameterization trick



A new estimator

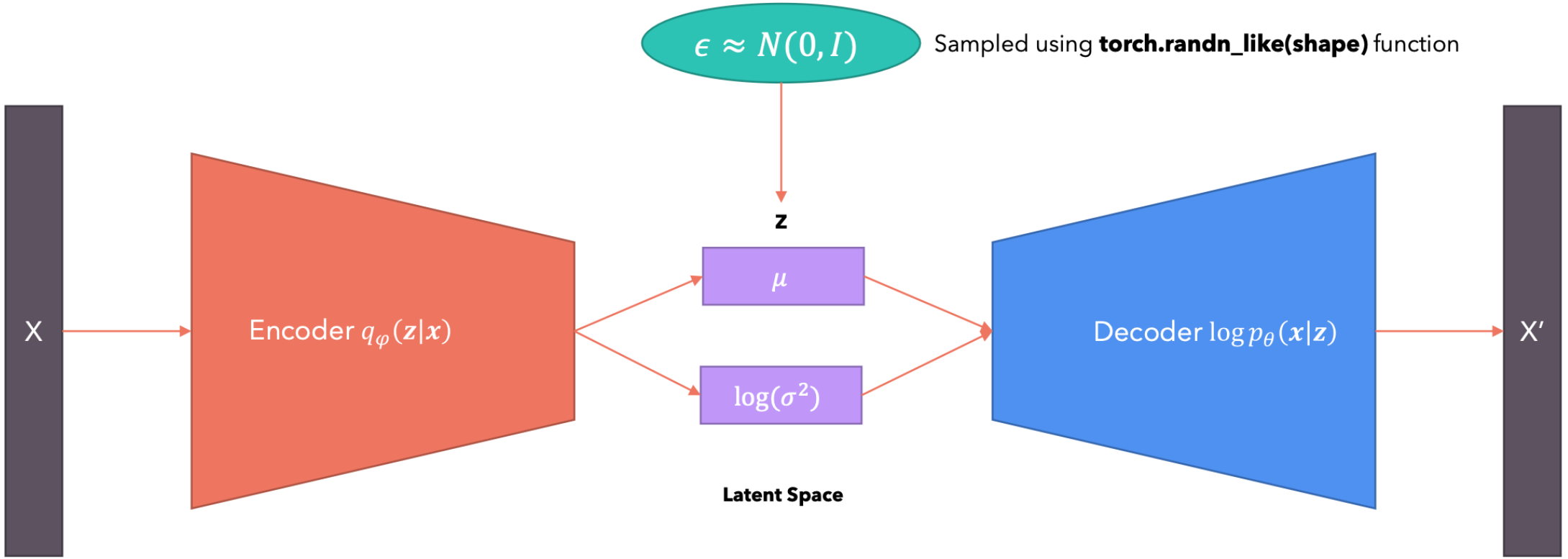
$$L(\theta, \varphi, \boldsymbol{x}) = \underbrace{E_{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]}_{\text{ELBO}} = E_{p(\epsilon)}\left[\log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right]$$

$$\epsilon \approx p(\epsilon)$$

$$z = g(\varphi, x, \epsilon)$$

$$E_{p(\epsilon)}\left[\log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}\right] \cong \tilde{L}(\theta, \varphi, \boldsymbol{x}) = \log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\varphi(\boldsymbol{z}|\boldsymbol{x})}$$

VAE: Example Network



We prefer learning $\log(\sigma^2)$ because it can be negative, so the model doesn't need to be forced to produce only positive values for it.

Stable Diffusion: Example

Paradise
Cosmic Beach

Input

Output

Pirate Ship

Input

Output

Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

The Components of Stable Diffusion

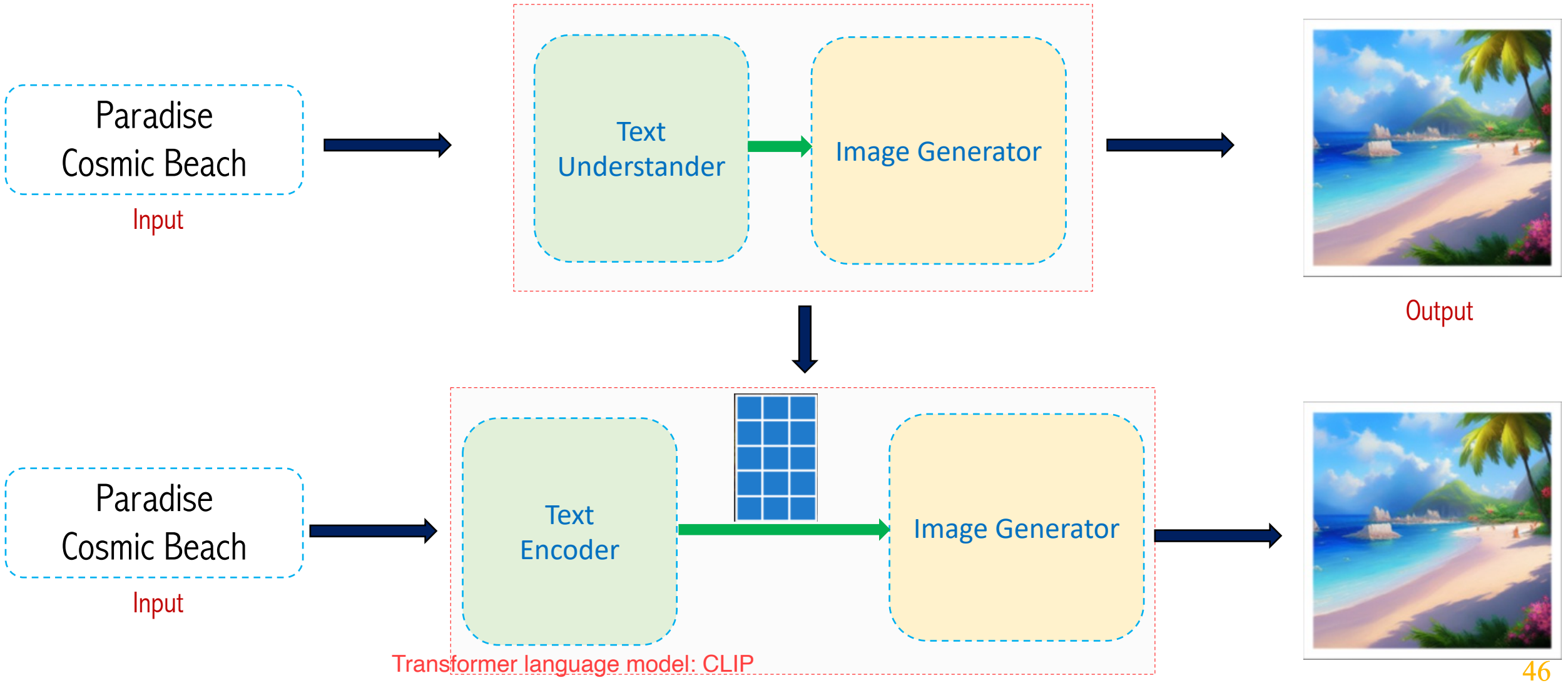
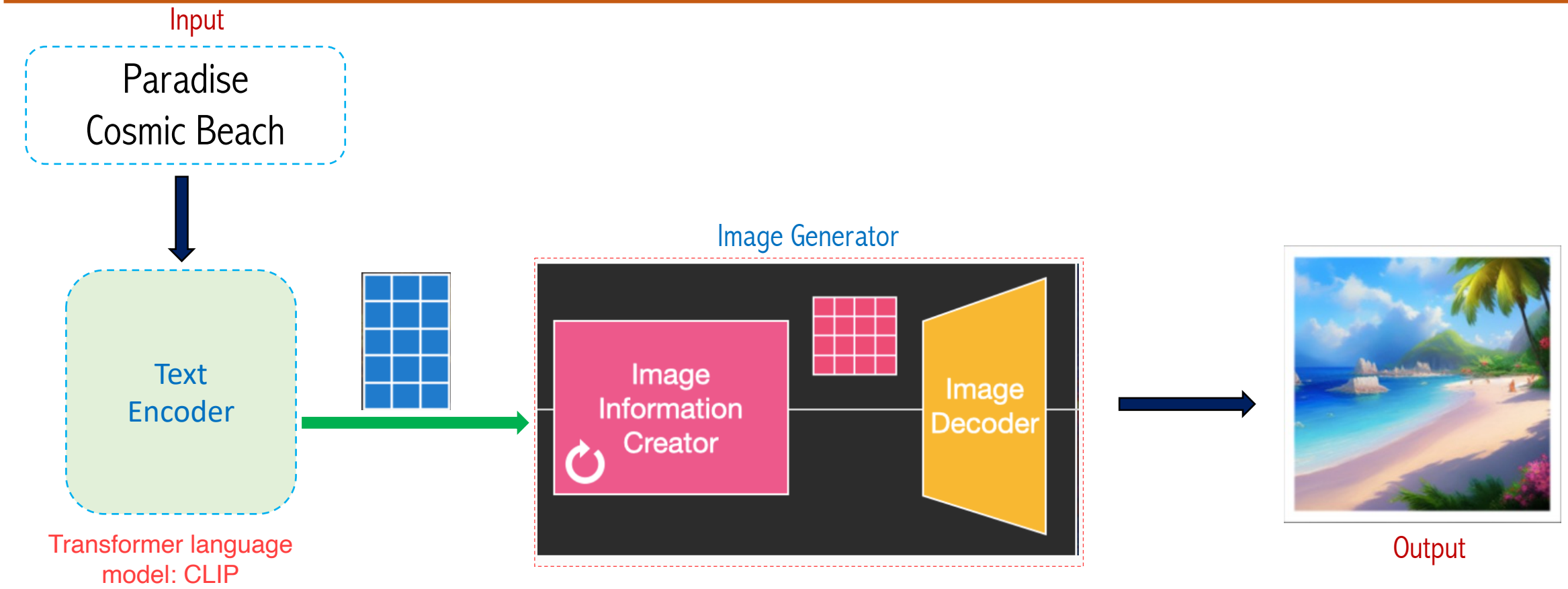
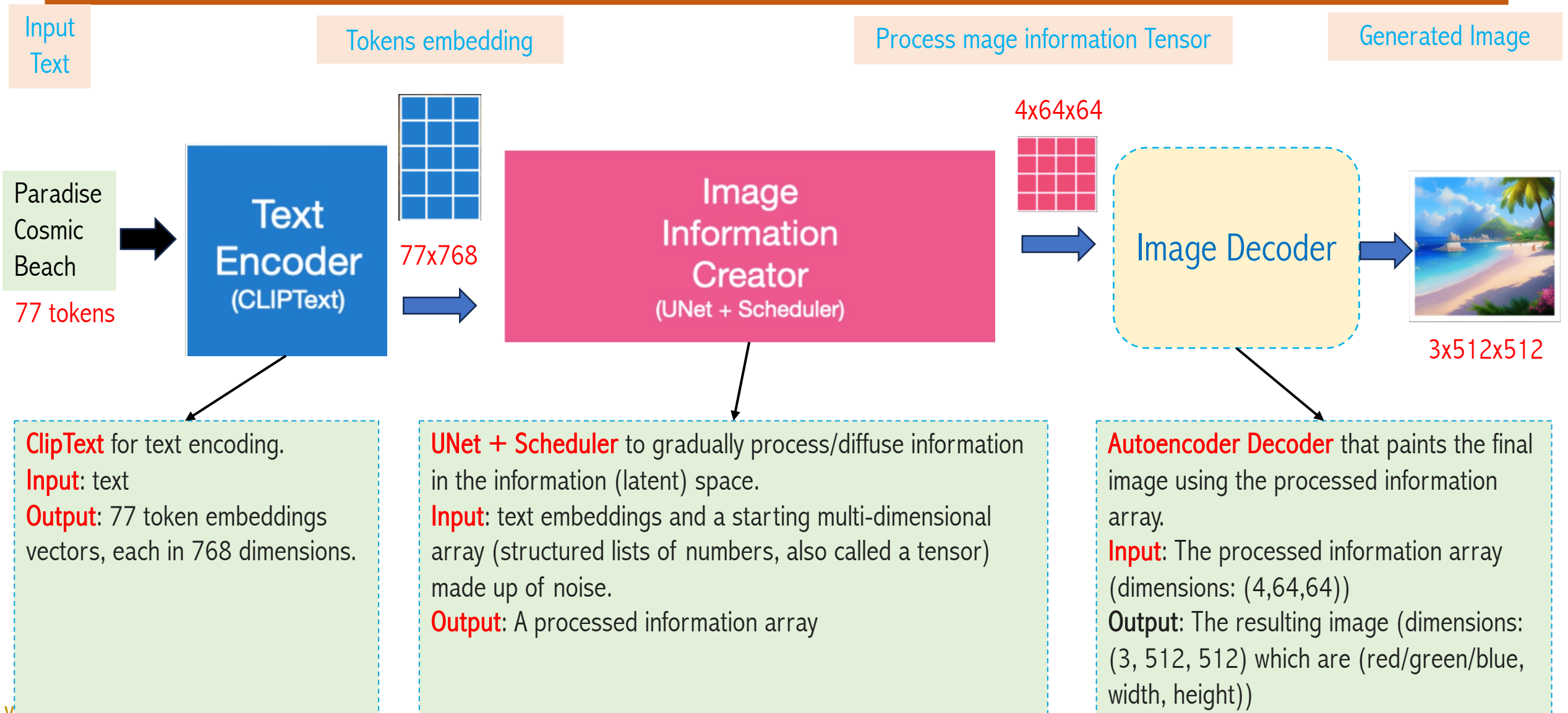


Image Generator



AI VIETNAM
All-in-One Course

Stable Diffusion: Model



ClipText for text encoding.
Input: text
Output: 77 token embeddings vectors, each in 768 dimensions.

UNet + Scheduler to gradually process/diffuse information in the information (latent) space.
Input: text embeddings and a starting multi-dimensional array (structured lists of numbers, also called a tensor) made up of noise.
Output: A processed information array

Autoencoder Decoder that paints the final image using the processed information array.
Input: The processed information array (dimensions: (4,64,64))
Output: The resulting image (dimensions: (3, 512, 512) which are (red/green/blue, width, height))

Stable Diffusion: Model

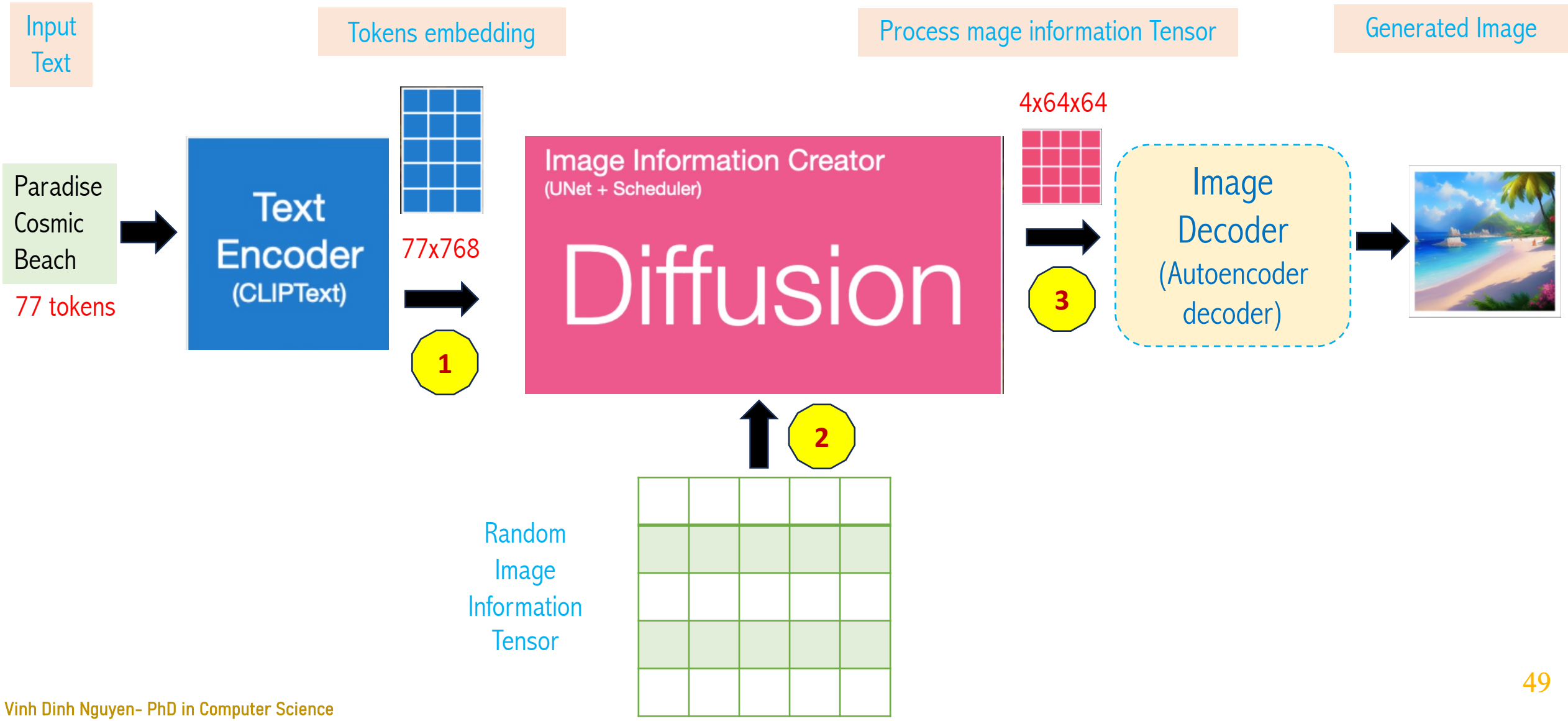
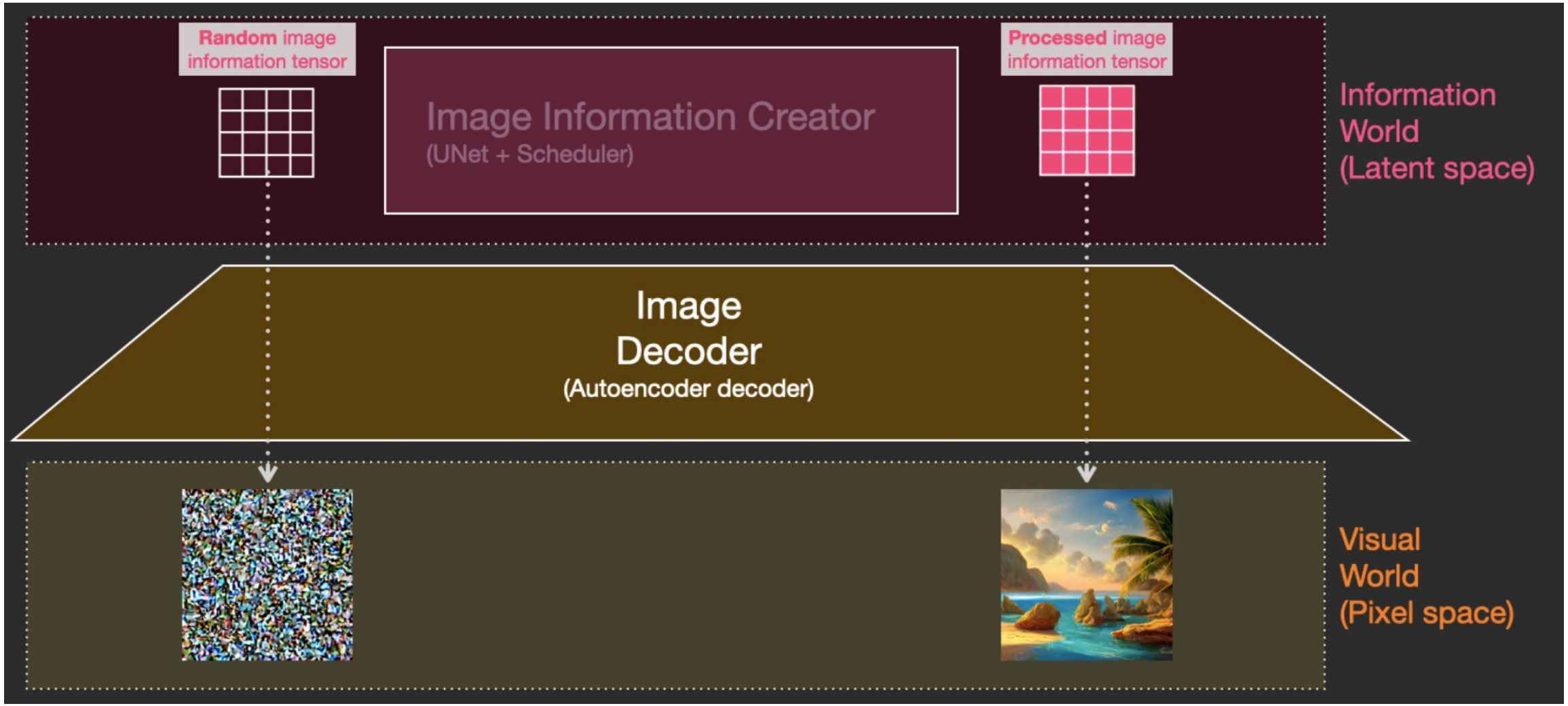
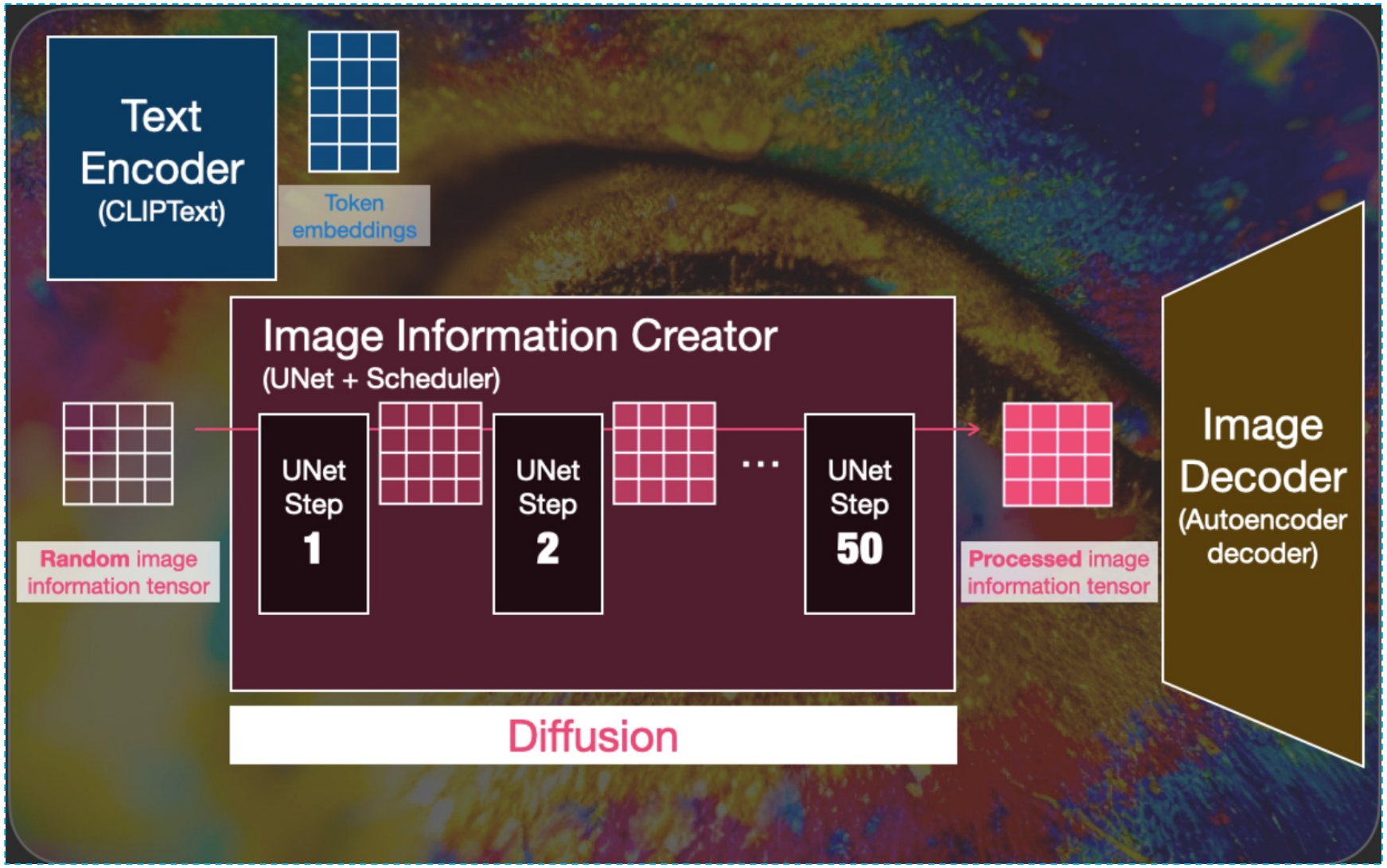


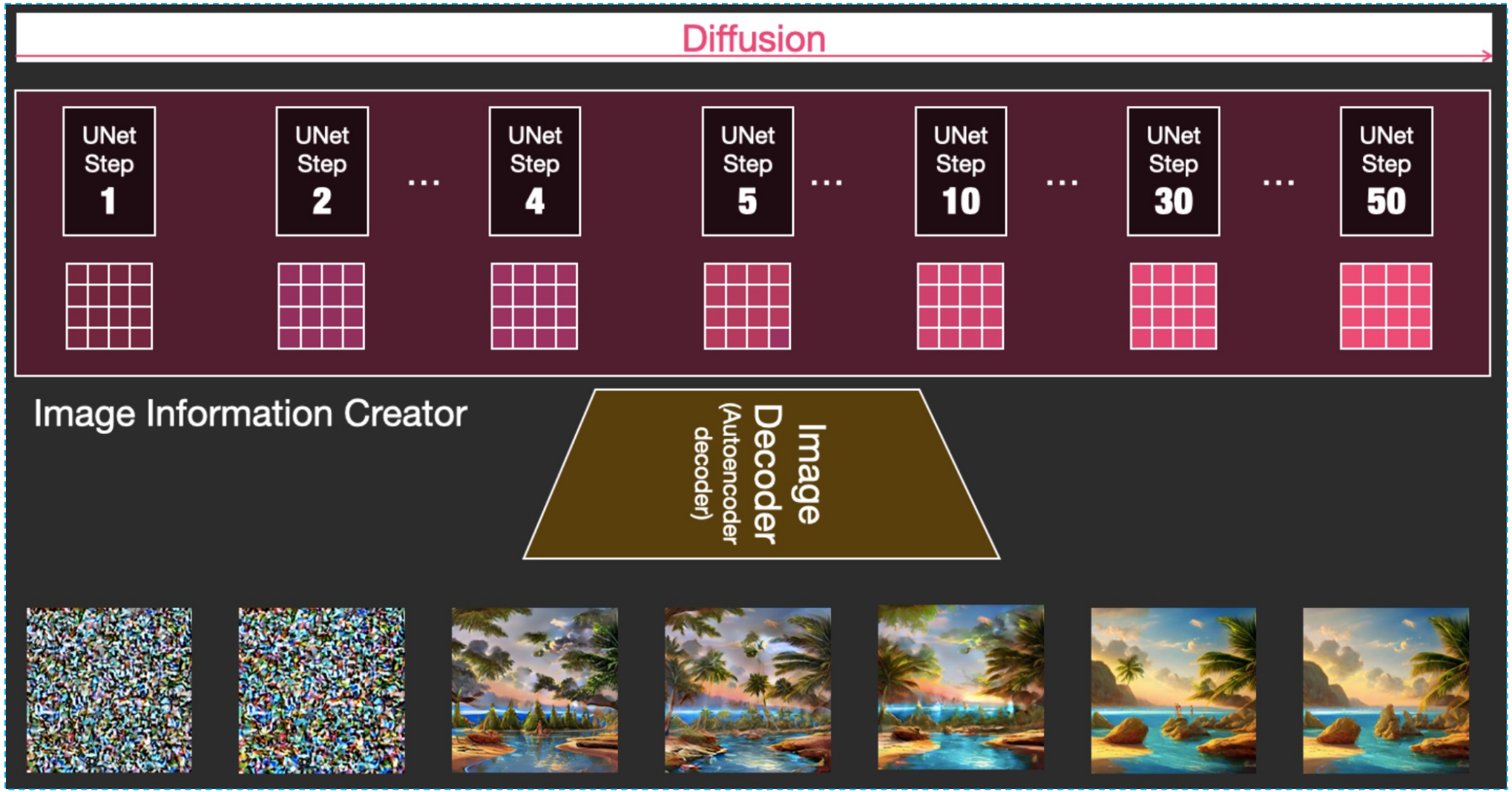
Image Decoder: Visualization



Diffusion Process: Where



Laten Feature: Visualization



Forward Diffusion: Image Generation

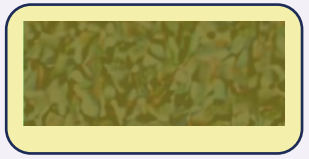
Pickup an image

Generate some random noise

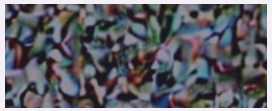
Pickup an amount of noise

0

1

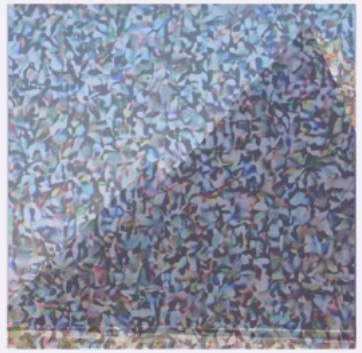


2



3

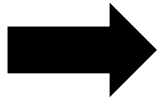
Add noise to the image in that amount



Diffusion Training: Noise Predictor

Noisy Amount	Noisy image
3	
14	
7	
42	
2	
21	

Noise Predictor
(UNet)



Noise sample






Diffusion Training: Noise Predictor

1 Pickup a training sample

Noisy Amount: 7

Noise Predictor
(UNet)

2 Predict the noise

UNet Prediction

3 Compare to actual noise

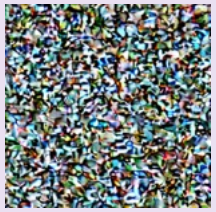
Actual noise(Label)

4 Update the Unet (back probagation)

Reverse Diffusion (Denoising)

Noise Amount: 2

Noise Predictor (UNet)



Predicted Noise sample 1

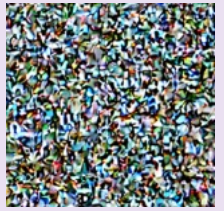
Subtract predicted noise from image

Slightly de-noised Image

Noise Amount: 1

Noise Predictor (UNet)

Subtract predicted nose from image



Predicted Noise sample 2

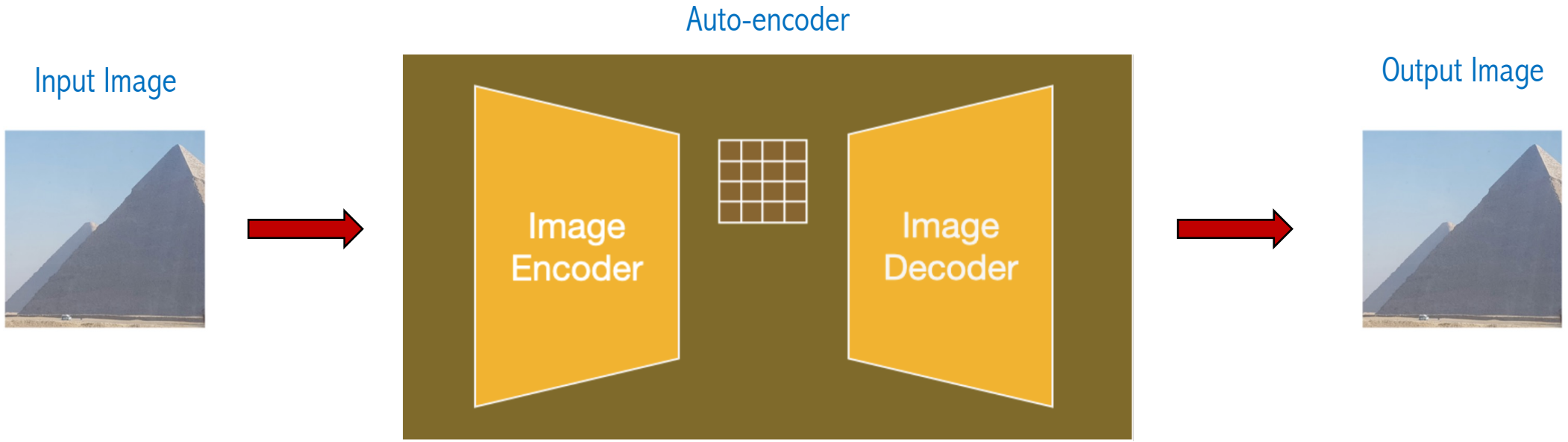
Generates images without using any text data

How text is incorporated in the process in order to control what type of image the model generates

Speed Boost

- Speed Boost: Diffusion on Compressed (Latent) Data Instead of the Pixel Image

The Stable Diffusion paper runs the diffusion process not on the pixel images themselves, but on a compressed version of the image



Forwarding Process

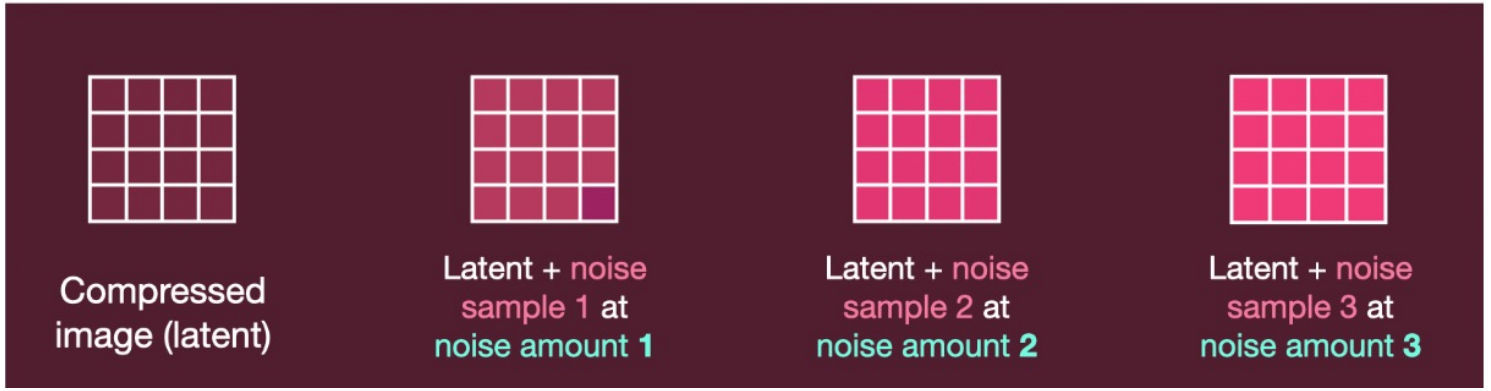
- Speed Boost: Diffusion on Compressed (Latent) Data Instead of the Pixel Image

The Stable Diffusion paper runs the diffusion process not on the pixel images themselves, but on a compressed version of the image

Input Image

Image Encoder

Generate training samples with different amounts of noise added to their compressed/latent version

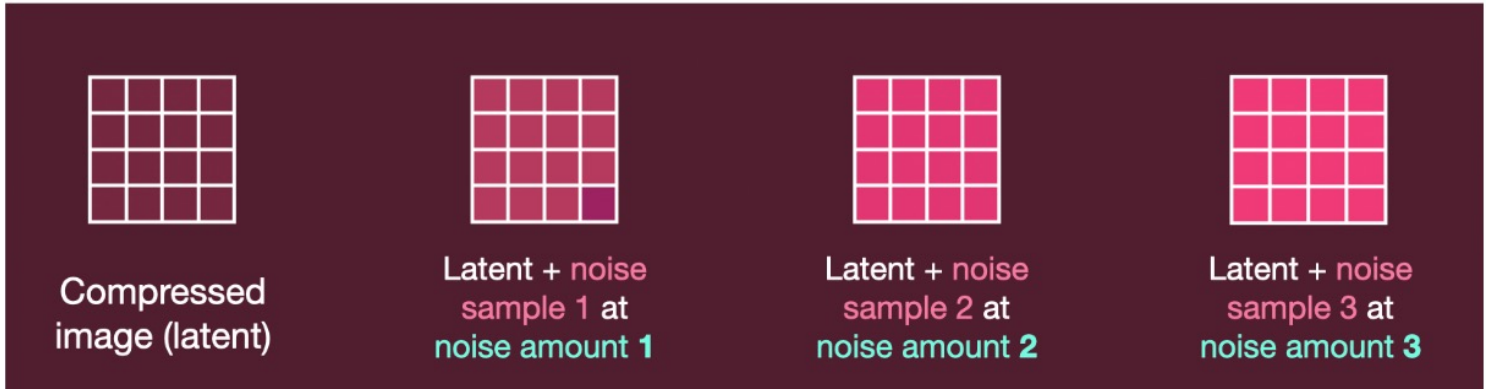


Reverse Process

Input Image

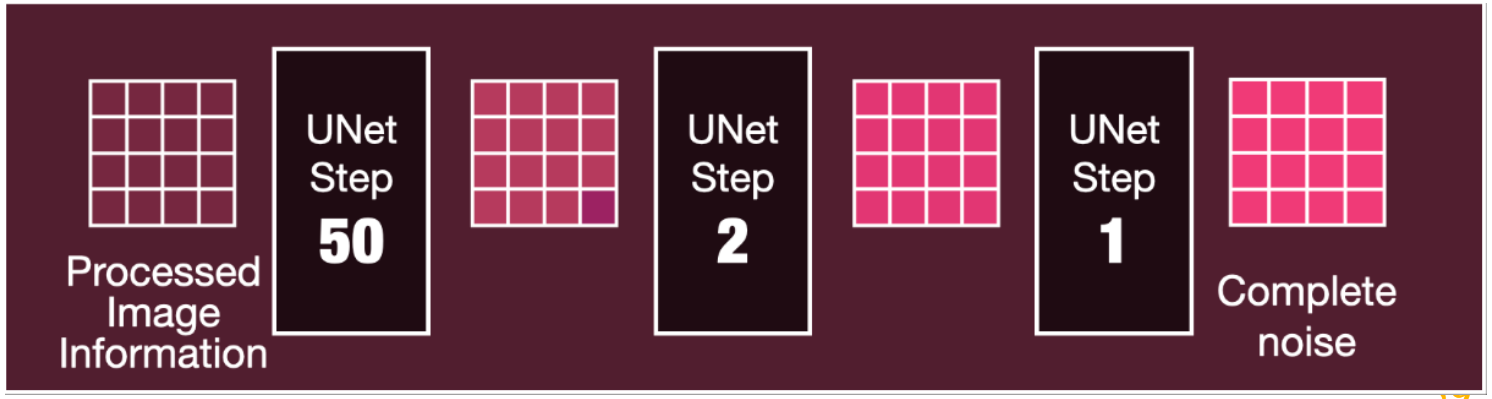
Image Encoder

Generate training samples with different amounts of noise added to their compressed/latent version



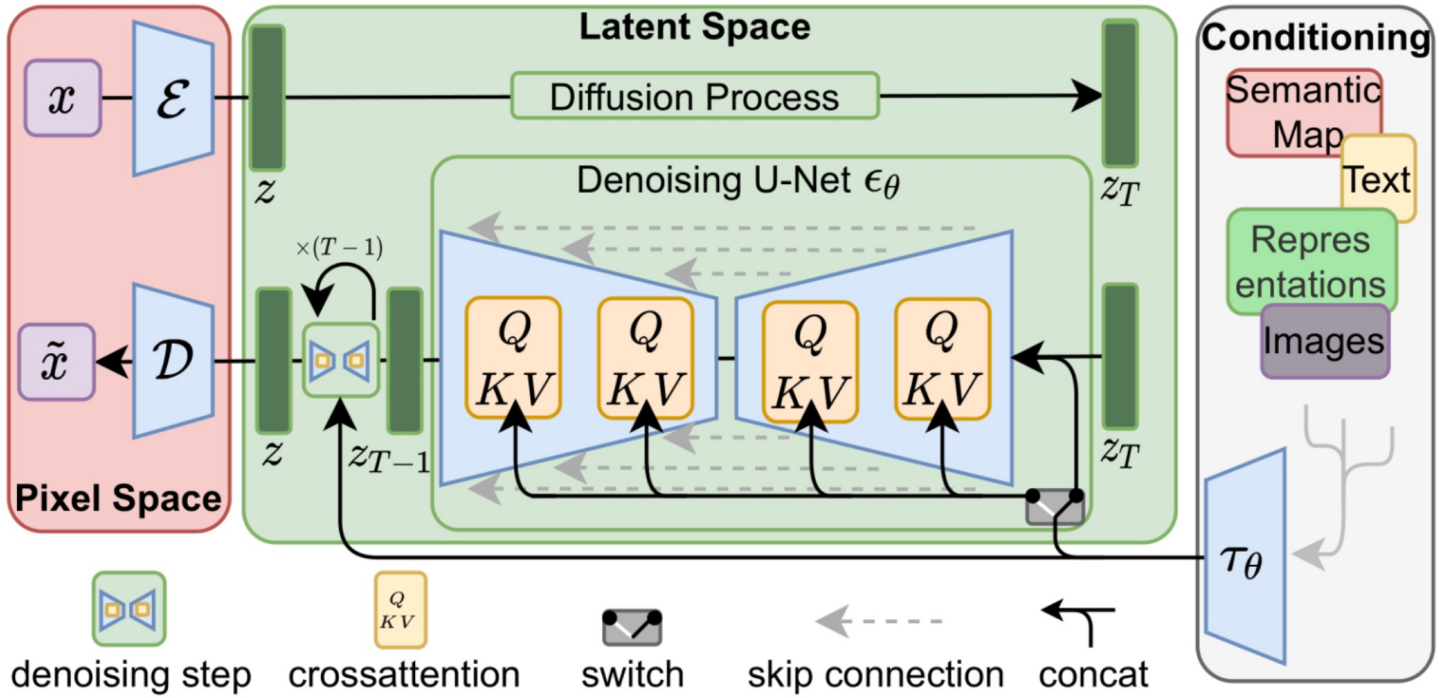
Generated Image

Image Decoder



LDM/Stable Diffusion

The text prompts describing what image the model should generate

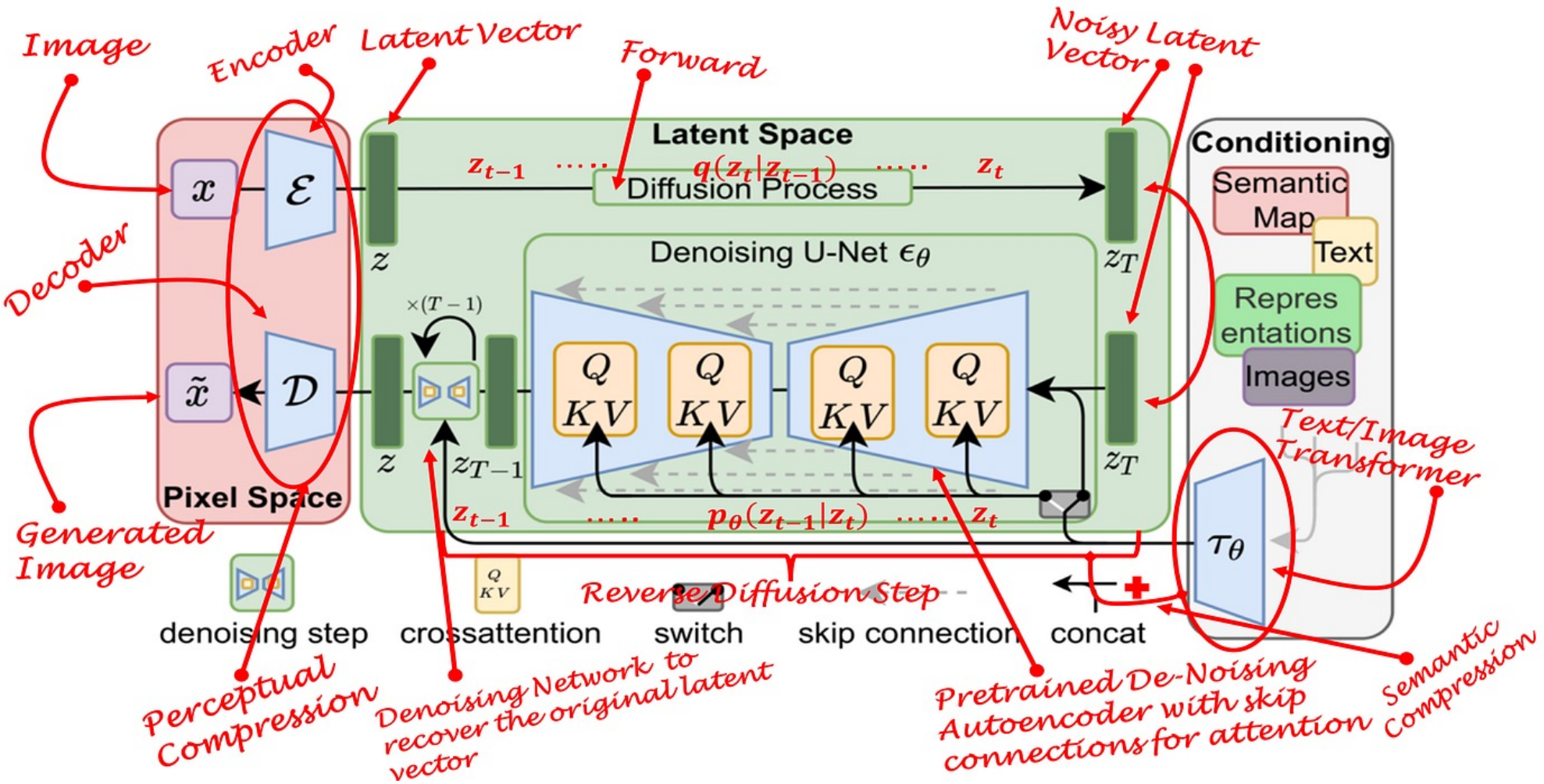


$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right) \cdot \mathbf{V}$$

where $\mathbf{Q} = \mathbf{W}_Q^{(i)} \cdot \varphi_i(\mathbf{z}_i)$, $\mathbf{K} = \mathbf{W}_K^{(i)} \cdot \tau_\theta(y)$, $\mathbf{V} = \mathbf{W}_V^{(i)} \cdot \tau_\theta(y)$

and $\mathbf{W}_Q^{(i)} \in \mathbb{R}^{d \times d_\epsilon^i}$, $\mathbf{W}_K^{(i)}, \mathbf{W}_V^{(i)} \in \mathbb{R}^{d \times d_\tau}$, $\varphi_i(\mathbf{z}_i) \in \mathbb{R}^{N \times d_\epsilon^i}$, $\tau_\theta(y) \in \mathbb{R}^{M \times d_\tau}$

LDM/Stable Diffusion



What is Self-Attention?

Self-Attention allows the model to relate words to each other.

In this simple case we consider the sequence length **seq** = 6 and $\mathbf{d_{model}} = \mathbf{d_k} = 512$.

The matrices **Q**, **K** and **V** are just the input sentence.

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$\text{softmax}\left(\frac{\mathbf{Q}\ (6, 512) \times \mathbf{K^T}\ (512, 6)}{\sqrt{512}}\right) =$$

	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

(6, 6)

* all values are random.

* for simplicity I considered only one head, which makes $d_{model} = d_k$.

What is Self-Attention?

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
A	0.210	0.128	0.206	0.212	0.119	0.125
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174
CAT	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6)

X

V

(6, 512)

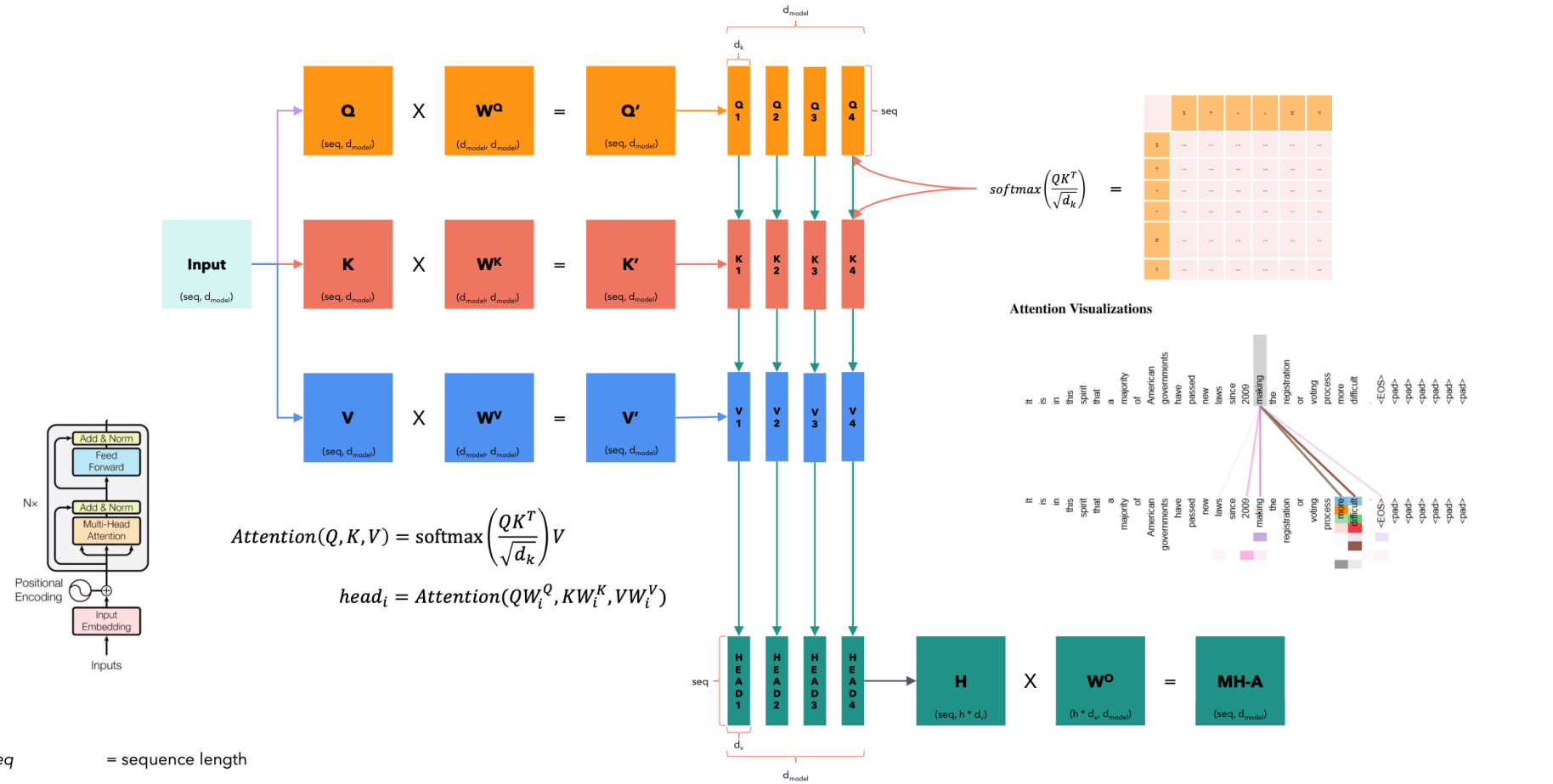
=

Attention

(6, 512)

Each row in this matrix captures not only the meaning (given by the embedding) or the position in the sentence (represented by the positional encodings) but also each word's interaction with other words.

What is Self-Attention?



$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

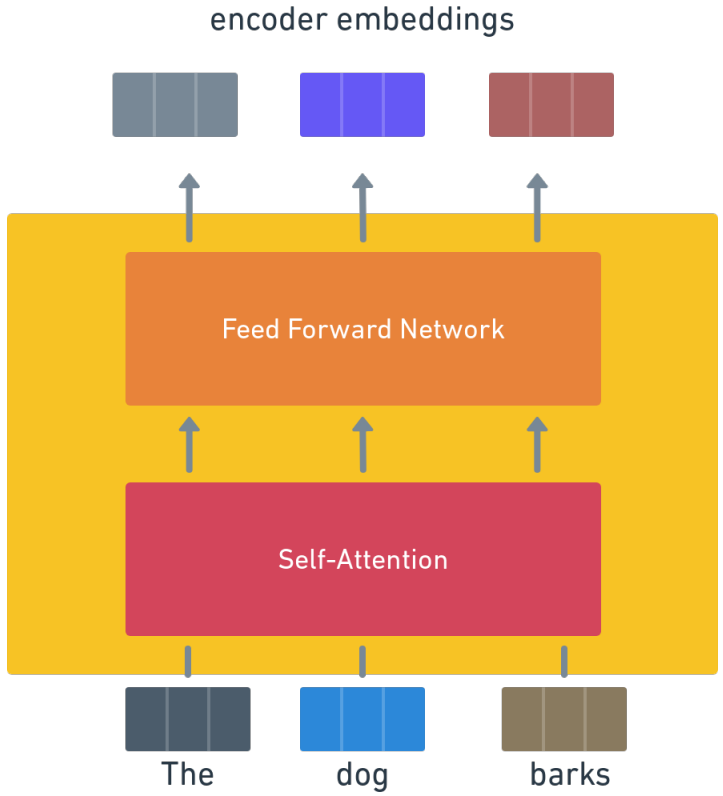
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$$MultiHead(Q,K,V) = Concat(head_1 \dots head_h)W^O$$

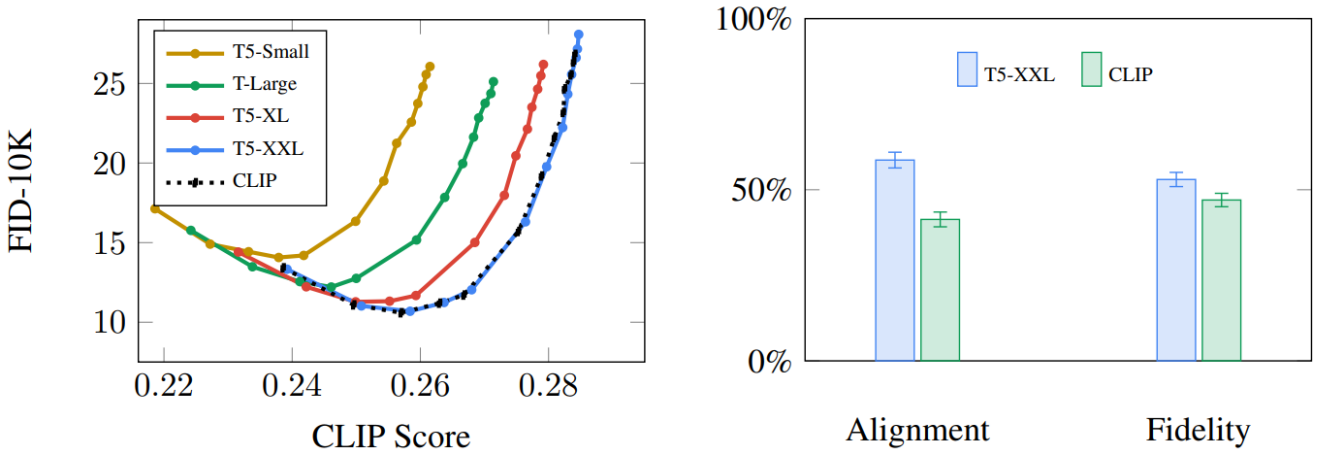
seq	= sequence length
d_{model}	= size of the embedding vector
h	= number of heads
$d_k = d_v$	= d_{model} / h

Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

The Text Encoder: A Transformer Language Model



Takes the text prompt and produces token embeddings



(a) Pareto curves comparing various text encoders.

(b) Comparing T5-XXL and CLIP on DrawBench.

Larger/better language models have a significant effect on the quality of image generation models

How CLIP is trained

Image

Caption

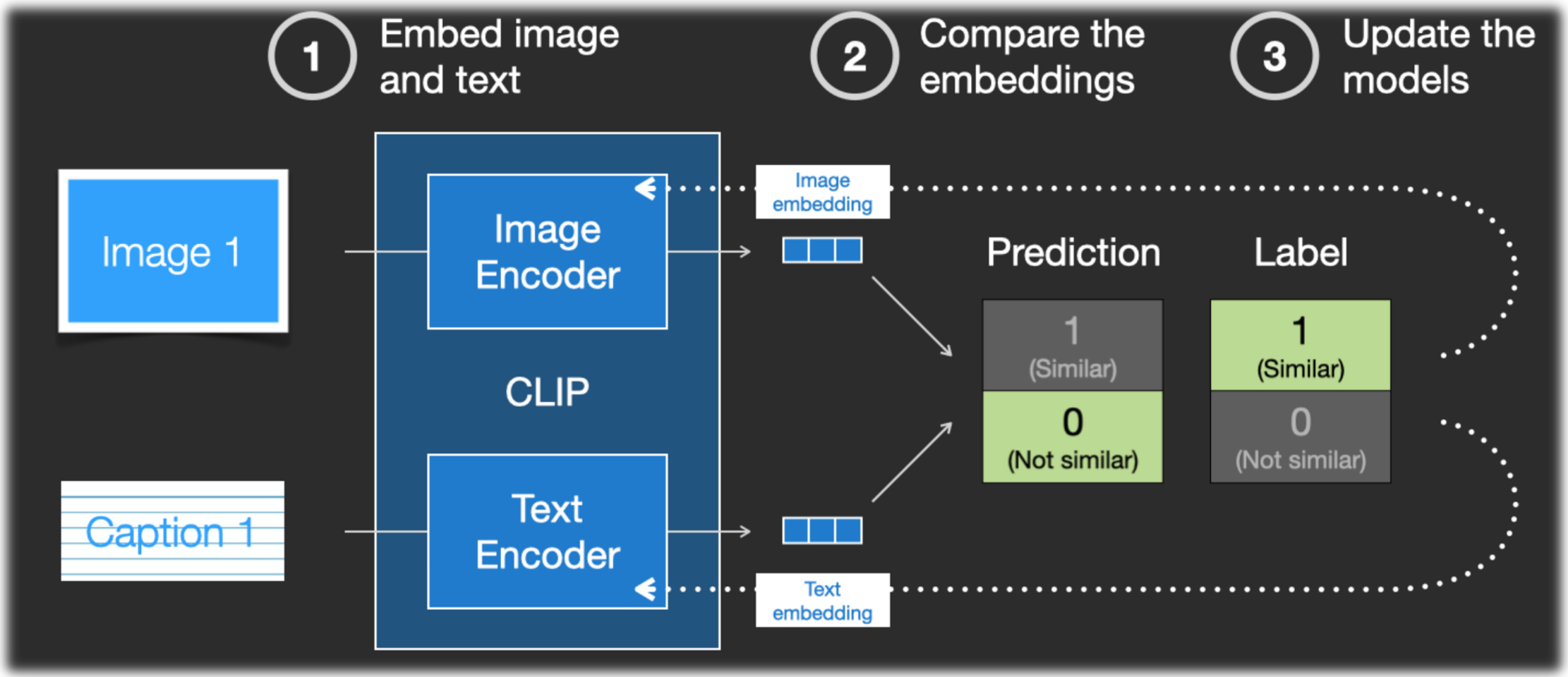
Photo pour Japanese pagoda and old house in Kyoto at twilight - image libre de droit

Soaring by Peter Eades

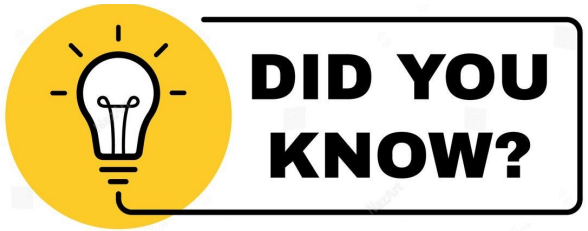
far cry 4 concept art is the reason why it 39 s a beautiful game vg247. Black Bedroom Furniture Sets. Home Design Ideas

The early Stable Diffusion models just plugged in the pre-trained ClipText model released by OpenAI

How CLIP is trained



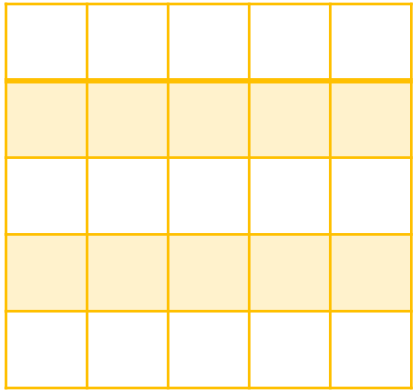
Feeding Text Information Into The Image Generation Process



Noise predictor with Text Input

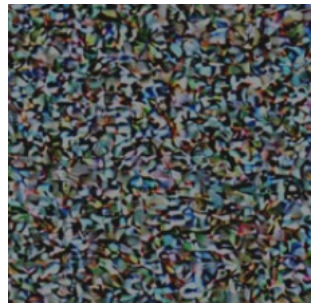
Noise Amount: 2

Prompt text information (token embeddings)

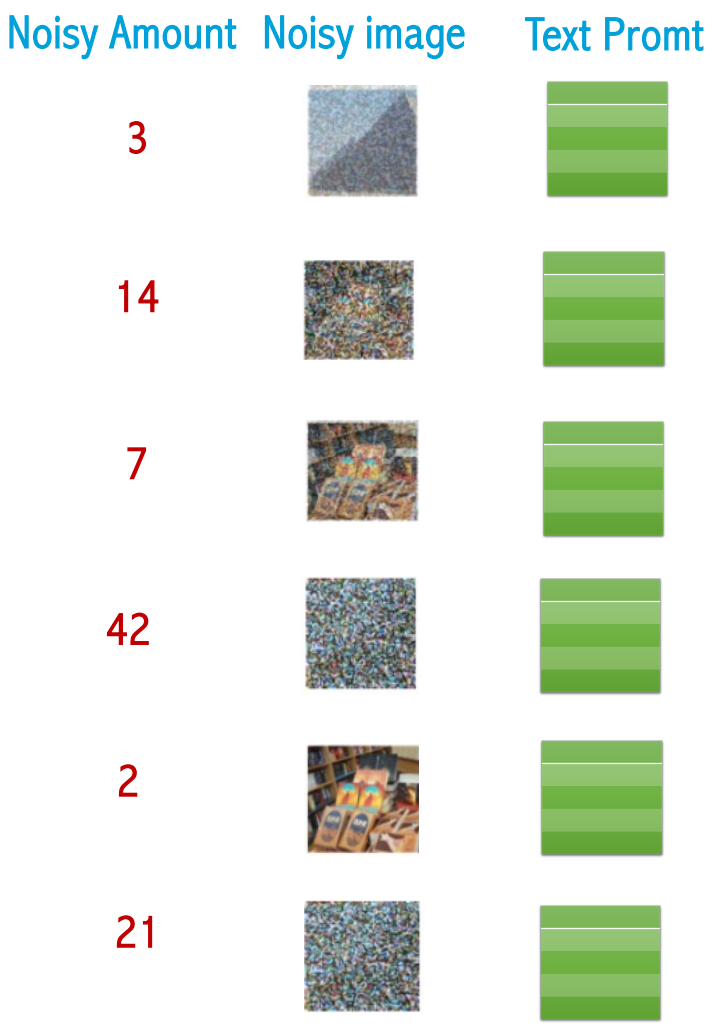


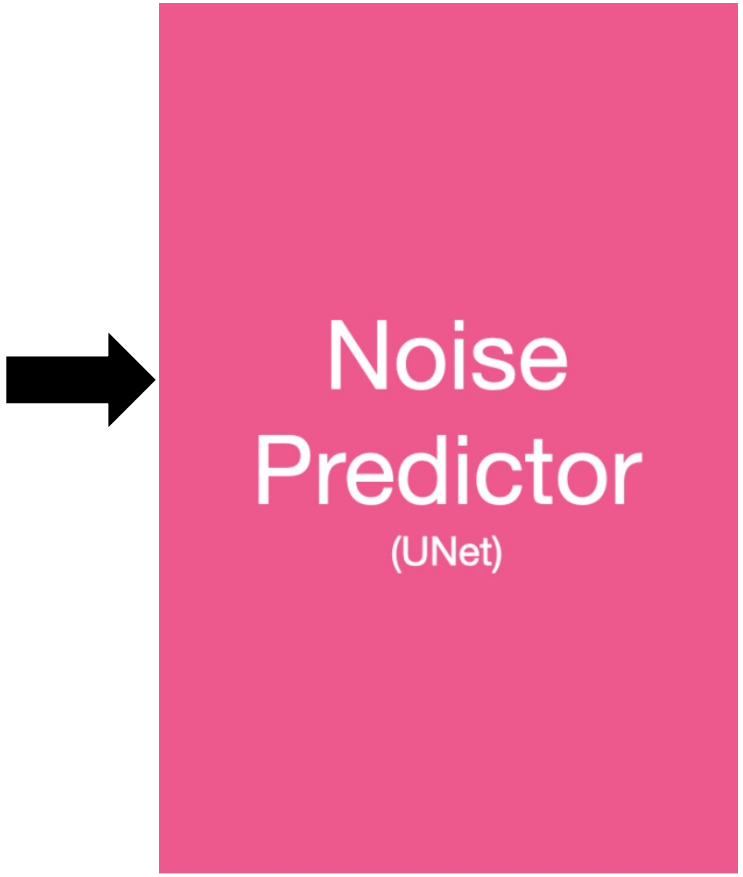
Noise Predictor (UNet) with Text Conditioning

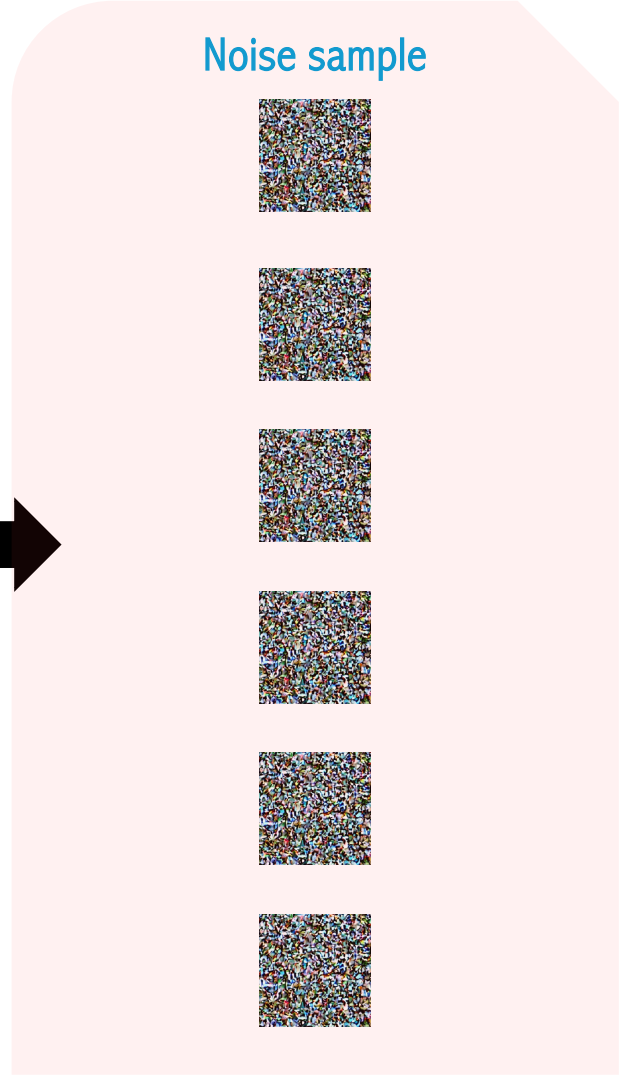
Predicted Noise Sample



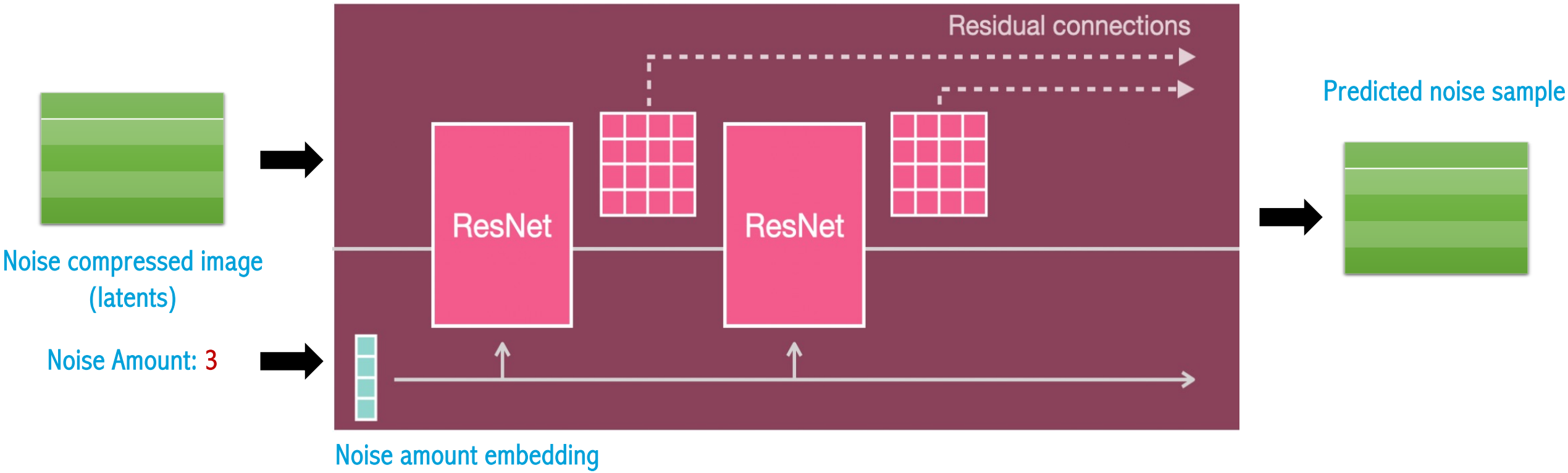
Diffusion Training: Noise Predictor



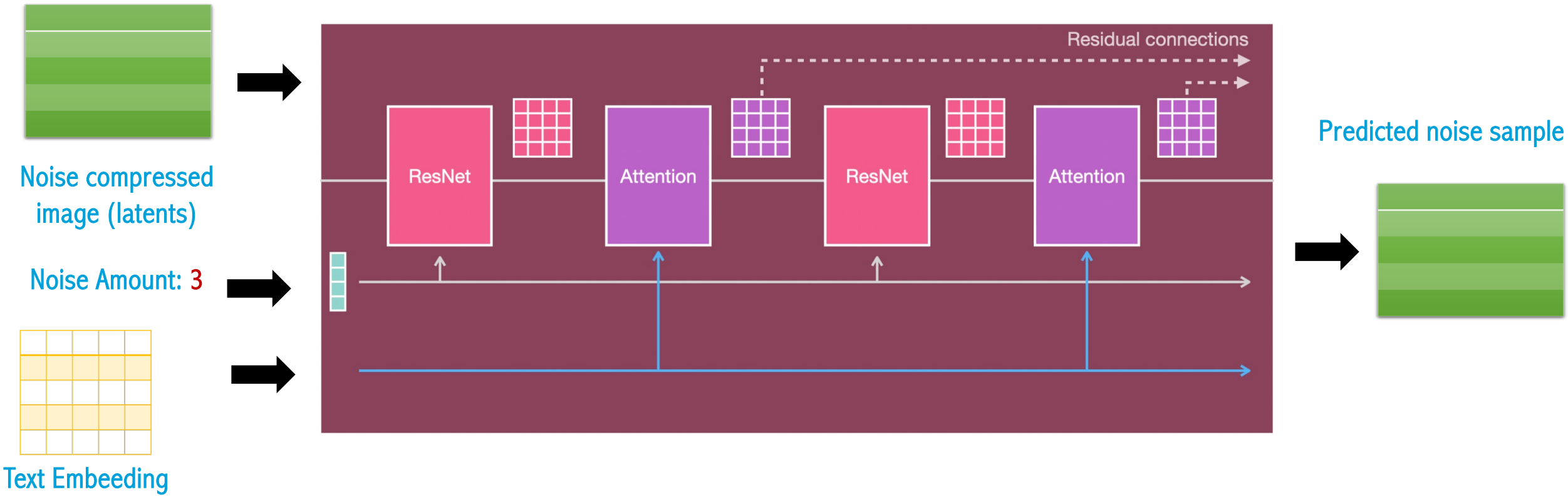




Unet Noise predictor (without text)



Unet Noise predictor (with text)



Latent Diffusion Loss: Discussion

$$L_{DM} = \mathbb{E}_{x,\epsilon\sim\mathcal{N}(0,1),t}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2\right]$$

$$L_{LDM} := \mathbb{E}_{\mathcal{E}(x),\epsilon\sim\mathcal{N}(0,1),t}\left[\|\epsilon - \epsilon_\theta(z_t, t)\|_2^2\right]$$

Latent Vector

Latent State of the Denoising U-Net at time 't'

Domain Specific Encoder (i.e. Transformer) on a condition 'y'

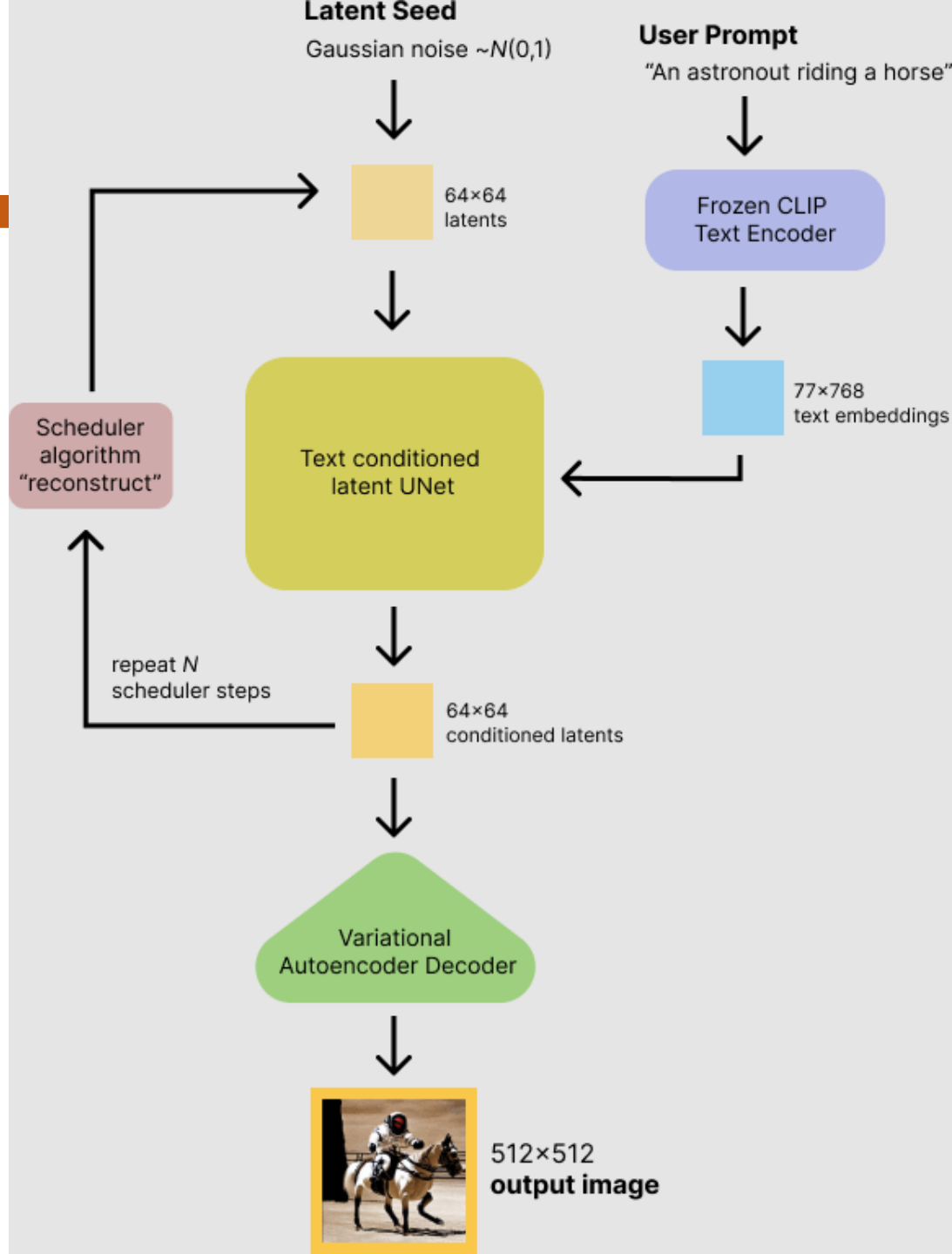
Conditional Loss Function

$$L_{LDM} := \mathbb{E}_{\mathcal{E}(x),y,\epsilon\sim\mathcal{N}(0,1),t}\left[\|\epsilon - \epsilon_\theta(z_t, t, \tau_\theta(y))\|_2^2\right]$$


QUIZ TIME

Stable Diffusion Inference Process

There are mainly three main components in latent diffusion:
1.An autoencoder (VAE).
2.A U-Net.
3.A text-encoder, *e.g.* CLIP's Text Encoder.



VAE: Implementation

During latent diffusion training, the encoder converts a 512*512*3 image into a low dimensional latent representation of image of size say 64*64*4 for the forward diffusion process

```
[2]  from torchvision import transforms as tfms
     from diffusers import AutoencoderKL

     # Load the autoencoder model which will be used to decode the latents into image space.
     vae = AutoencoderKL.from_pretrained("CompVis/stable-diffusion-v1-4", subfolder="vae")

     # To the GPU we go!
     vae = vae.to("cpu")

     # Convert PIL image to latents

     def pil_to_latent(input_im):
         # Single image -> single latent in a batch (so size 1, 4, 64, 64)
         with torch.no_grad():
             latent = vae.encode(tfms.ToTensor()(input_im).unsqueeze(0).to(torch_device)*2-1) # Note scaling
         return 0.18215 * latent.latent_dist.sample()
```

Unet: Implementation

The Unet that takes in the noisy latents (x) and predicts the noise. We use a conditional model that also takes in the timestep (t) and our text embedding as guidance.

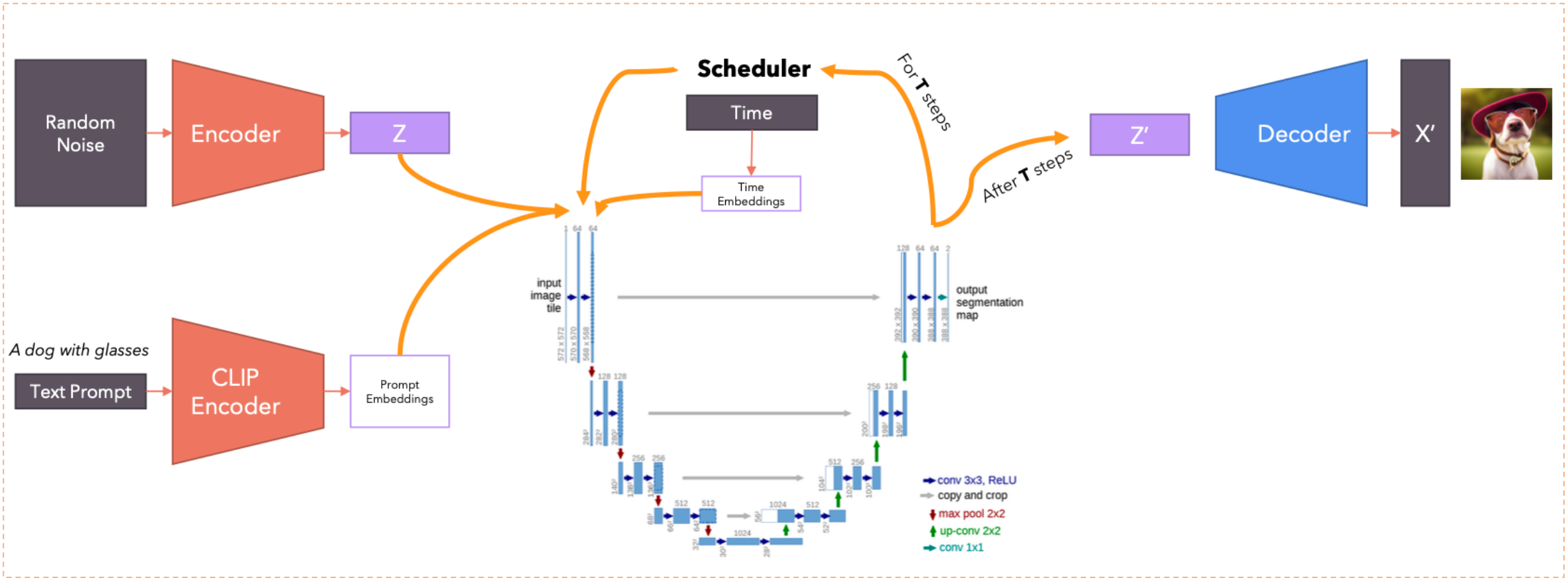
```
from diffusers import UNet2DConditionModel

# The UNet model for generating the latents.
unet = UNet2DConditionModel.from_pretrained("CompVis/stable-diffusion-v1-4", subfolder="unet")

# To the GPU
unet = unet.to(torch_device);
noise_pred = unet(latents, t, encoder_hidden_states=text_embeddings)["sample"]
```

The Text-encoder: Implementation

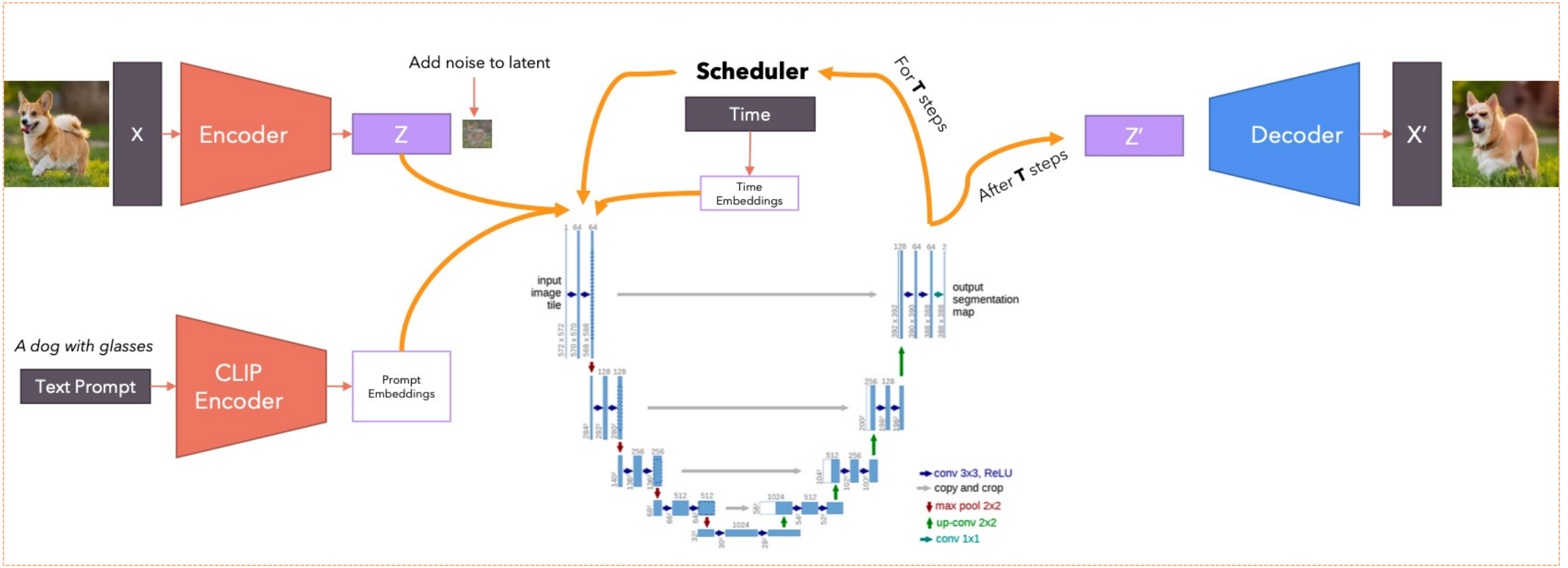
The text-encoder transforms the input prompt into an embedding space that goes as input to the U-Net

```
from transformers import CLIPTextModel, CLIPTokenizer

# Load the tokenizer and text encoder to tokenize and encode the text.
tokenizer = CLIPTokenizer.from_pretrained("openai/clip-vit-large-patch14")
text_encoder = CLIPTextModel.from_pretrained("openai/clip-vit-large-patch14")

# To the GPU
text_encoder = text_encoder.to(torch_device)

prompt = 'An astronaut riding a horse'
# Turn the text into a sequnce of tokens:
text_input = tokenizer(prompt, padding="max_length", max_length=tokenizer.model_max_length,
input_ids = text_input.input_ids.to(torch_device)
# Get output embeddings from tokens
output_embeddings = text_encoder(text_input.input_ids.to(torch_device))[0]
print('Shape:', output_embeddings.shape)
```

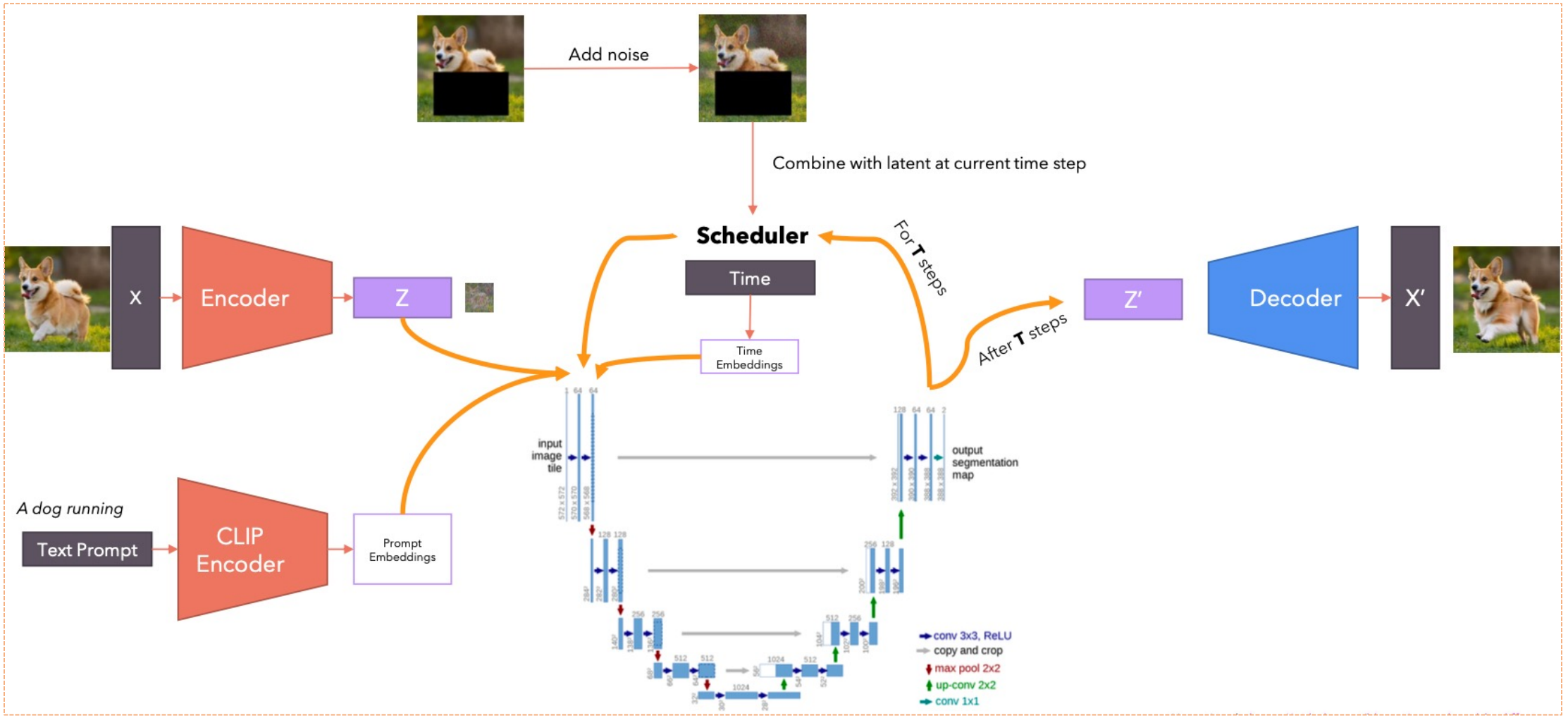
Achitecture (Text-To-Image)



Achitecture (Image-To-Image)



Achitecture (In-painting)



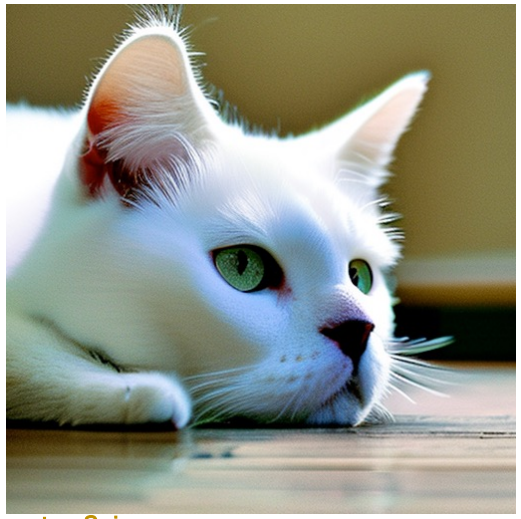
Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

AI VIETNAM
All-in-One Course

Stable Diffusion with API

```
from diffusers import StableDiffusionPipeline, DPMSolverMultistepScheduler

model_id = "stabilityai/stable-diffusion-2-1"

# Use the DPMSolverMultistepScheduler (DPM-Solver++) scheduler here instead
pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16)
pipe.scheduler = DPMSolverMultistepScheduler.from_config(pipe.scheduler.config)
pipe = pipe.to("cuda")

prompt = "a photo of a white cat"
image = pipe(prompt, height=512, width=512).images[0]
image
```



```
model_index.json: 100% 537/537 [00:00<00:00, 40.5kB/s]
Fetching 13 files: 100% 13/13 [00:40<00:00,  3.64s/it]
model.safetensors: 100% 1.36G/1.36G [00:30<00:00, 102MB/s]
(…)ature_extractor/preprocessor_config.json: 100% 342/342 [00:00<00:00, 14.7kB/s]
scheduler/scheduler_config.json: 100% 345/345 [00:00<00:00, 5.70kB/s]
tokenizer/tokenizer_config.json: 100% 824/824 [00:00<00:00, 9.01kB/s]
text_encoder/config.json: 100% 633/633 [00:00<00:00, 8.60kB/s]
tokenizer/special_tokens_map.json: 100% 460/460 [00:00<00:00, 5.91kB/s]
tokenizer/vocab.json: 100% 1.06M/1.06M [00:00<00:00, 6.39MB/s]
tokenizer/merges.txt: 100% 525k/525k [00:00<00:00, 3.17MB/s]
unet/config.json: 100% 939/939 [00:00<00:00, 16.2kB/s]
vae/config.json: 100% 611/611 [00:00<00:00, 10.8kB/s]
diffusion_pytorch_model.safetensors: 100% 3.46G/3.46G [00:39<00:00, 198MB/s]
diffusion_pytorch_model.safetensors: 100% 335M/335M [00:03<00:00, 100MB/s]
```

AI VIETNAM
All-in-One Course

Stable Diffusion with API

```
prompt = "a lovely cat running in the desert in Van Gogh style, trending art."
image = pipe(prompt).images[0]  # image here is in [PIL format](https://pillow.

# Now to display an image you can do either save it such as:
image.save(f"lovely_cat.png")
image
```

```
100% 50/50 [00:17<00:00, 2.72it/s]
```

Stable Diffusion with API

```
generator = torch.Generator("cuda").manual_seed(1024)

prompt = "a sleeping cat enjoying the sunshine."
image = pipe(prompt, generator=generator).images[0]  # image here is in [P

# Now to display an image you can do either save it such as:
image.save(f"lovely_cat_sun.png")
image
```

```
100% 50/50 [00:17<00:00, 2.69it/s]
```

Stable Diffusion with API

```
prompt = "a sleeping cat enjoying the sunshine."
image = pipe(prompt, generator=generator,
                negative_prompt="tree and leaves").images[0]  # 

# Now to display an image you can do either save it such as:
image.save(f"lovely_cat_sun_no_trees.png")
image
```

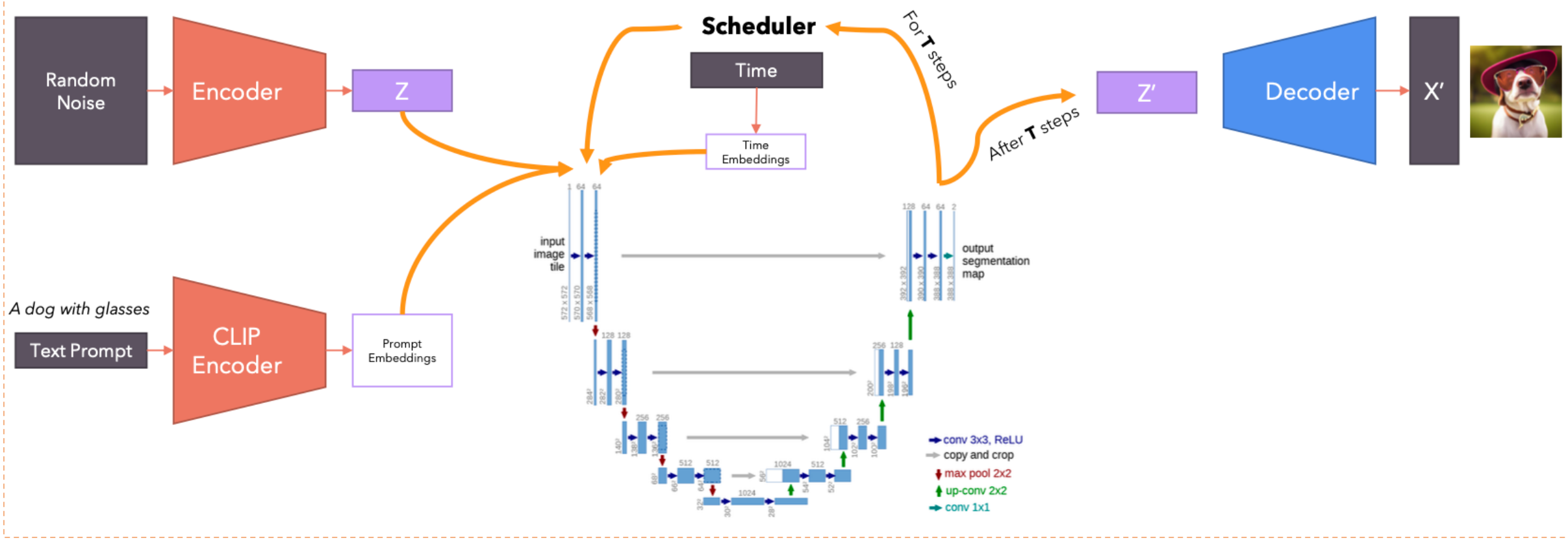
```
100% 50/50 [00:15<00:00, 3.14it/s]
```

Write a simple text2img sampling function

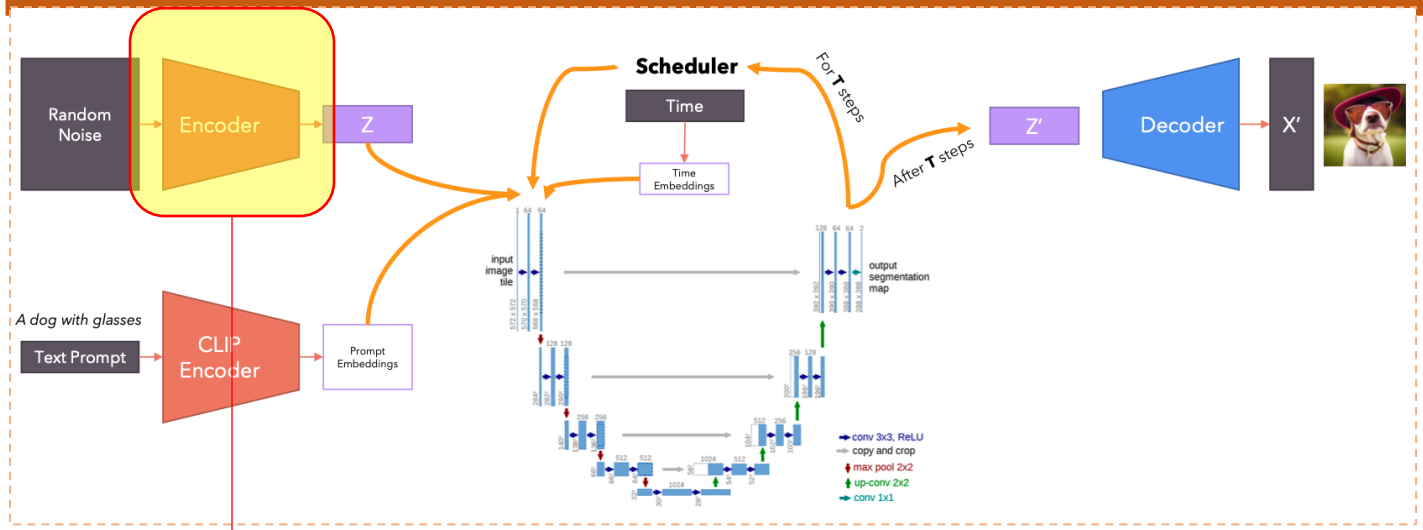
```
[23] @torch.no_grad()
     def generate_simplified(
         prompt = ["a lovely cat"],
         negative_prompt = [""],
         num_inference_steps = 50,
         guidance_scale = 7.5):
         # do_classifier_free_guidance
         batch_size = 1
         height, width = 512, 512
         generator = None
         # here `guidance_scale` is defined analog to the guidance weight `w` of equation (2)
         # of the Imagen paper: https://arxiv.org/pdf/2205.11487.pdf . `guidance_scale = 1`
         # corresponds to doing no classifier free guidance.

         # get prompt text embeddings
         text_inputs = pipe.tokenizer(
             prompt,
             padding="max_length",
             max_length=pipe.tokenizer.model_max_length,
             return_tensors="pt",
         )
```

```
image = generate_simplified(
    prompt = ["a lovely cat"],
    negative_prompt = ["Sunshine"],)
plt_show_image(image[0])
```

Stable Diffusion: Implementation



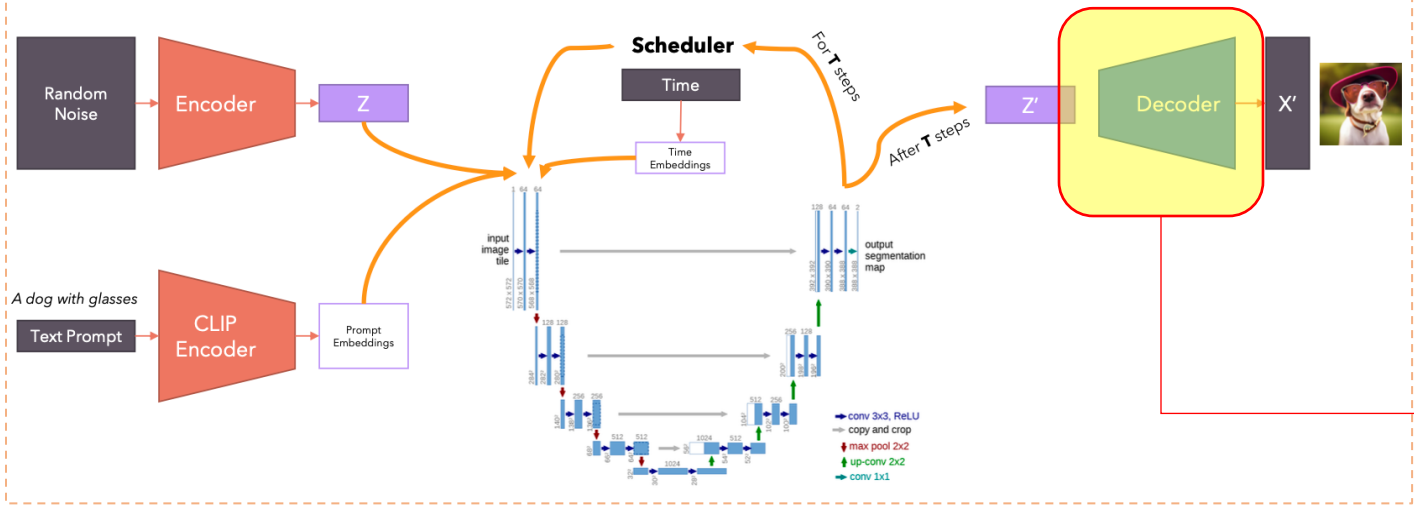
Encoder Implementation



Input: 512x512
Output: 64x64

```python
class VAE_Encoder(nn.Sequential):
    def __init__(self):
        super().__init__(
            # (Batch_Size, Channel, Height, Width) -> (Batch_Size, 128, Height, Width)
            nn.Conv2d(3, 128, kernel_size=3, padding=1),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height, Width)
            VAE_ResidualBlock(128, 128),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height, Width)
            VAE_ResidualBlock(128, 128),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height / 2, Width / 2)
            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=0),

            # (Batch_Size, 128, Height / 2, Width / 2) -> (Batch_Size, 256, Height / 2, Width / 2)
            VAE_ResidualBlock(128, 256),

            # (Batch_Size, 256, Height / 2, Width / 2) -> (Batch_Size, 256, Height / 2, Width / 2)
            VAE_ResidualBlock(256, 256),
```

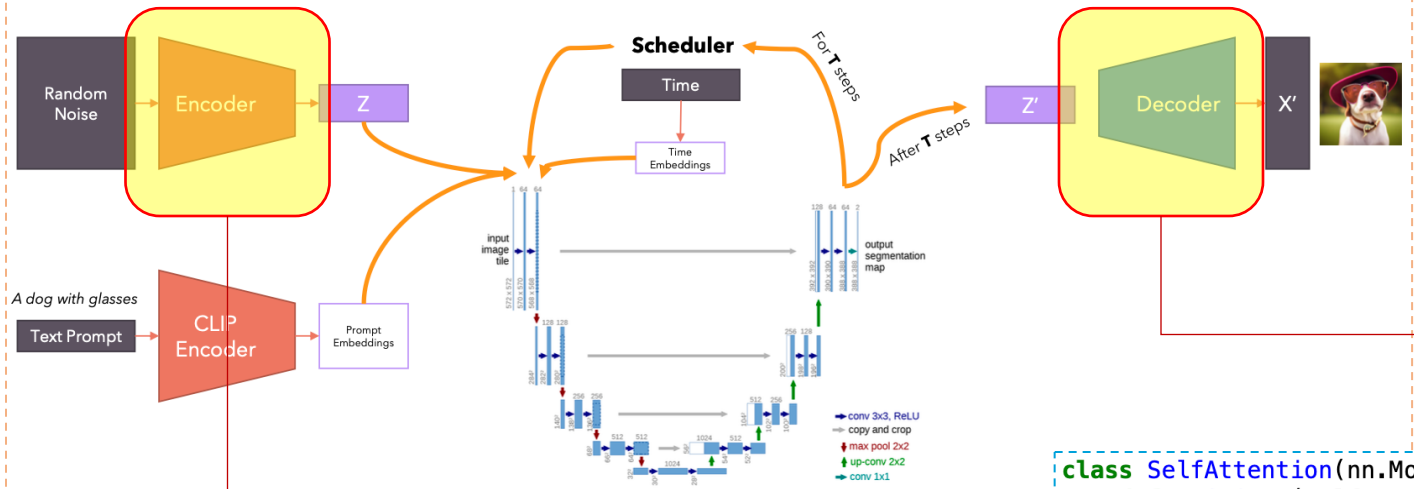
Encoder Implementation



Input: 64x64
Output: 512x512

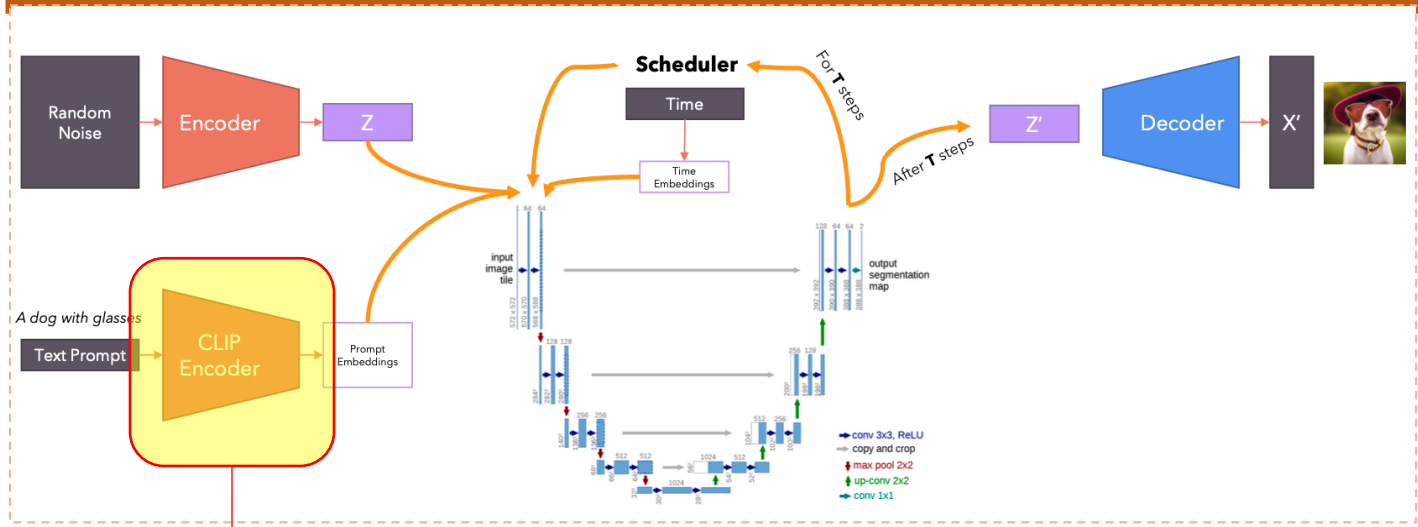
```python
class VAE_Encoder(nn.Sequential):
    def __init__(self):
        super().__init__(
            # (Batch_Size, Channel, Height, Width) -> (Batch_Size, 128, Height, Width)
            nn.Conv2d(3, 128, kernel_size=3, padding=1),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height, Width)
            VAE_ResidualBlock(128, 128),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height, Width)
            VAE_ResidualBlock(128, 128),

            # (Batch_Size, 128, Height, Width) -> (Batch_Size, 128, Height / 2, Width / 2)
            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=0),

            # (Batch_Size, 128, Height / 2, Width / 2) -> (Batch_Size, 256, Height / 2, Width / 2)
            VAE_ResidualBlock(128, 256),

            # (Batch_Size, 256, Height / 2, Width / 2) -> (Batch_Size, 256, Height / 2, Width / 2)
            VAE_ResidualBlock(256, 256),
```

Self-attention Implementation




```python
class SelfAttention(nn.Module):
    def __init__(self, n_heads, d_embed, in_proj_bias=True, out_proj_bias=True):
        super().__init__()
        # This combines the Wq, Wk and Wv matrices into one matrix
        self.in_proj = nn.Linear(d_embed, 3 * d_embed, bias=in_proj_bias)
        # This one represents the Wo matrix
        self.out_proj = nn.Linear(d_embed, d_embed, bias=out_proj_bias)
        self.n_heads = n_heads
        self.d_head = d_embed // n_heads

    def forward(self, x, causal_mask=False):
        # x: # (Batch_Size, Seq_Len, Dim)

        # (Batch_Size, Seq_Len, Dim)
        input_shape = x.shape

        # (Batch_Size, Seq_Len, Dim)
        batch_size, sequence_length, d_embed = input_shape

        # (Batch_Size, Seq_Len, H, Dim / H)
        interim_shape = (batch_size, sequence_length, self.n_heads, self.d_head)
```

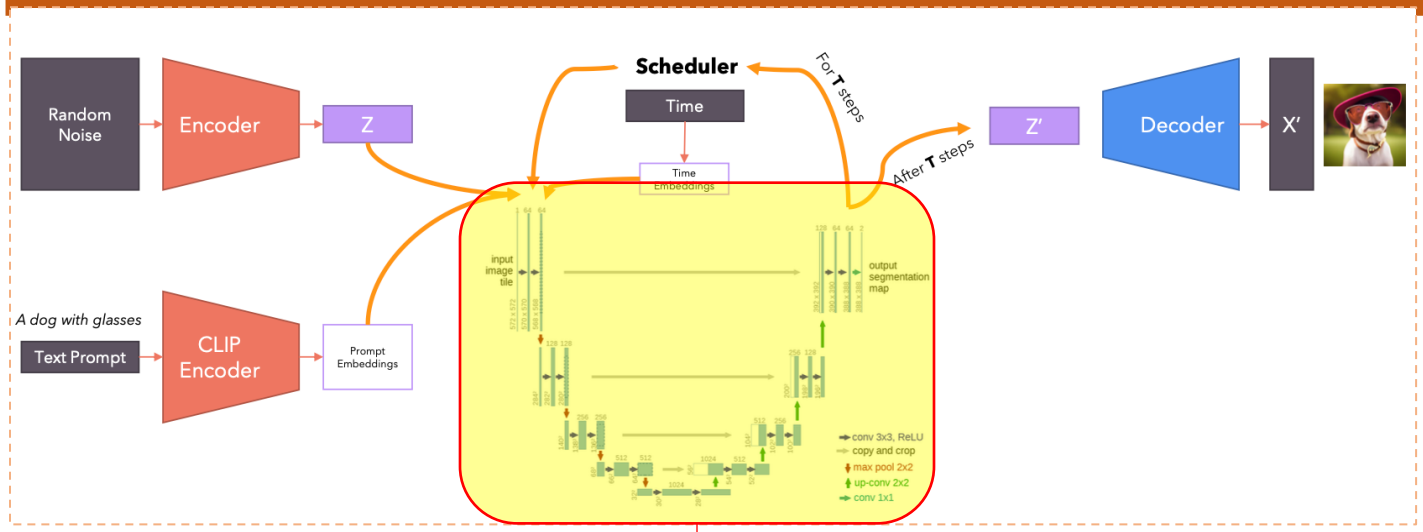
CLIP Implementation




```python
class CLIPEmbedding(nn.Module):
    def __init__(self, n_vocab: int, n_embd: int, n_token: int):
        super().__init__()

        self.token_embedding = nn.Embedding(n_vocab, n_embd)
        # A learnable weight matrix encodes the position information for each token
        self.position_embedding = nn.Parameter(torch.zeros((n_token, n_embd)))

    def forward(self, tokens):
        # (Batch_Size, Seq_Len) -> (Batch_Size, Seq_Len, Dim)
        x = self.token_embedding(tokens)
        # (Batch_Size, Seq_Len) -> (Batch_Size, Seq_Len, Dim)
        x += self.position_embedding

        return x

class CLIPLayer(nn.Module):
    def __init__(self, n_head: int, n_embd: int):
        super().__init__()

        # Pre-attention norm
        self.layernorm_1 = nn.LayerNorm(n_embd)
        # Self attention
        self.attention = SelfAttention(n_head, n_embd)
        # Pre-FNN norm
        self.layernorm_2 = nn.LayerNorm(n_embd)
        # Feedforward layer
```

AI VIETNAM
All-in-One Course

Unet Implementation

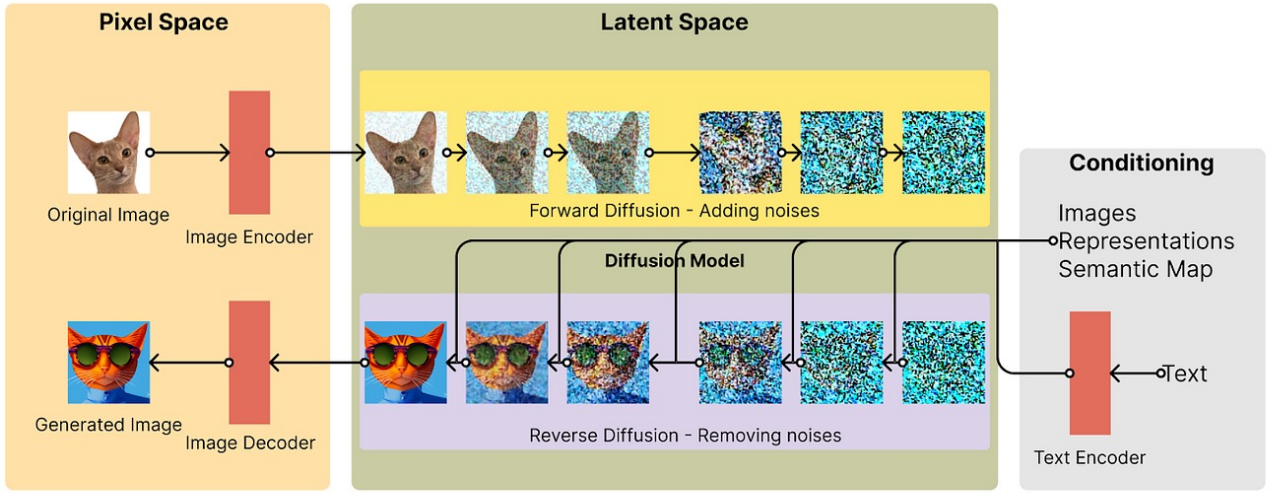



```python
class UNET_AttentionBlock(nn.Module):
    def __init__(self, n_head: int, n_embd: int, d_context=768):
        super().__init__()
        channels = n_head * n_embd

        self.groupnorm = nn.GroupNorm(32, channels, eps=1e-6)
        self.conv_input = nn.Conv2d(channels, channels, kernel_size=1, padding=0)

        self.layernorm_1 = nn.LayerNorm(channels)
        self.attention_1 = SelfAttention(n_head, channels, in_proj_bias=False)
        self.layernorm_2 = nn.LayerNorm(channels)
        self.attention_2 = CrossAttention(n_head, channels, d_context, in_proj_bias=False)
        self.layernorm_3 = nn.LayerNorm(channels)
        self.linear_geglu_1  = nn.Linear(channels, 4 * channels * 2)
        self.linear_geglu_2 = nn.Linear(4 * channels, channels)

        self.conv_output = nn.Conv2d(channels, channels, kernel_size=1, padding=0)

    def forward(self, x, context):
        # x: (Batch_Size, Features, Height, Width)
        # context: (Batch_Size, Seq_Len, Dim)

        residue_long = x
```

Outline

- Objective
- Introduction to Stable Diffusion
- Stable Diffusion: Motivation
- Stable Diffusion: Clearly Explained
- Stable Diffusion: Demo with API
- Summary

Summary



1. Principle of Diffusion models.
2. Model score function of images with UNet model
3. Understanding prompt through contextualized word embedding
4. Let text influence image through cross attention
5. Improve efficiency by adding an autoencoder


Thank you