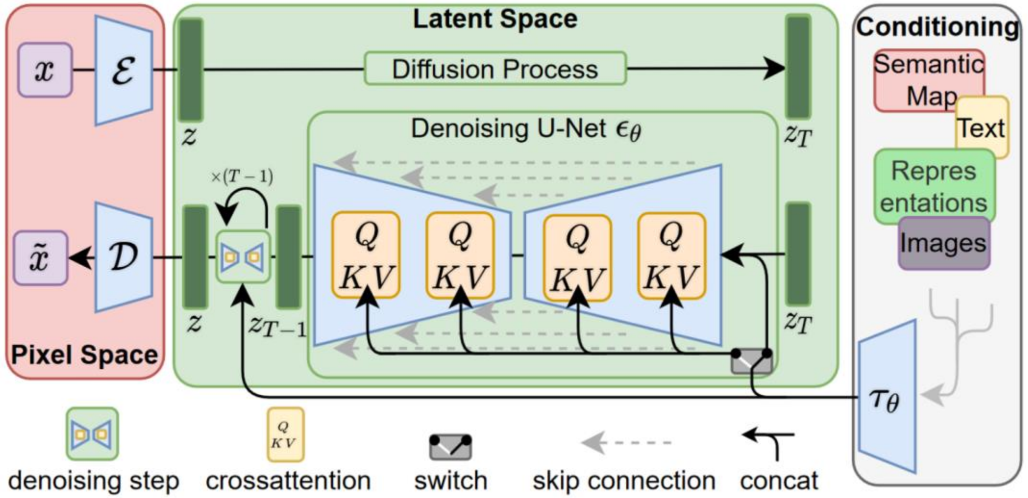


Coding Stable Diffusion form scratch in PyTorch

Umar Jamil

Topics and Prerequisites

Topics discussed

- Latent Diffusion Models (Stable Diffusion) from scratch in PyTorch. No other libraries used except for tokenizer.
- Maths of diffusion models as defined in the DDPM paper (simplified!)
- Classifier-Free Guidance
- Text – to – Image
- Image – to – Image
- Inpainting

Future videos

- Score-based models
- ODE and SDE theoretical framework for diffusion models
- Euler, Runge-Kutta and derived samplers.

Prerequisites

- Basics of probability and statistics (multivariate gaussian, conditional probability, marginal probability, likelihood, Bayers' rule).
    - I will give a non-maths intuition for most concepts.
- Basics of PyTorch and neural networks
- How the attention mechanism works (watch my video on the Transformer model).
- How convolution layers work

What is 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

Output

Prompt

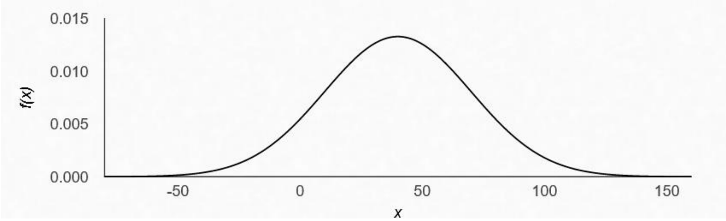
Picture of a dog with glasses

What is a generative model?

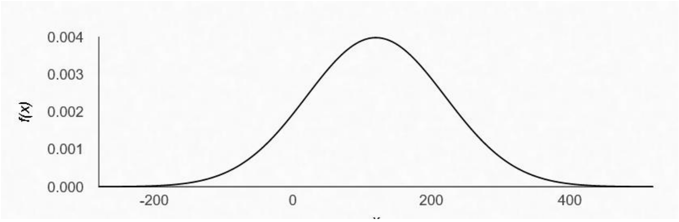
A generative model learns a probability distribution of the data set such that we can then sample from the distribution to create new instances of data. For example, if we have many pictures of cats and we train a generative model on it, we then sample from this distribution to create new images of cats.

Why do we model data as distributions?

- Imagine you're a criminal, 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.

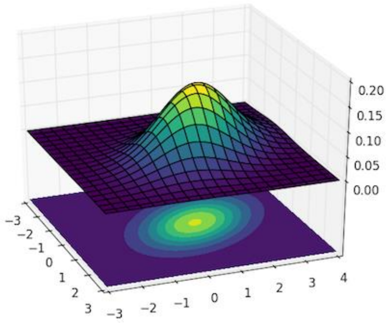


Age: **N**$(40, 30^2)$



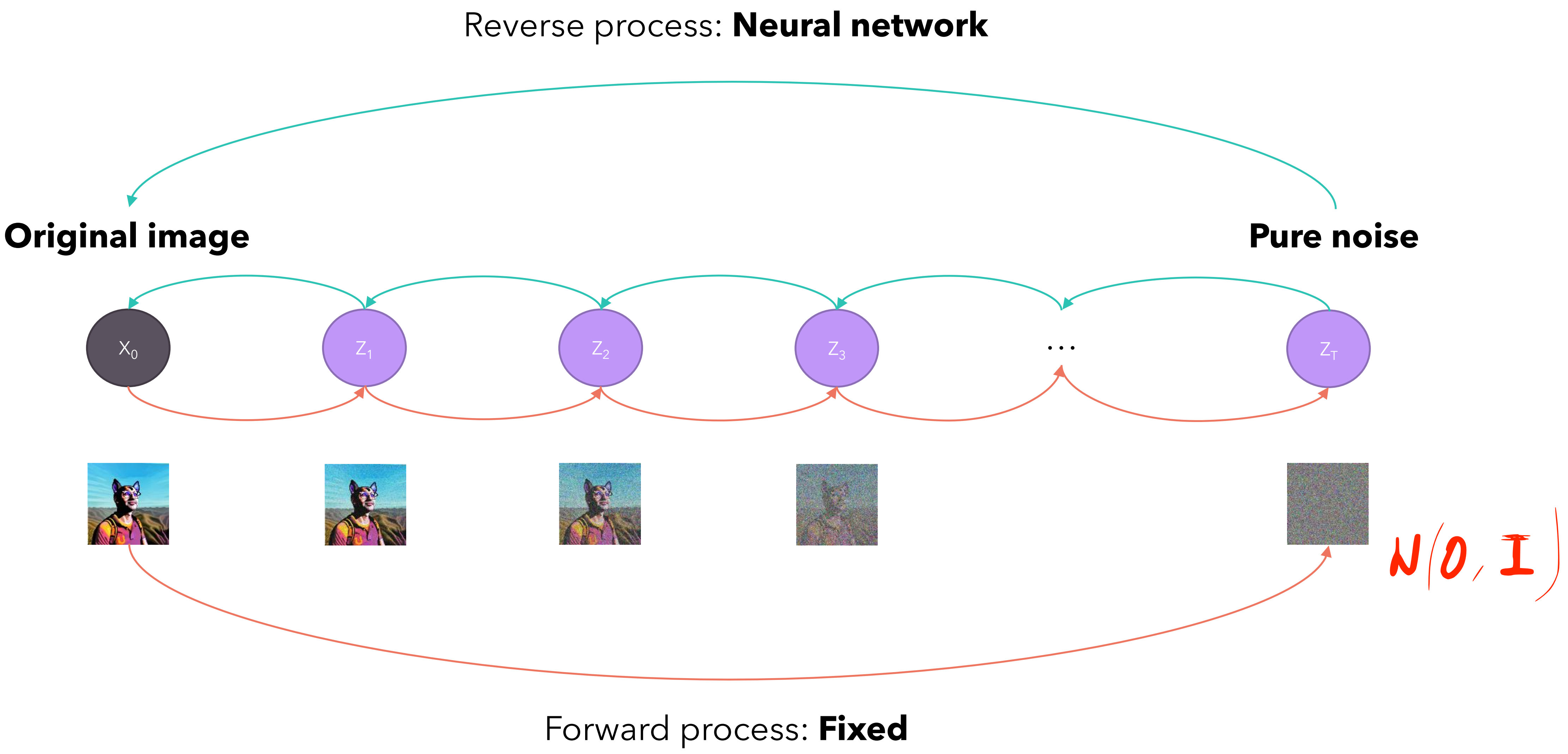
Height: **N**$(120, 100^2)$

- At first, you may sample from each distribution independently to create a fake identity, but that would produce unreasonable pairs of (Age, Height).
- To generate fake identities that make sense, you need the **joint distribution**, otherwise you may end up with an unreasonable pair of (Age, Height)
- We can also evaluate probabilities on one of the two variables using **conditional probability** and/or by **marginalizing** a variable.



Learning the distribution p(x) of our data.

- We have a data set made of up images, and we want to learn a very complex distribution that we can then use to **sample** from.


Reverse process: **Neural network**
Original image
Pure noise
X_0
Z_1
Z_2
Z_3
...
Z_T
$N(0, I)$
Forward process: **Fixed**

The math of diffusion models… simplified!

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 $\mathbf{x}_1, \dots, \mathbf{x}_T$ 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 $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$:

unkown mean, unknown variance -> learn with neural network), we will fix the variance and let the NN only learn the mean:)

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

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

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

Evidence Lower Bound **(ELBO)**

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

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

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

Forward process **q**

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

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

Algorithm 1 Training

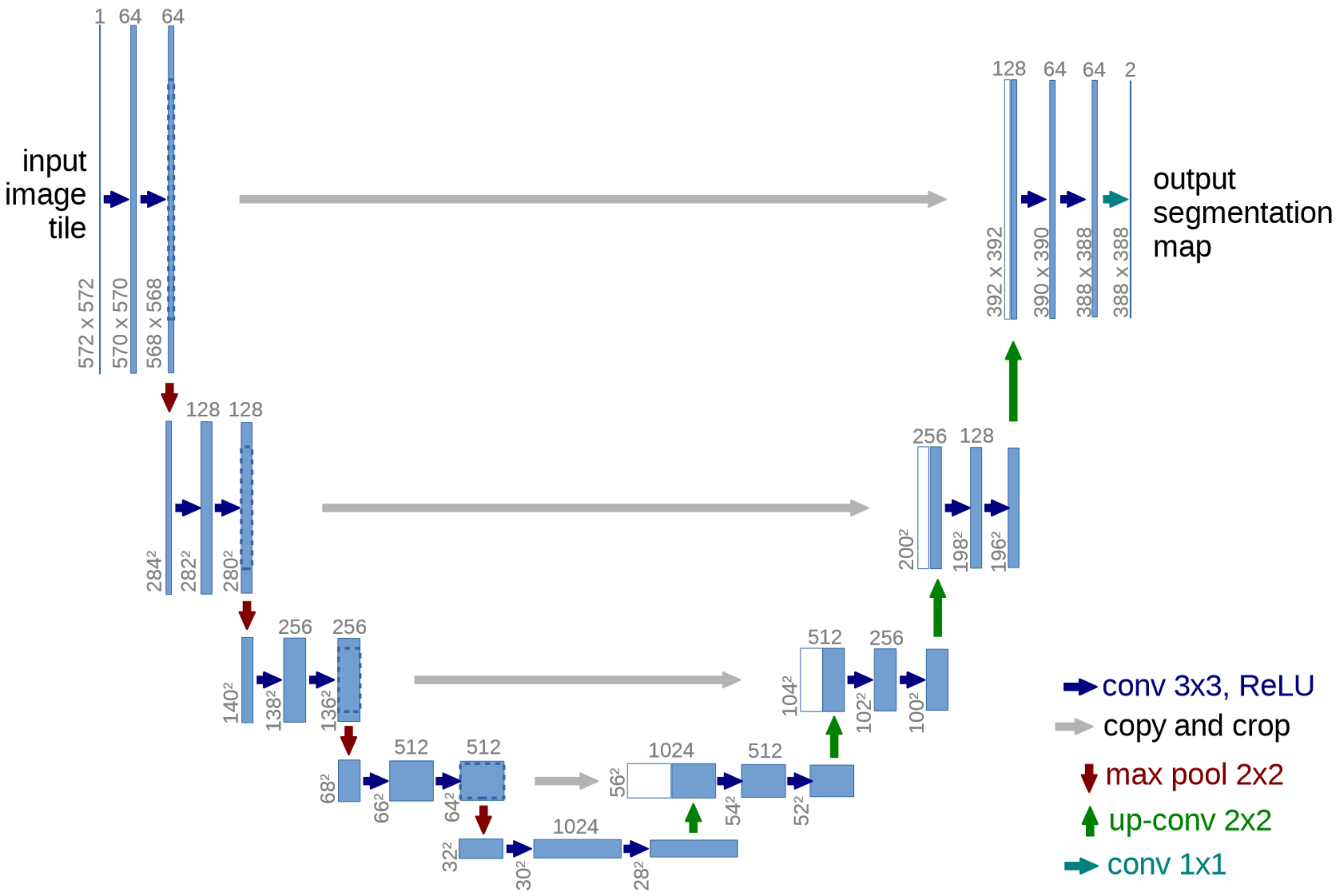
repeat
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ We take a sample from our dataset
  $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$ We generate a random number t, between 1 and T
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ We sample some noise
  Take gradient descent step on
    $\nabla_\theta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

Algorithm 2 Sampling

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

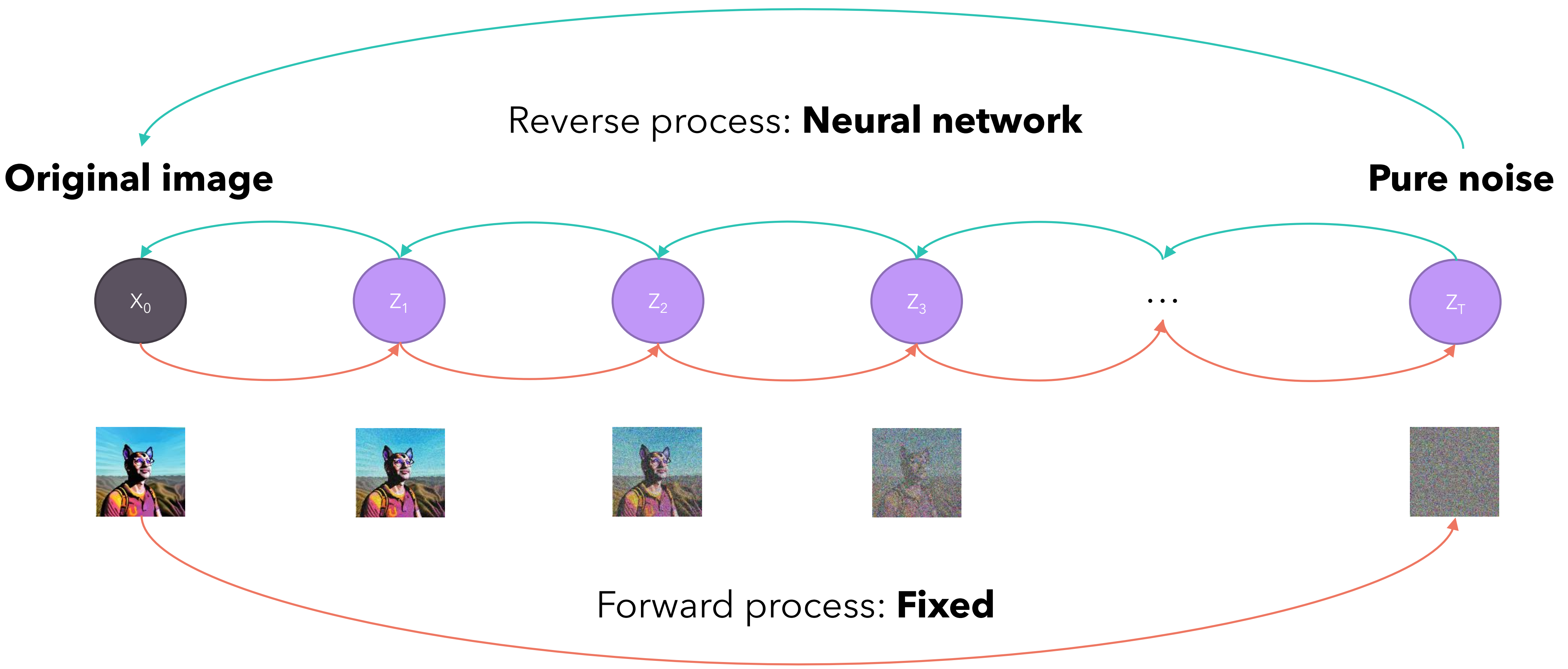
We keep denoising the image progressively for T steps.

U-Net



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

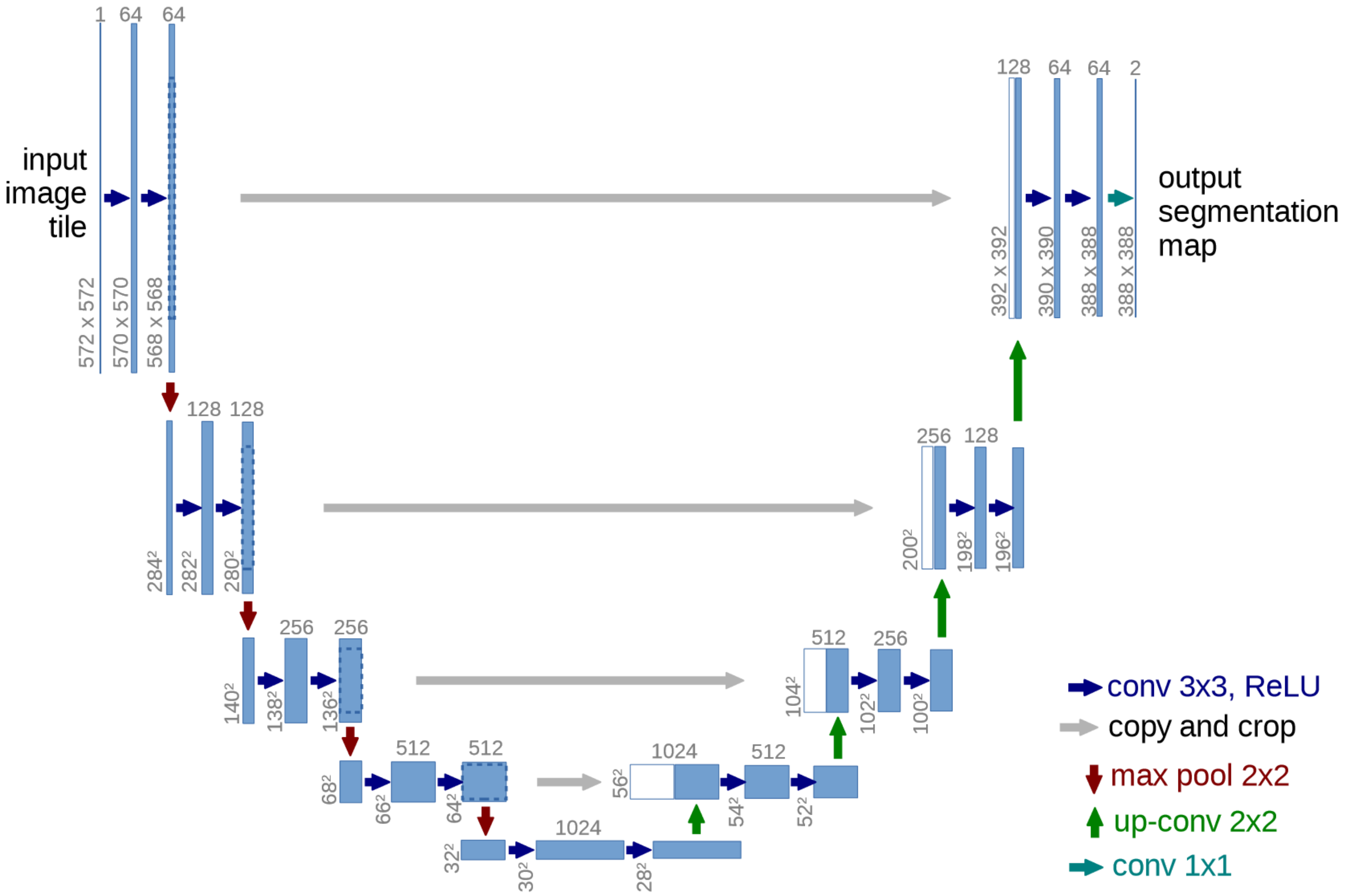
How to generate new data?



How to condition the reverse process?

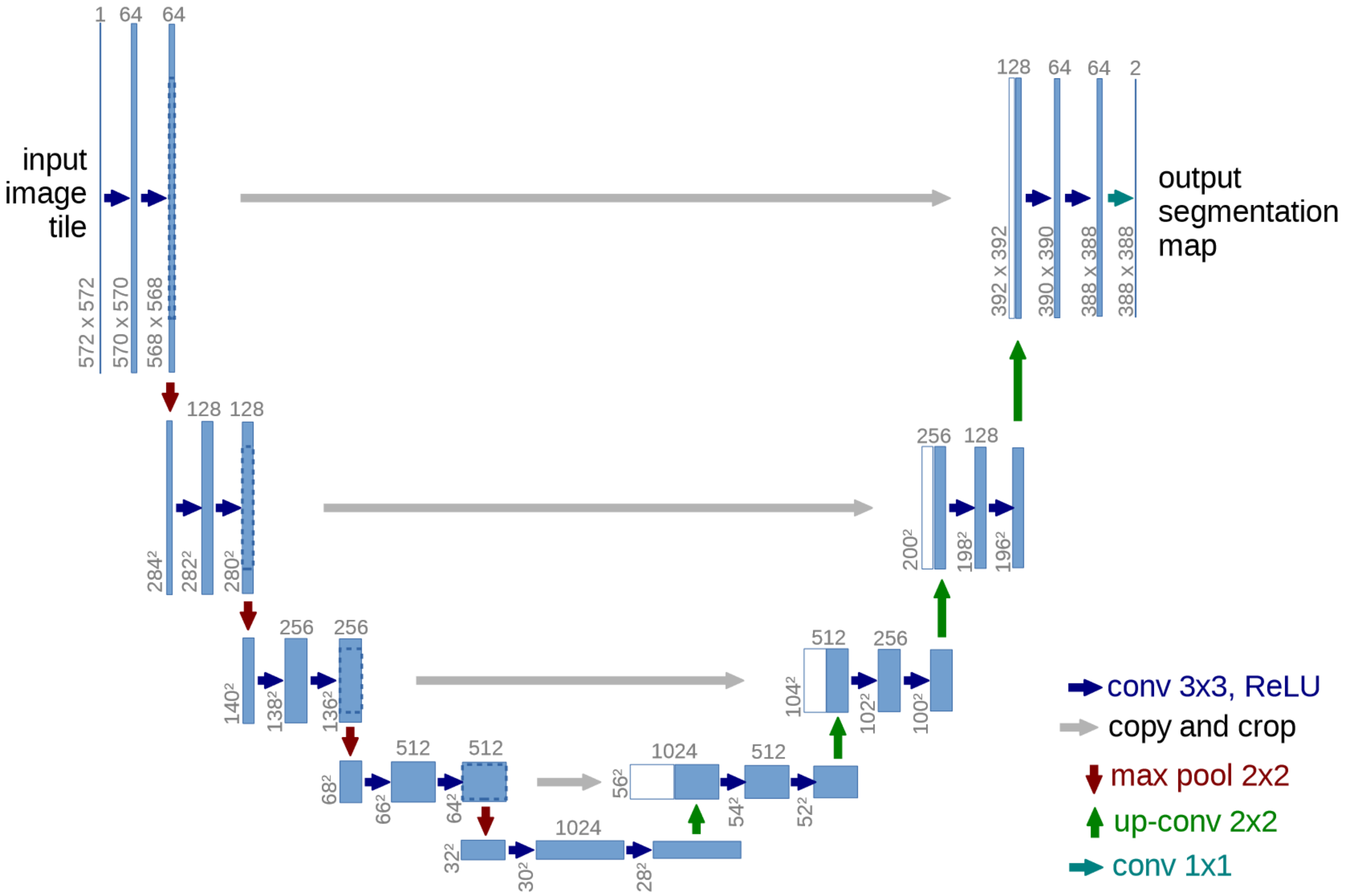
- Since we start from noise in the reserve process, how can the model know what we want as output? How can the model understand our prompt? This is why we need to condition the reverse process.
- If we want to condition our network, we could train a model to learn a joint distribution of the data and the conditioning signal $p(x, c)$, and then sample from this joint distribution. This, however, requires the training of a model for each separate conditioning signal.
- Another approach, called **classifier guidance**, involves the training of a separate model to condition the output.
- The latest and most successful approach is called **classifier-free guidance**, in which, instead of training two networks, one conditional network and an unconditional network, we train a single network and during training, with some probability, we set the conditional signal to zero, this way the network becomes a mix of conditioned and unconditioned network, and we can take the conditioned and unconditioned output and combine them with a weight that indicates how much we want the network to pay attention to the conditioning signal.

Classifier Free Guidance (Training)



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

Classifier Free Guidance (Inference)



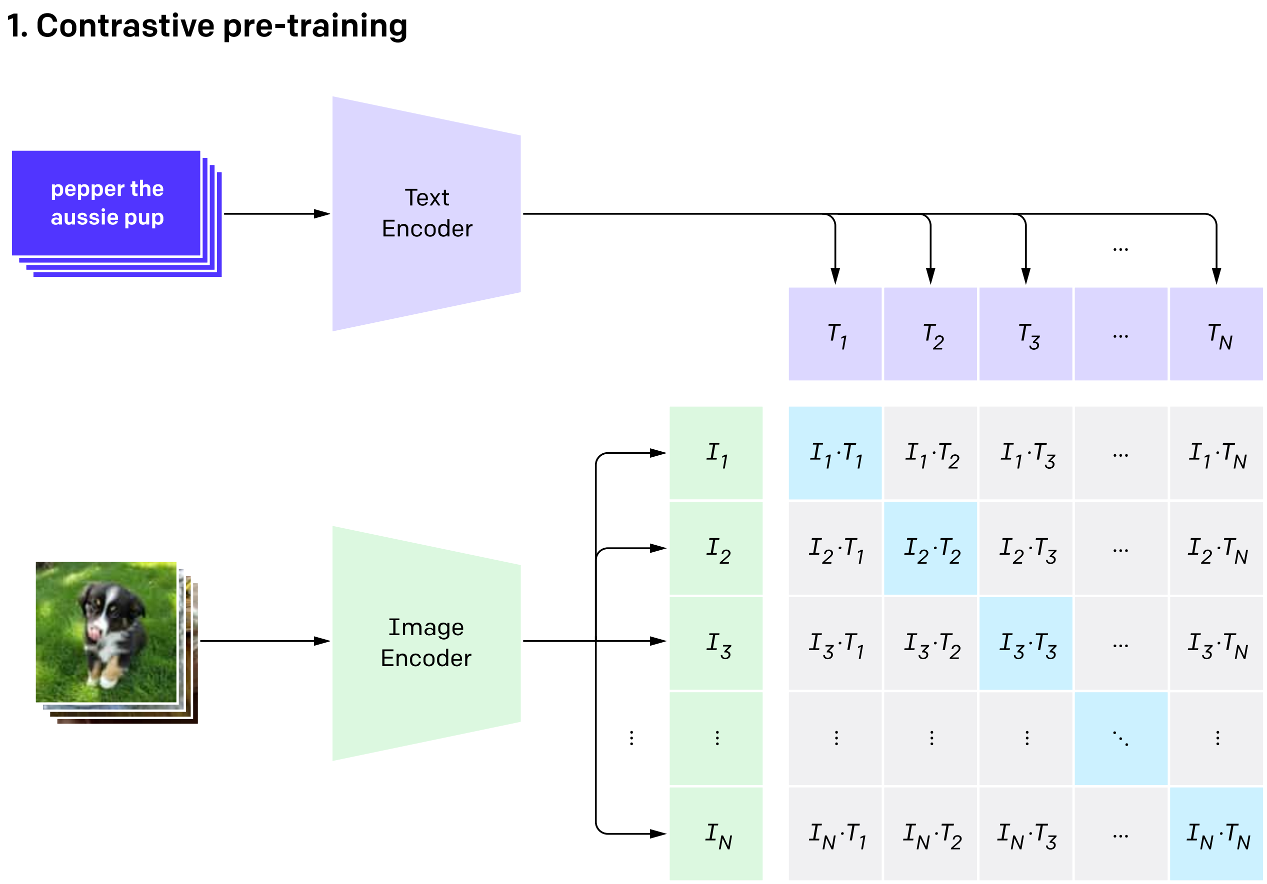
Ronneberger, O., Fischer, P. and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18* (pp. 234-241). Springer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

Classifier Free Guidance (Combine output)

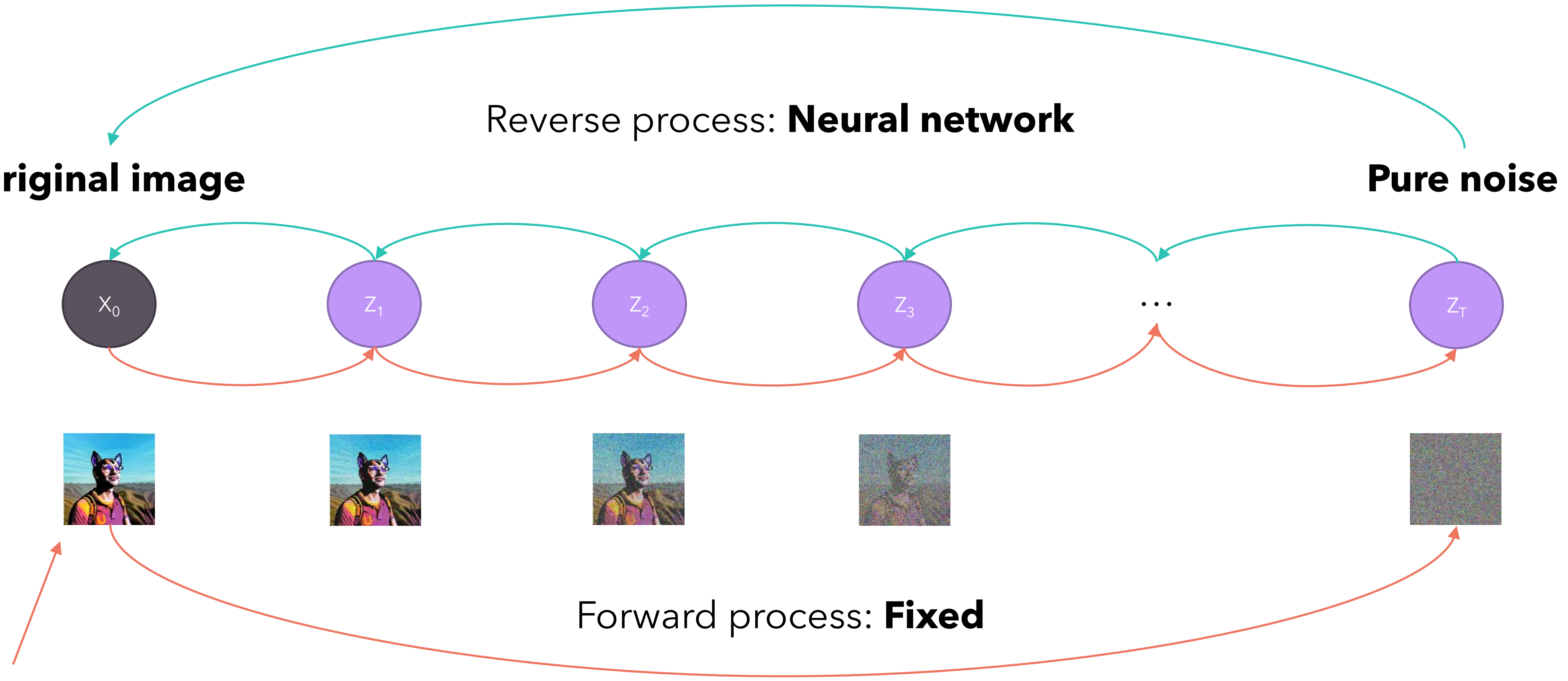
$$output = w\ * (output_{conditioned} - output_{unconditioned}) + output_{unconditioned}$$

A weight that indicates how much we want the model to pay attention to the conditioning signal (prompt).

CLIP (Contrastive Language–Image Pre-training)



Performing many steps on big images is **slow**



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)?

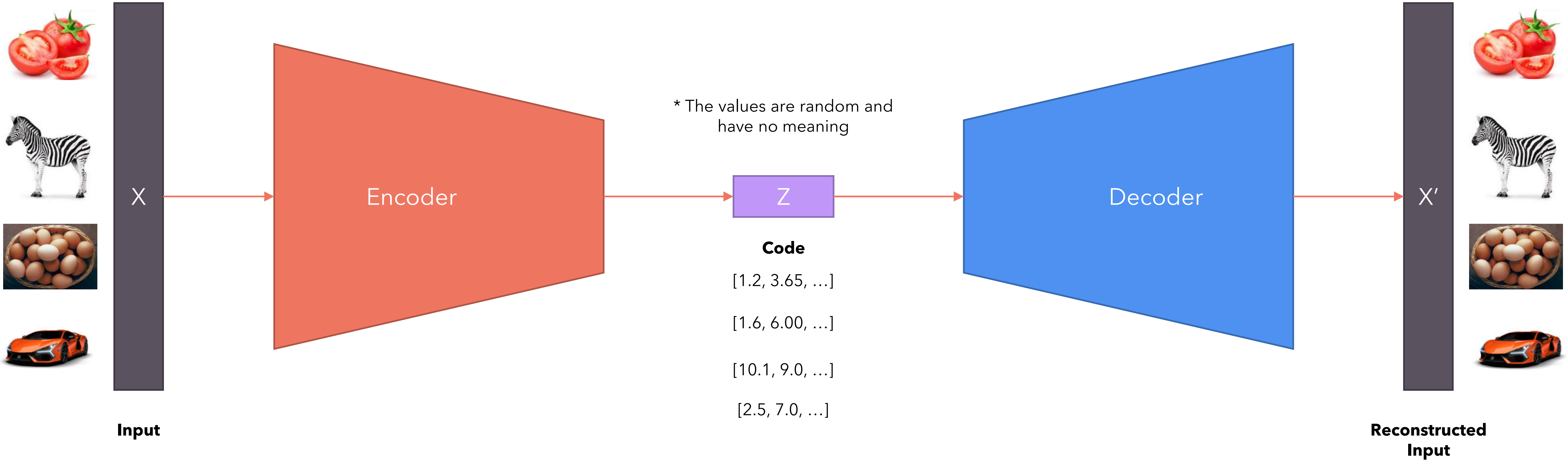
Latent Diffusion Model

- 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**.
- 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

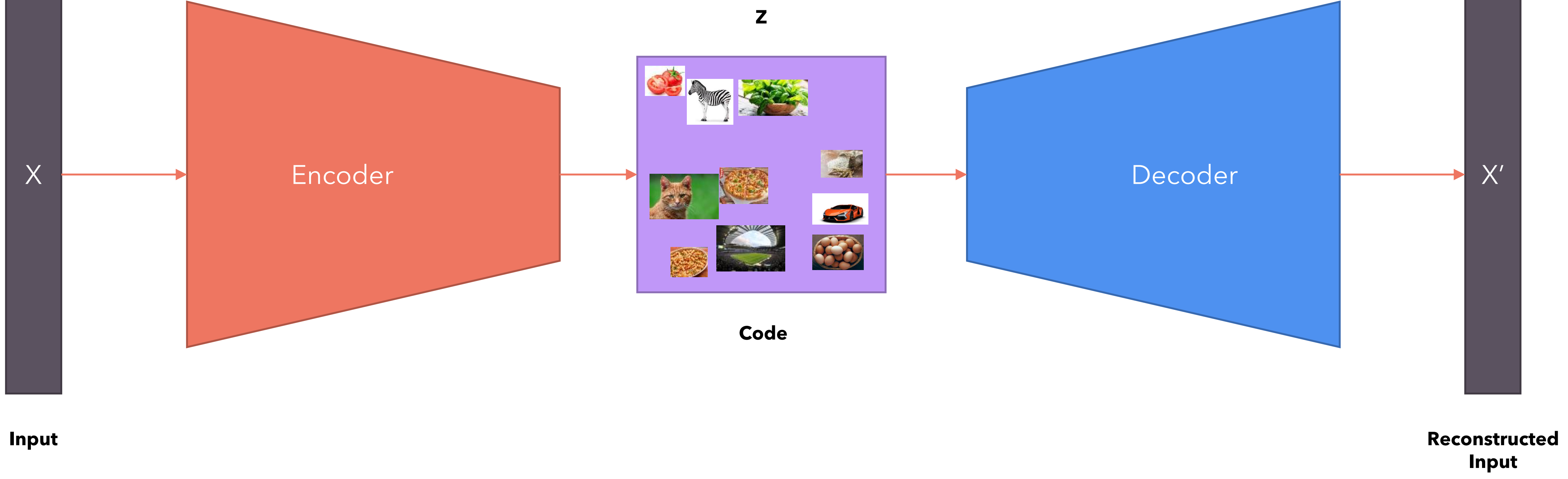
Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

What is an Autoencoder?



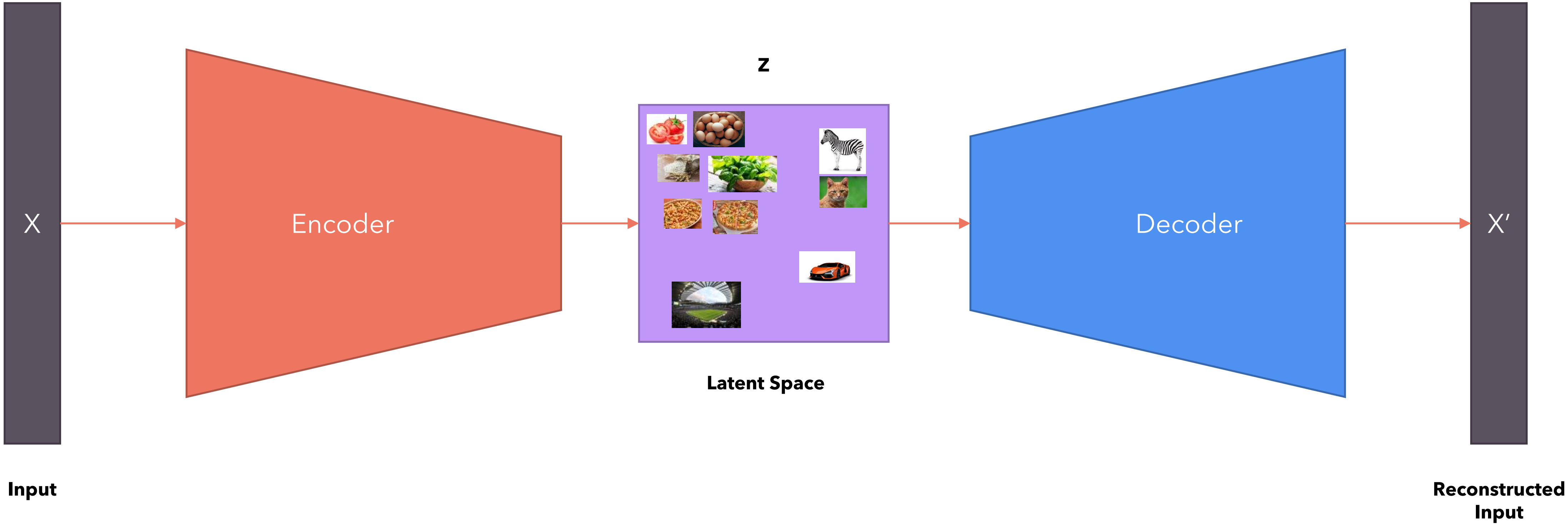
What's the problem with Autoencoders?

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



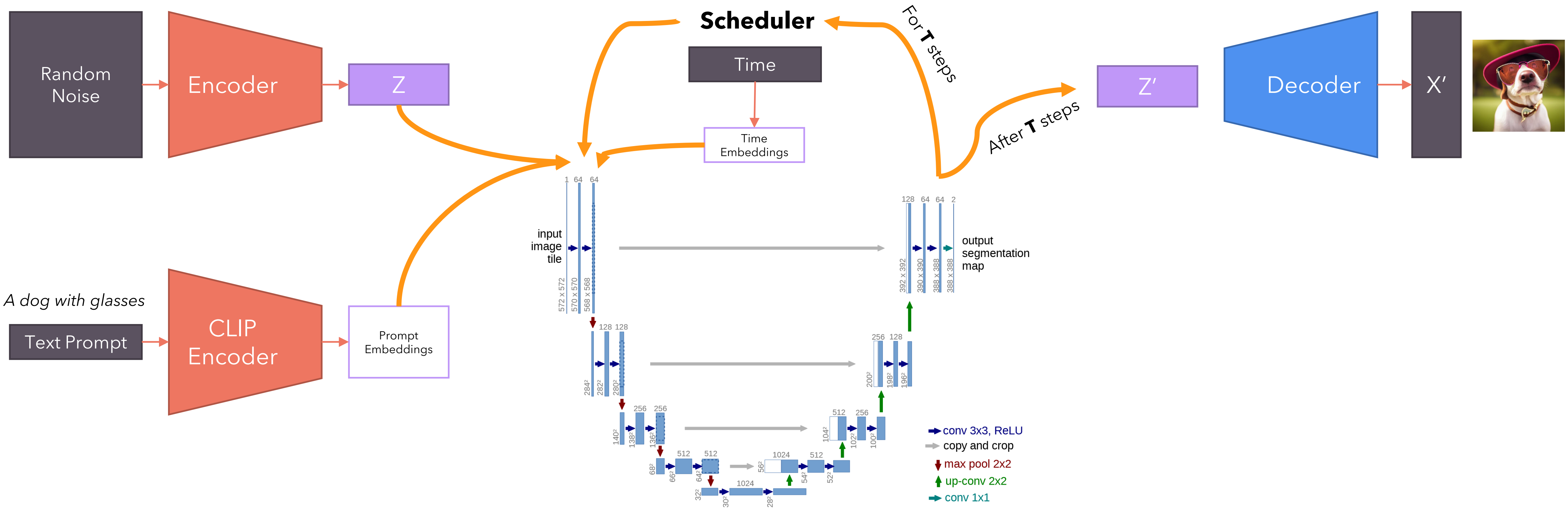
Introducing the Variational Autoencoder

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



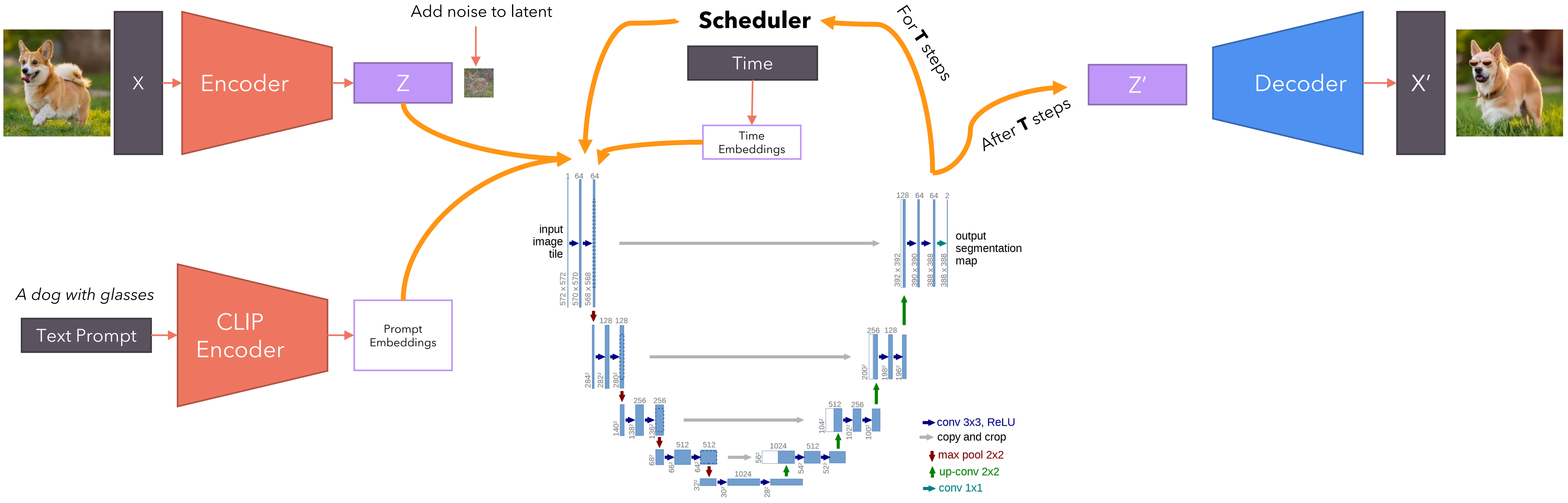
Architecture (Text-To-Image)

S.D.



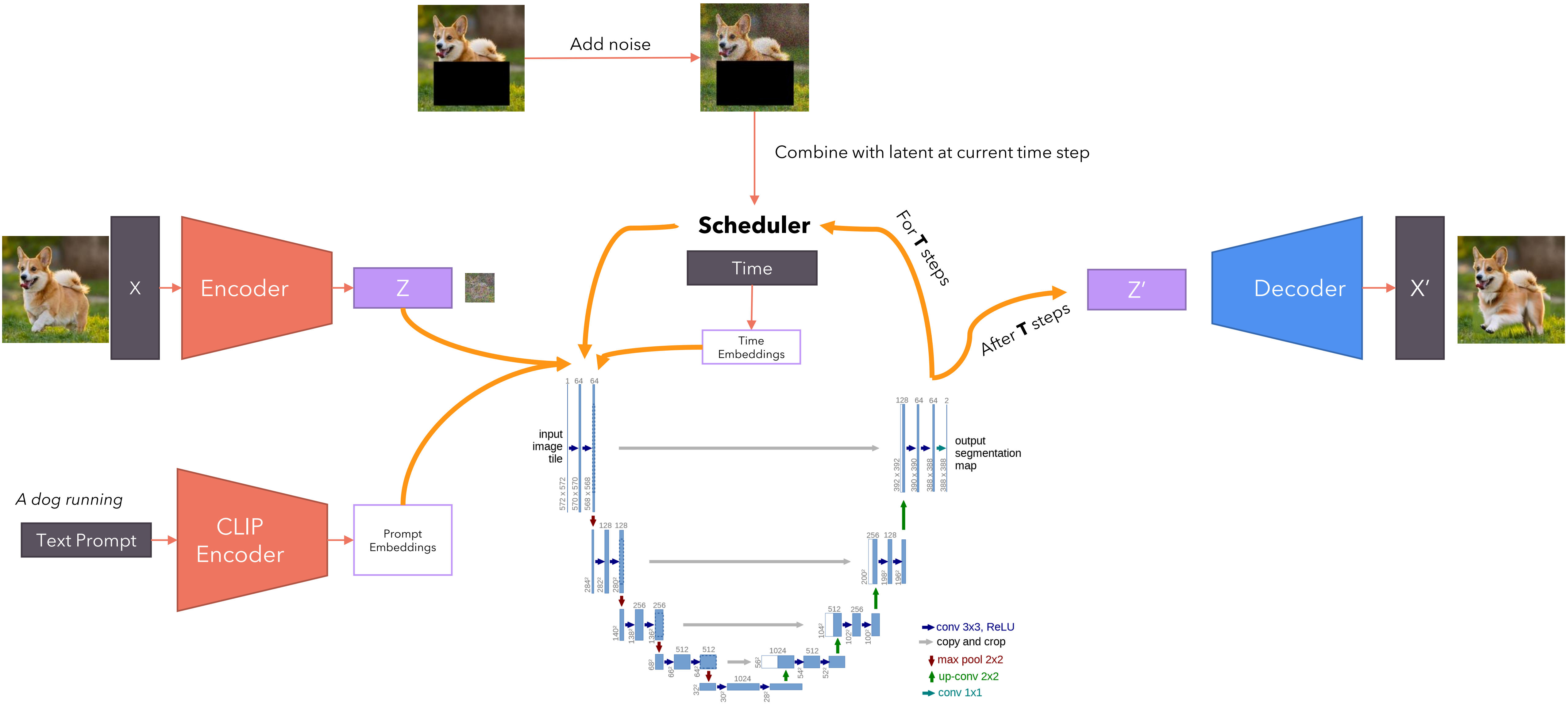
Umar Jamil - https://github.com/hkproj/pytorch-stable-diffusion

Architecture (Image-To-Image)



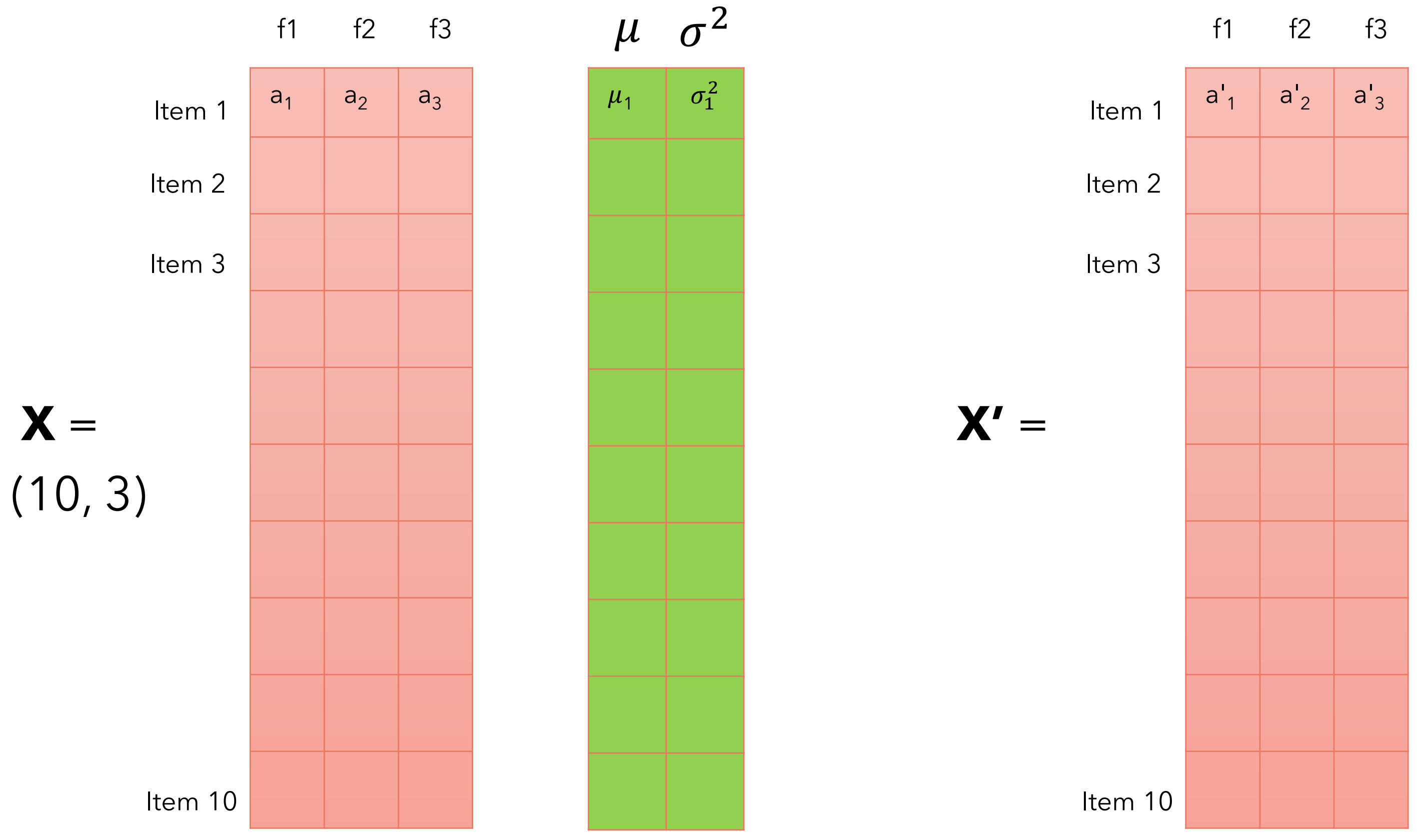
Umar Jamil - https://github.com/hkproj/pytorch-stable-diffusion

Architecture (In-Painting): how to fool models



Umar Jamil - https://github.com/hkproj/pytorch-stable-diffusion

Layer Normalization



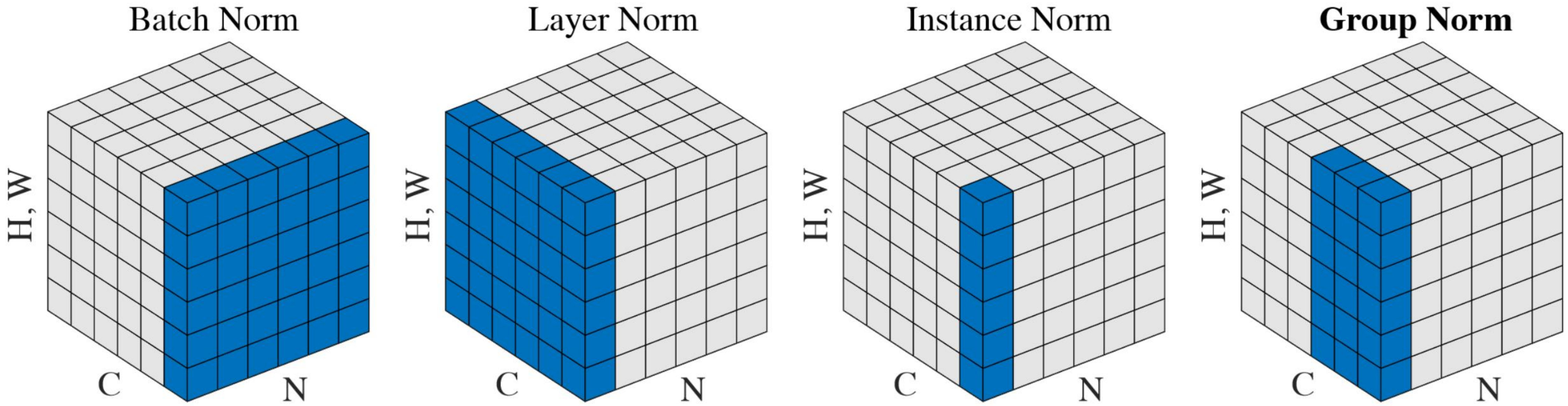
$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Each item is updated with its normalized value, which will turn it into a normal distribution with 0 mean and variance of 1.
- The two parameters **gamma** and **beta** are learnable parameters that allow the model to “amplify” the scale of each feature or apply a translation to the feature according to the needs of the loss function.

With batch normalization we normalize by **columns (features)**

With layer normalization we normalize by **rows (data items)**

Group Normalization



The full code is available on GitHub!

Full code: https://github.com/hkproj/pytorch-stable-diffusion

Special thanks to:

1. https://github.com/CompVis/stable-diffusion/
2. https://github.com/divamgupta/stable-diffusion-tensorflow
3. https://github.com/kjsman/stable-diffusion-pytorch
4. https://github.com/huggingface/diffusers/

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!