CLIP and Diffusion Models

MAC5921 - Deep Learning

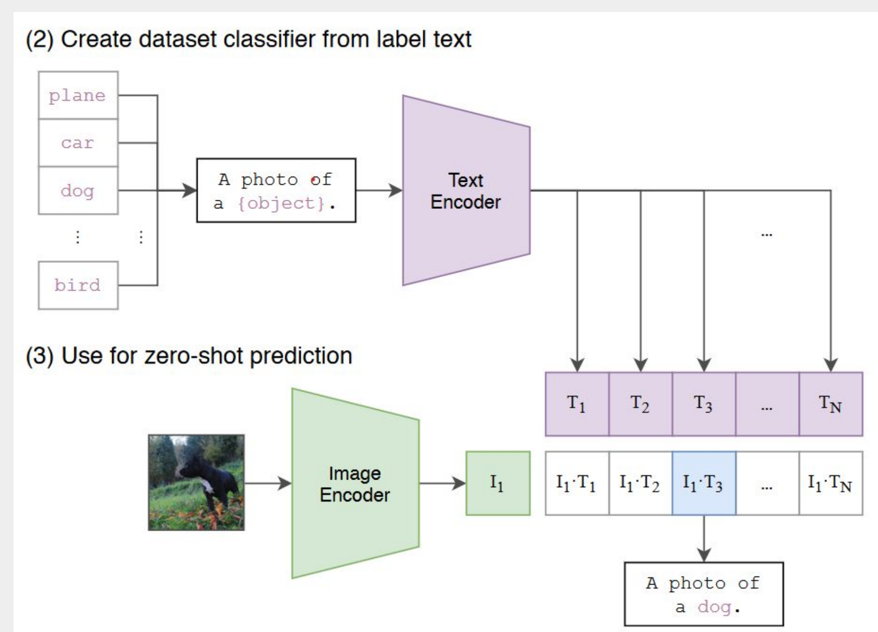
Instituto de Matemática e Estatística - IME-USP

Giulio Cesare Mastrocinque Santo

São Paulo, 14/11/2023

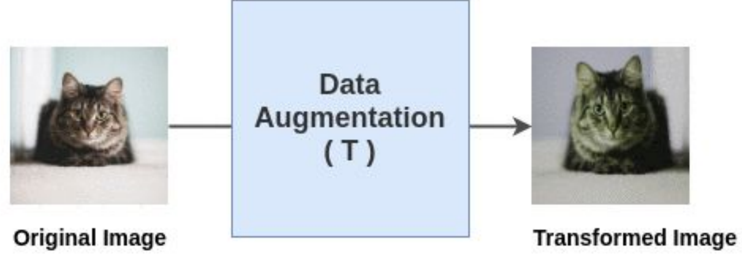
CLIP Model

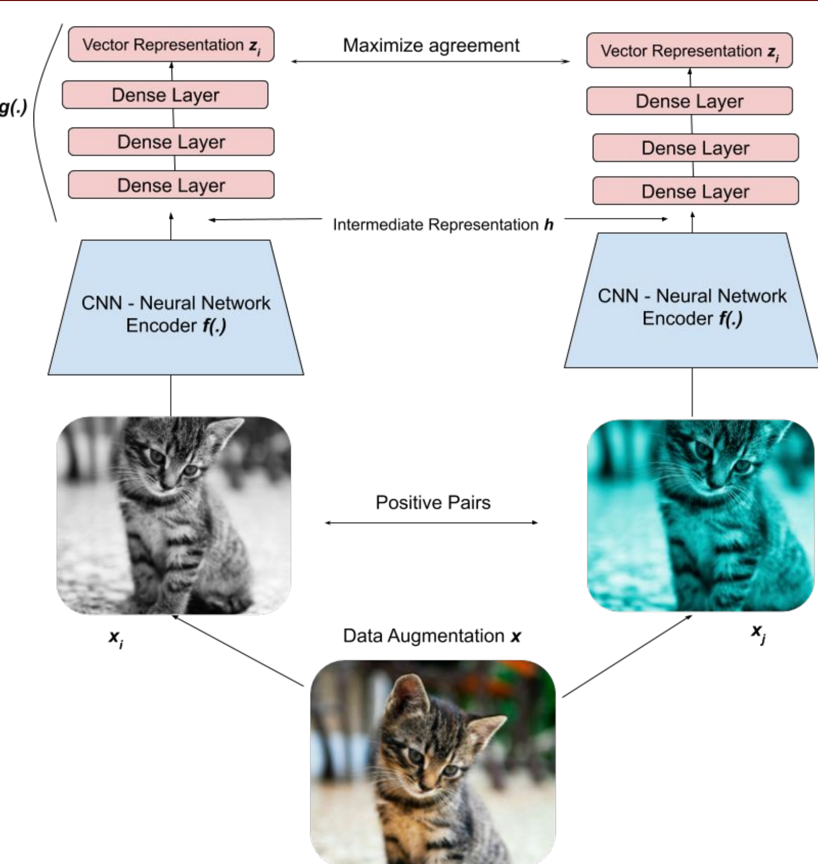
(Contrastive Language-Image Pre-Training)
2021



Images Source: *Learning Transferable Visual Models From Natural Language Supervision*

Contrastive Learning





Images Source: [Understanding Contrastive Learning]

CLIP - Overview

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description N>, <Image N>)

Image Encoder

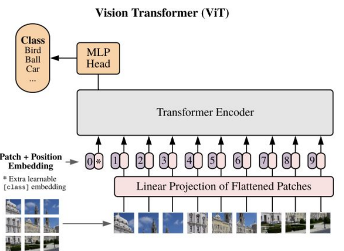
$\mathbb{R}^{I'}$

"An Image of A Bernese Mountain Dog

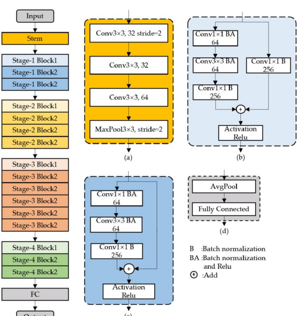
Text Encoder

$\mathbb{R}^{T'}$

CLIP - Overview

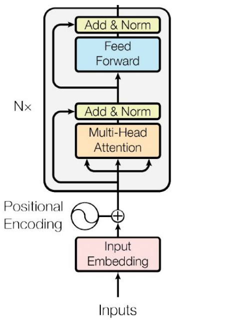


ViT (Vision Transformer)

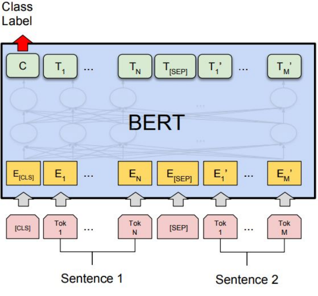


ResNet

...



Transformer Encoder



BERT

...

CLIP - Overview

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description N>, <Image N>)

Image Encoder

$\mathbb{R}^{I'}$

“An Image of A Bernese Mountain Dog

Text Encoder

$\mathbb{R}^{T'}$

CLIP - Overview

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description N>, <Image N>)

Image Encoder

$\mathbb{R}^{I'}$

MatMul

nn.Linear($\mathbb{R}^{I}$,$\mathbb{R}^{E}$)

$\mathbb{R}^{E}$

$\mathbb{R}^{I}$

"An Image of A Bernese Mountain Dog

Text Encoder

$\mathbb{R}^{T'}$

CLIP - Overview

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description N>, <Image N>)

Image Encoder

$\mathbb{R}^{I'}$

MatMul

$\mathbb{R}^{I}$

nn.Linear($\mathbb{R}^{I}$,$\mathbb{R}^{E}$)

$\mathbb{R}^{E}$

"An Image of A Bernese Mountain Dog

Text Encoder

$\mathbb{R}^{T'}$

MatMul

$\mathbb{R}^{T}$

nn.Linear($\mathbb{R}^{T}$,$\mathbb{R}^{E}$)

$\mathbb{R}^{E}$

CLIP - Overview

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description N>, <Image N>)

Image Encoder

$\mathbb{R}^{I'}$

MatMul

$\mathbb{R}^{I}$

nn.Linear($\mathbb{R}^{I}$,$\mathbb{R}^{E}$)

$\mathbb{R}^{E}$

L2 Norm

$\mathbb{R}^{E}$

Cosine

"An Image of A Bernese Mountain Dog

Text Encoder

$\mathbb{R}^{T'}$

MatMul

$\mathbb{R}^{T}$

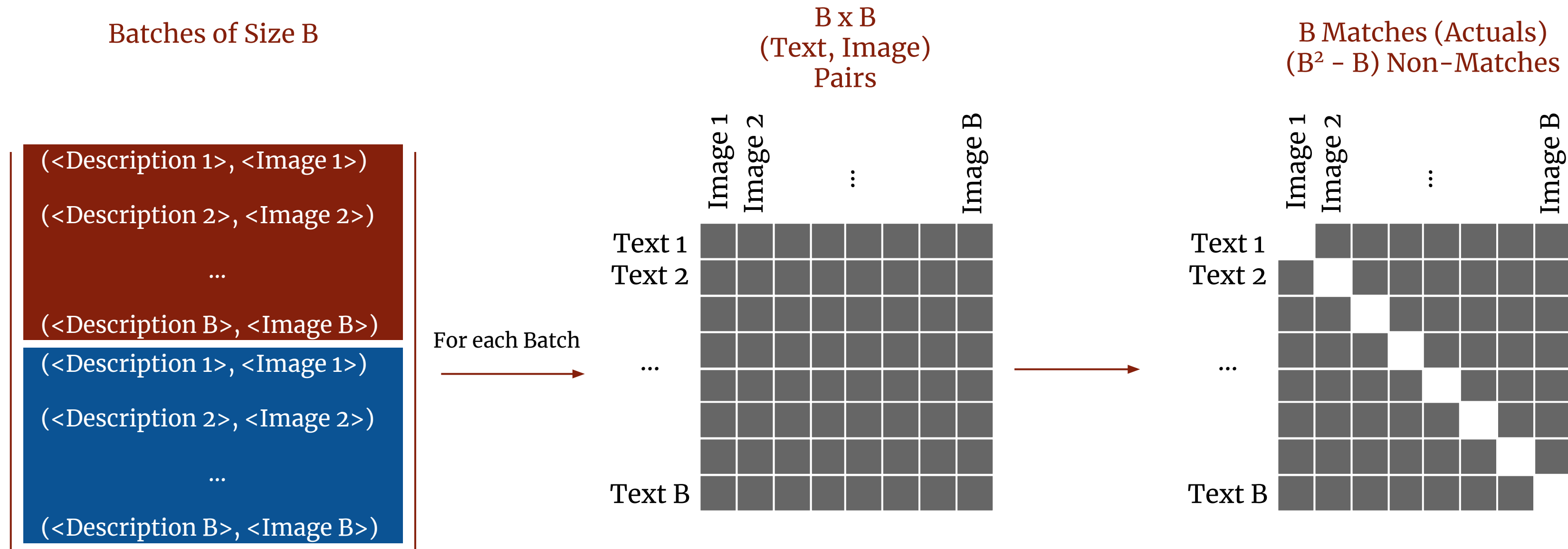
nn.Linear($\mathbb{R}^{T}$,$\mathbb{R}^{E}$)

$\mathbb{R}^{E}$

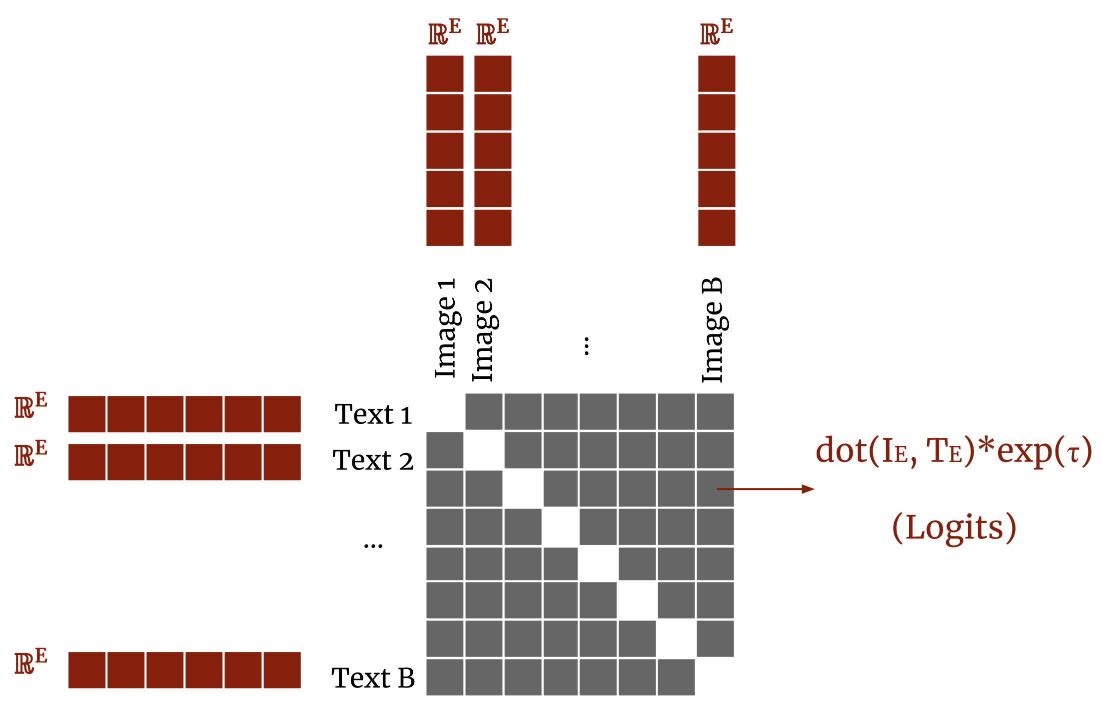
L2 Norm

$\mathbb{R}^{E}$

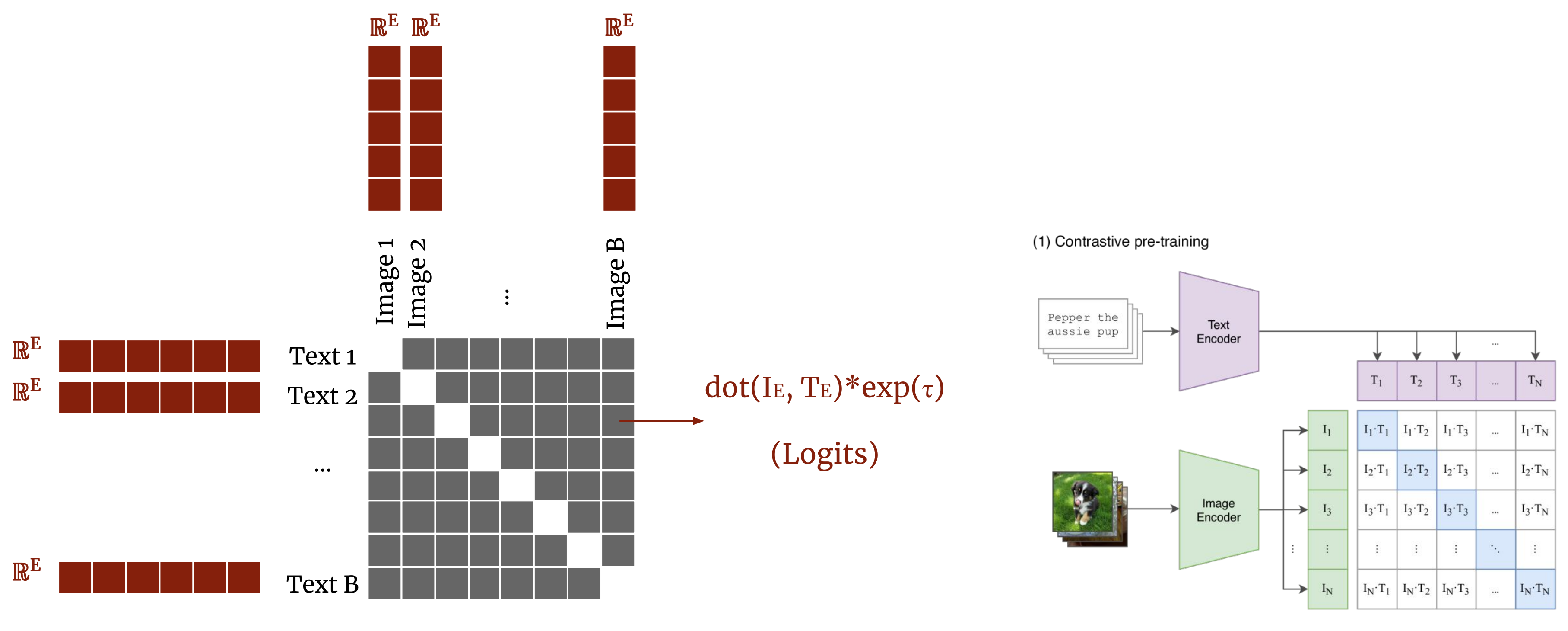
CLIP - Training Process



CLIP - Training Process



CLIP - Training Process



Loss Function

Batches of Size B

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description B>, <Image B>)

...

"B" Classes Multiclass Classification Problem

	"Tabular Dataset"				Actuals	Actuals One-Hot Encoded			
Text 1 (Sample 1)	$p(0)$	$p(1)$	...	$p(B)$	0	1	0	...	0
Text 1 (Sample 2)	$p(0)$	$p(1)$	...	$p(B)$	1	0	1	...	0
	...	...	...	...		...	...	...	...
Text 1 (Sample B)	$p(0)$	$p(1)$	...	$p(B)$	B	0	0	...	1

B x B Matrix
Rows: Samples
Column j: Probability of Belong to Class j

Loss Function

Batches of Size B

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description B>, <Image B>)

...

What About The Images???

Text 1 (Sample 1)

Text 1 (Sample 2)

Text 1 (Sample B)

"B" Classes Multiclass Classification Problem

"Tabular Dataset"

$p(0)$	$p(1)$	...	$p(B)$
$p(0)$	$p(1)$	...	$p(B)$
...	...	...	...
$p(0)$	$p(1)$	...	$p(B)$

B x B Matrix
Rows: Samples
Column j: Probability of Belong to Class j

Actuals

0
1
B

Actuals One-Hot Encoded

1	0	...	0
0	1	...	0
...	...	...	...
0	0	...	1

Loss Function

Batches of Size B

(<Description 1>, <Image 1>)

(<Description 2>, <Image 2>)

...

(<Description B>, <Image B>)

...

Repeat For the Images!!!

Image 1 (Sample 1)

Image 2 (Sample 1)

Image B (Sample 1)

"B" Classes Multiclass Classification Problem

"Tabular Dataset"

$p(0)$	$p(1)$	...	$p(B)$
$p(0)$	$p(1)$	...	$p(B)$
...	...	...	...
$p(0)$	$p(1)$	...	$p(B)$

B x B Matrix
Rows: Samples
Column j: Probability of Belong to Class j

Actuals

0
1
B

Actuals One-Hot Encoded

1	0	...	0
0	1	...	0
...	...	...	...
0	0	...	1

End-to-End Pseudo-Code

```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Image Source: Learning Transferable Visual Models From Natural Language Supervision

End-to-End Pseudo-Code

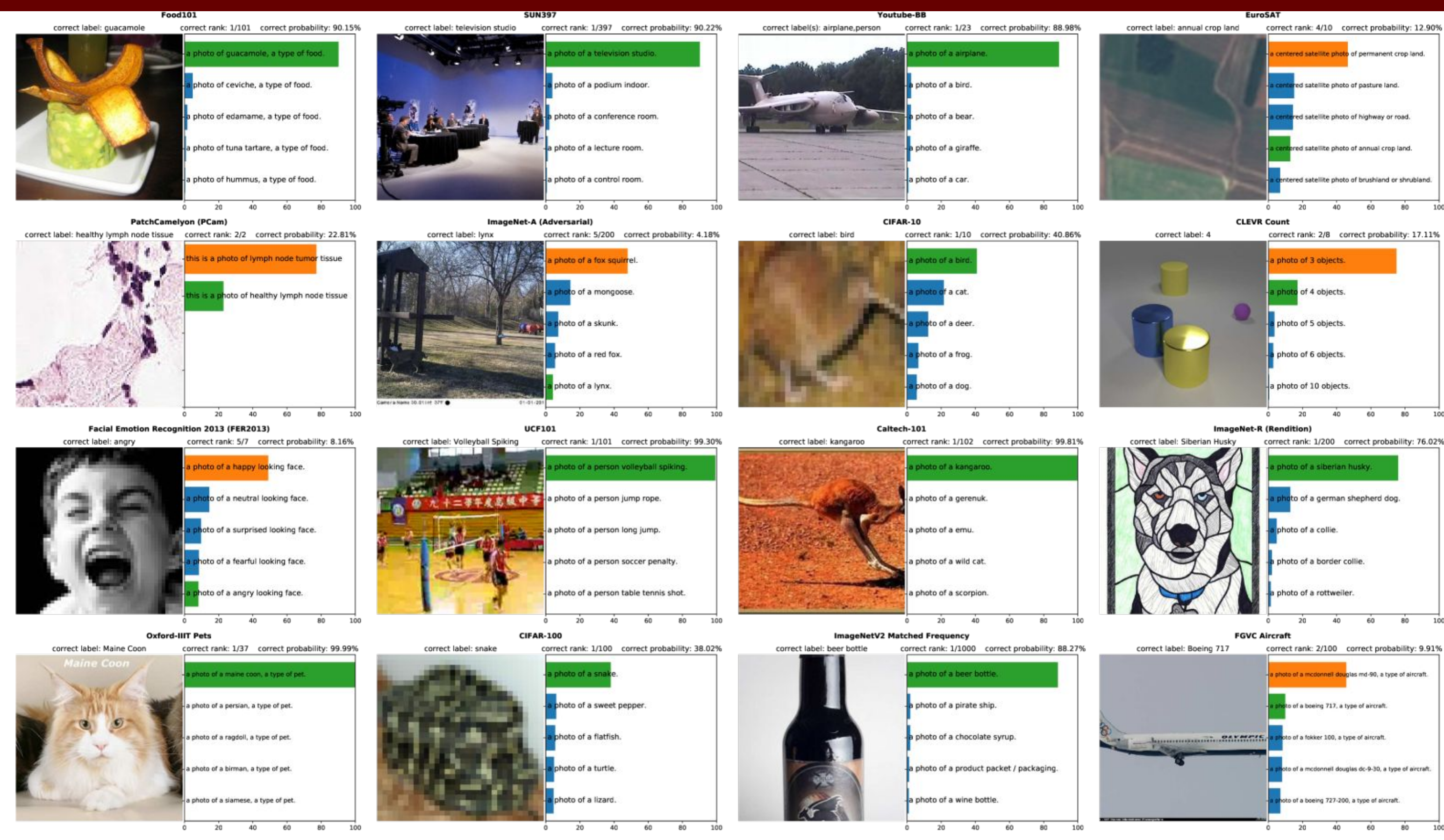
```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Table 19: CLIP-ResNet hyperparameters

Hyperparameter	Value
Batch size	32768
Vocabulary size	49408
Training epochs	32
Maximum temperature	100.0
Weight decay	0.2
Warm-up iterations	2000
Adam β_1	0.9
Adam β_2	0.999 (ResNet), 0.98 (ViT)
Adam ϵ	10^{-8} (ResNet), 10^{-6} (ViT)

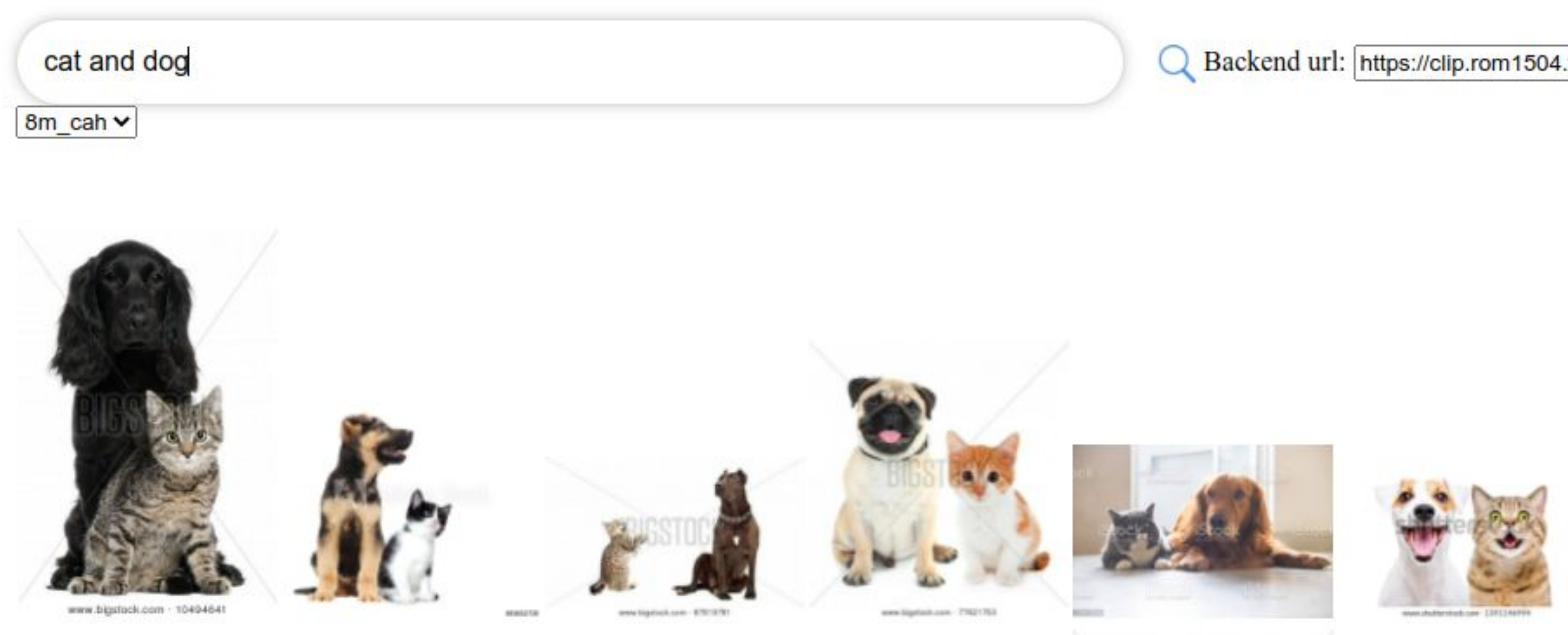
Images Source: *Learning Transferable Visual Models From Natural Language Supervision*

Use Cases - Zero-shot Classification



Images Source: [Learning Transferable Visual Models From Natural Language Supervision]

Use Cases - Text-Image Retrieval



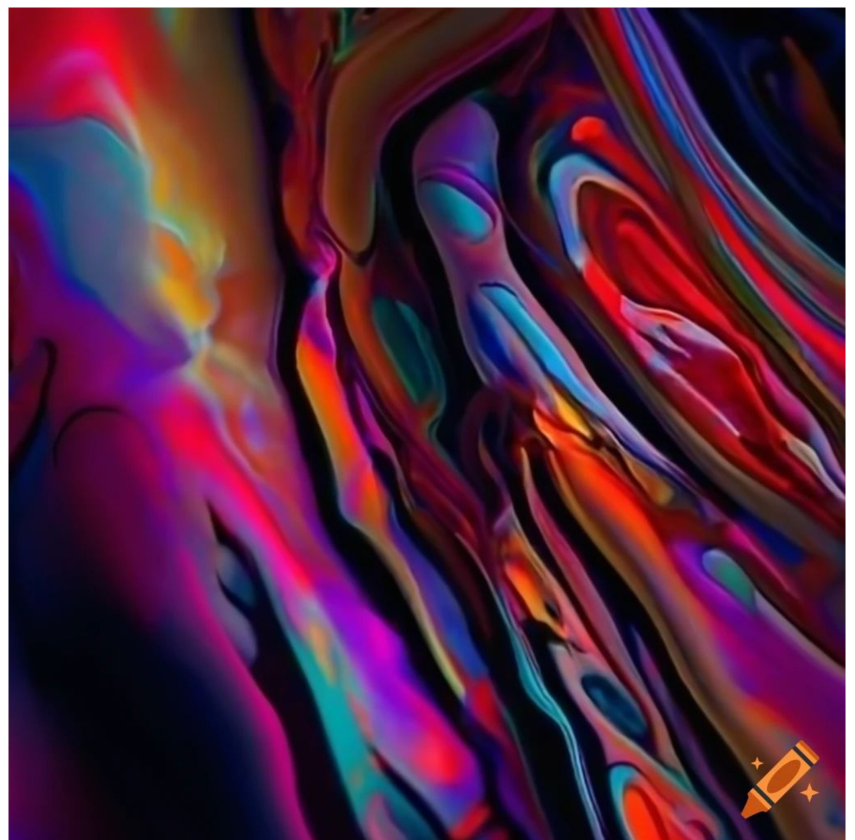
Images Source: https://pypi.org/project/clip-retrieval/

CLIP References

[1] Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G. & Sutskever, I.. (2021). Learning Transferable Visual Models From Natural Language Supervision. <i>Proceedings of the 38th International Conference on Machine Learning</i>, in <i>Proceedings of Machine Learning Research</i> 139:8748-8763 Available from https://proceedings.mlr.press/v139/radford21a.html.

[2] Understanding Contrastive Learning

Diffusion Models



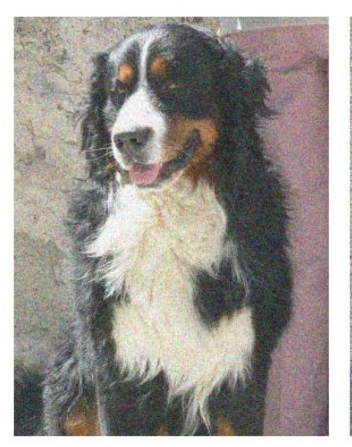
"An abstract image about the subject Diffusion Models" Generated with craiyon.com (DALL·E mini)

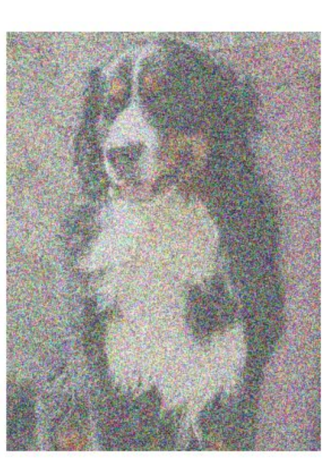
Why Diffusion?

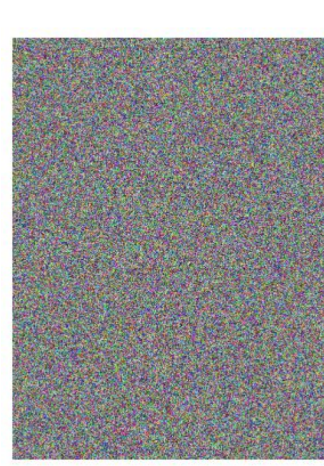
Thermodynamics

Diffusion: Move from high Concentration to low concentration until equilibrium

Diffusion Models

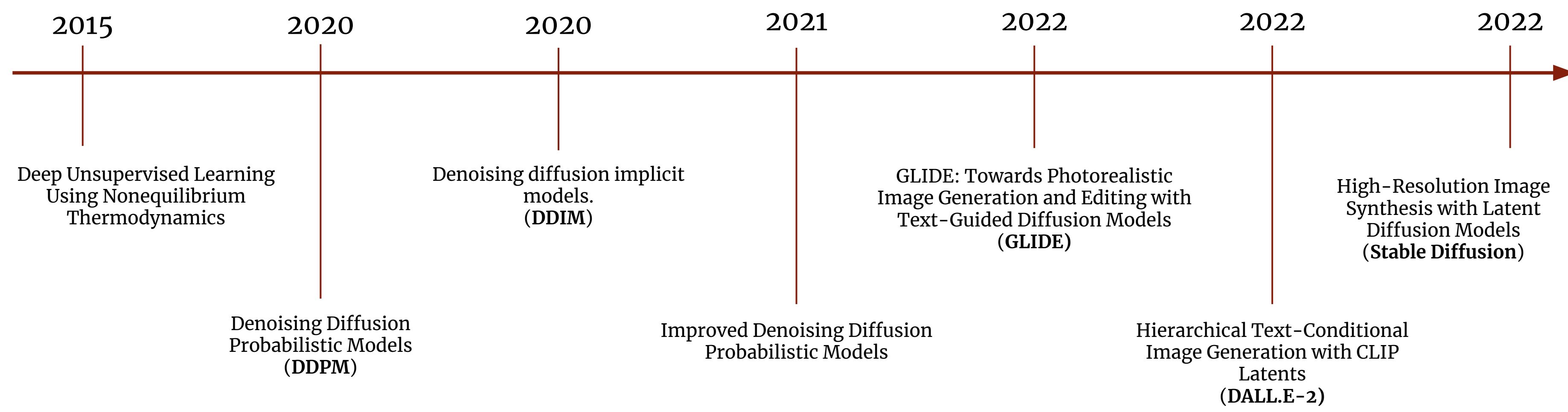






Diffusing noise into the image

Diffusion Models Timeline



Comparing Generative Models

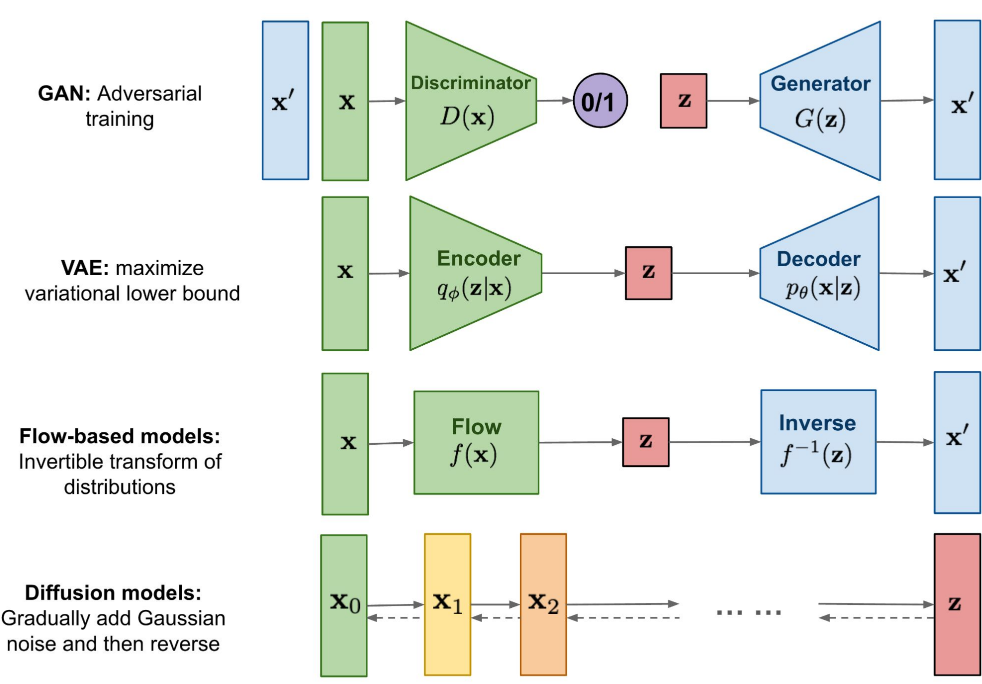


Image Source: Weng, Lilian. (Jul 2021). What are diffusion models?

Similarities

1. Learn $p(X)$ through $p_\theta(X)$, i.e., make $p(X) \sim p_\theta(X)$
2. Generate new data by sampling from $p_\theta(X)$

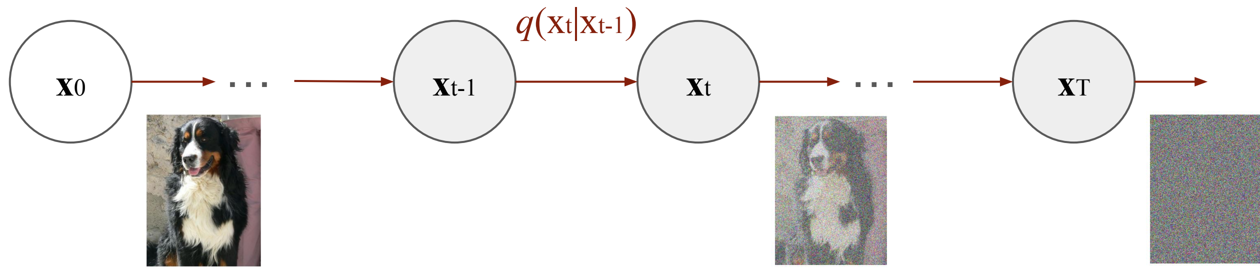
Differences

GAN, VAE: Noise → Image in a ***Single Step***; Encoder-Decoder Architecture

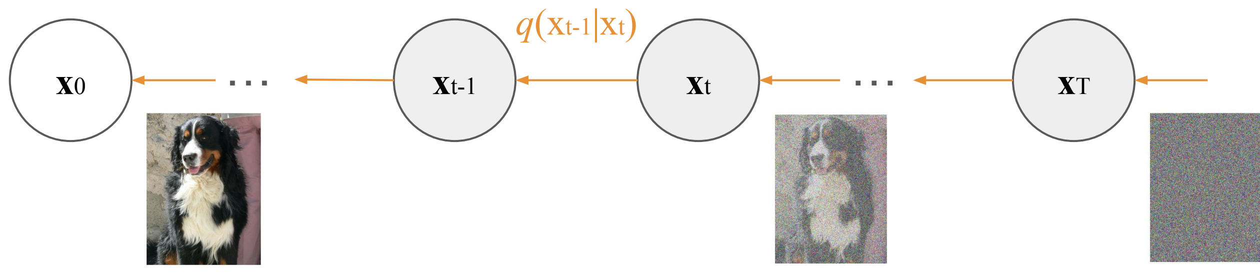
Diffusion: Noise → Image in ***Multiple Steps***; Markov Chain

Diffusion Models - **Forward** and **Reverse**

"Encoder"
Forward Process

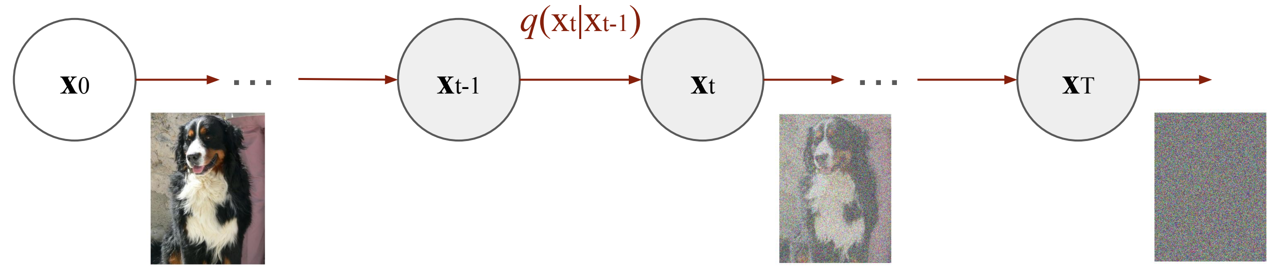


"Decoder"
Reverse Process

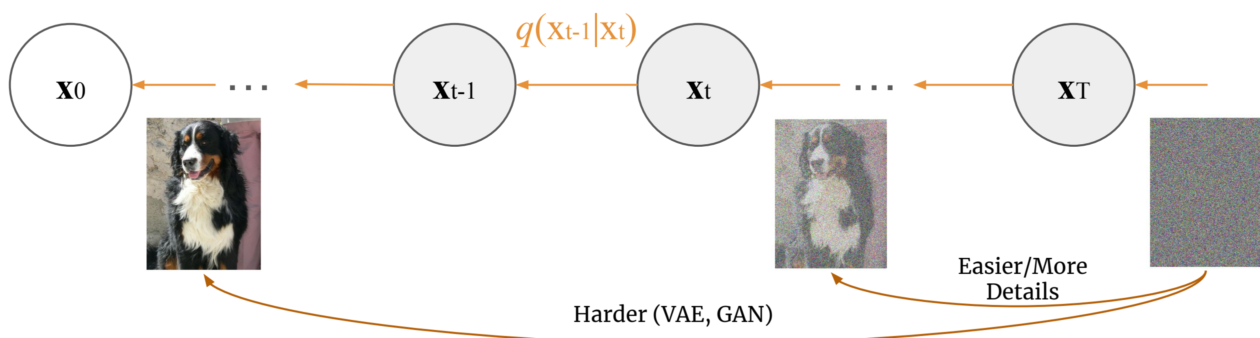


Diffusion Models - **Forward** and **Reverse**

“Encoder”
Forward Process



“Decoder”
Reverse Process



Forward Diffusion Process

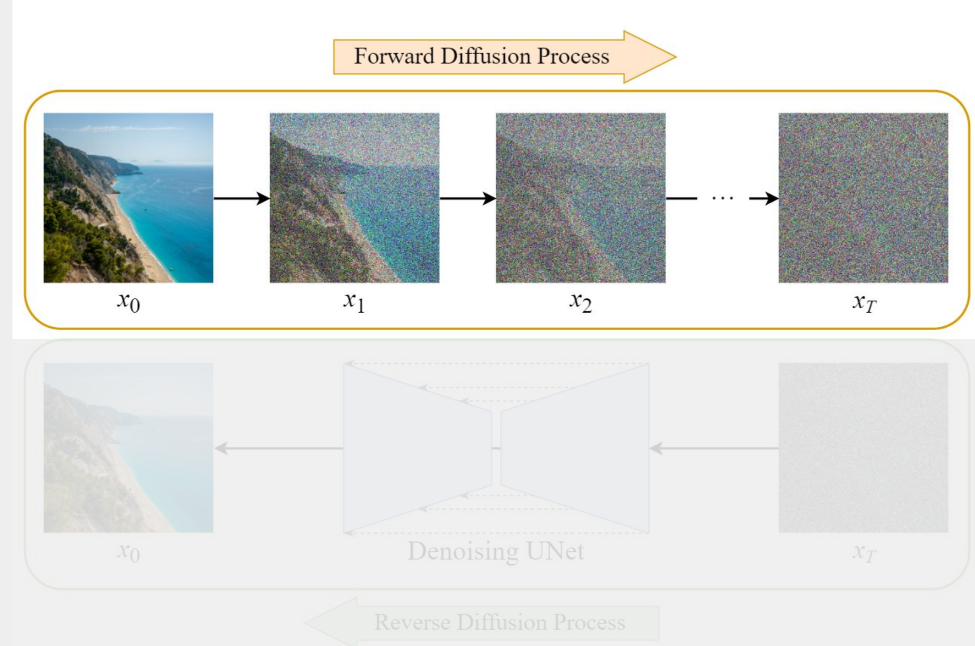
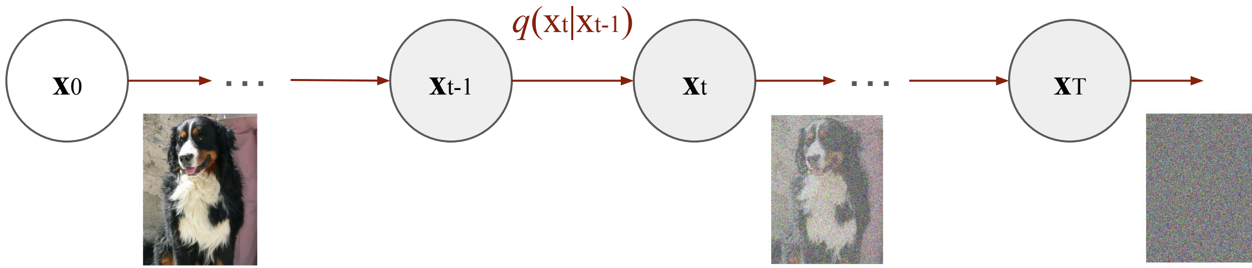


Image Source: Diffusion Model Clearly Explained!

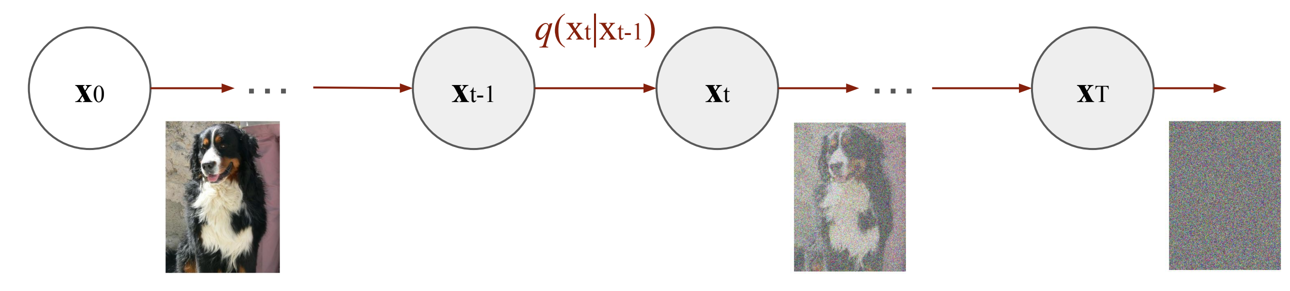
“Encoder”
Forward Process



- Include more noise at each step until pure noise N(0,1)
- The noise follow a scheduling

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?

"Encoder" Forward Process

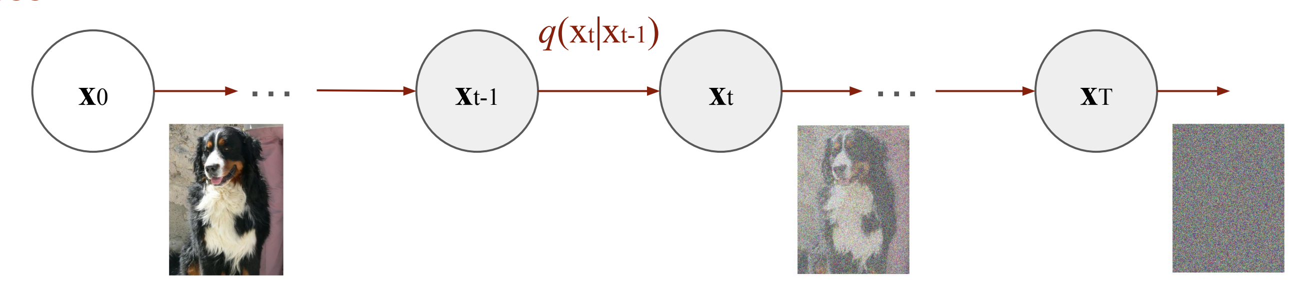


Noise Distribution at *t*	$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$
Image With Noise at T from X0	$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$
Noise Schedule	$\{\beta_t \in (0,1)\}_{t=1}^{T}$

Diffusion Models - **Forward Process**

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?

"Encoder" Forward Process



Noise Distribution at *t*	$q(\mathbf{x}_t\|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$
Image With Noise at T from X0	$q(\mathbf{x}_{1:T}\|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t\|\mathbf{x}_{t-1})$
Noise Schedule	$\{\beta_t \in (0,1)\}_{t=1}^{T}$

Closed Form →

$$\alpha_t = 1 - \beta_t$$
$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$$

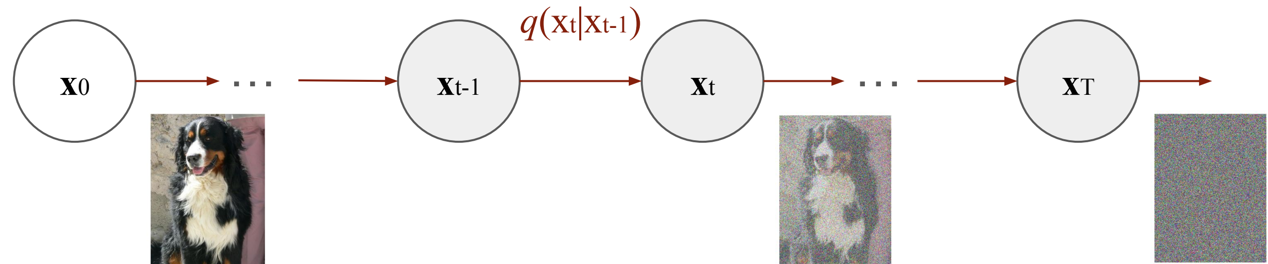
$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}\right)$$

$$\begin{aligned}
\mathbf{x}_t &= \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \\
&= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \\
&= \dots \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon
\end{aligned}$$

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}\right)$$

Diffusion Models - **Forward Process**

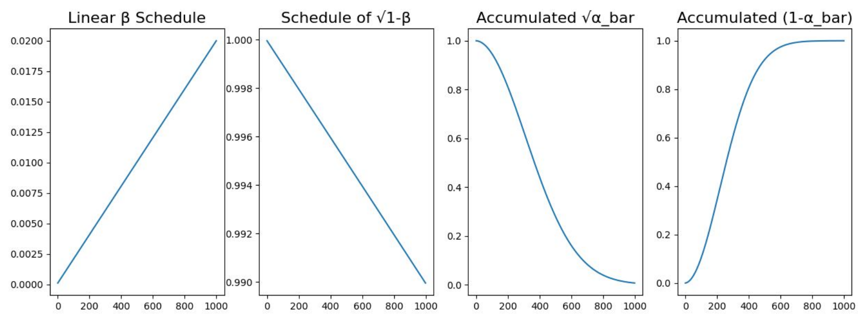
“Encoder”
Forward Process



$$\alpha_t = 1 - \beta_t$$
$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$$

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0,\, (1 - \bar{\alpha}_t)\mathbf{I}\right)$$

```
timesteps = 1000

β_0 = 1e-4
β_final = 0.02
β_t_list = (β_final - β_0) * np.linspace(0, 1, timesteps) + β_0

α_t_list = []
α_bar_t_list = []
for t in range(timesteps):
    α_t = 1 - β_t_list[t]
    α_t_list.append(α_t)
    α_bar_t = 1
    for i in range(len(α_t_list)):
        α_bar_t*=α_t_list[i]
    α_bar_t_list.append(α_bar_t)
```



Reverse Diffusion Process

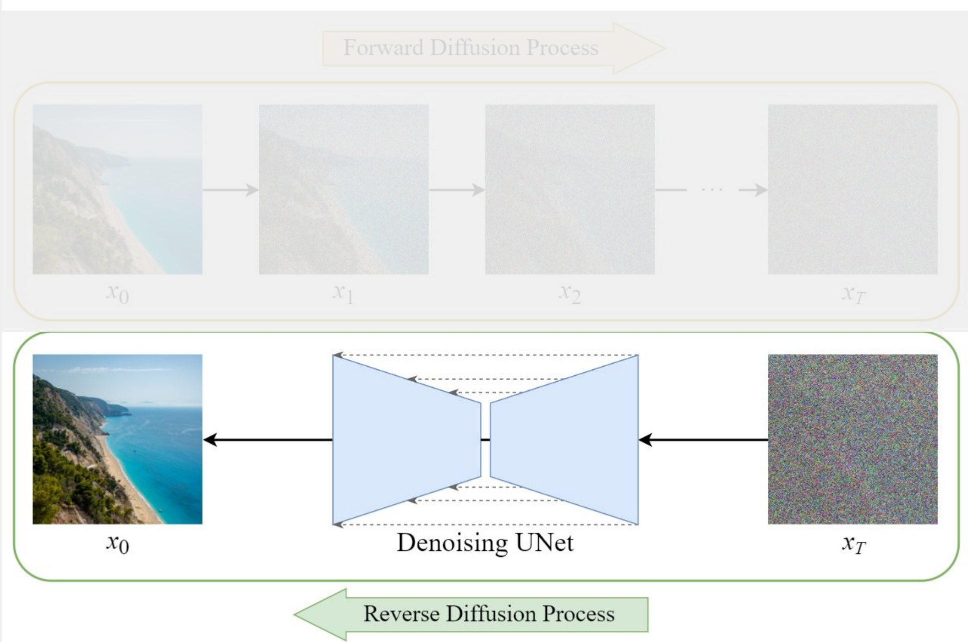
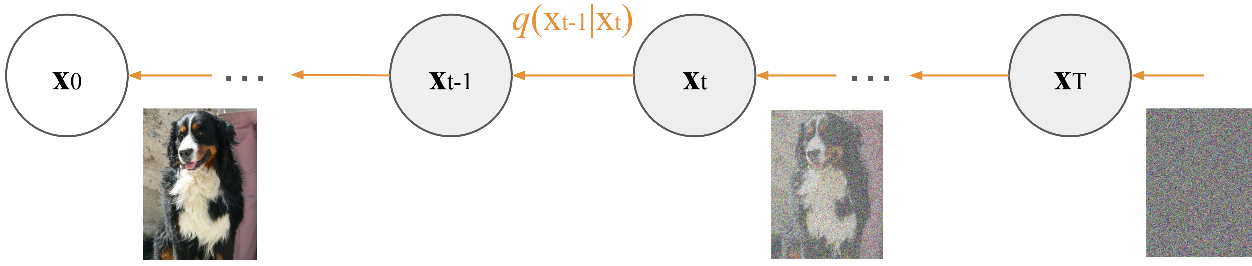


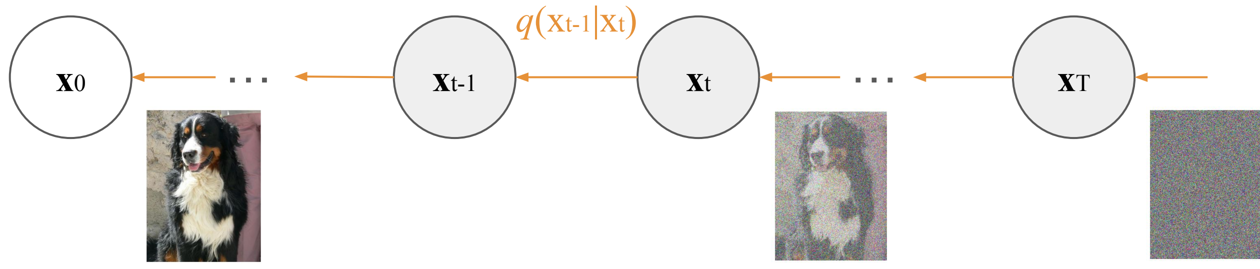
Image Source: Diffusion Model Clearly Explained!

"Decoder"
Reverse Process



- Gradually remove the noise until the image is fully recovered
- The idea is to estimate somehow $q(x_{t-1}|x_t)$

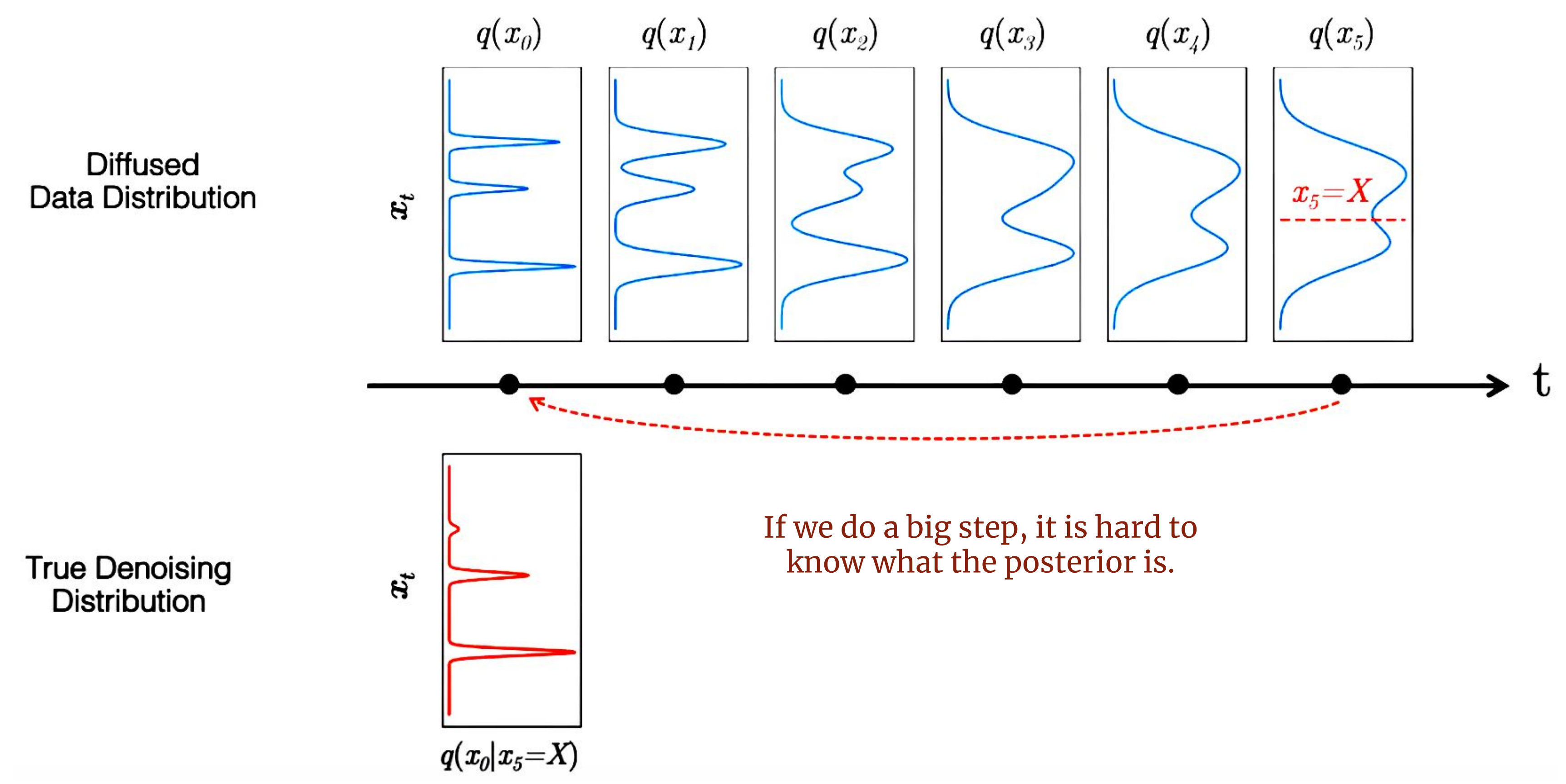
“Decoder”
Reverse Process



- Gradually remove the noise until the image is fully recovered
- The idea is to estimate somehow $q(\mathrm{x}_{t-1}|\mathrm{x}_t)$

$$q(x_{t-1} \mid x_t) = q(x_t \mid x_{t-1}) \frac{q(x_{t-1})}{q(x_t)} \qquad q(x_t) = \int q(x_t \mid x_{t-1}) q(x_{t-1}) \mathrm{d}x$$

Diffusion Models - **Reverse Process**



If we do a big step, it is hard to know what the posterior is.

Image Source: CS 198-126: Lecture 12 - Diffusion Models

Diffusion Models - **Reverse Process**

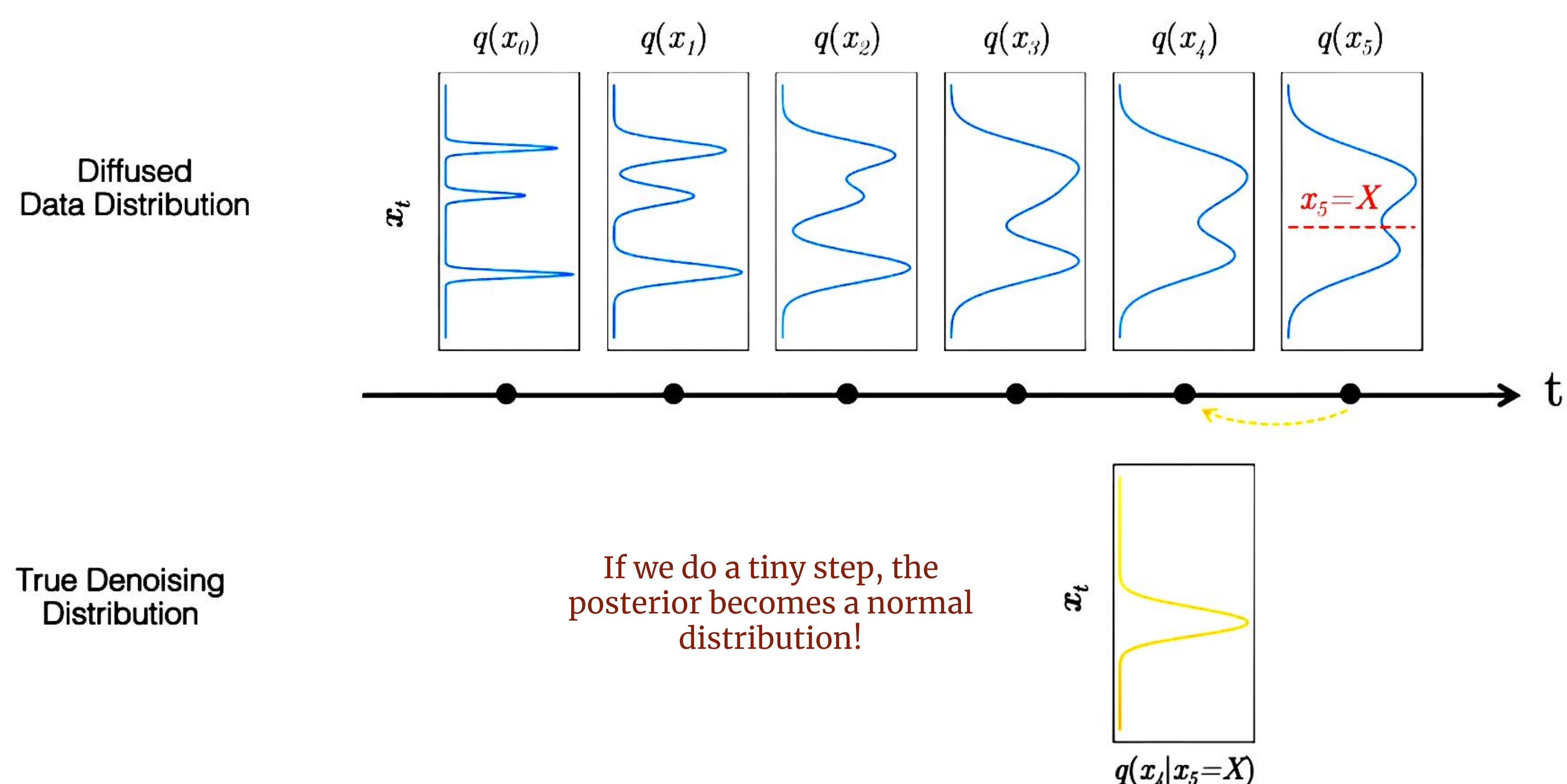
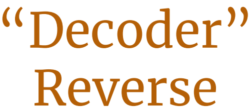
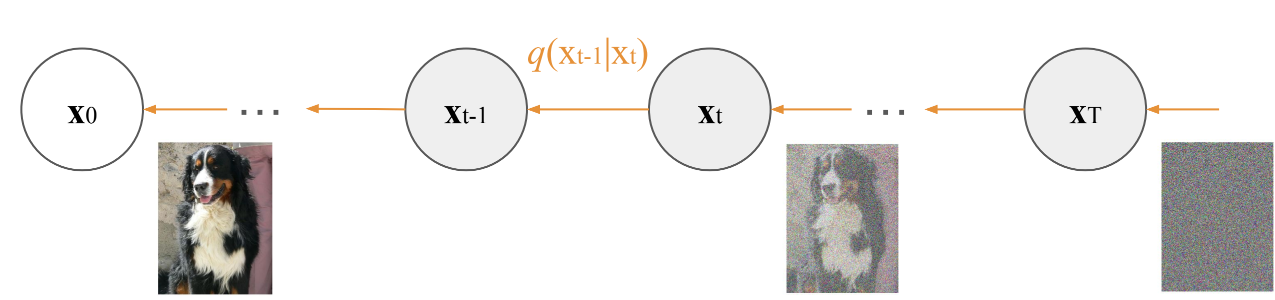


Image Source: CS 198-126: Lecture 12 - Diffusion Models

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?





Approximating the posterior by a Gaussian Distribution:

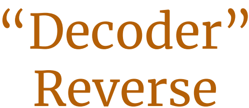
Posterior Conditioned on X0

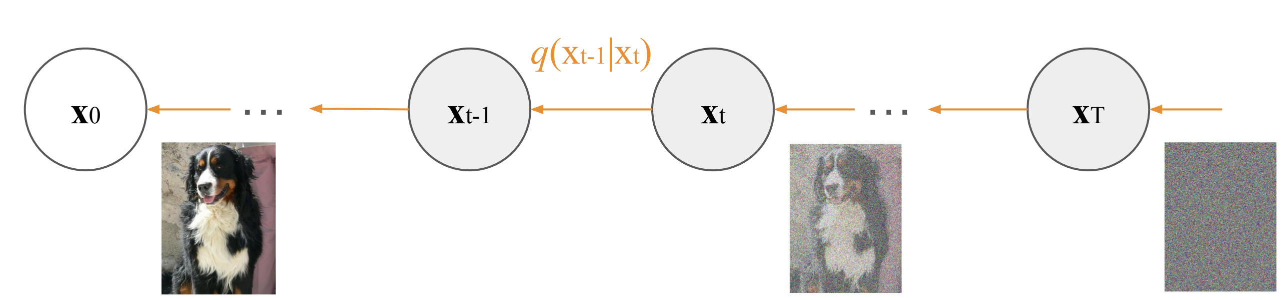
$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$

$$\tilde{\beta}_t = 1/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t$$

$$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$$

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?





Approximating the posterior by a Gaussian Distribution:

Posterior Conditioned on X0

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$

$$\tilde{\beta}_t = 1/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t$$

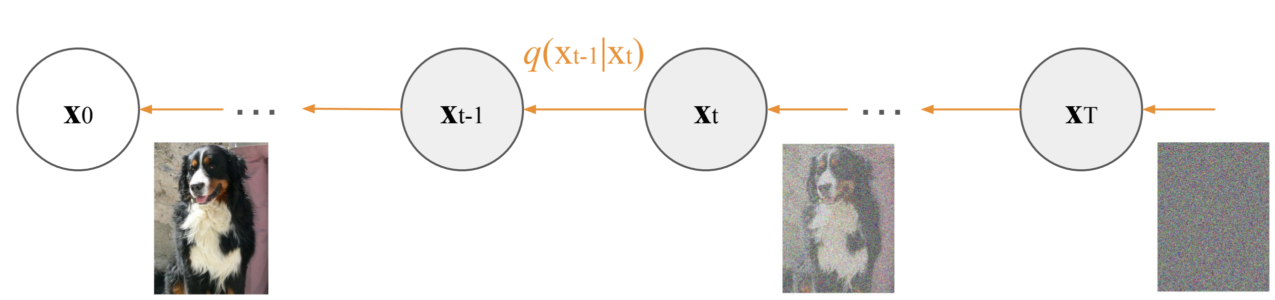
$$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$$

Estimation (Model)

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?



Approximating the posterior by a Gaussian Distribution:

Posterior Conditioned on X0

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$

$$\tilde{\beta}_t = 1/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t$$

$$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$$

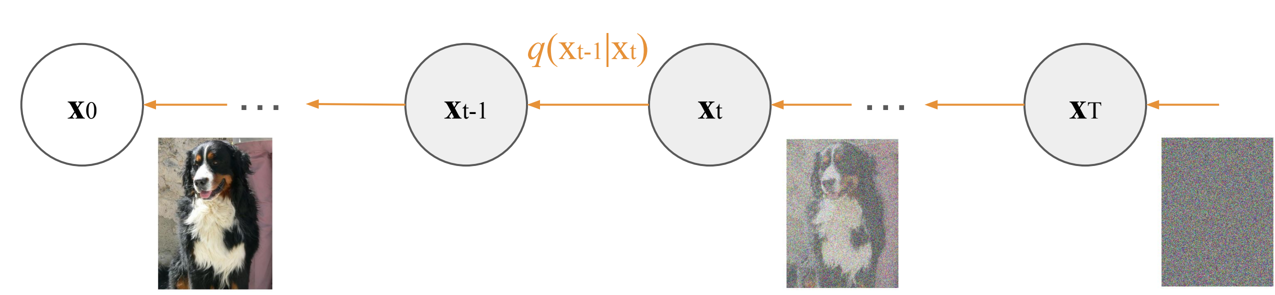
Estimation (Model)

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Recovered Image

Equations Source: Weng, Lilian. (Jul 2021). What are diffusion models?



Approximating the posterior by a Gaussian Distribution:

Loss Function	$L_t = \mathbb{E}_{\mathbf{x}_0,\epsilon}\Big[\frac{1}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t)\|_2^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t,t)\|^2\Big]$ (PS: There are more components! See the reference!)
Actual Mean	$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$
Prediction	$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t,t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t))$ → Machine Learning Model

Hands-On

End-to-End - **Version 1**

Pre-compute Scheduling

```
timesteps = 1000

β_0 = 1e-4
β_final = 0.02
β_t_list = (β_final - β_0) * np.linspace(0, 1, timesteps) + β_0

α_t_list = []
α_bar_t_list = []
for t in range(timesteps):
    α_t = 1 - β_t_list[t]
    α_t_list.append(α_t)
    α_bar_t = 1
    for i in range(len(α_t_list)):
        α_bar_t*=α_t_list[i]
    α_bar_t_list.append(α_bar_t)
```

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

The pseudo-code on the right was based on https://github.com/ThiagoLira/ToyDiffusion

The pseudo-code on the left was based on code provided in How Diffusion Models Work *DeepLearni.AI Short Course*

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):          # Diffusion Steps

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Apply Noise

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}\right)$$

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Actual and Prediction

$$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$$

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I})$$

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Actual and Prediction

$$\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0$$

Notice: The Network Must Learn with t, so it understand the noise level at step t

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Loss Function and Backpropagation

$$L_t = \mathbb{E}_{\mathbf{x}_0,\epsilon}\left[\frac{1}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t,t)\|_2^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t,t)\|^2\right]$$

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Start From Noise

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Remember We Are Only Predicting μ

$$\tilde{\beta}_t = 1/(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t$$

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

Neural Network Prediction

End-to-End - **Version 1**

```
###Training
Ts = 1000
for epoch in epochs:
    for t in range (0, Ts):

        # α_bar_t
        α_bar_t = α_bar_t_list[t]
        cov_t = torch.eye(X.shape[0])*(1 - α_bar_t)

        #Forward
        Xt = torch.distributions.MultivariateNormal(
            loc = torch.sqrt(α_bar_t)*X0,
            covariance_matrix = cov_t
        ).sample()

        #Reverse
        #Get q(Xt-1|Xt, X0)
        actual_μ = get_actual_μ(Xt, X0)
        pred_μ = NN(Xt, t)

        L = (actual_μ-pred_μ)**2
        L.backward()

###Denoising
X = noise
for t in range (Ts, 0):
    α_bar_t = α_bar_t[t]
    α_bar_t_1 = α_bar_t[t-1]

    β_t_tilde = β_t*(1 - α_bar_t_1)/(1 - α_bar_t)
    pred_μ = NN(X, t)
    X = torch.distributions.MultivariateNormal(
        loc = pred_μ,
        covariance_matrix = torch.eye(X.shape[0])*β_t_tilde
    ).sample()
```

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

End-to-End - **Version 2**

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

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

Image Source: Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc., Red Hook, NY, USA, Article 574, 6840–6851.

End-to-End - **Version 2**

```
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar\alpha_t}\mathbf{x}_0 + \sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

The pseudo-code on the left was based on code provided in How Diffusion Models Work *DeepLearni.AI Short Course*

End-to-End - **Version 2**

```
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar\alpha_t}\mathbf{x}_0 + \sqrt{1-\bar\alpha_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

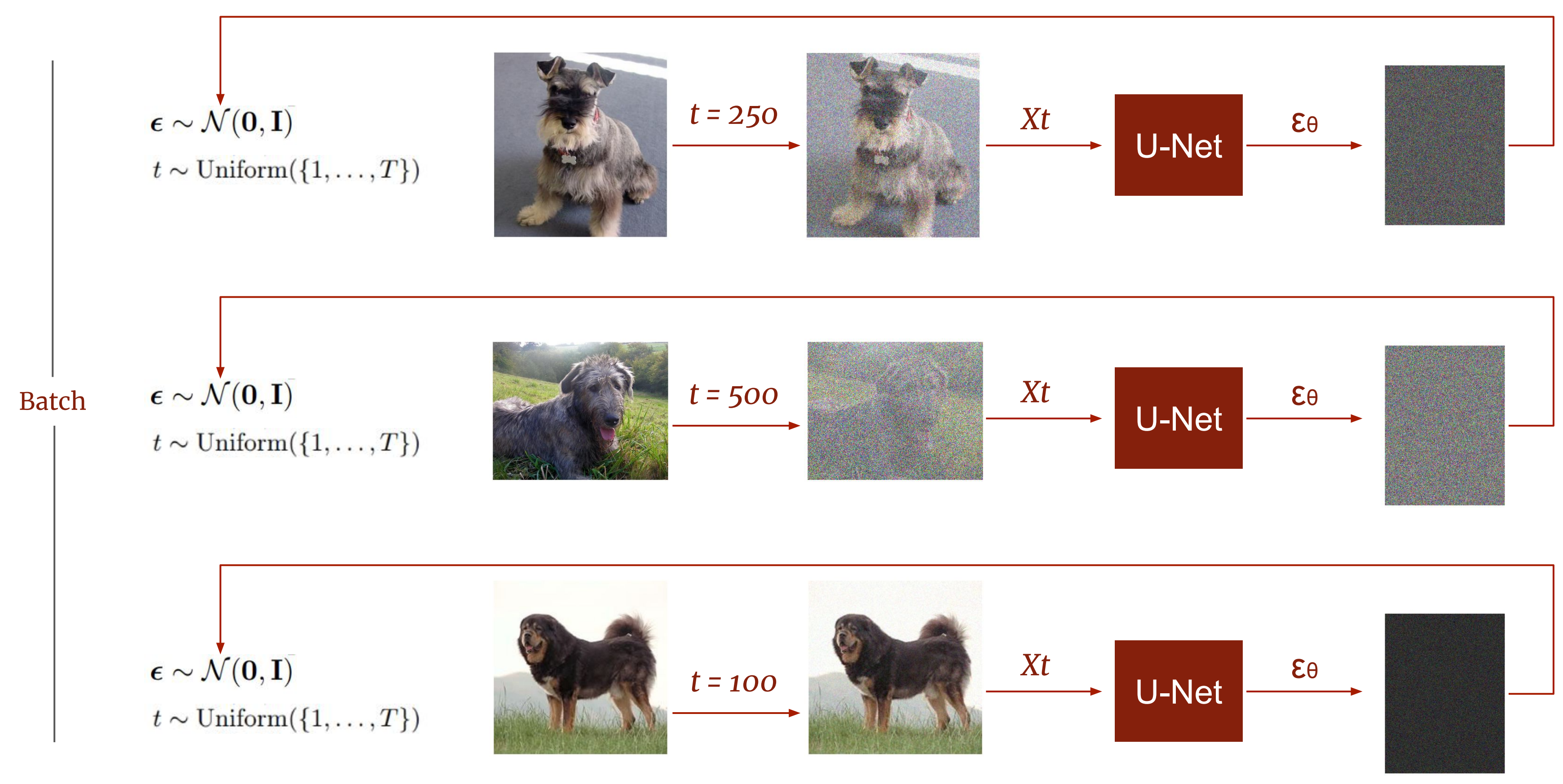
End-to-End - **Version 2**

```
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

End-to-End - **Version 2**



End-to-End - **Version 2**

```
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 2 Sampling

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

End-to-End - **Version 2**

```
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 2 Sampling

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

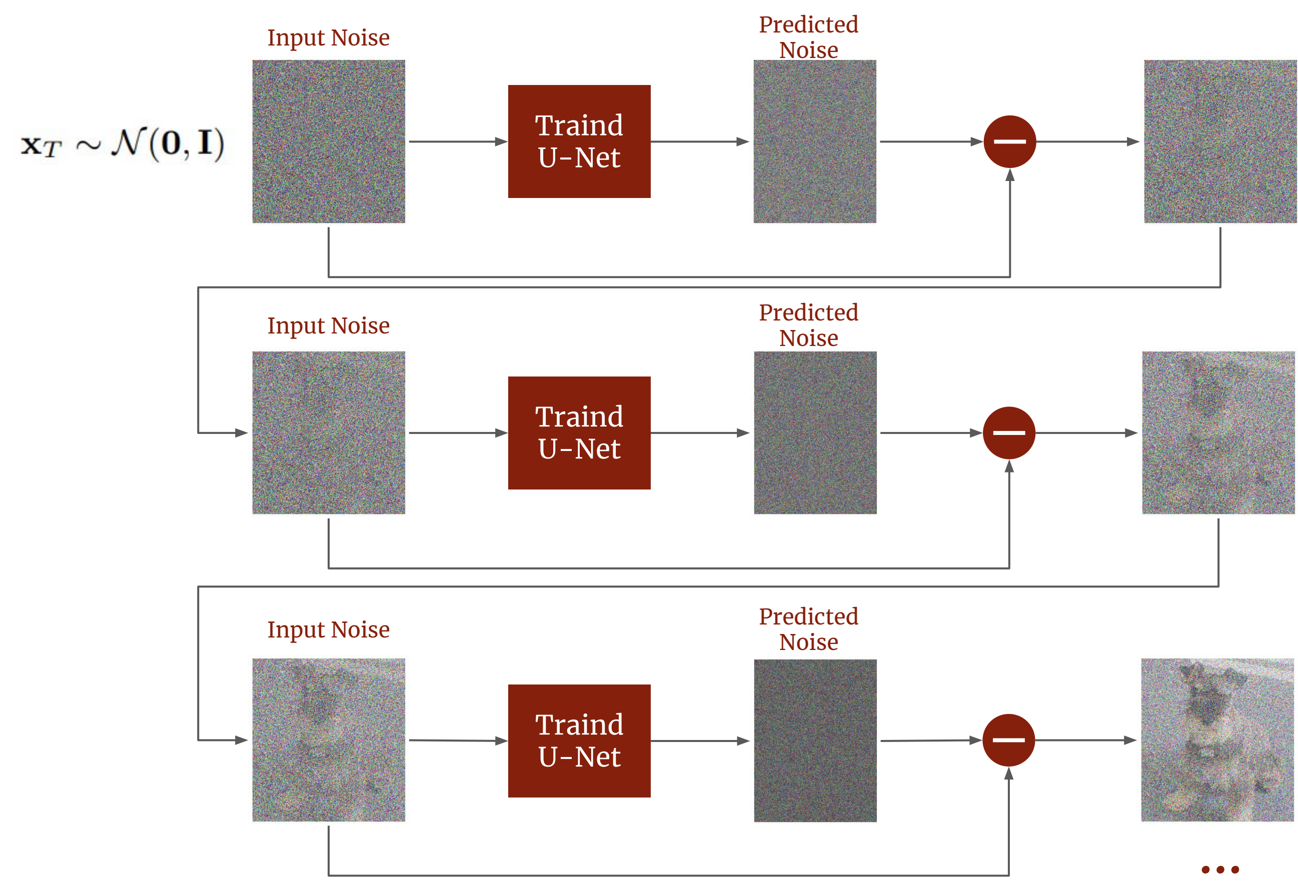
End-to-End - **Version 2**

```python
##Training
Ts = 1000
for epoch in epochs:
    for image_batch in dataloader:

        t_batch = torch.randint(1, Ts + 1, (image_batch.shape[0],)) #One t for each image in the batch

        noise_batch = torch.randn_like(image_batch)

        α_bar_t_batch = α_bar_t_list[t_batch] #One for each image in the batch

        Xt_batch = torch.sqrt(α_bar_t_batch)*image_batch + (1 - α_bar_t_batch)*noise_batch

        pred_noise_batch = nn_model(Xt_batch, t_batch)

        loss = mse_loss(pred_noise_batch, noise_batch)

        loss.backward()

##Denoising
Xt = torch.randn(n_sample, 3, height, height).to(device) #Generating batch of images

for t in range (Ts, 0):

    α_bar_t = α_bar_t_list[t]
    α_t = α_t_list[t]
    β_t = β_t_list[t]

    if t > 1:
        z = torch.randn_like(x)
    else:
        z = torch.zeros_like(x)

    t = torch.tensor([t])[:, None, None, None] #one t for each image

    pred_noise_batch = nn_model(Xt, t)

    Xt = (1/torch.sqrt(α_t))*(
        Xt - ((1-α_t)/(torch.sqrt(1-α_bar_t)))*pred_noise_batch
    ) + torch.sqrt(β_t) * z
```

Algorithm 2 Sampling

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

End-to-End - **Version 2**



Model Architecture and Guidance

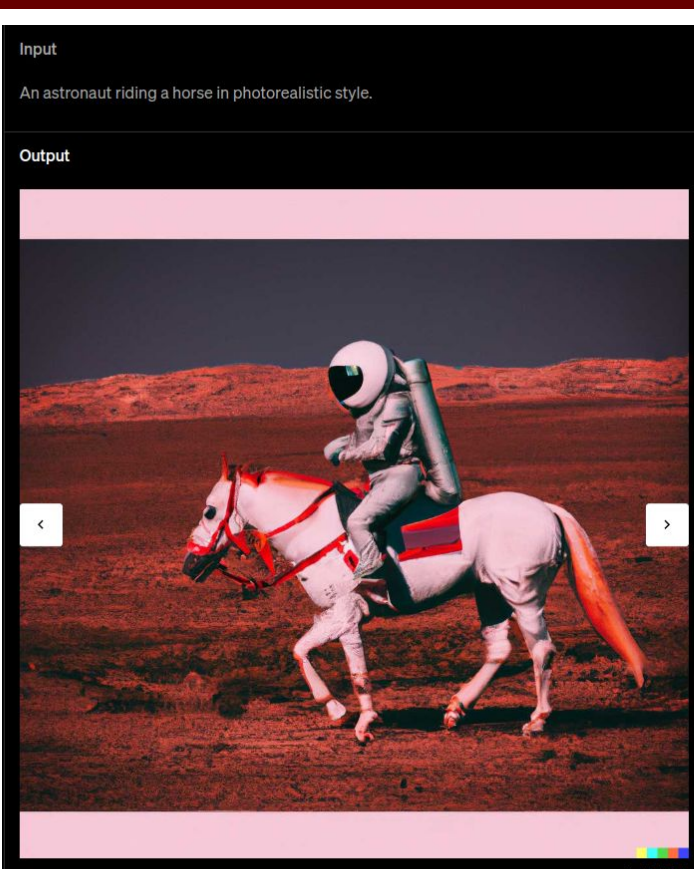


Image Source: https://openai.com/dall-e-2

Model Architecture and Guidance

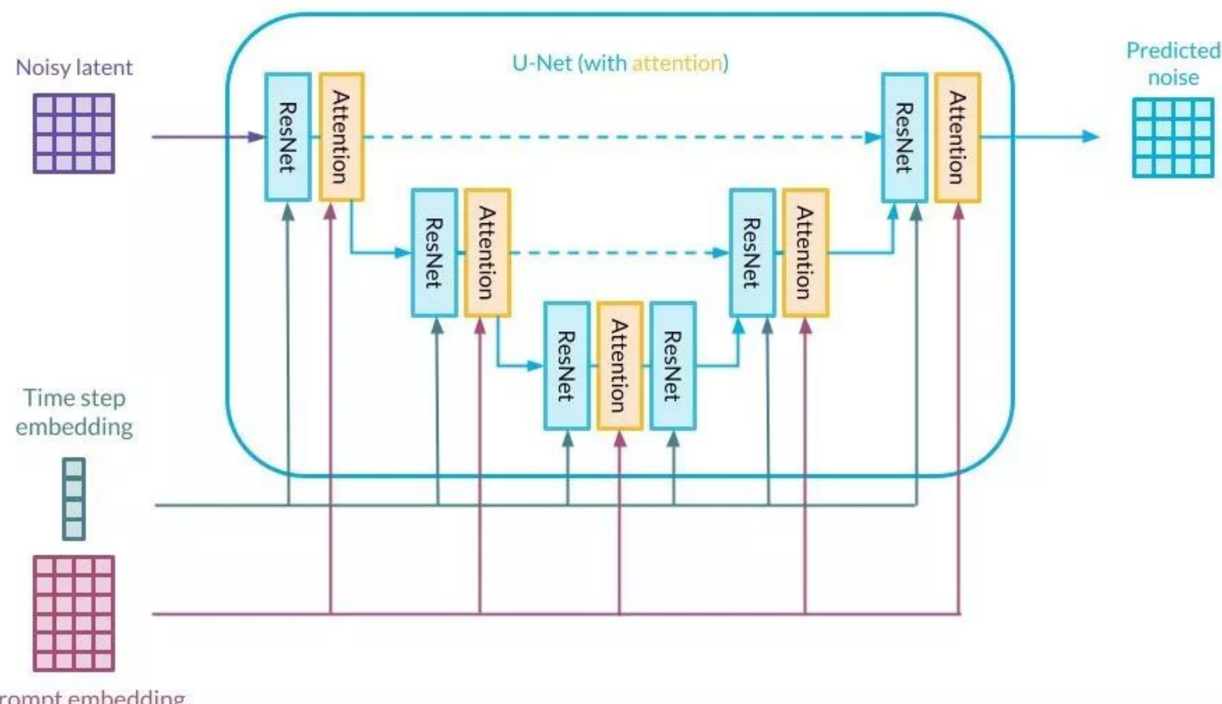
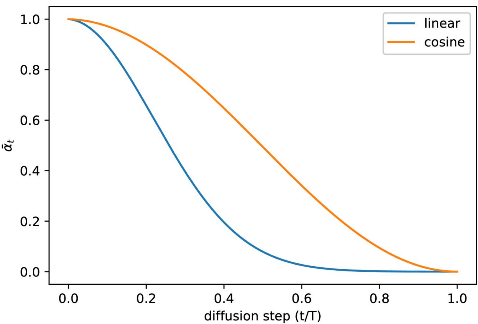


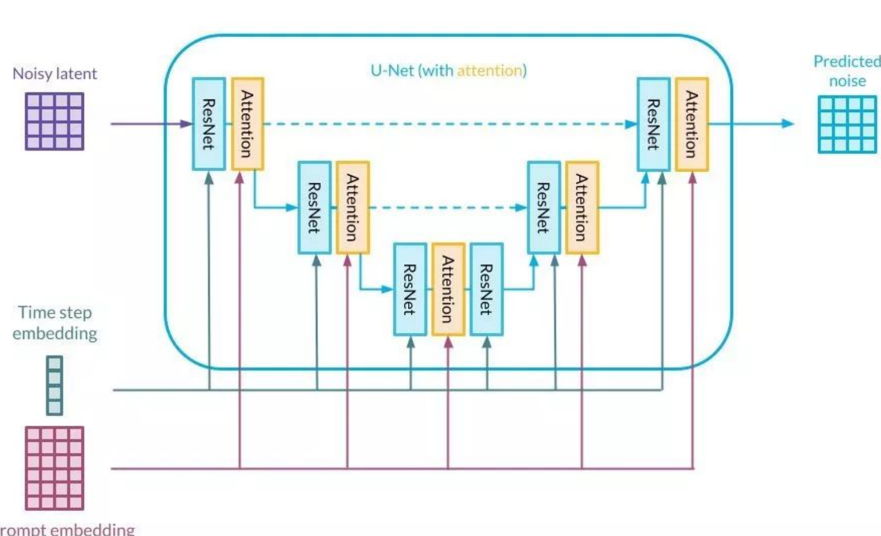
Image Source: Diffusion models in practice. Part 1: A primers

Linear VS Cosine Schedule



Images Source: *Nichol, Alex and Prafulla Dhariwal. "Improved Denoising Diffusion Probabilistic Models."*

Model Architecture and Guidance



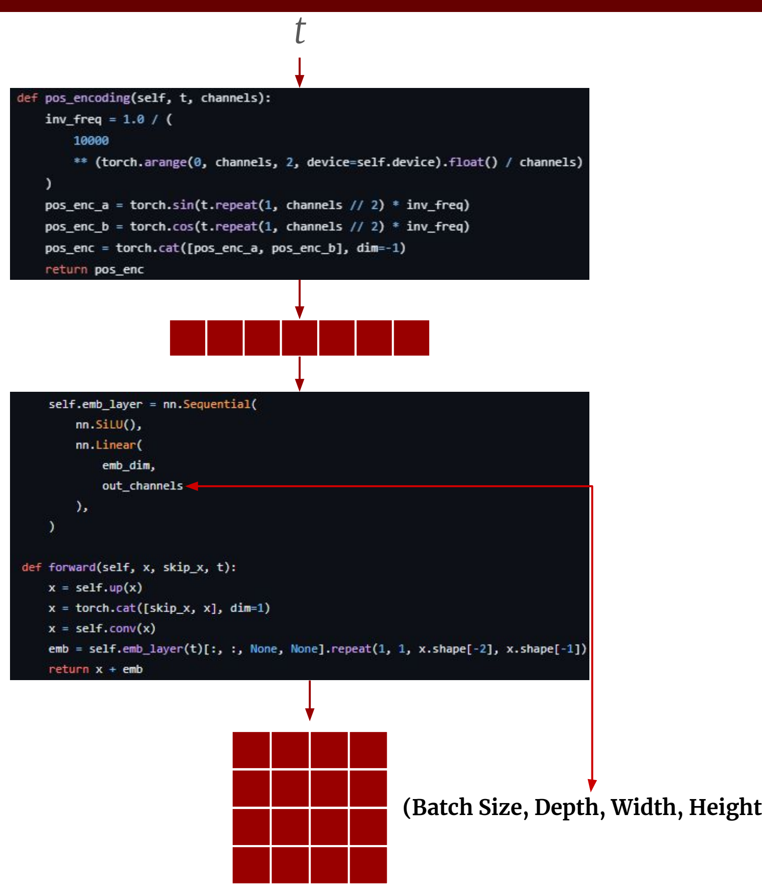


Image Source: Diffusion models in practice. Part 1: A primers

Code Extracted From https://github.com/dome272/Diffusion-Models-pytorch/blob/main/modules.py

Diffusion Training

- Text go through a Transformer Encoder
- “The final token embedding is used in place of a class embedding in the Diffusion model” - GLIDE paper
- “The last layer of token embeddings (a sequence of K feature vectors) is separately projected to the dimensionality of each attention layer throughout the Diffusion Model” - GLIDE paper

Diffusion Inference

- CLIP is first trained on noisy images;
- CLIP Guidance:
  - *cosine_similarity*(denoised image, text) -> CLIP Score
  - Add the gradient of the CLIP Score to the predicted image

$$\hat{\mu}_\theta(x_t|c) = \mu_\theta(x_t|c) + s \cdot \Sigma_\theta(x_t|c)\nabla_{x_t}(f(x_t) \cdot g(c))$$

DALL.E-2

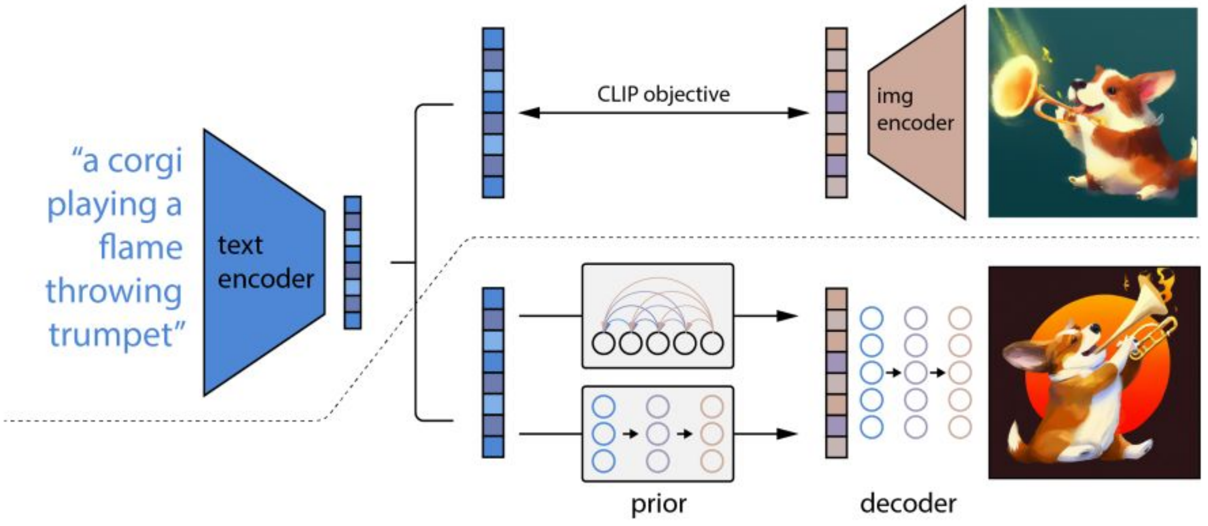


Figure 2: A high-level overview of unCLIP. Above the dotted line, we depict the CLIP training process, through which we learn a joint representation space for text and images. Below the dotted line, we depict our text-to-image generation process: a CLIP text embedding is first fed to an autoregressive or diffusion prior to produce an image embedding, and then this embedding is used to condition a diffusion decoder which produces a final image. Note that the CLIP model is frozen during training of the prior and decoder.

__Source__: Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical Text-Conditional Image Generation with CLIP Latents. ArXiv, abs/2204.06125.

DALL.E-2

***Source**: Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical Text-Conditional Image Generation with CLIP Latents. ArXiv, abs/2204.06125.*

Diffusion Models References

[1] What are Diffusion Models? – Video About Diffusion Models

[2] Diffusion Models | Paper Explanation | Math Explained – Video About Diffusion Models

[3] Diffusion models explained. How does OpenAI's GLIDE work? – Video About Diffusion Models

[4] CS 198-126: Lecture 12 – Diffusion Models – Machine Learning at Berkeley Lecture

[5] How do DDPMs relate to Stable Diffusion? (Denoising Diffusion Probabilistic Models; Midjourney) – Interview with Ajay Jain, co-author of DDPM

[6] Weng, Lilian. (Jul 2021). What are diffusion models? Lil'Log. https://lilianweng.github.io/posts/2021-07-11-diffusion-models/.

[7] How Diffusion Models Work, Sharon Zhou - DeepLearning.AI Short Course

[8] A Toy Diffusion model you can run on your laptop; https://github.com/ThiagoLira/ToyDiffusion

[9] *Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc., Red Hook, NY, USA, Article 574, 6840–6851.*

[10] Nichol, Alex and Prafulla Dhariwal. "Improved Denoising Diffusion Probabilistic Models." *ArXiv* abs/2102.09672 (2021).

[11] *Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao and Li Fei-Fei. **Novel dataset for Fine-Grained Image Categorization**. First Workshop on Fine-Grained Visual Categorization (FGVC), IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2011*

[12] *Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical Text-Conditional Image Generation with CLIP Latents. ArXiv, abs/2204.06125.*

[13] *Nichol, A.Q., Dhariwal, P., Ramesh, A., Shyam, P., Mishkin, P., Mcgrew, B., Sutskever, I. & Chen, M.. (2022). GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models. <i>Proceedings of the 39th International Conference on Machine Learning</i>, in <i>Proceedings of Machine Learning Research</i> 162:16784-16804 Available from https://proceedings.mlr.press/v162/nichol22a.html.*