10-423/10-623 Generative AI

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Homework 2 Recitation
Diffusion Models
Variational Inference

Sept. 26, 2025

Agenda

1. HW2 Starter Code Overview
   - U-Net review
   - Forward/reverse process algorithms

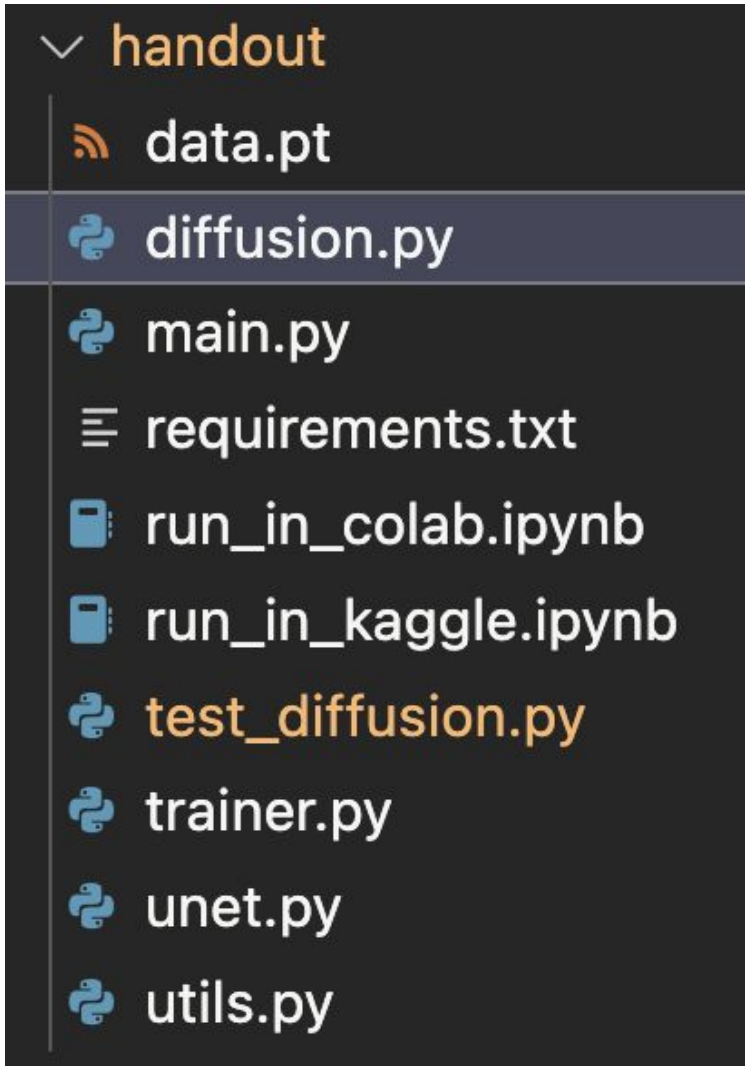
2. HW2 Written Overview
   - Review of Diffusion Models
   - Fréchet Inception Distance (FID)
   - Evidence Lower Bound (ELBO) and reparameterization

3. Helpful functions & practice reading documentation

Homework 2 Starter Code Overview

follow along on the HW2 handout!

File overview



1. diffusion.py: the only file you modify
2. main.py: run code locally
3. requirements.txt: make a conda environment with this
4. run_in_colab.ipynb: run code with GPUs
5. run_in_kaggle.ipynb: ^
6. trainer.py: train loop, Trainer class
7. unet.py: U-Net Model
8. utils.py: helper functions, wandb logging

Key ingredients to implementing this diffusion model

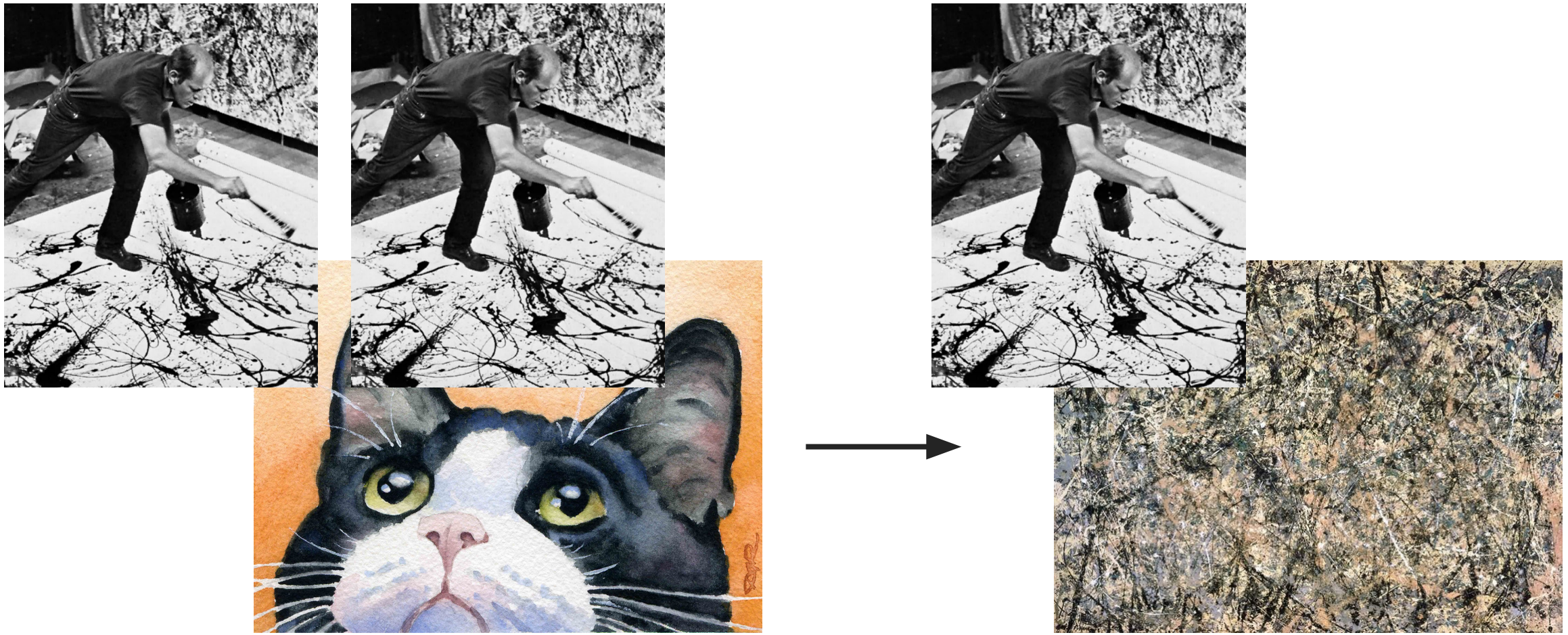
1. The U-Net itself -> done
2. Noise scheduler
3. Training algorithm (forward)
4. Sampling algorithm

In diffusion.py, you'll write 6 functions to implement steps 2-4. More on this later…

U-Nets and Diffusion Algorithms

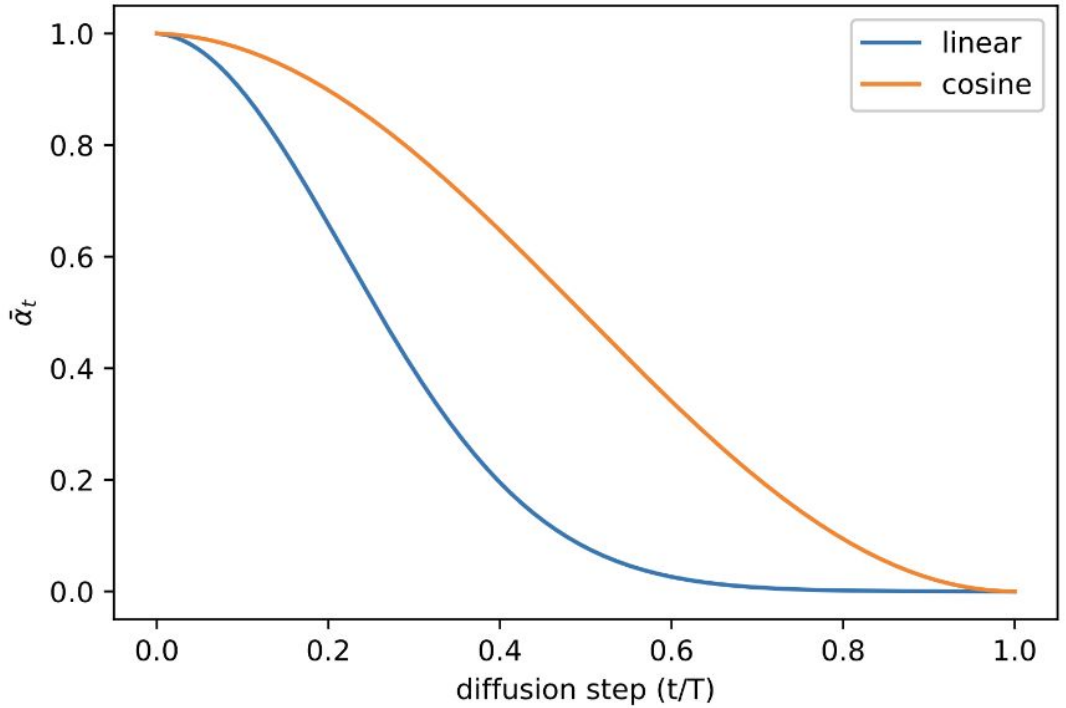
sometimes the conceptual is the hardest part

Diffusion Model Analogy - forward process (adding noise)



The forward process uses a noise scheduler

- Control the amount of noise we add in each step of the diffusion forward process

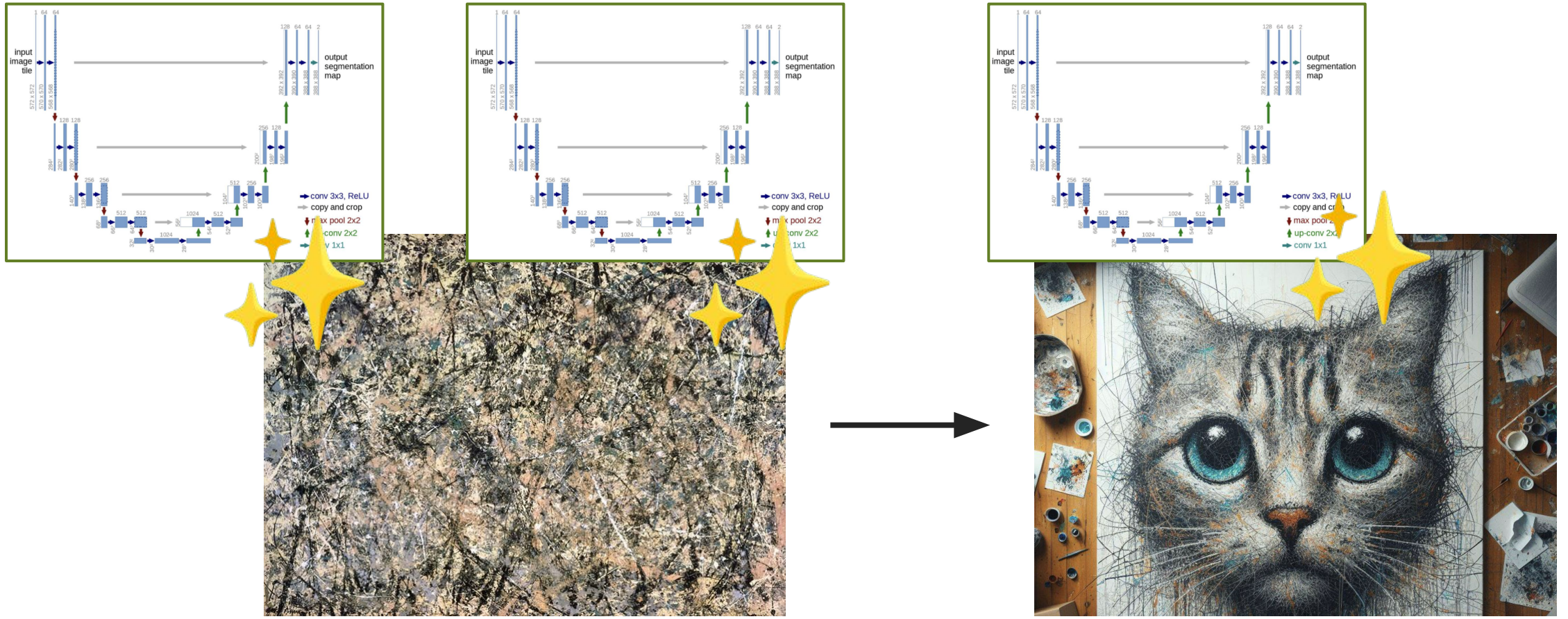


- We adopt the improved *cosine-based* variance schedule, introduced in (Nichol & Dhariwal, 2021)

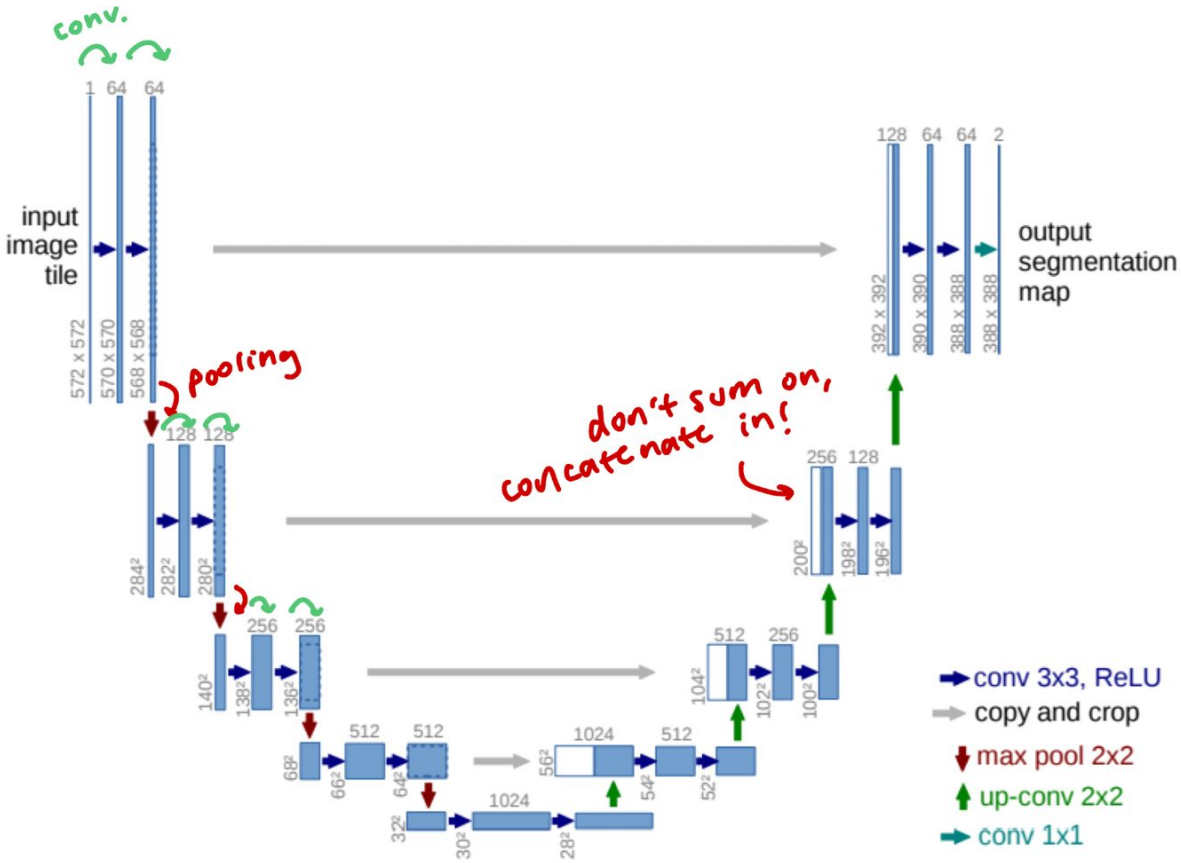
$$\alpha_t = \text{clip}\left(\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, 0.001, 1\right), \bar{\alpha}_t = \frac{f(t)}{f(0)},$$

$$\text{where } f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2,$$

Diffusion Model Analogy - reverse process (denoising)



U-Net models the denoising function



- U-Net removes a little bit of noise *at each step* of the reverse process
- We're training the U-Net model to be good at this so our output images come out well
- U-Net can capture multi-scale features

How do we train a denoising U-Net model?

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, ..., T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
5: $\mathbf{x}_t \leftarrow \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ forward diffusion process
6: Take optimizer step on L_1 loss, $\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|_1$
7: **until** converged

1. Sample a training image
2. Pick a random time step
3. Run forward diffusion to generate a noisy version of image at that time step
4. Use our model to predict the noise that was added
5. Calculate the loss between the actual noise and the predicted noise

How do we generate images with a U-Net model?

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(0, I)$
for $t = T, ..., 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ if $t > 1$, else $\mathbf{z} = 0$
  $\boldsymbol{\epsilon}_t \leftarrow \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ ▷ predicted noise
  $\hat{\mathbf{x}}_0 \leftarrow \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t\right)$ ▷ estimated $\hat{\mathbf{x}}_0$
  $\hat{\mathbf{x}}_0 \leftarrow \mathit{clamp}(\hat{\mathbf{x}}_0, -1, 1)$ ▷ rectify $\hat{\mathbf{x}}_0$
  $\tilde{\boldsymbol{\mu}}_t \leftarrow \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0$ ▷ posterior mean of x_{t-1}
  $\sigma_t^2 \leftarrow \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}(1-\alpha_t)$ ▷ posterior variance of x_{t-1}
  $\mathbf{x}_{t-1} \leftarrow \tilde{\boldsymbol{\mu}}_t + \sigma_t \mathbf{z}$ ▷ reverse diffusion process
return $\mathbf{x}_0$

1. Start from a noisy “image” (sample image-shaped noise from a normal distribution)
2. Denoise in a loop for T timesteps

How do we generate images with a U-Net model?

Idea #2: Choose μ_θ based on $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$, i.e. we want $\mu_\theta(\mathbf{x}_t, t)$ to be close to $\tilde{\mu}_q(\mathbf{x}_t, \mathbf{x}_0)$. Here are three ways we could parameterize this:

Option A: Learn a network that approximates $\tilde{\mu}_q(\mathbf{x}_t, \mathbf{x}_0)$ directly from $\mathbf{x}_t$ and t:

$$\mu_\theta(\mathbf{x}_t, t) = \text{UNet}_\theta(\mathbf{x}_t, t)$$

where t is treated as an extra feature in UNet

Option B: Learn a network that approximates the real $\mathbf{x}_0$ from only $\mathbf{x}_t$ and t:

$$\mu_\theta(\mathbf{x}_t, t) = \alpha_t^{(0)} \mathbf{x}_\theta^{(0)}(\mathbf{x}_t, t) + \alpha_t^{(t)} \mathbf{x}_t$$

where $\mathbf{x}_\theta^{(0)}(\mathbf{x}_t, t) = \text{UNet}_\theta(\mathbf{x}_t, t)$

Option C: Learn a network that approximates the $\boldsymbol{\epsilon}$ that gave rise to $\mathbf{x}_t$ from $\mathbf{x}_0$ in the forward process from $\mathbf{x}_t$ and t:

$$\mu_\theta(\mathbf{x}_t, t) = \alpha_t^{(0)} \mathbf{x}_\theta^{(0)}(\mathbf{x}_t, t) + \alpha_t^{(t)} \mathbf{x}_t$$

where $\mathbf{x}_\theta^{(0)}(\mathbf{x}_t, t) = \left(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) / \sqrt{\bar{\alpha}_t}$

where $\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) = \text{UNet}_\theta(\mathbf{x}_t, t)$

Option C is the best empirically

How do we generate images with a U-Net model?

Algorithm 1 Sampling (Option C)

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t \in \{1, \ldots, T\}$ **do**
3: $\quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4: $\quad \hat{\mathbf{x}}_0 \leftarrow \left(\mathbf{x}_0 + (1 - \bar{\alpha}_t)\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) / \sqrt{\bar{\alpha}_t}$
5: $\quad \hat{\boldsymbol{\mu}}_t \leftarrow \alpha_t^{(0)} \hat{\mathbf{x}}_0 + \alpha_t^{(t)} \mathbf{x}_t$
6: $\quad \mathbf{x}_{t-1} \leftarrow \hat{\boldsymbol{\mu}}_t + \sigma_t^2 \boldsymbol{\epsilon}$
7: **return** $\mathbf{x}_0$

$$\alpha_t = \text{clip}\left(\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, 0.001, 1\right), \bar{\alpha}_t = \frac{f(t)}{f(0)},$$

$$\text{where } f(t) = \cos\left(\frac{t/T + s}{1 + s} \cdot \frac{\pi}{2}\right)^2,$$

Intuition:

Given noise, use U-Net to predict what some “original” x0 would be. Use this to estimate the mean and variance from $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$

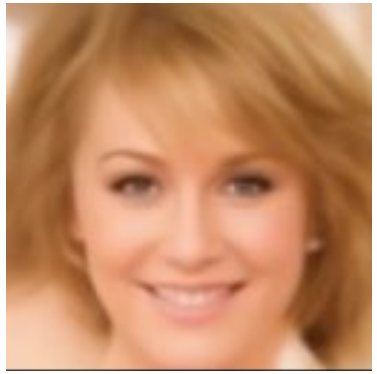
Use this mean + variance to sample a slightly denoised image (by *just one timestep, t-1*), then repeat process in loop

Sampling an image

t=4

t=3

t=1



t=0

After training, our goal is that our model can revert images with any amount of noise t=n to the previous step t=n-1

This allows us to generate images by repeatedly denoising them, one timestep at a time

Functions that you'll write

- Training
  - Forward
  - P_loss
  - Q_sample
- Sampling
  - Sample
  - P_sample_loop
  - P_sample

Flags

Description	Parameter	Default Value
Directory from which to load data	`data_path`	(See starter notebook)
Number of iterations to train the model	`train_steps`	(See handout below)
Enable FID calculation	`fid`	(See handout below)
Frequency of periodic save, sample and (optionally) FID calculation	`save_and_sample_every`	(See handout below)

Table 2: Useful parameters for `run_in_colab/kaggle.ipynb`

Description	Parameter	Default Value
Dataloader worker threads	`dataloader_workers`	16
Directory where the model is stored	`save_folder`	`./results/`
Path of a trained model	`load_path`	`./results/model.pt`

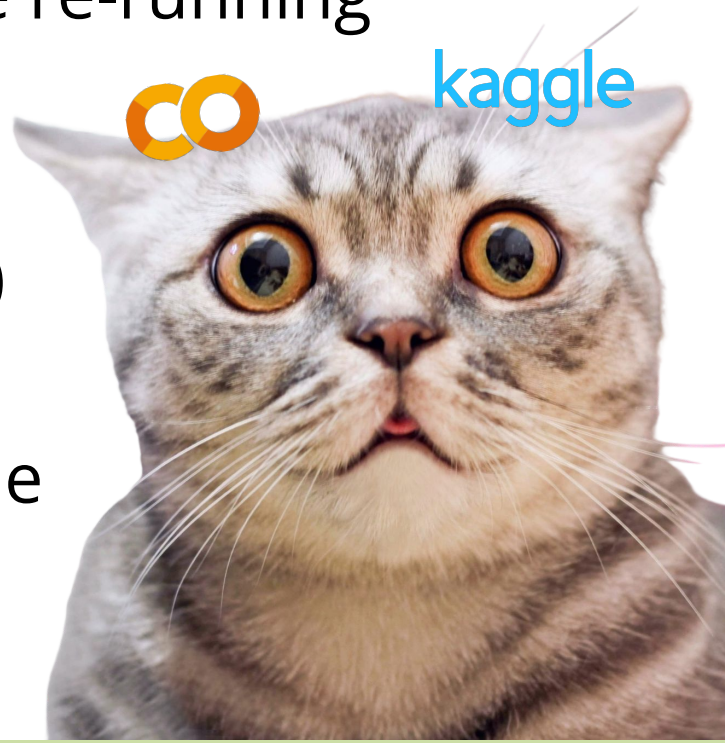
Table 3: Additional parameters for `run_in_colab/kaggle.ipynb`. You likely won't need to change these.

Description	Parameter	Default Value
Model image size	`image_size`	32
Model batch size	`batch_size`	32
Data domain of AFHQ dataset	`data_class`	`cat`
Number of steps of diffusion process, T	`time_steps`	50
Number of output channels of the first layer in U-Net	`unet_dim`	16
Learning rate in training	`learning_rate`	1e-3
U-Net architecture	`unet_dim_mults`	[1, 2, 4, 8]

Table 4: Additional parameters for `run_in_colab/kaggle.ipynb`. These won't need to be changed from default values for this homework.

General advice: you will be training for a while

- Start early! This is the homework with the most train time
- GPUs are a necessity. The experiments we ask you to run will take 2-3 hours on a Colab T4 GPU, and you will likely be re-running these experiments as you debug
- Test your code for a few timesteps (locally or GPU)
- See writeup for guidance on setup for Colab/Kaggle



Colab Pro is FREE for students

Pay As You Go

$9.99 for 100 Compute Units

$49.99 for 500 Compute Units

You currently have 0 compute units.

Compute units expire after 90 days.
Purchase more as you need them.

- ✓ No subscription required.
  Only pay for what you use.
- ✓ Faster GPUs
  Upgrade to more powerful GPUs.

Recommended

Colab Pro

$9.99 per month

Colab Pro for Education

No cost for students and educators

- ✓ 100 compute units per month
  Compute units expire after 90 days.
  Purchase more as you need them.
- ✓ Faster GPUs
  Upgrade to more powerful GPUs.
- ✓ More memory
  Access our highest memory machines.

Colab Pro+

$49.99 per month

🔥 Limited time offer of an additional 100 compute units, totaling **600 per month**.

All of the benefits of Pro, plus:

- ✓ An additional ~~400~~ **500** compute units per month
  Compute units expire after 90 days.
  Purchase more as you need them.
- ✓ Faster GPUs
  Priority access to upgrade to more powerful premium GPUs.
- ✓ Background execution
  With compute units, your actively running notebook will continue running for up to

Colab Enterprise

Pay for what you use

- ✓ Integrated
  Tightly integrated with Google Cloud services like BigQuery and Vertex AI.
- ✓ Enterprise notebook storage
  Replace your usage of Google Drive notebooks with GCP notebooks, stored and shared within your cloud console.
- ✓ Productive
  Generative AI powered code completion and generation.

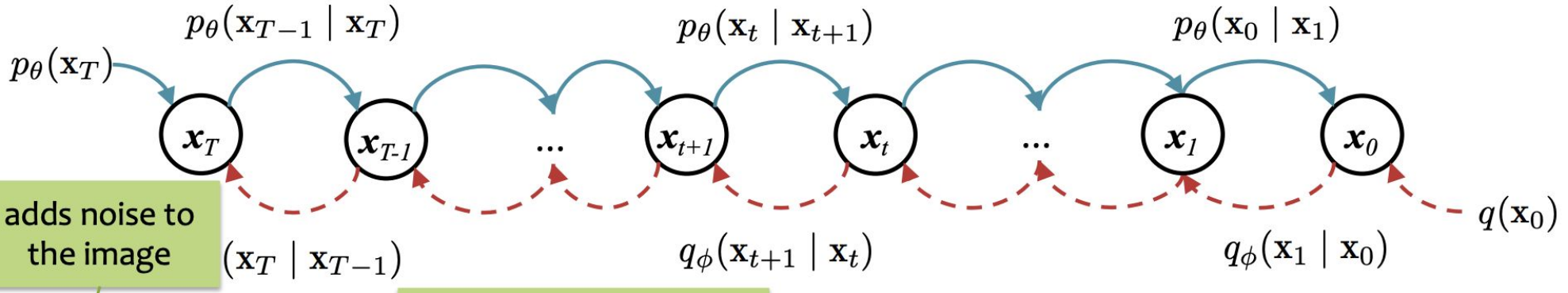
Homework 2 Written Overview

going over a few key topics!

Diffusion Models

forward! …reverse!

Diffusion Models



adds noise to the image

if we could sample from this we'd be done

Forward Process:

$$q_\phi(\mathbf{x}_{0:T}) = q(\mathbf{x}_0) \prod_{t=1}^{T} q_\phi(\mathbf{x}_t \mid \mathbf{x}_{t-1})$$

(Learned) Reverse Process: ← removes noise

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

goal is to learn this

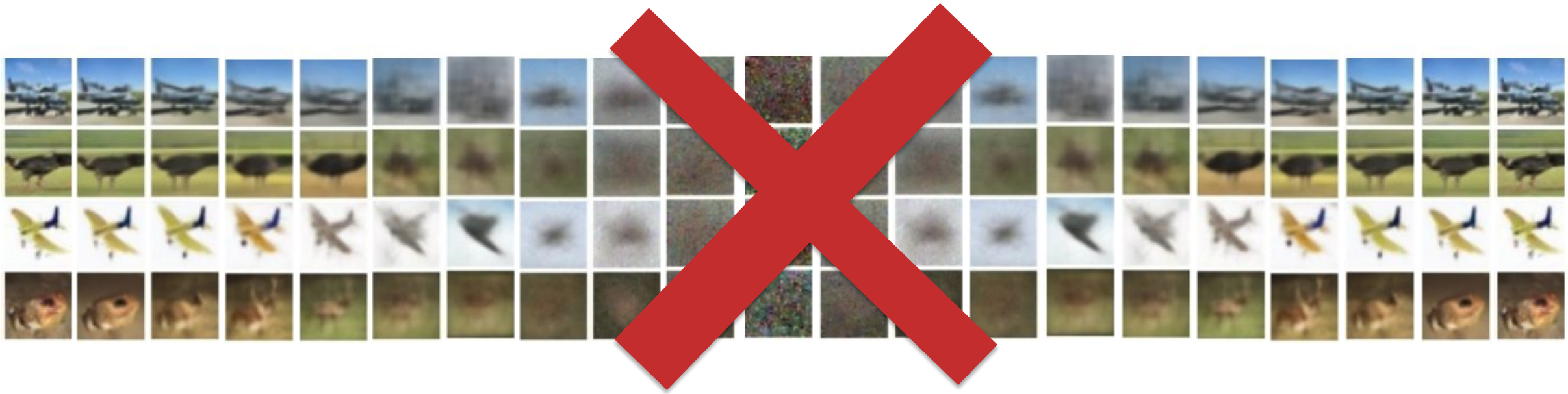
(Exact) Reverse Process:

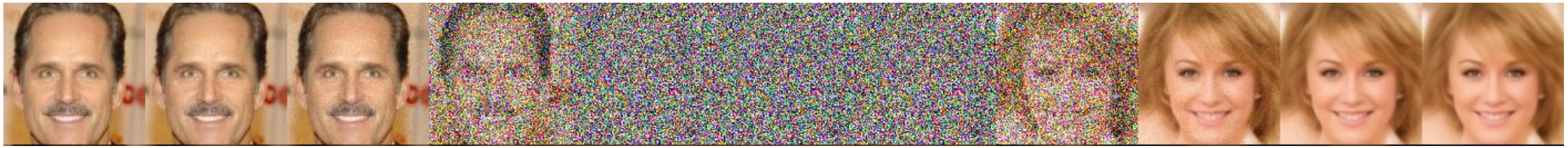
$$q_\phi(\mathbf{x}_{1:T}) = q_\phi(\mathbf{x}_T) \prod_{t-1}^{T} q_\phi(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$$

The *exact* reverse process requires inference. And even though $q_\phi(\mathbf{x}_t \mid \mathbf{x}_{t-1})$ is simple, computing $q_\phi(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ is intractable! Why? Because $q(\mathbf{x}_0)$ might be not-so-simple.

Denoising is not image recovery…

…it’s image generation!

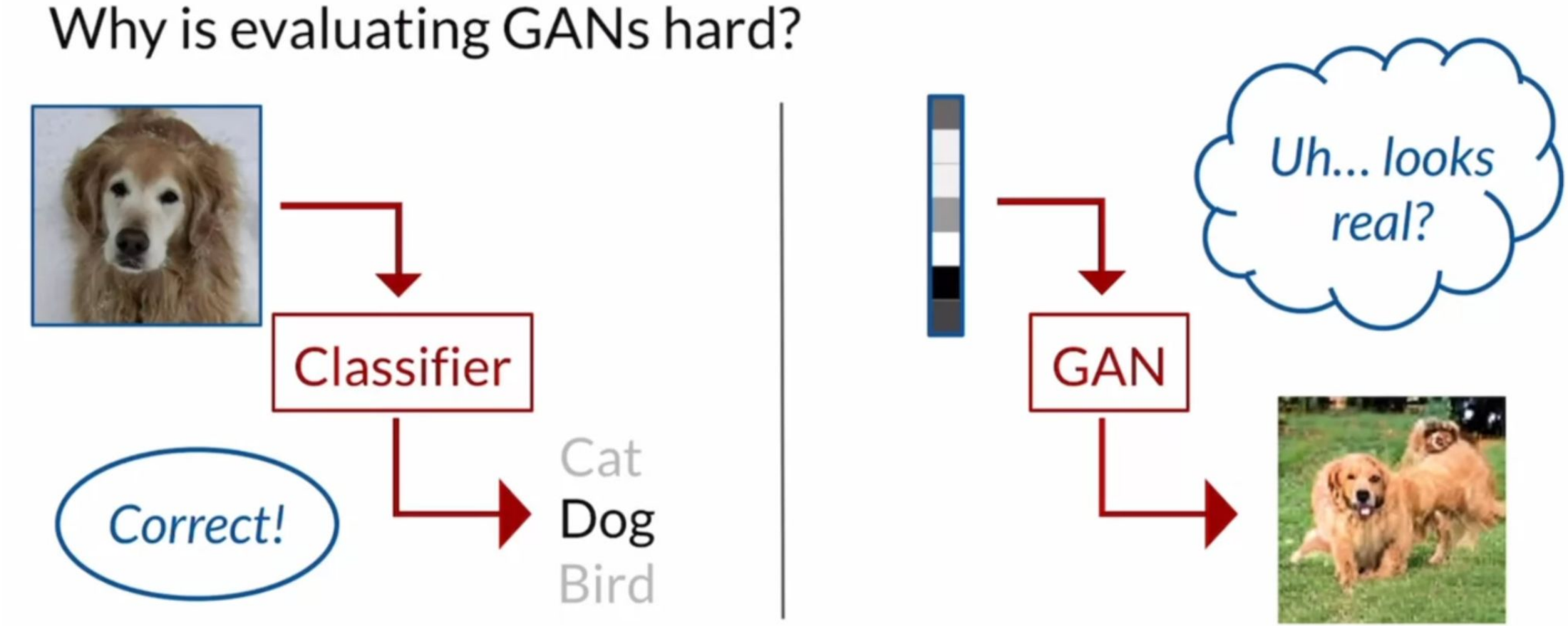




Fréchet Inception Distance (FID)

one way to evaluate image generation models

How do we evaluate these models?



Source

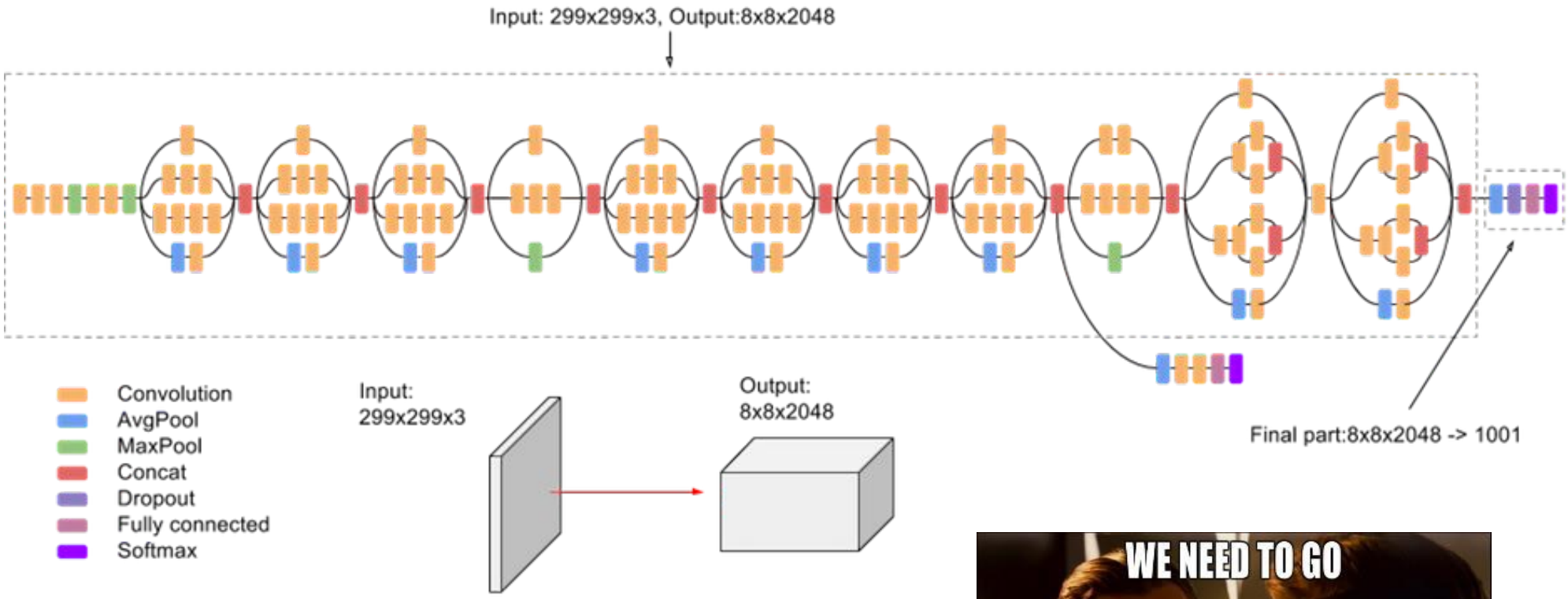
FID – Fréchet Inception Distance

How do we measure the quality of a generated image?

Target image distribution

Model generated image distribution

FID Inception Model



48 layers
SOTA in 2015 on ImageNet top-5 error
Named after an internet meme

Putting it all together

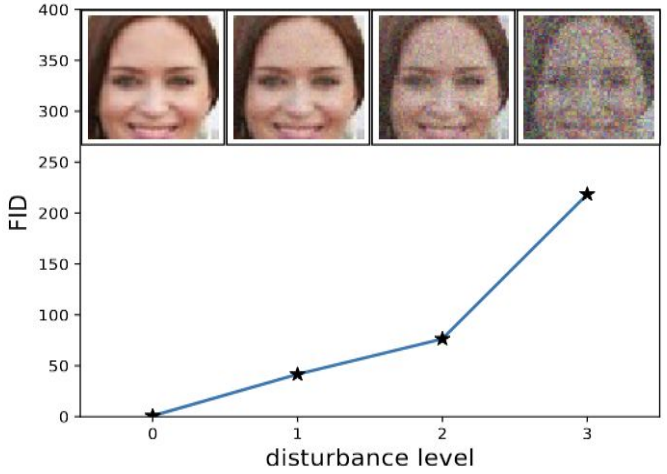
Summary

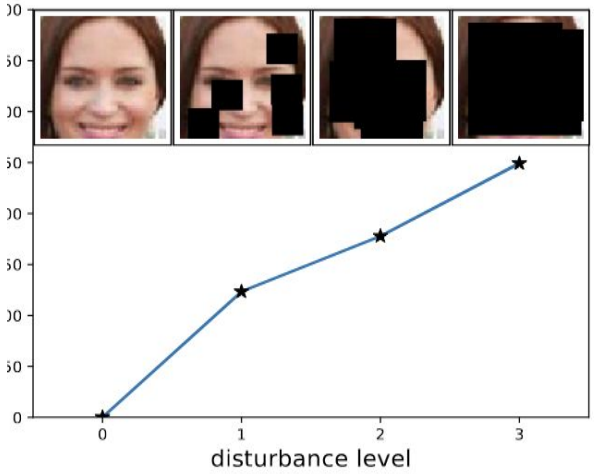
1. Extract the features from real images and generated images using an Inceptionv3 model.
2. Find the distance between the two distribution of features

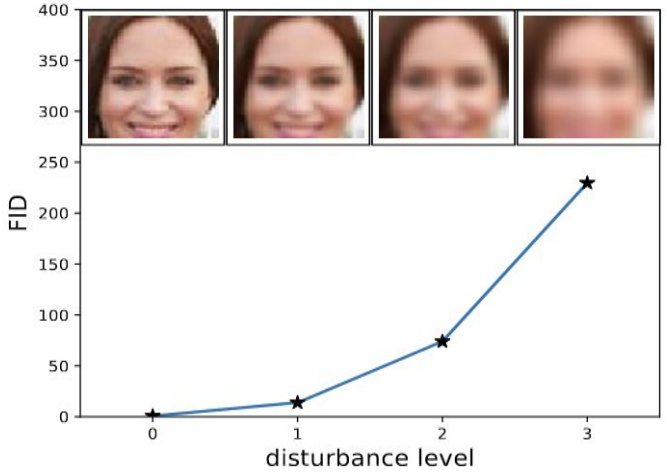
Code snippet

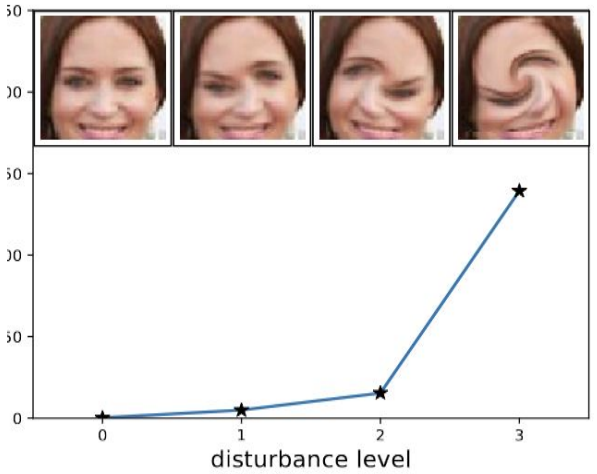
```
Python 3.7.10 (default, Feb 26 2021, 18:47:35)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Empirical examples of FID score









The closer the image is to the target distribution, the lower the FID score

FID ranges between 0 and infinity

We use FID to track the quality of our generated images in HW2

Evidence Lower BOund (ELBO)

the workhorse of VAEs

What is ELBO?

$$\mathbb{E}_{z\sim q(z|x)}[\log p_\theta(x \mid z)] - D_{KL}\left(q_\phi(z \mid x) \,\|\, p(z)\right)$$

reconstruction error

KL-divergence

You are a crime scene reporter!

Your job is to reproduce details of the crime scene in your article.

You must be very accurate and not overly stray from the details of the scene.

You are a *viral* crime scene reporter!

BUT:

You also want your article to have the most clicks and views!

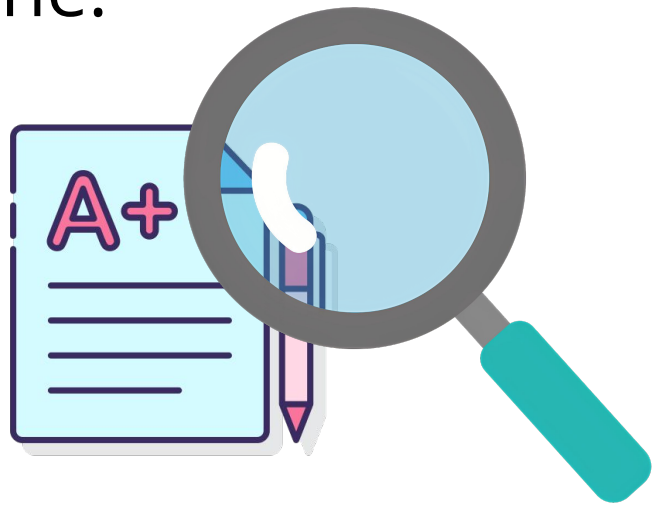
How would you do this?

Reconstruction Error

$$\mathbb{E}_{z \sim q(z|x)}[\log p_\theta(x \mid z)]$$

reconstruction error

You write the **best possible** detective report that explains the crime scene.



What is ELBO?

$$\mathbb{E}_{z\sim q(z|x)}[\log p_\theta(x \mid z)] - D_{KL}\left(q_\phi(z \mid x) \,\|\, p(z)\right)$$

reconstruction error

KL-divergence

KL Divergence

While **staying consistent** with what you generally know about getting the most clicks and views (the prior) :)

$$D_{KL}(q_\phi(z \mid x) \parallel p(z))$$

KL-divergence

Okay…this sounds great…but how does this connect to VAEs and the bigger picture??

$$\mathbb{E}_{z\sim q(z|x)}[\log p_\theta(x \mid z)] - D_{KL}(q_\phi(z \mid x) \,\|\, p(z))$$

reconstruction error

KL-divergence

Deriving the Variational Autoencoder

1. *Problem:* Cannot sample from an autoencoder.
   Solution: Define a decoder that we can sample from:
   $p_\theta(\mathbf{z}_T) \sim \mathcal{N}(0, \mathbf{I})$
   $p_\theta(\mathbf{x} \mid \mathbf{z}) \sim \mathcal{N}(\mu_\theta(\mathbf{z}), \Sigma_\theta(\mathbf{z}))$ where $\mu_\theta(\mathbf{z})$ and $\Sigma_\theta(\mathbf{z})$ are neural nets.

2. *Problem:* Intractable to maximize data likelihood with this decoder:
   $\hat{\theta} = \text{argmin}_\theta \log p_\theta(\mathbf{x}) = \text{argmin}_\theta \int_z p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$
   Solution: Maximize a lower bound (ELBO) instead by introducing an encoder distribution:
   $q_\phi(\mathbf{z} \mid \mathbf{x}) \sim \mathcal{N}(\mu_\phi(\mathbf{x}), \Sigma_\phi(\mathbf{x}))$
   $\hat{\theta}, \hat{\phi} = \text{argmax}_{\theta,\phi}$ ELBO(q_ϕ, p_θ) where $\log p_\theta(\mathbf{x}) \geq$ ELBO(q_ϕ, p_θ)

Evidence Lower BOund (ELBO)

Jensen's Inequality

If function $f(\cdot)$ is concave, then:

$$\mathbb{E}[f(X)] \leq f(\mathbb{E}[X])$$

Evidence Lower BOund (ELBO)

ELBO via Jensen's Inequality

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

We use marginalization To make latent variable appear

Evidence Lower BOund (ELBO)

ELBO via Jensen's Inequality

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

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

We need this term for the expectation of Jensen's

Evidence Lower BOund (ELBO)

ELBO via Jensen's Inequality

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

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

We cancel out to preserve The equality

Evidence Lower BOund (ELBO)

ELBO via Jensen's Inequality

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

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

It is a concave function!

Evidence Lower BOund (ELBO)

ELBO via Jensen's Inequality

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

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

$$\geq \int q(\boldsymbol{z}) \log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} d\boldsymbol{z}$$

We use Jensen's to swap log and expectation

Evidence Lower BOund (ELBO)

ELBO

$$\log p(\boldsymbol{x}) \geq \int q(\boldsymbol{z}) \log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} d\boldsymbol{z}$$

ELBO is your best friend

Evidence Lower BOund (ELBO)

ELBO

$$\log p(\boldsymbol{x}) \geq \int q(\boldsymbol{z}) \log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} d\boldsymbol{z}$$

$$= \mathbb{E}_q \left[\log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} \right]$$

ELBO is your best friend

Maximize this

The Reparameterization Trick

Reparameterization is a method of generating random numbers by transforming some base distribution $p(\boldsymbol{\epsilon})$ to a desired distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$

$$\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) \longrightarrow g(\epsilon; \boldsymbol{\theta}) \longrightarrow p_{\boldsymbol{\theta}}(\boldsymbol{z})$$

The Reparameterization Trick

Reparameterization is a method of generating random numbers by transforming some base distribution $p(\boldsymbol{\epsilon})$ to a desired distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$

$$\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) \longrightarrow g(\epsilon; \boldsymbol{\theta}) \longrightarrow p_{\boldsymbol{\theta}}(\boldsymbol{z})$$

A simple distribution to sample from

The Reparameterization Trick

Reparameterization is a method of generating random numbers by transforming some base distribution $p(\boldsymbol{\epsilon})$ to a desired distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$

$$\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) \longrightarrow g(\epsilon; \boldsymbol{\theta}) \longrightarrow p_{\boldsymbol{\theta}}(\boldsymbol{z})$$

A simple transformation

The Reparameterization Trick

Gaussian Distribution:

We want samples from $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I})$

The Reparameterization Trick

Gaussian Distribution

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

We sample standard Normal

The Reparameterization Trick

Gaussian Distribution

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

We sample standard Normal

$$\boldsymbol{x} = \boldsymbol{\mu} + \sigma\boldsymbol{\epsilon}$$

We apply linear transformation

The Reparameterization Trick

Gaussian Distribution

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$

We sample standard Normal

$$\boldsymbol{x} = \boldsymbol{\mu} + \sigma\boldsymbol{\epsilon}$$

We apply linear transformation

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2\boldsymbol{I})$$

Transformed sample comes from the desired Gaussian distribution

Helpful Functions and Methods

you may enjoy the homework more with these

noise_like

```
def noise_like(self, shape, device):
    """
    Generates noise with the same shape as the input.
    Args:
        shape: The shape of the noise.
        device: The device on which to create the noise.
    Returns:
        The generated noise.
    """
    noise = lambda: torch.randn(shape, device=device)
    return noise()
```

Why do we want to use noise_like as opposed to torch.randn?

noise_like

```
def noise_like(self, shape, device):
    """
    Generates noise with the same shape as the input.
    Args:
        shape: The shape of the noise.
        device: The device on which to create the noise.
    Returns:
        The generated noise.
    """
    noise = lambda: torch.randn(shape, device=device)
    return noise()
```

Why do we want to use noise_like as opposed to torch.randn?

To stay consistent, decrease randomness across submissions.

extract

```python
def extract(a, t, x_shape):
    """
    This function abstracts away the tedious indexing that would otherwise have
    to be done to properly compute the diffusion equations from lecture. This
    is necessary because we train data in batches, while the math taught in
    lecture only considers a single sample.

    To use this function, consider the example
        alpha_t * x
    To compute this in code, we would write
        extract(alpha, t, x.shape) * x

    Args:
        a: 1D tensor containing the value at each time step.
        t: 1D tensor containing a batch of time indices.
        x_shape: The reference shape.
    Returns:
        The extracted tensor.
    """
    b, *_ = t.shape
    out = a.gather(-1, t)
    return out.reshape(b, *((1,) * (len(x_shape) - 1)))
```

Why/when should we use extract?

extract

```python
def extract(a, t, x_shape):
    """
    This function abstracts away the tedious indexing that would otherwise have
    to be done to properly compute the diffusion equations from lecture. This
    is necessary because we train data in batches, while the math taught in
    lecture only considers a single sample.

    To use this function, consider the example
        alpha_t * x
    To compute this in code, we would write
        extract(alpha, t, x.shape) * x

    Args:
        a: 1D tensor containing the value at each time step.
        t: 1D tensor containing a batch of time indices.
        x_shape: The reference shape.
    Returns:
        The extracted tensor.
    """
    b, *_ = t.shape
    out = a.gather(-1, t)
    return out.reshape(b, *((1,) * (len(x_shape) - 1)))
```

Why/when should we use extract?

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, ..., T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
5: $\mathbf{x}_t \leftarrow \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ forward diffusion process
6: Take optimizer step on L_1 loss, $\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|_1$
7: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(0, I)$
2: **for** $t = T, ..., 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ if $t > 1$, else $\mathbf{z} = 0$
4: $\boldsymbol{\epsilon}_t \leftarrow \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$ ▷ predicted noise
5: $\hat{\mathbf{x}}_0 \leftarrow \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}_t\right)$ ▷ estimated $\hat{\mathbf{x}}_0$
6: $\hat{\mathbf{x}}_0 \leftarrow \textit{clamp}(\hat{\mathbf{x}}_0, -1, 1)$ ▷ rectify $\hat{\mathbf{x}}_0$
7: $\tilde{\boldsymbol{\mu}}_t \leftarrow \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)}{1-\bar{\alpha}_t}\hat{\mathbf{x}}_0$ ▷ posterior mean of x_{t-1}
8: $\sigma_t^2 \leftarrow \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}(1 - \alpha_t)$ ▷ posterior variance of x_{t-1}
9: $\mathbf{x}_{t-1} \leftarrow \tilde{\boldsymbol{\mu}}_t + \sigma_t \mathbf{z}$ ▷ reverse diffusion process
return $\mathbf{x}_0$

extract

```python
def extract(a, t, x_shape):
    """
    This function abstracts away the tedious indexing that would otherwise have
    to be done to properly compute the diffusion equations from lecture. This
    is necessary because we train data in batches, while the math taught in
    lecture only considers a single sample.

    To use this function, consider the example
        alpha_t * x
    To compute this in code, we would write
        extract(alpha, t, x.shape) * x

    Args:
        a: 1D tensor containing the value at each time step.
        t: 1D tensor containing a batch of time indices.
        x_shape: The reference shape.
    Returns:
        The extracted tensor.
    """
    b, *_ = t.shape
    out = a.gather(-1, t)
    return out.reshape(b, *((1,) * (len(x_shape) - 1)))
```

Why/when should we use extract?

- Allows us to pre-calculate important values and extract just the timesteps we need
- Less computationally expensive than recalculating every time

torch.cumprod

TORCH.CUMPROD

torch.cumprod(*input*, *dim*, ***, *dtype=None*, *out=None*) → Tensor

Returns the cumulative product of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements.

$$y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i$$

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to do the operation over

Keyword Arguments

- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.
- **out** (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3, 4, 5]])
y = torch.cumprod(x, 0)
print(y)
```

torch.cumprod

TORCH.CUMPROD

torch.cumprod(*input*, *dim*, ***, *dtype=None*, *out=None*) → Tensor

Returns the cumulative product of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements.

$$y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i$$

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to do the operation over

Keyword Arguments

- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.
- **out** (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3, 4, 5]])
y = torch.cumprod(x, 0)
print(y)
```

Output:
tensor([[1, 2, 3, 4, 5]])

torch.cumprod

TORCH.CUMPROD

torch.cumprod(*input*, *dim*, ***, *dtype=None*, *out=None*) → Tensor

Returns the cumulative product of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements.

$$y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i$$

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to do the operation over

Keyword Arguments

- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.
- **out** (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3, 4, 5]])
y = torch.cumprod(x, 1)
print(y)
```

torch.cumprod

TORCH.CUMPROD

torch.cumprod(*input*, *dim*, ***, *dtype=None*, *out=None*) → Tensor

Returns the cumulative product of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements.

$$y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i$$

Parameters

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to do the operation over

Keyword Arguments

- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.
- **out** (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3, 4, 5]])
y = torch.cumprod(x, 1)
print(y)
```

Output:
tensor([[1, 2, 6, 24, 120]])

torch.clamp

TORCH.CLAMP

torch.clamp(*input*, *min=None*, *max=None*, *, *out=None*) → Tensor

Clamps all elements in `input` into the range [`min` , `max`]. Letting min_value and max_value be `min` and `max` , respectively, this returns:

$$y_i = \min(\max(x_i, \text{min\_value}_i), \text{max\_value}_i)$$

If `min` is `None` , there is no lower bound. Or, if `max` is `None` there is no upper bound.

• NOTE

If `min` is greater than `max` `torch.clamp(..., min, max)` sets all elements in `input` to the value of `max` .

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number or Tensor, optional*) – lower-bound of the range to be clamped to
- **max** (*Number or Tensor, optional*) – upper-bound of the range to be clamped to

Keyword Arguments

out (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
y = torch.clamp(x, min=4, max=6)
print(y)
```

torch.clamp

TORCH.CLAMP

torch.clamp(*input*, *min=None*, *max=None*, ***, *out=None*) → Tensor

Clamps all elements in `input` into the range [`min` , `max`]. Letting min_value and max_value be `min` and `max` , respectively, this returns:

$$y_i = \min(\max(x_i, \text{min\_value}_i), \text{max\_value}_i)$$

If `min` is `None` , there is no lower bound. Or, if `max` is `None` there is no upper bound.

• NOTE

If `min` is greater than `max` `torch.clamp(..., min, max)` sets all elements in `input` to the value of `max` .

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number or Tensor, optional*) – lower-bound of the range to be clamped to
- **max** (*Number or Tensor, optional*) – upper-bound of the range to be clamped to

Keyword Arguments

out (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
y = torch.clamp(x, min=4, max=6)
print(y)
```

TypeError: clamp() received an invalid combination of arguments - got (numpy.ndarray, max=int, min=int), but expected one of: * (Tensor input, Tensor min, Tensor max, *, Tensor out) * (Tensor input, Number min, Number max, *, Tensor out)

torch.clamp

TORCH.CLAMP

torch.clamp(*input*, *min=None*, *max=None*, *, *out=None*) → Tensor

Clamps all elements in `input` into the range [`min` , `max`]. Letting min_value and max_value be `min` and `max` , respectively, this returns:

$$y_i = \min(\max(x_i, \text{min\_value}_i), \text{max\_value}_i)$$

If `min` is `None` , there is no lower bound. Or, if `max` is `None` there is no upper bound.

• NOTE

If `min` is greater than `max` `torch.clamp(..., min, max)` sets all elements in `input` to the value of `max` .

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number or Tensor, optional*) – lower-bound of the range to be clamped to
- **max** (*Number or Tensor, optional*) – upper-bound of the range to be clamped to

Keyword Arguments

out (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
y = torch.clamp(x, min=4, max=6)
print(y)
```

torch.clamp

TORCH.CLAMP

torch.clamp(*input*, *min=None*, *max=None*, ***, *out=None*) → Tensor

Clamps all elements in `input` into the range [`min` , `max`]. Letting min_value and max_value be `min` and `max` , respectively, this returns:

$$y_i = \min(\max(x_i, \text{min\_value}_i), \text{max\_value}_i)$$

If `min` is `None` , there is no lower bound. Or, if `max` is `None` there is no upper bound.

• NOTE

If `min` is greater than `max` `torch.clamp(..., min, max)` sets all elements in `input` to the value of `max` .

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number or Tensor, optional*) – lower-bound of the range to be clamped to
- **max** (*Number or Tensor, optional*) – upper-bound of the range to be clamped to

Keyword Arguments

out (*Tensor, optional*) – the output tensor.

What does the following code snippet return?

```
import torch

x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
y = torch.clamp(x, min=4, max=6)
print(y)
```

Output:
tensor([[4, 4, 4], [4, 5, 6], [6, 6, 6]])

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full(2, 3, 3)
x2 = torch.ones(2,3) * 3

print(x1 == x2)
```

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full(2, 3, 3)
x2 = torch.ones(2,3) * 3

print(x1 == x2)
```

Output:

TypeError: full() received an invalid combination of arguments - got (int, int, int), but expected one of: * (tuple of ints size, Number fill_value, *, tuple of names names, torch.dtype dtype, torch.layout layout, torch.device device, bool pin_memory, bool requires_grad) * (tuple of ints size, Number fill_value, *, Tensor out, torch.dtype dtype, torch.layout layout, torch.device device, bool pin_memory, bool requires_grad)

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full((2, 3), 3)
x2 = torch.ones(2,3) * 3

print(x1 == x2)
```

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full((2, 3), 3)
x2 = torch.ones(2,3) * 3

print(x1 == x2)
```

Output:

tensor([[True, True, True], [True, True, True]])

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full((2,3), 3)
x2 = torch.ones(2,3) * 3

print((x1 == x2).all())
```

torch.full

TORCH.FULL

torch.full(*size*, *fill_value*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a tensor of size `size` filled with `fill_value`. The tensor's dtype is inferred from `fill_value`.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **fill_value** (*Scalar*) – the value to fill the output tensor with.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

What does the following code snippet return?

```
import torch

x1 = torch.full((2,3), 3)
x2 = torch.ones(2,3) * 3

print((x1 == x2).all())
```

Output:
tensor(True)

Thanks everyone!

happy diffusing! -Natalie, Rithvik, Irene, Ziming