Computer Vision in Python

Day 3, Part 1/4

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this lecture

1. What are **generative models** in deep learning?
2. What are **Generative Adversarial Networks** (**GANs**) and their uses?
3. What are the **advanced techniques** in GANs
4. What are **deepfakes and ethical considerations** when using generative models?
5. What are **diffusion models**?

What are generative models?

Definition (Computer Vision generative models):

In computer vision, **generative models** are models that can generate new visual data (typically new images) that are plausible and statistically similar to some training data.

In other words, we are talking about models that can generate realistic images out of thin air.

As such, generative models are the opposite of the computer vision models we have seen until now: the image is the output, and no longer the input of the AI model.

Typical uses of generative models

Many uses for computer vision generative models:

- **Image and video generation:** generating an image based on a given description
- **Super-resolution and denoising:** increasing the resolution of an image, or removing noise from an image
- **Style transfer:** transforming a picture into a similar picture but different style.
- Etc.

GAN definition

Definition (GAN):

A **Generative Adversarial Network** (or **GAN**, in short) is a particular type of architecture, which attempts to learn to reproduce the samples found in a given dataset X.

- Train a model G, with noise sample inputs $z \in \mathbb{R}^K$ drawn from normalized Gaussian distributions.

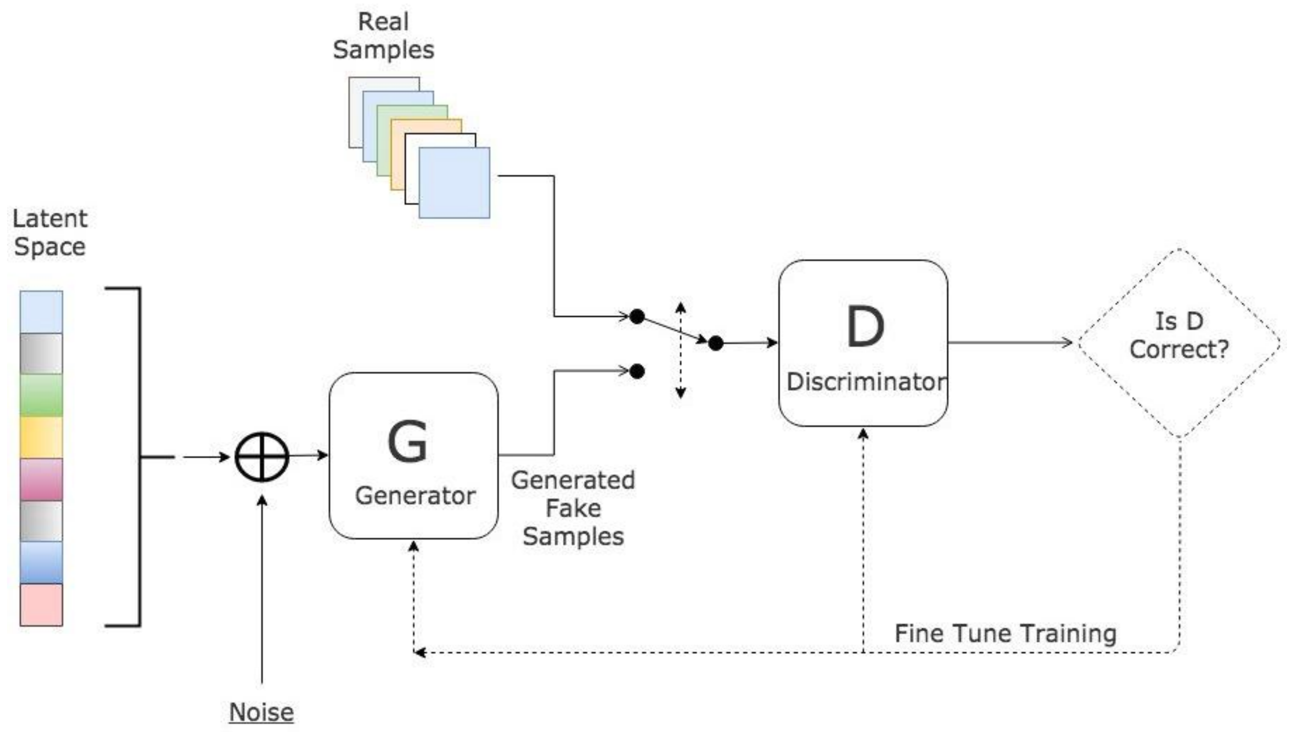
$$\hat{x} = G(z), \qquad z \to N(0_K, I_K)$$

The training objective for G, is then to ensure that the distribution of $\hat{X}$ matches the one of our original dataset X.

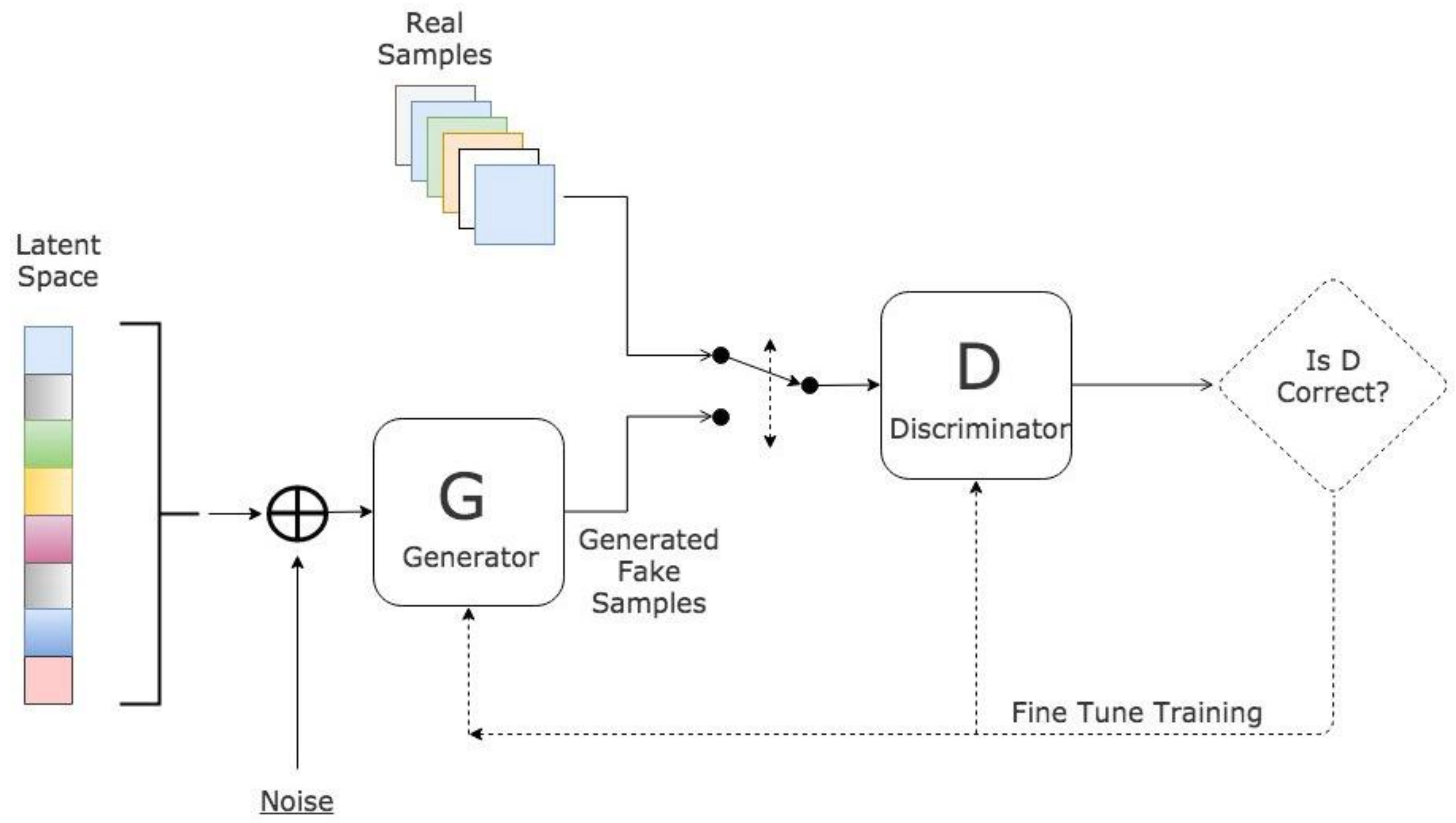
Vanilla GAN architecture

To train a GAN, we need five components.

1. **Noise sample generator $\boldsymbol{Z}$**
2. **Generator $\boldsymbol{G}$**
3. **Discriminator $\boldsymbol{D}$**
4. **Loss $\boldsymbol{L_D}$ on discriminator $\boldsymbol{D}$**
5. **Loss $\boldsymbol{L_G}$ on generator $\boldsymbol{G}$**



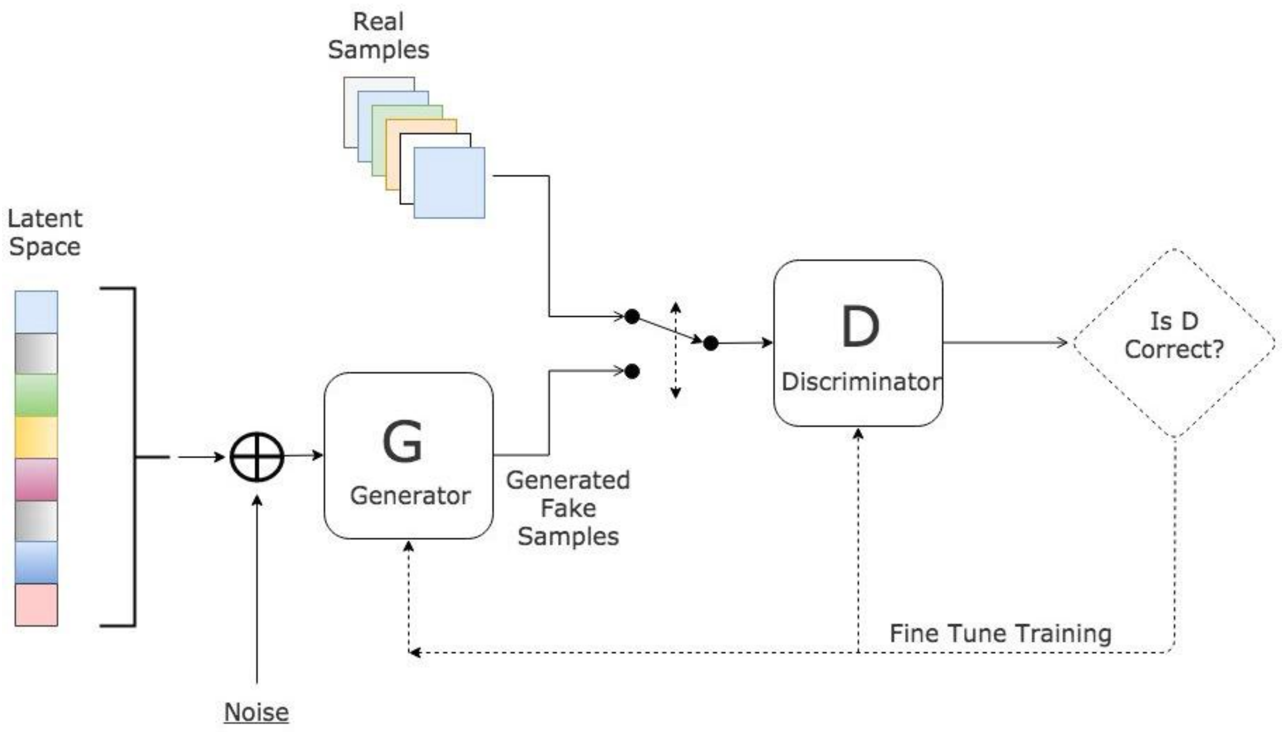
Vanilla GAN architecture



Vanilla GAN architecture

To train a GAN, we need five components.

1. **Noise sample generator Z**
2. **Generator G**
3. **Discriminator D**
4. **Loss L_D on discriminator D**
5. **Loss L_G on generator G**



Vanilla GAN architecture

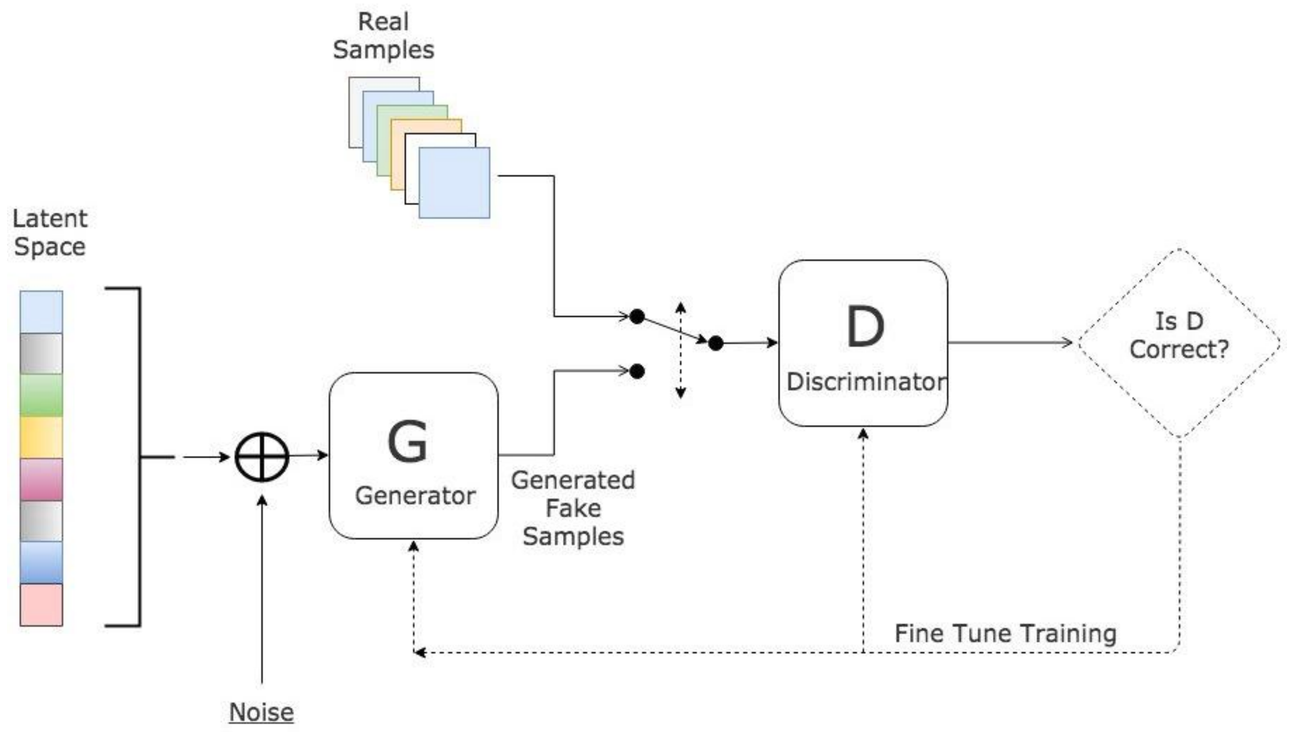
To train a GAN, we need five components.

1. **Noise sample generator Z**

Simply draw some random vector $z \in \mathbb{R}^K$, by using random noise.

$$z \to N(0_K, I_K)$$

It will be fed to our generator G.



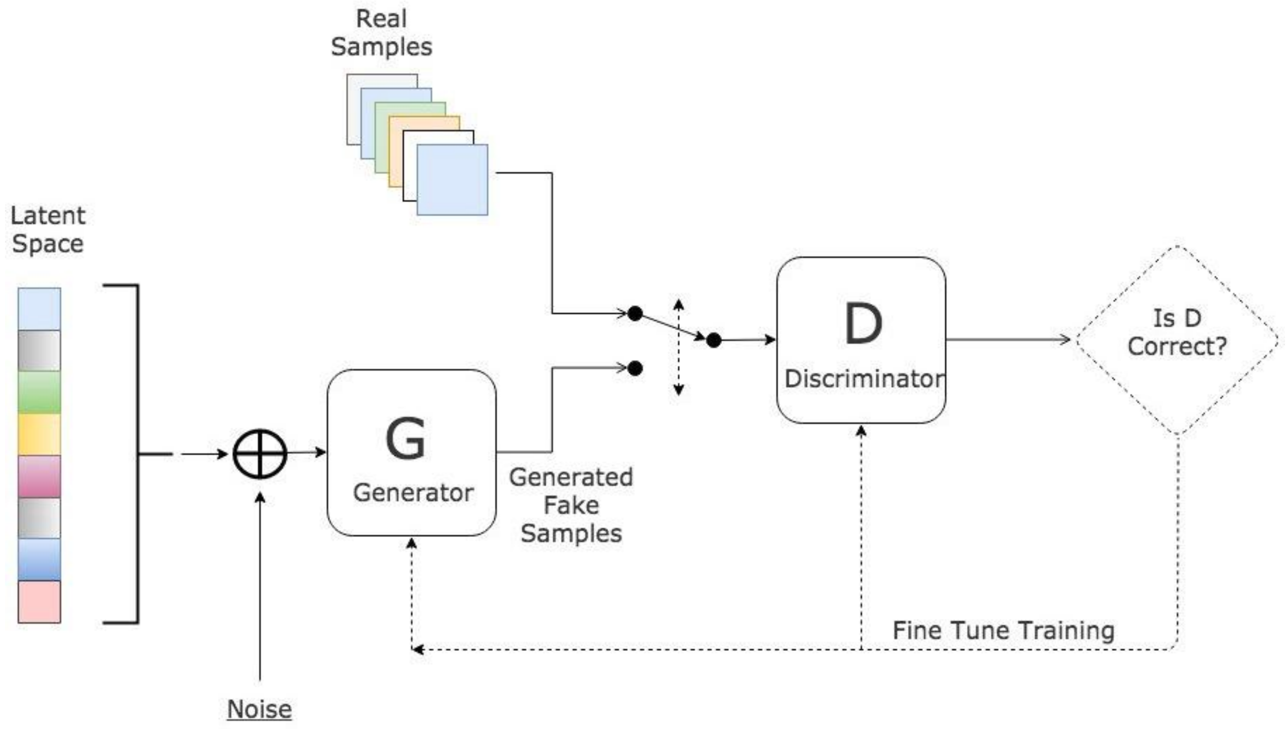
Vanilla GAN architecture

To train a GAN, we need five components.

2. Generator G

Receives the noise vector z as input and produces a (fake) image $\hat{x}$ as output.

In terms of architecture, same logic as the decoder part of our AE/VAE, i.e. **upsampling FC** or **TransposeConv** layers!



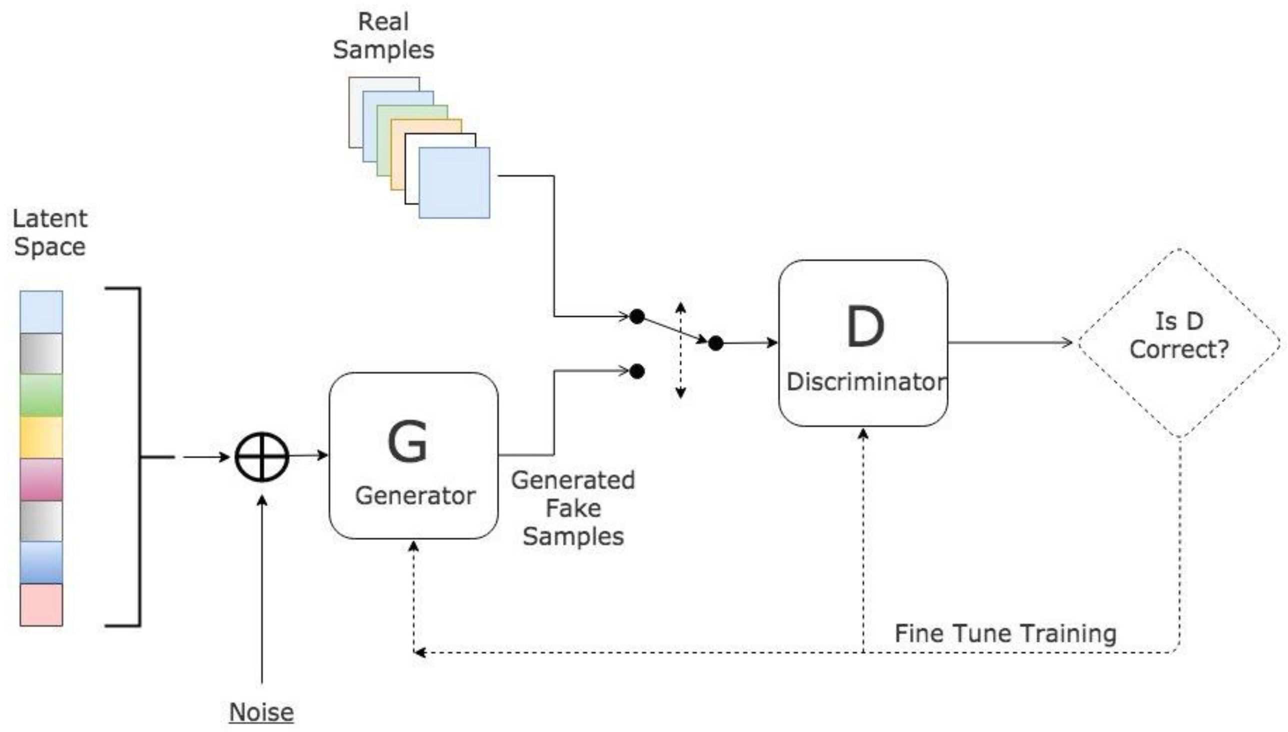
Vanilla GAN architecture

To train a GAN, we need five components.

3. Discriminator $\boldsymbol{D}$

Will receive

- an image x from the dataset X half of the time,
- an image $\hat{x} = G(z)$ from the generator the other half.



Binary classification: needs to classify the image as fake (0) or real (1).

Vanilla GAN architecture

To train a GAN, we need five components.

4. **Loss $\boldsymbol{L_D}$ on discriminator $\boldsymbol{D}$**

The purpose of D is to correctly guess whether the image is real or was generated by G (BCE loss!)

- The first component checks if the discriminator is correctly classifying the real samples.
- The second component checks if the discriminator correctly classifies the fake samples.

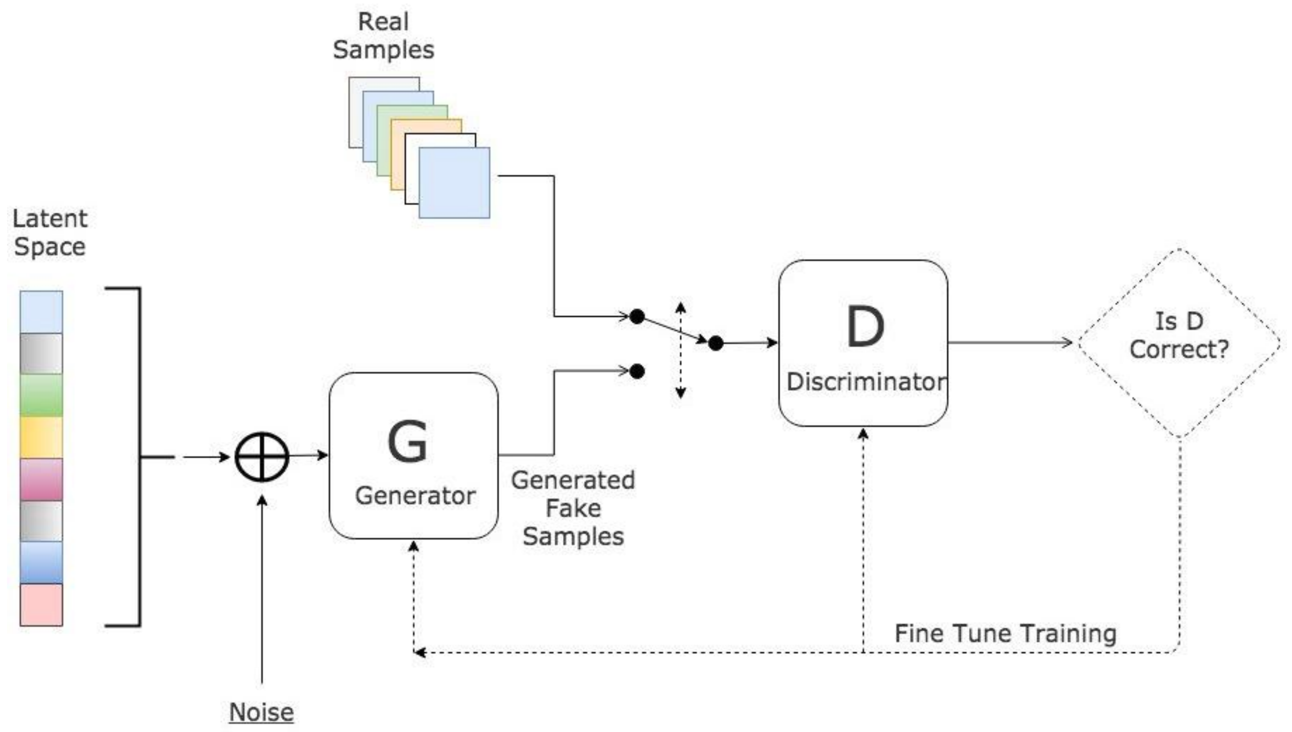
$$L_D = \frac{1}{N} \sum_{x_i \in minibatch(X_N)} -\log\big(D(x_i)\big) + \frac{1}{N} \sum_{z_i \in minibatch(N)} -\log\Big(1 - D\big(G(z_i)\big)\Big)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. **Loss L_G on generator G**

The purpose of G is to create images that are good enough to fool the discriminator D.



$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1.\log\left(D\left(G(z_i)\right)\right)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. **Loss L_G on generator G**

The purpose of G is to create images that are good enough to fool the discriminator D.

Intuitively,

- This loss checks that the discriminator classifies the **generated samples $G(z_i)$**
- as real samples (1), instead of fake ones (0).

$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1.\log\left(D\left(G(z_i)\right)\right)$$

GAN implementation: dataset

```python
# Image transform to be applied to dataset
# - Tensor conversion
transform = transforms.Compose([transforms.ToTensor()])
```

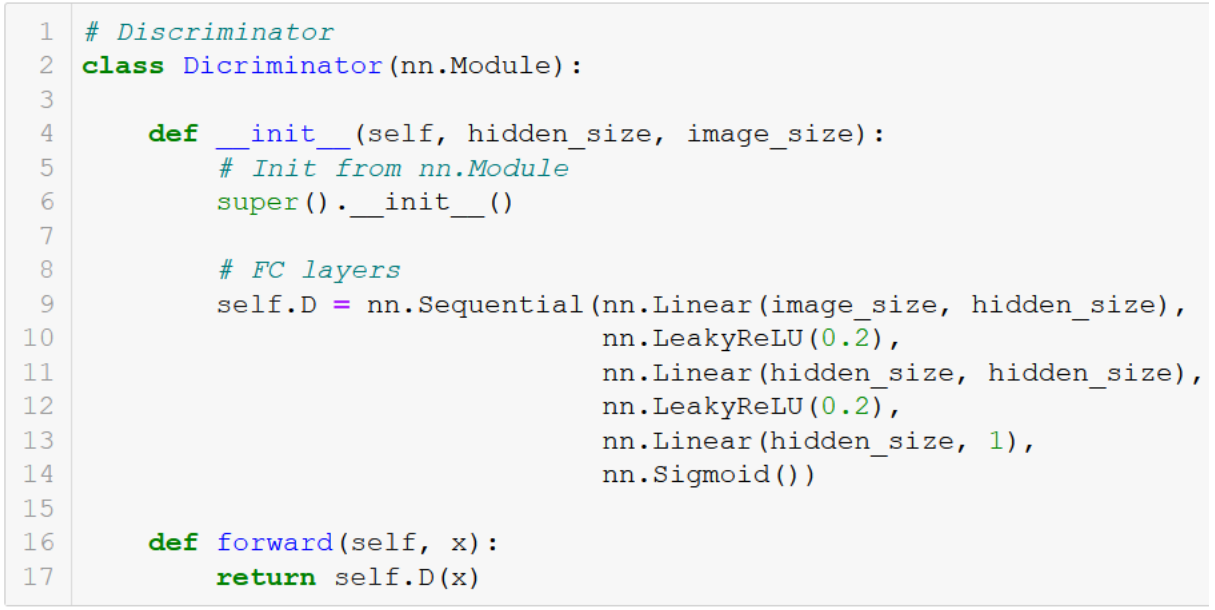
```python
# MNIST train dataset
mnist = torchvision.datasets.MNIST(root = './data/',
                                   train = True,
                                   transform = transform,
                                   download = True)
```

```python
# Data loader
batch_size = 32
data_loader = torch.utils.data.DataLoader(dataset = mnist,
                                          batch_size = batch_size,
                                          shuffle = True)
```

GAN implementation: discriminator

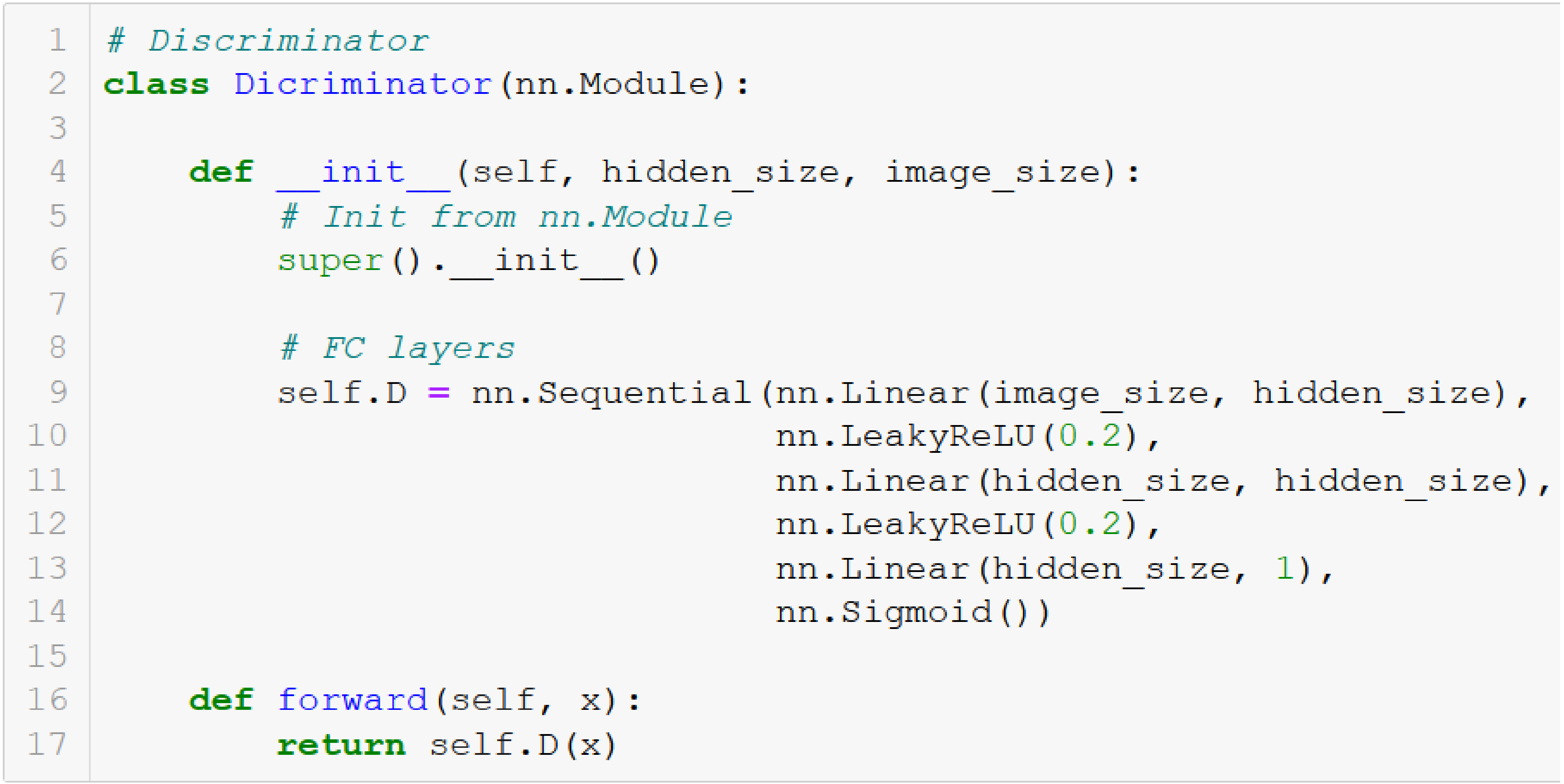
- For simplicity, our discriminator, will be designed as stacked downsampling FC layers.
- Freely decide on the hidden layer sizes.

- (Later on, we might want to replace these layers with Conv2d, which are easier to train and better at working with images!)



```python
# Discriminator
class Dicriminator(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1),
                               nn.Sigmoid())

    def forward(self, x):
        return self.D(x)
```

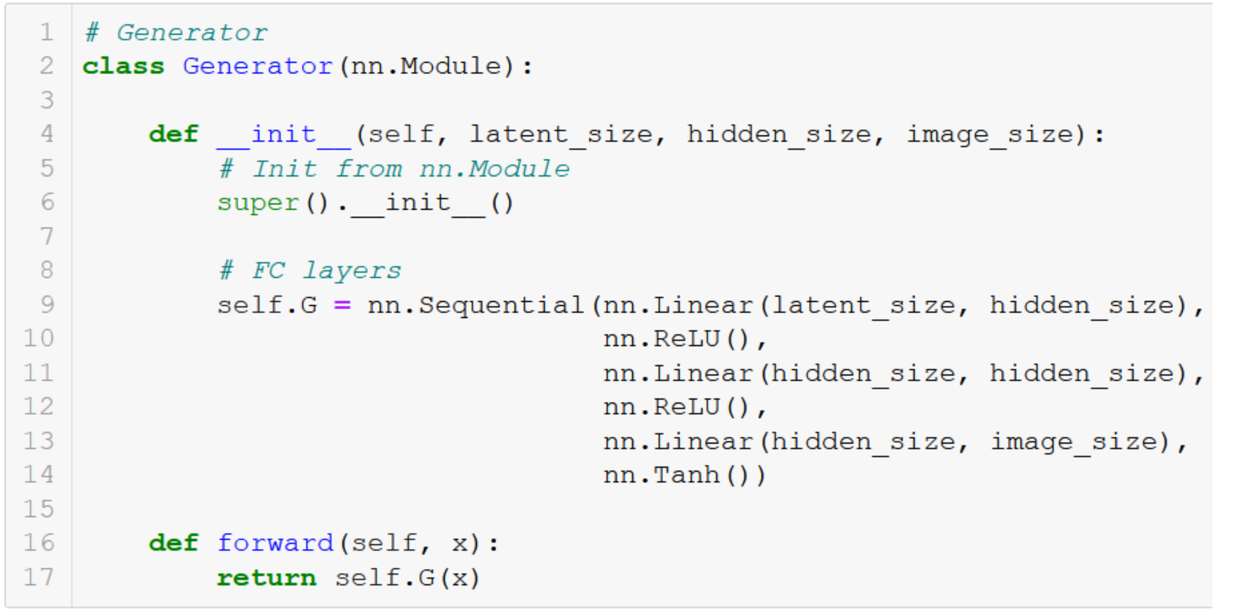
GAN implementation: discriminator



```python
# Discriminator
class Dicriminator(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1),
                               nn.Sigmoid())

    def forward(self, x):
        return self.D(x)
```

GAN implementation: generator

- For simplicity, our generator will mirror the operations of our discriminator.
- It will therefore consist of upsampling FC layers.



```
# Generator
class Generator(nn.Module):

    def __init__(self, latent_size, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.G = nn.Sequential(nn.Linear(latent_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, image_size),
                               nn.Tanh())

    def forward(self, x):
        return self.G(x)
```

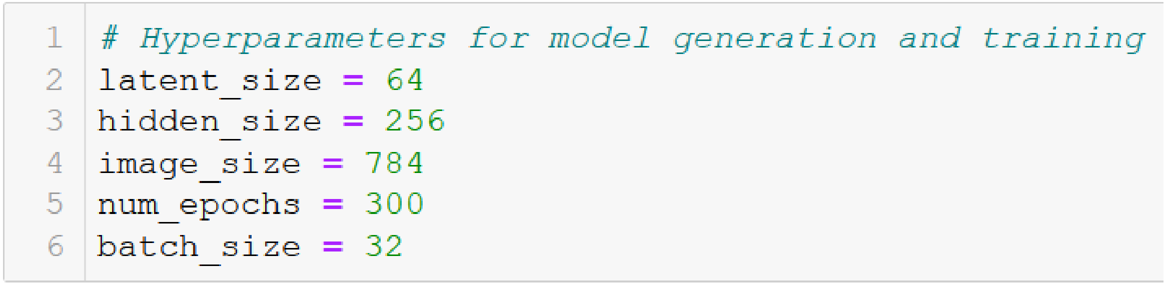
GAN implementation: generator

```python
# Generator
class Generator(nn.Module):

    def __init__(self, latent_size, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.G = nn.Sequential(nn.Linear(latent_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, image_size),
                               nn.Tanh())

    def forward(self, x):
        return self.G(x)
```

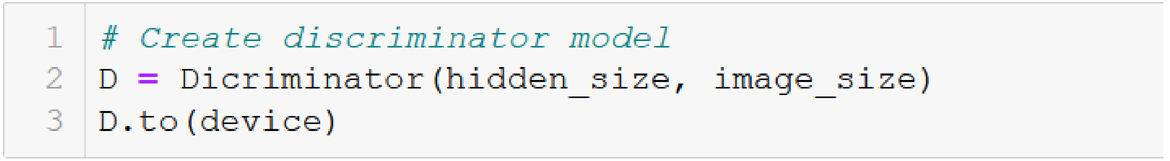
GAN implementation: trainer

Start by defining hyperparameters and models.

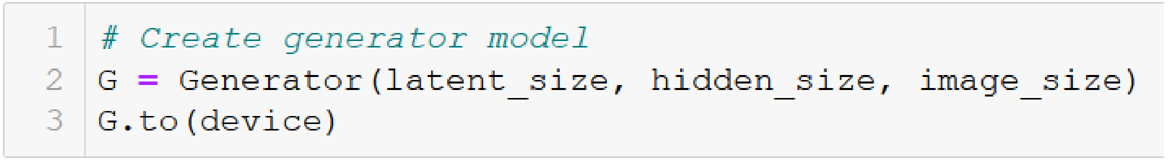
- Latent size for noise samples fed to generator arbitrarily fixed.
- Adam, with almost default parameters, arbitrarily chosen.
- Losses on D and G for training curves later on.
- Also, accuracy scores of D on real and fake samples for visualization after training.



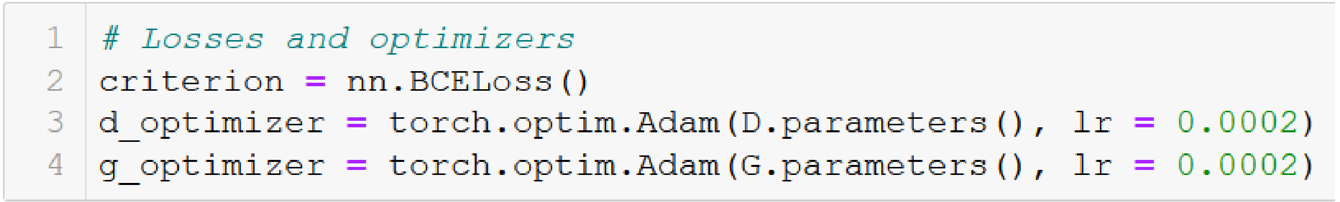
```python
# Hyperparameters for model generation and training
latent_size = 64
hidden_size = 256
image_size = 784
num_epochs = 300
batch_size = 32
```



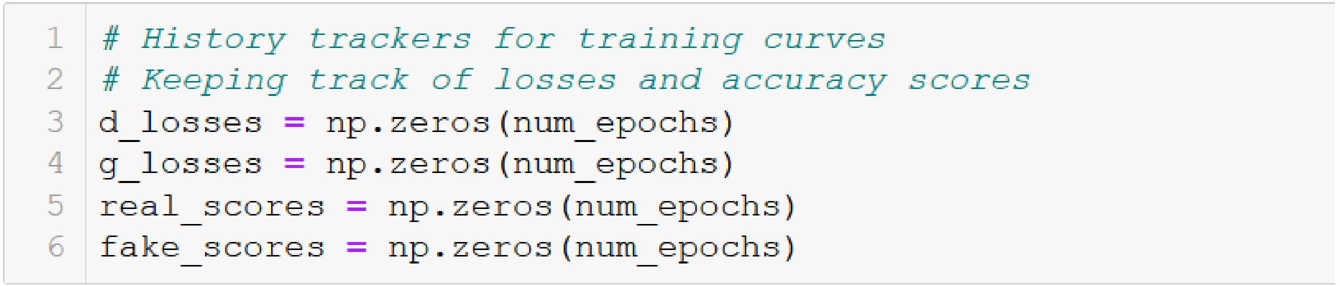
```python
# Create discriminator model
D = Dicriminator(hidden_size, image_size)
D.to(device)
```



```python
# Create generator model
G = Generator(latent_size, hidden_size, image_size)
G.to(device)
```



```python
# Losses and optimizers
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr = 0.0002)
g_optimizer = torch.optim.Adam(G.parameters(), lr = 0.0002)
```

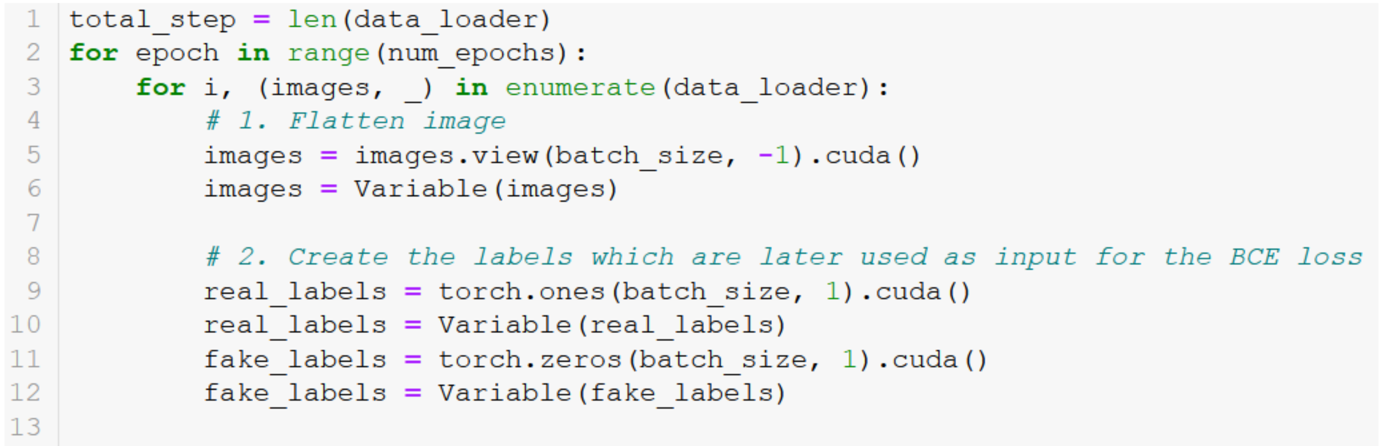


```python
# History trackers for training curves
# Keeping track of losses and accuracy scores
d_losses = np.zeros(num_epochs)
g_losses = np.zeros(num_epochs)
real_scores = np.zeros(num_epochs)
fake_scores = np.zeros(num_epochs)
```

GAN implementation: trainer

Start by processing the samples

- Generate N (mini-batch size) random noise samples, which will later be fed to generator. Make their labels 0.
- Draw N (mini-batch size) samples from the dataset X, which will be fed to discriminator later on. Make their labels 1.
- Flatten images (FC layers!)



```python
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        # 1. Flatten image
        images = images.view(batch_size, -1).cuda()
        images = Variable(images)

        # 2. Create the labels which are later used as input for the BCE loss
        real_labels = torch.ones(batch_size, 1).cuda()
        real_labels = Variable(real_labels)
        fake_labels = torch.zeros(batch_size, 1).cuda()
        fake_labels = Variable(fake_labels)

```

GAN implementation: trainer

```python
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        # 1. Flatten image
        images = images.view(batch_size, -1).cuda()
        images = Variable(images)

        # 2. Create the labels which are later used as input for the BCE loss
        real_labels = torch.ones(batch_size, 1).cuda()
        real_labels = Variable(real_labels)
        fake_labels = torch.zeros(batch_size, 1).cuda()
        fake_labels = Variable(fake_labels)

```

GAN implementation: trainer

Train the discriminator

1. Pass real samples to D, check if it is able to classify these samples correctly as real (1).
2. Compute the first half of the loss and the accuracy of D on these real samples.
3. Pass noise samples to generator G, and its outputs to discriminator D, check if it is able to classify these samples correctly as fake (0).
4. Compute the second half of the loss and the accuracy of D on these fake samples.
5. Backpropagate D on the computed combined loss.

GAN implementation: trainer

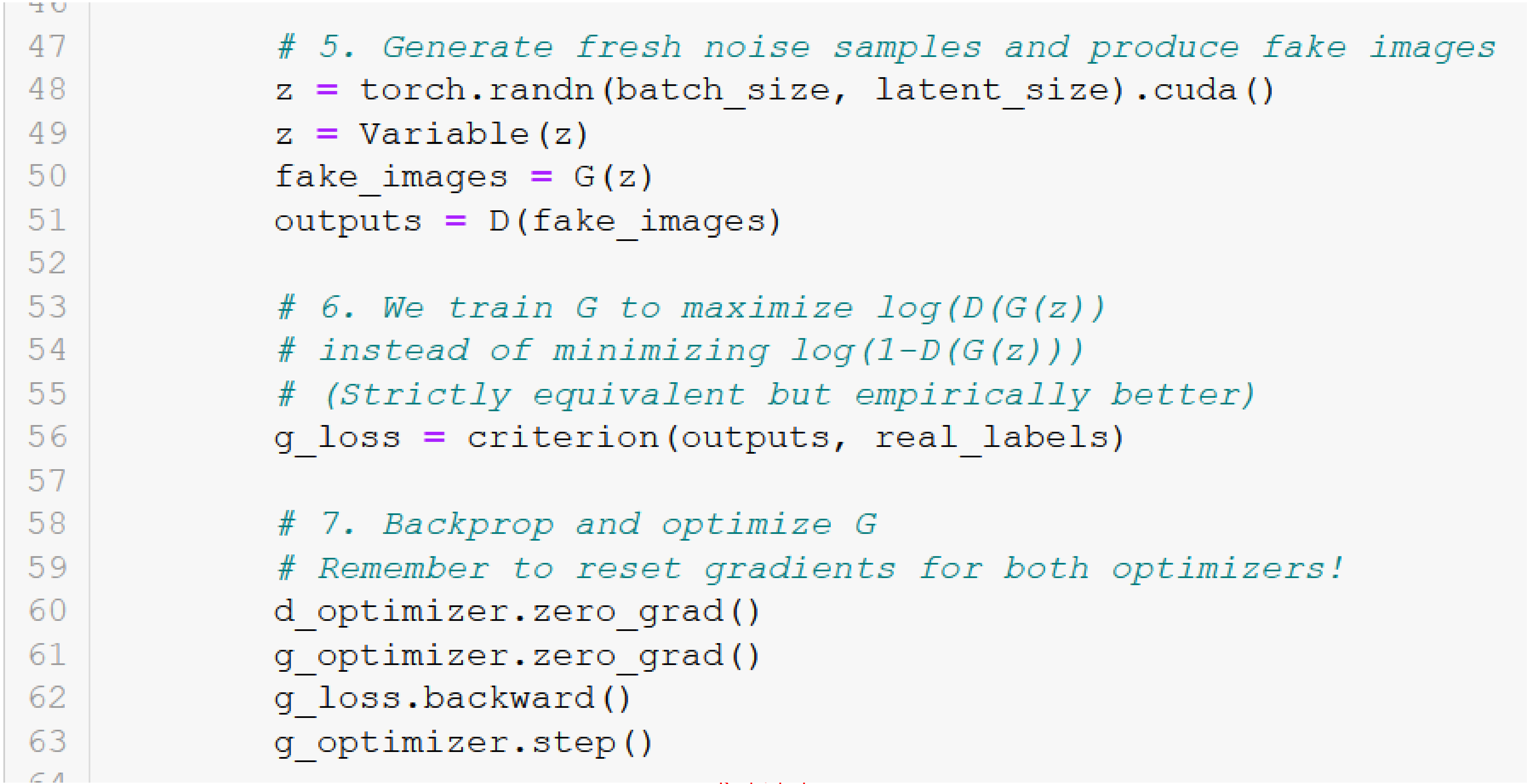
```
        # 3. Compute BCE_Loss using real images
        # Here, BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
        # Second term of the loss is always zero since real_labels = 1
        outputs = D(images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs

        # 3.bis. Compute BCELoss using fake images
        # Here, BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
        # First term of the loss is always zero since fake_labels = 0
        z = torch.randn(batch_size, latent_size).cuda()
        z = Variable(z)
        fake_images = G(z)
        outputs = D(fake_images)
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs

        # 4. Backprop and optimize for D
        # Remember to reset gradients for both optimizers!
        d_loss = d_loss_real + d_loss_fake
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()
        d_loss.backward()
        d_optimizer.step()
```

GAN implementation: trainer

Train the generator

1. Produce a fresh batch of noise samples to be fed to the generator.
2. Produce fake images by feeding these noise samples to generator G.
3. Pass fake images to discriminator D, check if it is misclassifying these samples as real (1) instead of fake.
4. Backpropagate G on the computed loss.

GAN implementation: trainer



```
        # 5. Generate fresh noise samples and produce fake images
        z = torch.randn(batch_size, latent_size).cuda()
        z = Variable(z)
        fake_images = G(z)
        outputs = D(fake_images)

        # 6. We train G to maximize log(D(G(z))
        # instead of minimizing log(1-D(G(z)))
        # (Strictly equivalent but empirically better)
        g_loss = criterion(outputs, real_labels)

        # 7. Backprop and optimize G
        # Remember to reset gradients for both optimizers!
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()
        g_loss.backward()
        g_optimizer.step()
```

GAN implementation: trainer

Definition (interleaved training):

In general, we like to train a discriminator, while the generator is fixed, and vice-versa. We then alternate a few rounds of training on the discriminator/generator.

This prevents the discriminator from getting used to what the generator is producing and in turn, forces the generator to generate better samples, with higher chance of fooling the discriminator.

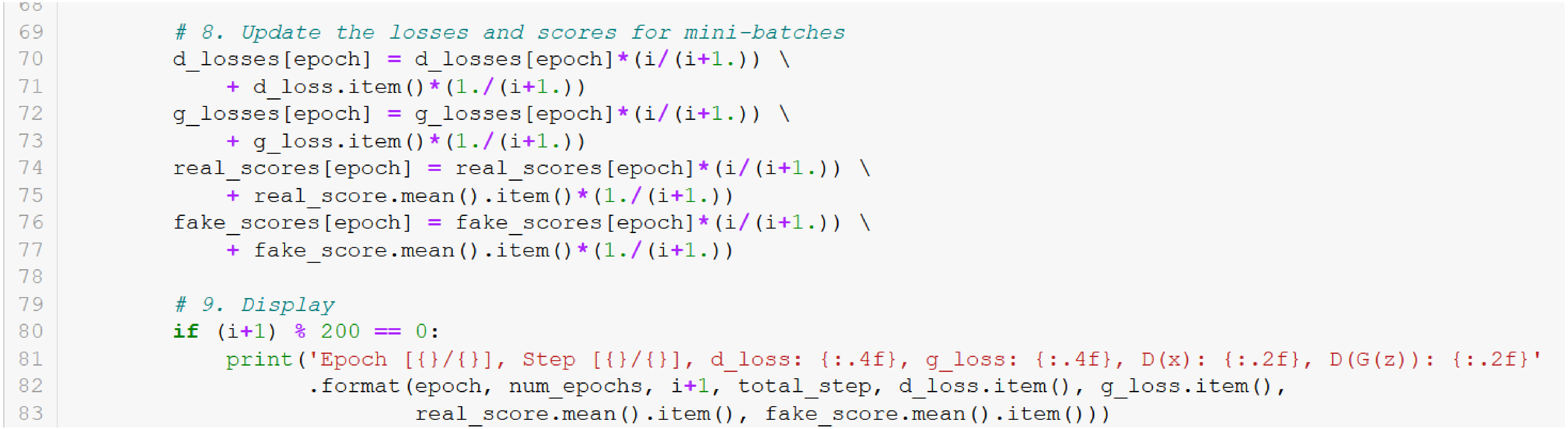
This is called an **interleaved training**, and is very common in GANs, or **game theory** with multiple players trying to figure out their best strategies.

However, this does not ensure that the GAN training will converge!

GAN implementation: trainer

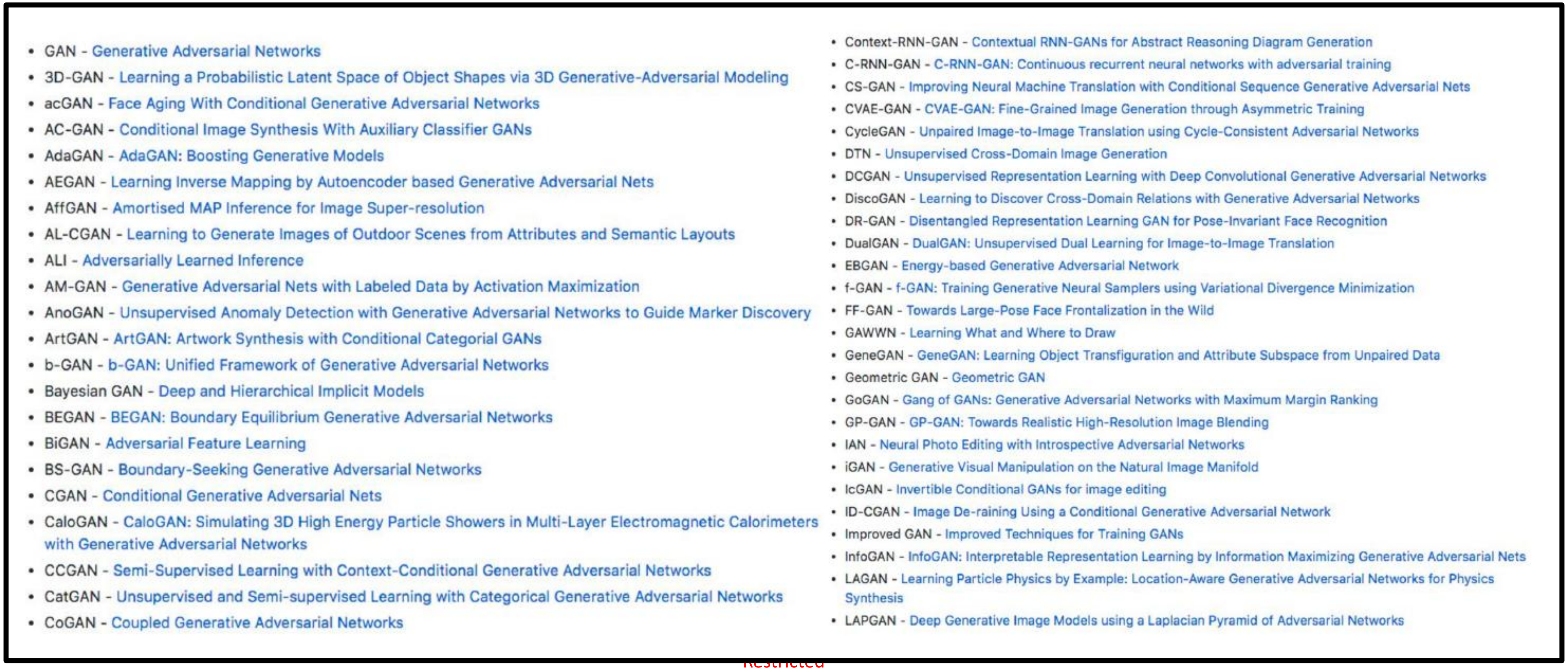
Update loss and accuracy history after each mini-batch.

Display on periodic epoch values for convenience.



```
        # 8. Update the losses and scores for mini-batches
        d_losses[epoch] = d_losses[epoch]*(i/(i+1.)) \
            + d_loss.item()*(1./(i+1.))
        g_losses[epoch] = g_losses[epoch]*(i/(i+1.)) \
            + g_loss.item()*(1./(i+1.))
        real_scores[epoch] = real_scores[epoch]*(i/(i+1.)) \
            + real_score.mean().item()*(1./(i+1.))
        fake_scores[epoch] = fake_scores[epoch]*(i/(i+1.)) \
            + fake_score.mean().item()*(1./(i+1.))

        # 9. Display
        if (i+1) % 200 == 0:
            print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}, D(x): {:.2f}, D(G(z)): {:.2f}'
                  .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.item(),
                          real_score.mean().item(), fake_score.mean().item()))
```

Many variations to this vanilla GAN exist!



- GAN - Generative Adversarial Networks
- 3D-GAN - Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling
- acGAN - Face Aging With Conditional Generative Adversarial Networks
- AC-GAN - Conditional Image Synthesis With Auxiliary Classifier GANs
- AdaGAN - AdaGAN: Boosting Generative Models
- AEGAN - Learning Inverse Mapping by Autoencoder based Generative Adversarial Nets
- AffGAN - Amortised MAP Inference for Image Super-resolution
- AL-CGAN - Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts
- ALI - Adversarially Learned Inference
- AM-GAN - Generative Adversarial Nets with Labeled Data by Activation Maximization
- AnoGAN - Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery
- ArtGAN - ArtGAN: Artwork Synthesis with Conditional Categorial GANs
- b-GAN - b-GAN: Unified Framework of Generative Adversarial Networks
- Bayesian GAN - Deep and Hierarchical Implicit Models
- BEGAN - BEGAN: Boundary Equilibrium Generative Adversarial Networks
- BiGAN - Adversarial Feature Learning
- BS-GAN - Boundary-Seeking Generative Adversarial Networks
- CGAN - Conditional Generative Adversarial Nets
- CaloGAN - CaloGAN: Simulating 3D High Energy Particle Showers in Multi-Layer Electromagnetic Calorimeters with Generative Adversarial Networks
- CCGAN - Semi-Supervised Learning with Context-Conditional Generative Adversarial Networks
- CatGAN - Unsupervised and Semi-supervised Learning with Categorical Generative Adversarial Networks
- CoGAN - Coupled Generative Adversarial Networks
- Context-RNN-GAN - Contextual RNN-GANs for Abstract Reasoning Diagram Generation
- C-RNN-GAN - C-RNN-GAN: Continuous recurrent neural networks with adversarial training
- CS-GAN - Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets
- CVAE-GAN - CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training
- CycleGAN - Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- DTN - Unsupervised Cross-Domain Image Generation
- DCGAN - Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
- DiscoGAN - Learning to Discover Cross-Domain Relations with Generative Adversarial Networks
- DR-GAN - Disentangled Representation Learning GAN for Pose-Invariant Face Recognition
- DualGAN - DualGAN: Unsupervised Dual Learning for Image-to-Image Translation
- EBGAN - Energy-based Generative Adversarial Network
- f-GAN - f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
- FF-GAN - Towards Large-Pose Face Frontalization in the Wild
- GAWWN - Learning What and Where to Draw
- GeneGAN - GeneGAN: Learning Object Transfiguration and Attribute Subspace from Unpaired Data
- Geometric GAN - Geometric GAN
- GoGAN - Gang of GANs: Generative Adversarial Networks with Maximum Margin Ranking
- GP-GAN - GP-GAN: Towards Realistic High-Resolution Image Blending
- IAN - Neural Photo Editing with Introspective Adversarial Networks
- iGAN - Generative Visual Manipulation on the Natural Image Manifold
- IcGAN - Invertible Conditional GANs for image editing
- ID-CGAN - Image De-raining Using a Conditional Generative Adversarial Network
- Improved GAN - Improved Techniques for Training GANs
- InfoGAN - InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
- LAGAN - Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis
- LAPGAN - Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks

Limits of vanilla/Wasserstein GANs

Issue #1: GANs are effective at generating images, but

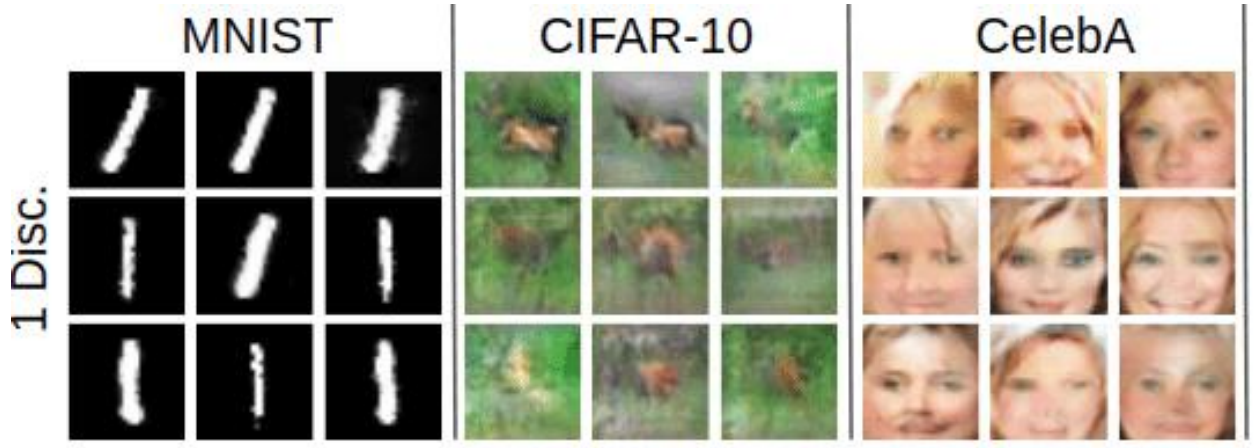
- they are limited to small images sizes,
- because of model stability,
- and training required.

- **Generating high-resolution images is challenging for GANs, because the generator must learn how to produce both large structure and fine details at the same time.**

Limits of vanilla/Wasserstein GANs

Issue #2: Lack of control on the generators outputs

- Limited understanding of the impact of noise vectors used as input in generators (amount of randomness? structure of the latent space?)
- How to disentangle features/properties in images and add controls for these properties to the generator model?



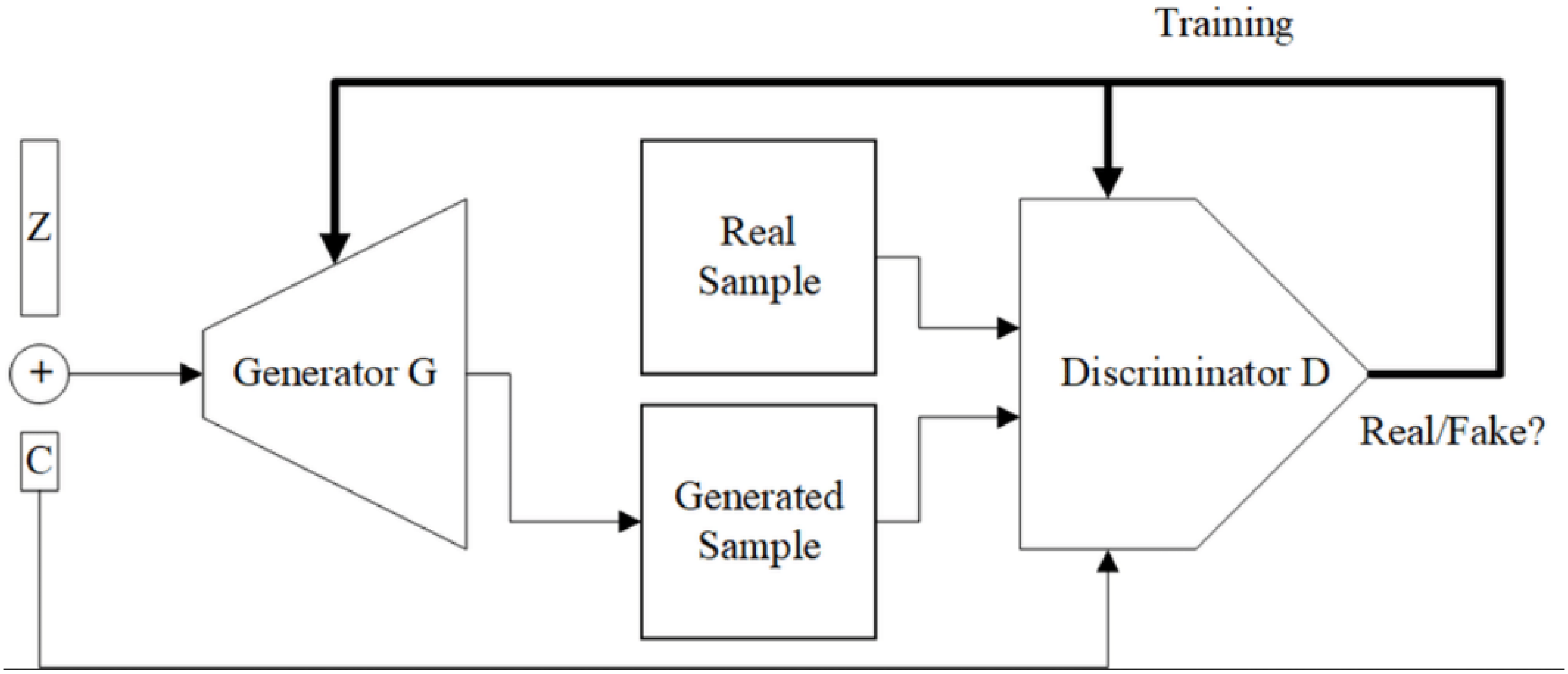
Conditional GANs

- **Issue #2:** Lack of control on the generators outputs.

Core idea of Conditional GANs [CondGAN]**:** Add some deterministic values to the noise samples used in the generator.

- This deterministic value corresponds to the class you expect the sample to have (e.g. generate a handwritten 7).
- Generator task is to fool the discriminator/critic by generating a sample, which will be classified as a “real 7” and not anything else (“fake 7”, “real 6” are both considered failure cases).

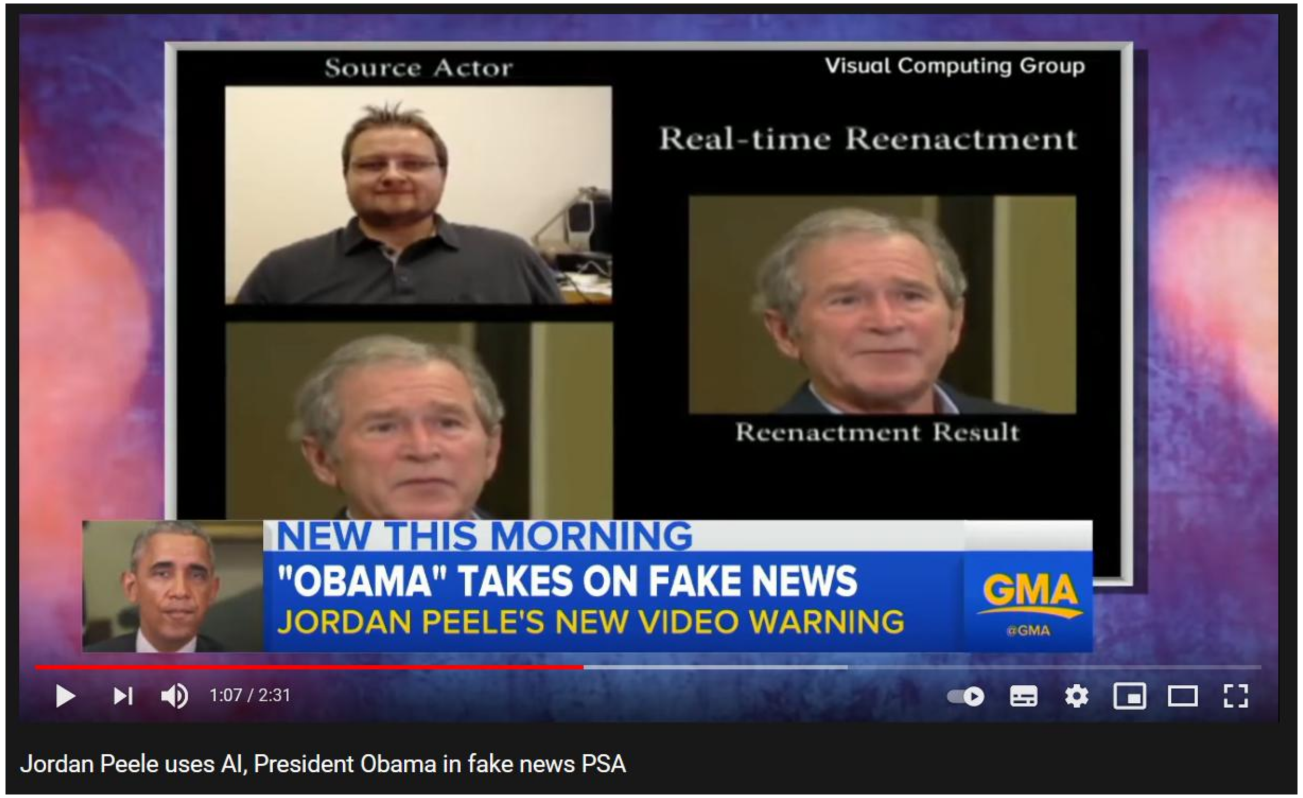
Conditional GANs



CycleGAN

- **Core idea of CycleGANs** [CycleGAN]: do not use noise as input for generator, use another image that you wish to transform!
- Train an autoencoder style of architecture, with two generators.
- One generator G_1 that transforms images of zebras into horses, and one generator G_2 that transforms images of horses into zebras.
- Train both generators to accomplish $G_1(G_2(x)) = x$.
- By doing so, you get a generator, which transforms images of a certain type into images of another type!

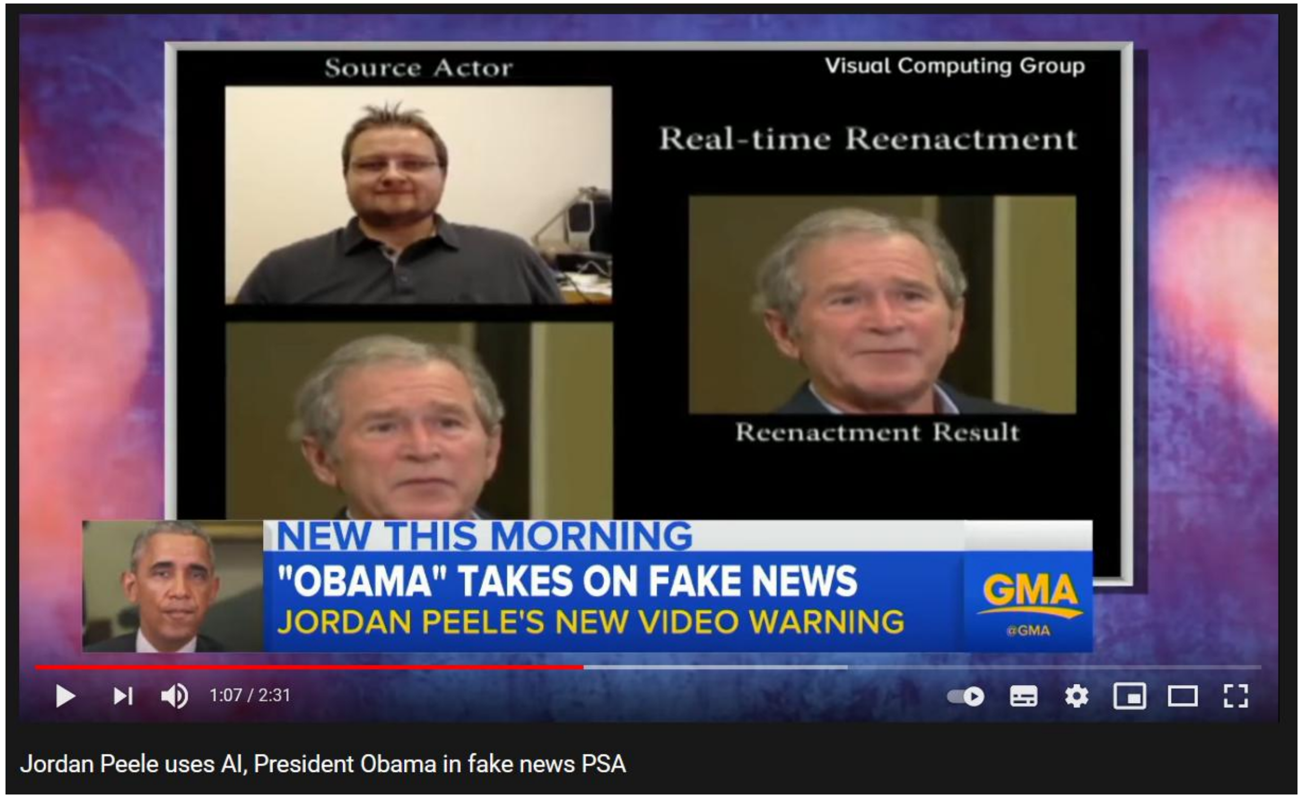
GANs and ethics



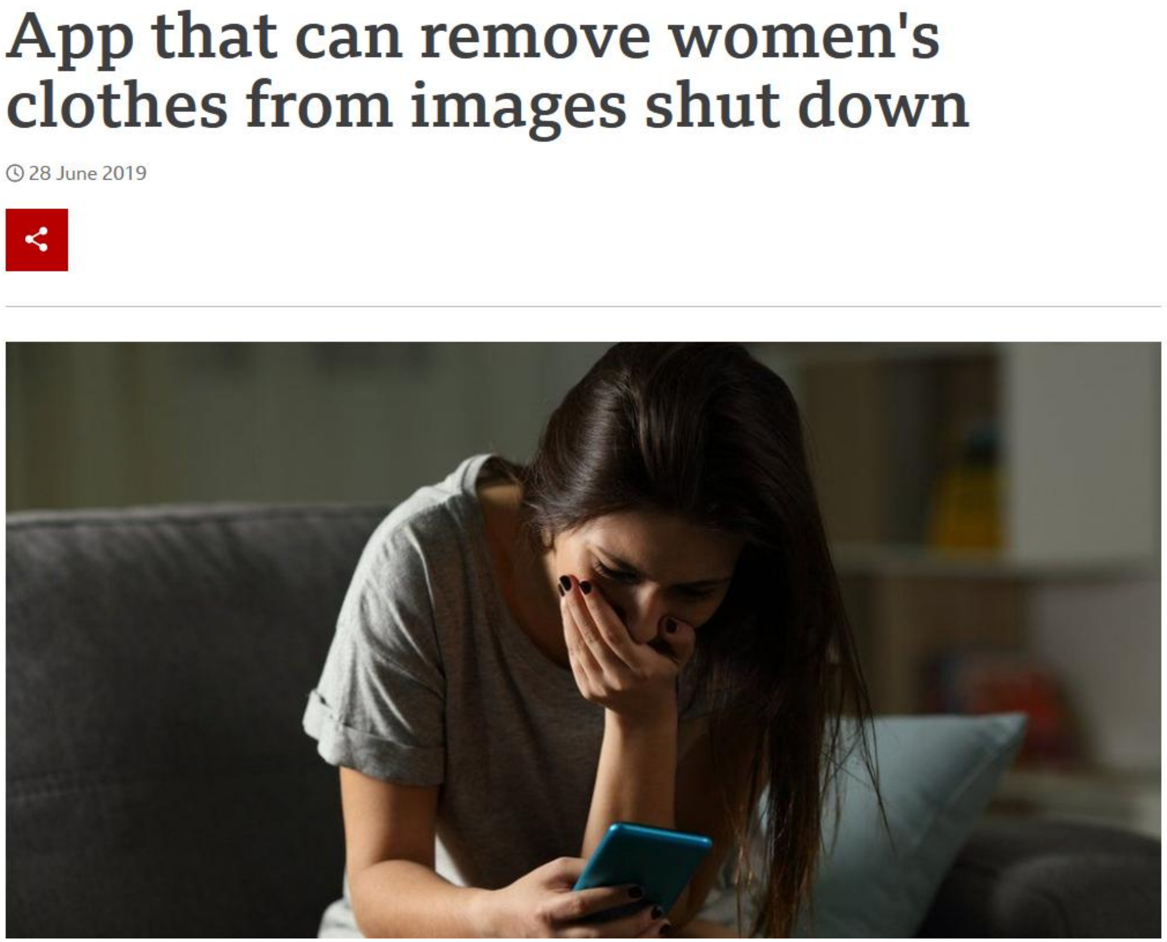
Jordan Peele uses AI, President Obama in fake news PSA

https://www.youtube.com/watch?v=bE1KWpoX9Hk

GANs and ethics



Jordan Peele uses AI, President Obama in fake news PSA



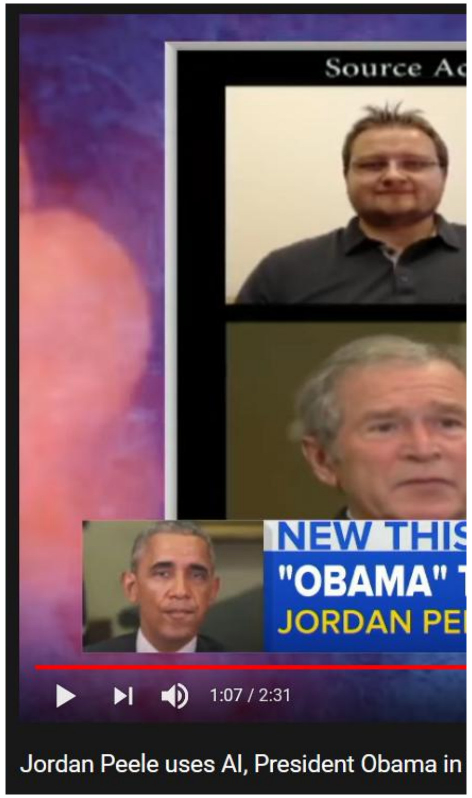
App that can remove women's clothes from images shut down

28 June 2019

https://www.youtube.com/watch?v=bE1KWpoX9Hk

https://www.bbc.com/news/technology-48799045

GANs a



Jordan Peele uses AI, President Obama in

A Zelensky Deepfake Was Quickly Defeated. The Next One Might Not Be

PHOTOGRAPH: EMIN SANSAR/GETTY IMAGES

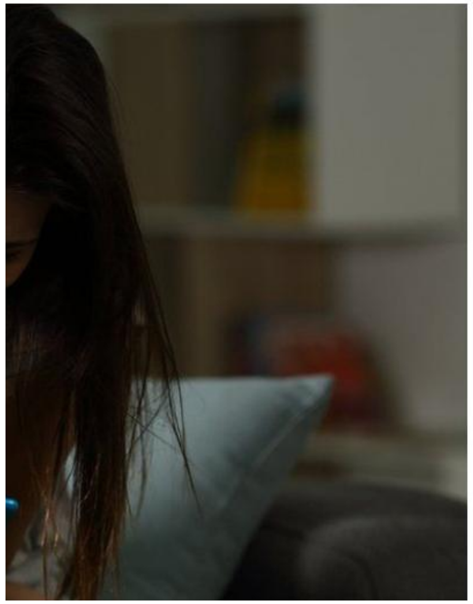
The AI Database →

APPLICATION: DEEPFAKES, SAFETY SECTOR: PUBLIC SAFETY, SOCIAL MEDIA

SOURCE DATA: VIDEO

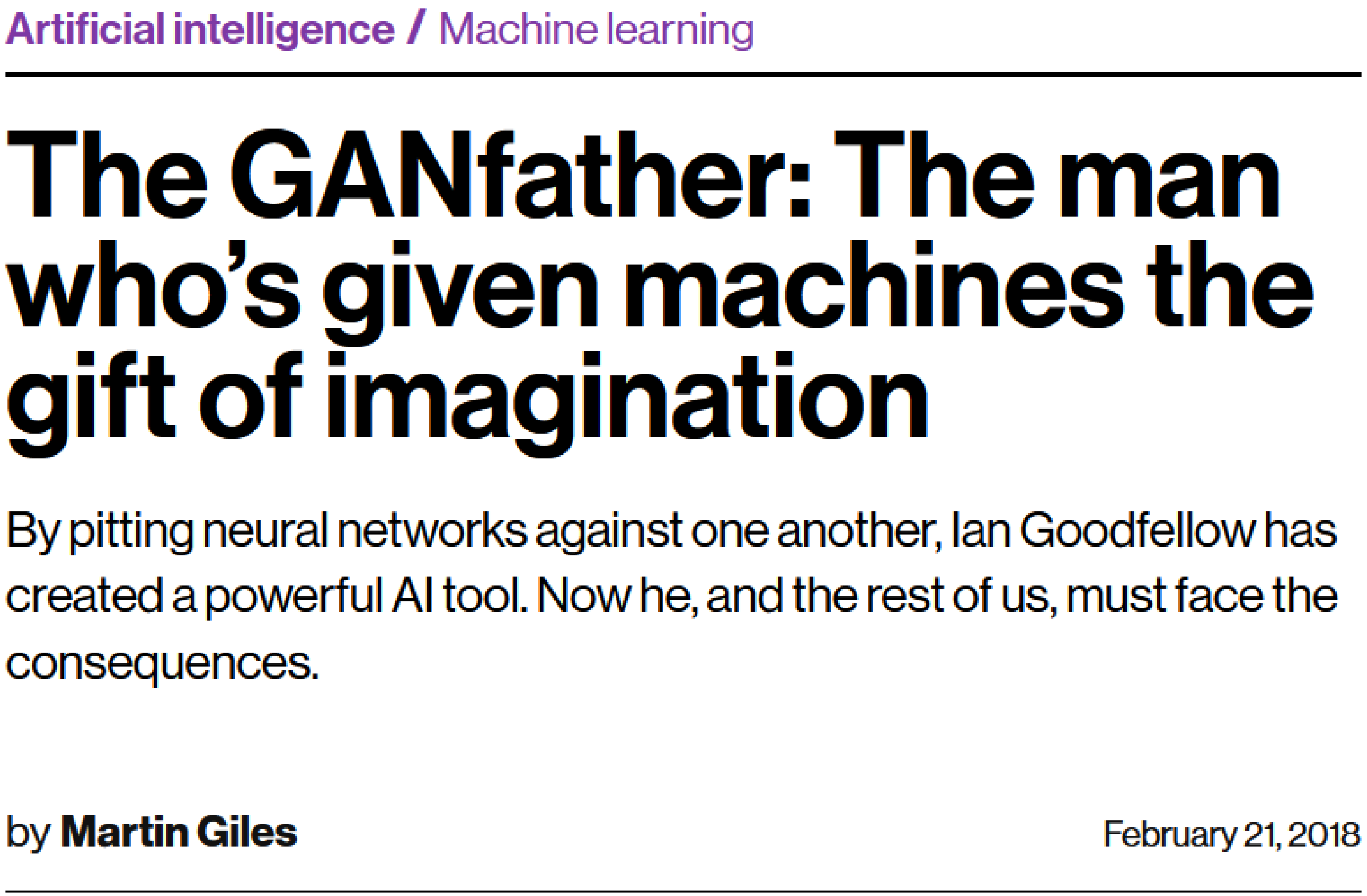
ON MARCH 2, the Ukraine government's Center for Strategic Communication warned that its enemies might be preparing a "deepfake" video that appeared to show president Volodymyr Zelensky announcing his surrender to Russia's invasion. On Wednesday, that warning appeared prescient.

women's
ıut down



GANs and ethics

App that can remove women's
ıut down

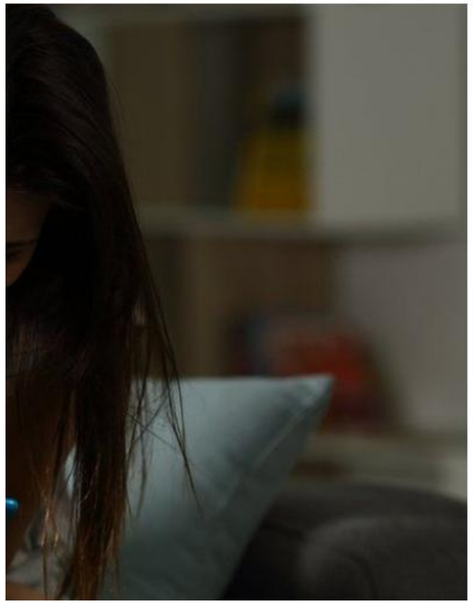


Artificial intelligence / Machine learning

The GANfather: The man who's given machines the gift of imagination

By pitting neural networks against one another, Ian Goodfellow has created a powerful AI tool. Now he, and the rest of us, must face the consequences.

by **Martin Giles** February 21, 2018



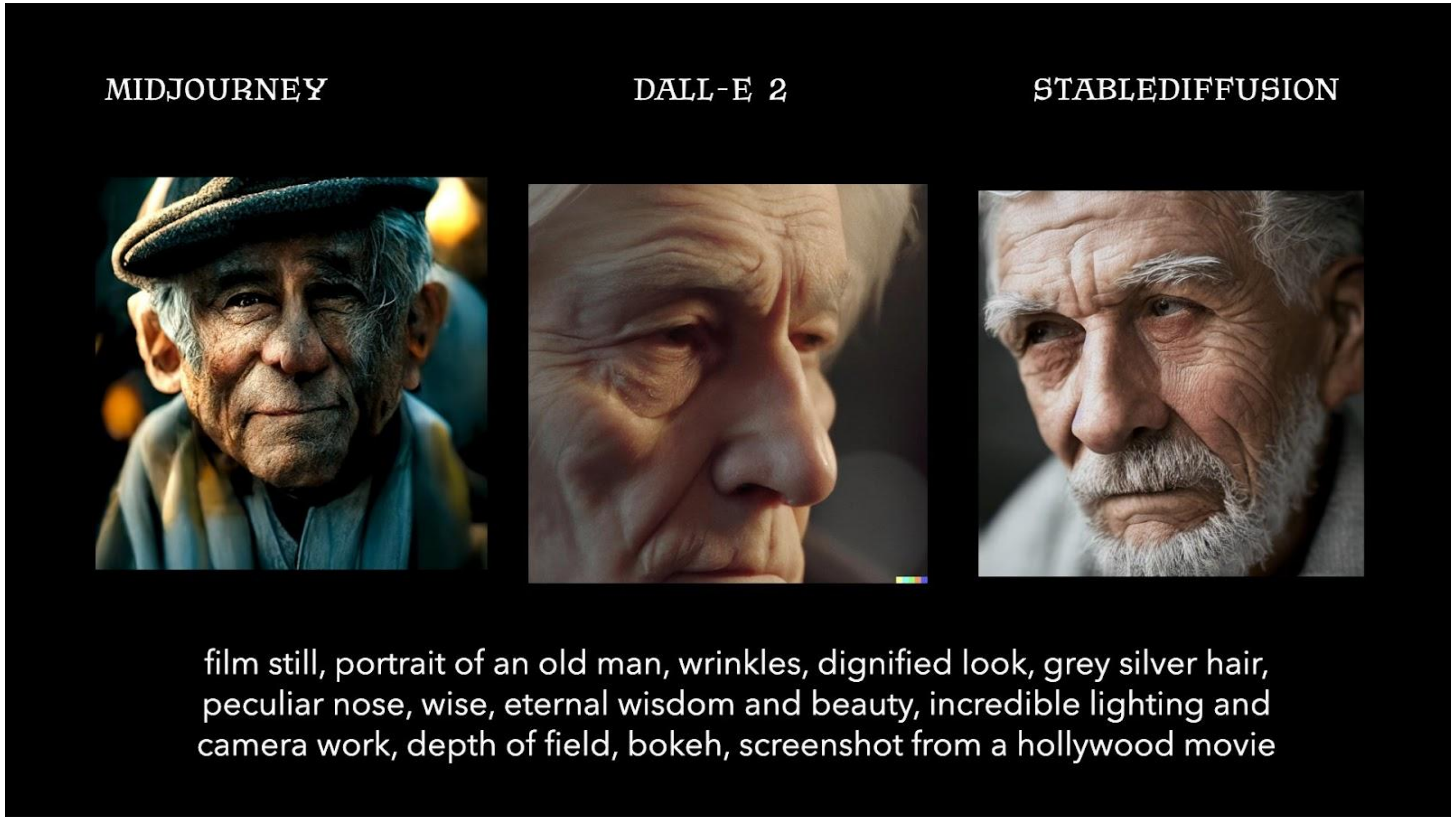
https://www.technologyreview.com/2018/02/21/145289/the-ganfather-the-man-whos-given-machines-the-gift-of-imagination/

What about diffusion models?!

You might have heard **diffusion models**…

- E.g. Dall-E2, StableDiffusion, MidJourney, etc.

Turns out: special type of generative models!


MIDJOURNEY
DALL-E 2
STABLEDIFFUSION
film still, portrait of an old man, wrinkles, dignified look, grey silver hair, peculiar nose, wise, eternal wisdom and beauty, incredible lighting and camera work, depth of field, bokeh, screenshot from a hollywood movie

Diffusion Models: a definition

Definition (Diffusion Models):

Diffusion models are a class of **stochastic mathematical models** that have typically been used to describe the random motion of particles, often referred to as “Brownian motion.“

They are widely used in various fields, such as physics, chemistry, biology, and finance, for instance to model the stochastic behaviour of particles or other quantities that exhibit random fluctuations over time.

The fundamental concept behind diffusion models is the idea of Stochastic Differential Equations (SDEs).

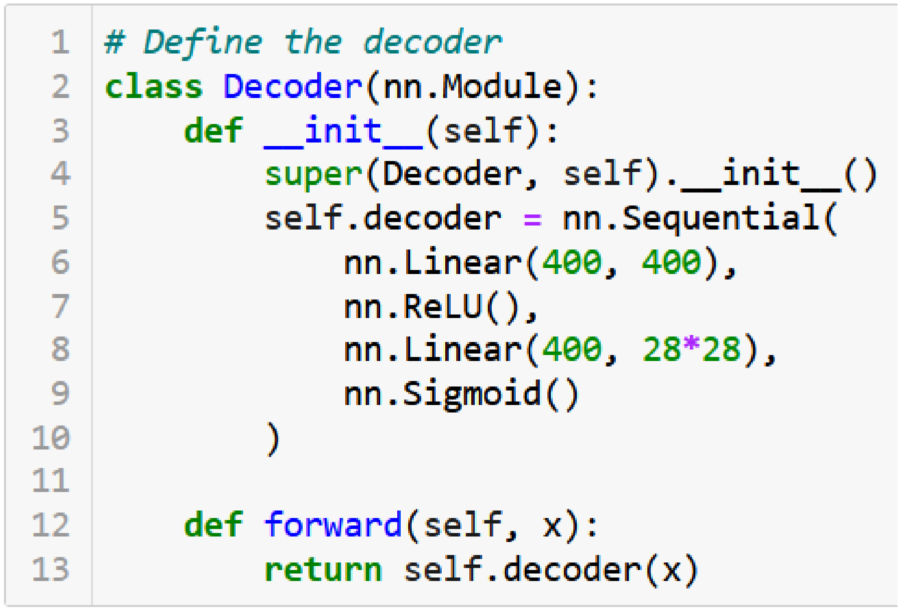
A first task, denoising images

Let us start by defining the four components of our ML problem.

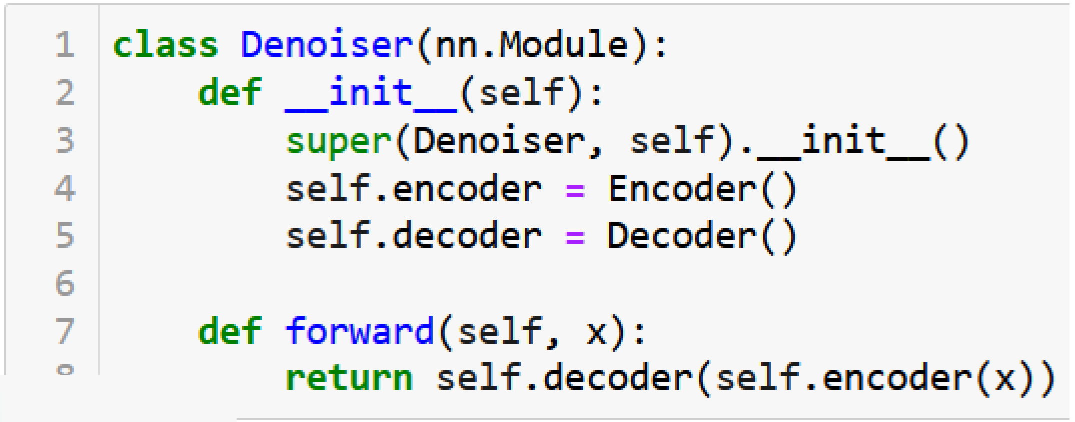
- **Task:** take images from MNIST and noise them. The model should try its best to denoise them an get back to the original image.
- **Dataset:** MNIST images, noised (inputs) and original (output).
- **Model:** an encoder-decoder model of some sort, similar to an autoencoder (almost).
- **Loss and training procedure:** As in the case of autoencoders, a simple MSE loss and a basic training procedure with Adam optimizer will do.

A first task, denoising images

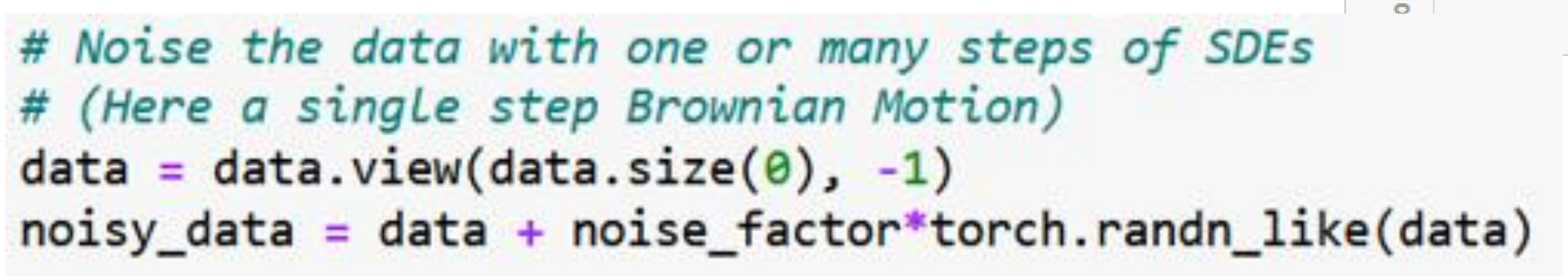
- For simplicity, we consider an encoder-decoder model, running on a simple noising operation and a denoiser consisting of simple linear layers.
- And a basic training procedure…



```python
# Define the decoder
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(400, 400),
            nn.ReLU(),
            nn.Linear(400, 28*28),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.decoder(x)
```



```python
class Denoiser(nn.Module):
    def __init__(self):
        super(Denoiser, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        return self.decoder(self.encoder(x))
```



```python
# Noise the data with one or many steps of SDEs
# (Here a single step Brownian Motion)
data = data.view(data.size(0), -1)
noisy_data = data + noise_factor*torch.randn_like(data)
```

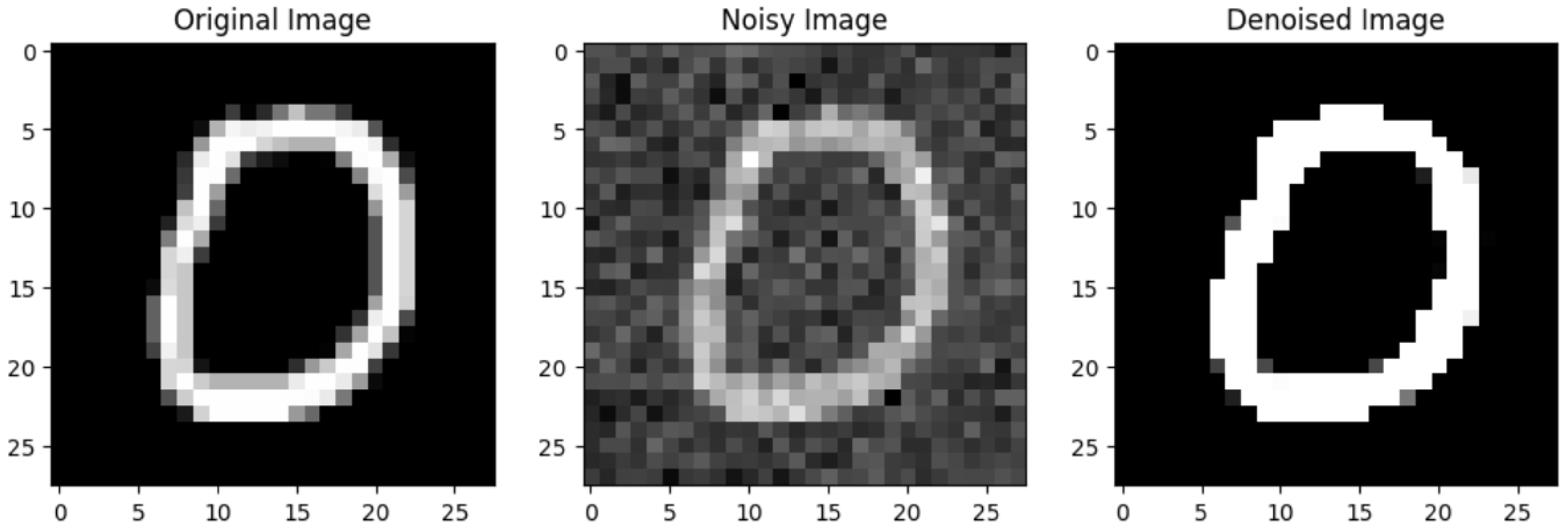
```python
# Hyperparameters
epochs = 10
learning_rate = 1e-3
noise_factor = 0.5
```

```python
# Initialize the model, loss function, and optimizer
model = Denoiser()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr = learning_rate)
```

```python
# Training Loop
for epoch in range(epochs):
    for batch_idx, (data, _) in enumerate(train_loader):
        # Noise the data with one or many steps of SDEs
        # (Here a single step Brownian Motion)
        data = data.view(data.size(0), -1)
        noisy_data = data + noise_factor*torch.randn_like(data)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(noisy_data)

        # Backprop
        loss = criterion(outputs, data)
        loss.backward()
        optimizer.step()

        # Display
        if batch_idx % 100 == 0:
            print(f'Epoch: {epoch+1}/{epochs}, Batch: {batch_idx}/{len(train_loader)}, Loss: {loss.item()}')
```

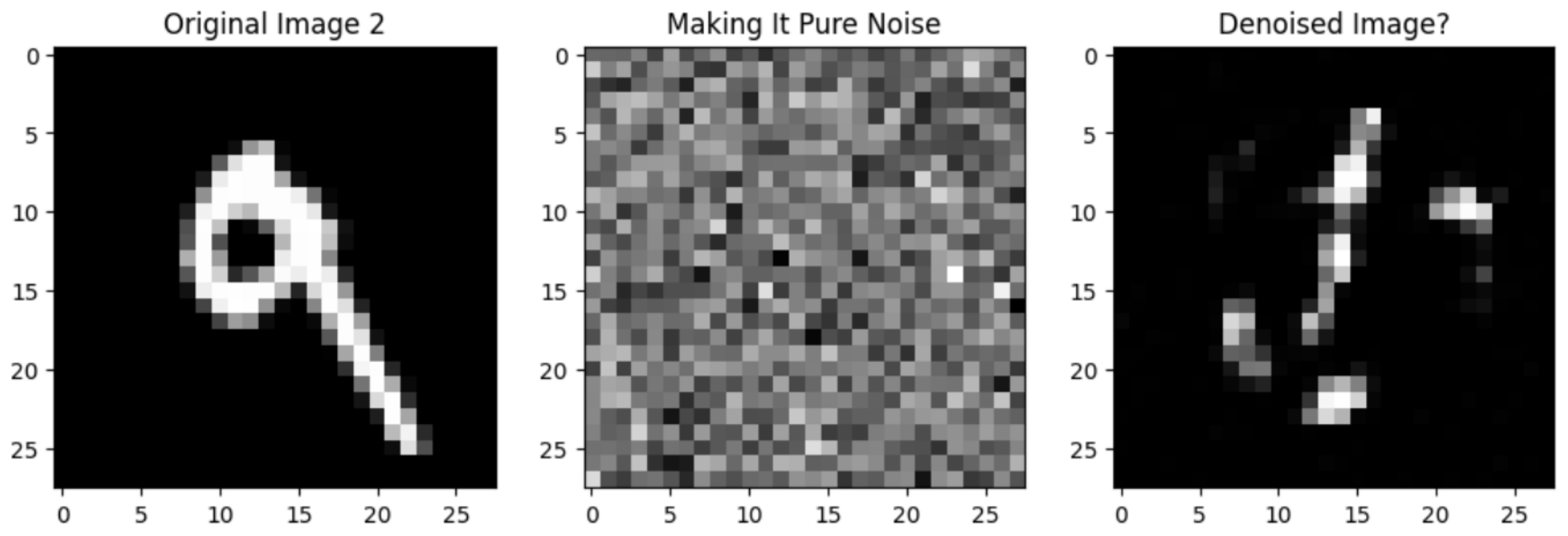
A first task, denoising images

- The model trains and seems able to denoise lightly noised images.



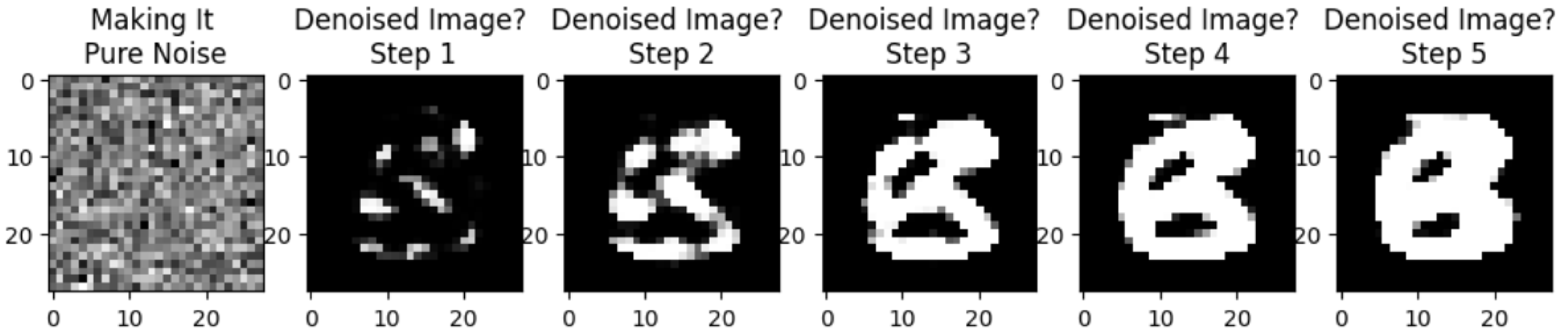
A first task, denoising images

- The model trains and seems able to denoise lightly noised images.
- **Could it denoise a pure noise image, though?** Not really.



A first task, denoising images

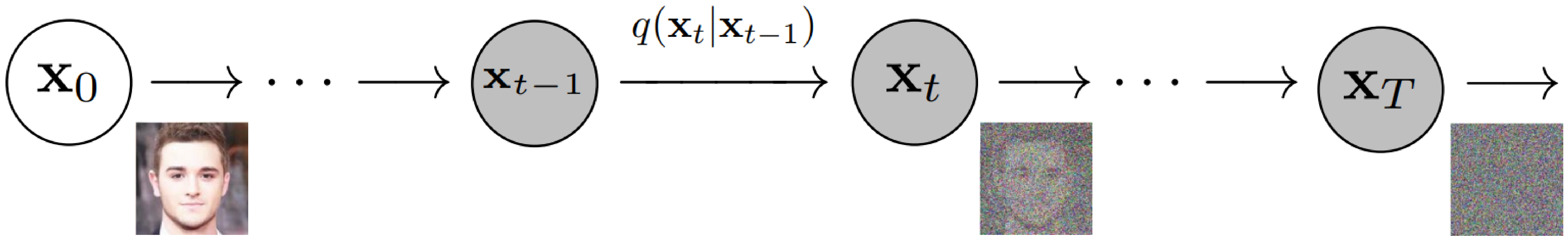
- The model trains and seems able to denoise lightly noised images.
- **Could it denoise a pure noise image, though?** Not really.
- **What if we used the model many (e.g. 5) times in a row on our full noise image?** Eventually would obtain something B&W in the likes of MNIST but somehow non-sensical in terms of shapes…?!



A (not so?) crazy idea

Idea: what if instead of noising the image in one step, and denoising it in one call of the model, we try iterative noising and denoising?

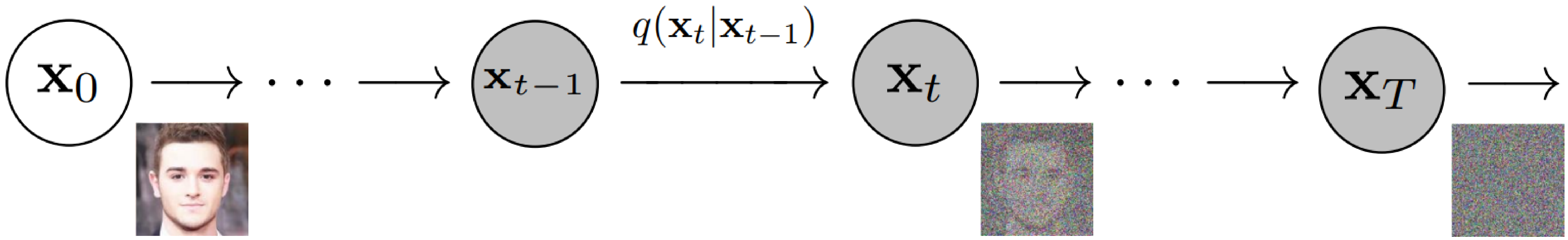
(Somehow similar to the idea of modifying the image iteratively during the iterated versions of our attacks in W5!)



A (not so?) crazy idea

Idea: what if instead of noising the image in one step, and denoising it in one call of the model, we try iterative noising and denoising?

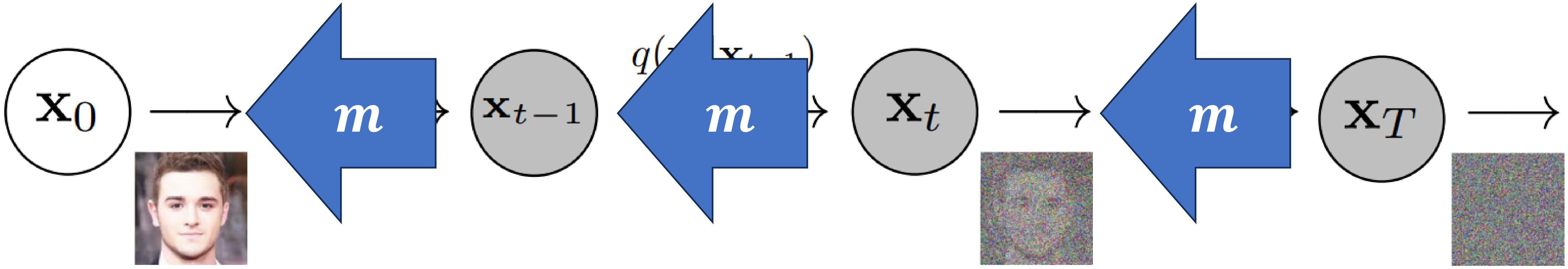
- **Step 1:** Add progressive noising T times in a row, until the image becomes pure noise (or so).
- The noising process follows some sort of an SDE equation describing the progressive noising of the original image (our Forward here!).



A (not so?) crazy idea

Idea: what if instead of noising the image in one step, and denoising it in one call of the model, we try iterative noising and denoising?

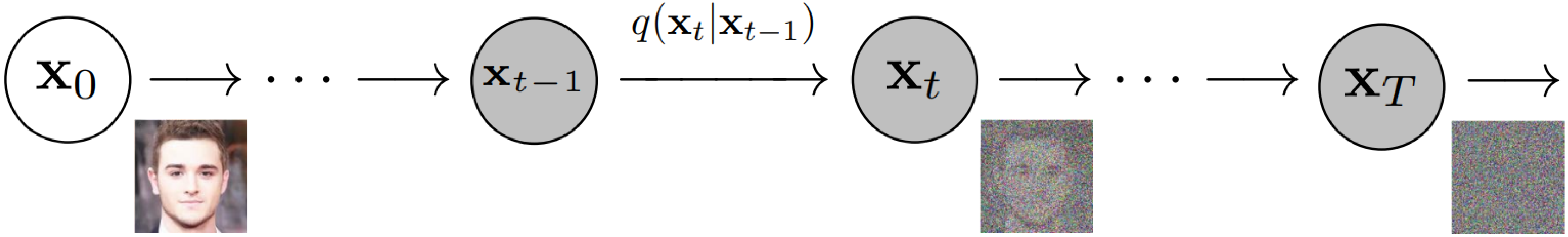
- **Step 2:** Train a single model m to try and cancel noising iteratively.
- Our model m receives image x_t and tries to produce x_{t-1}, do for each noising iteration t (going backwards here!).



A (not so?) crazy idea

Idea: what if instead of noising the image in one step, and denoising it in one call of the model, we try iterative noising and denoising?

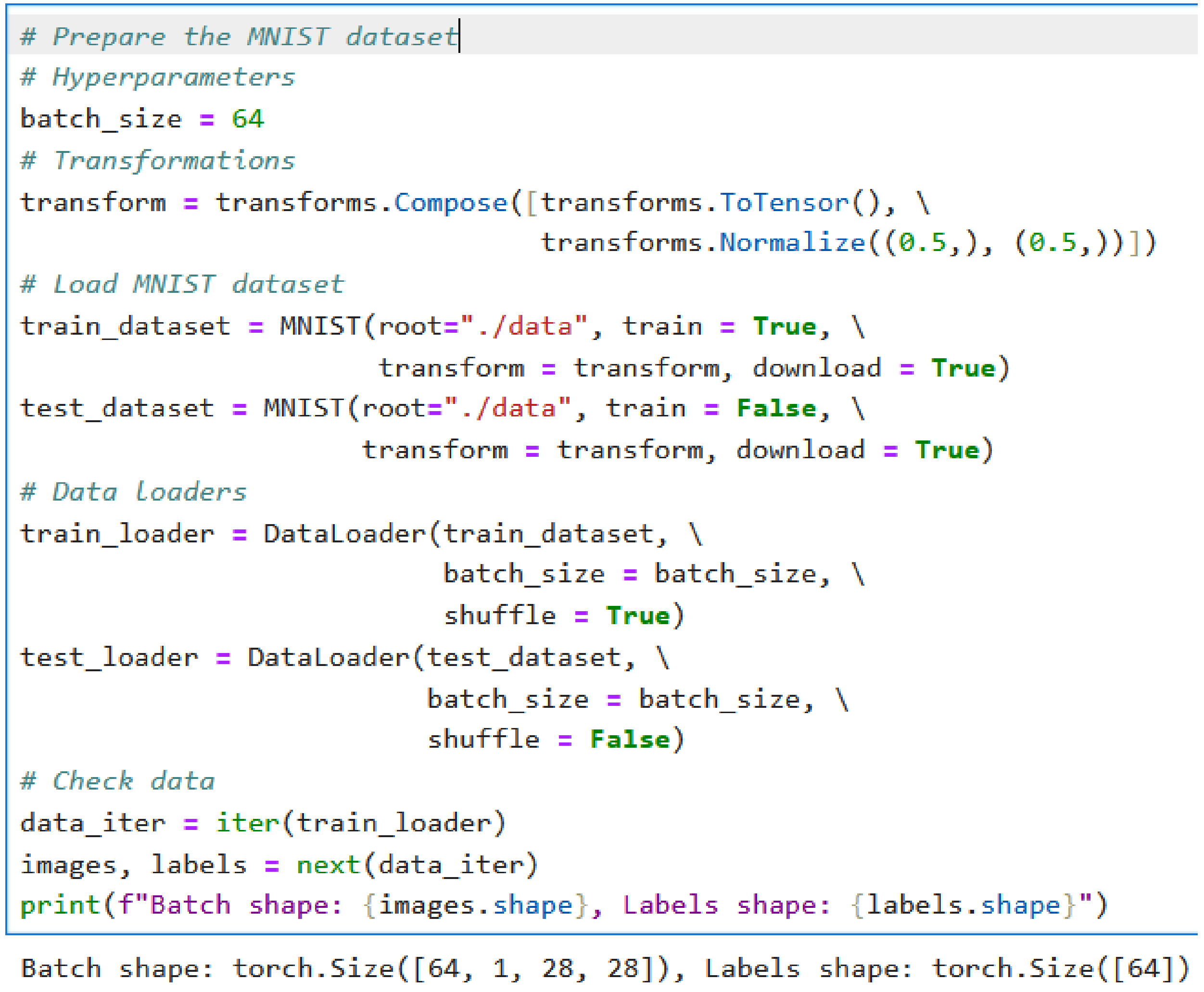
- **Training procedure:** Use original images from MNIST and noise them T times in a row. Then go backwards and train model m on denoising each step and pair of images (x_t, x_{t-1}).
- Use MSE loss and training procedure similar as before.



A basic Diffusion Model setup

First, we need a toy dataset.

- For simplicity, we will rely on MNIST and will attempt to train a model that can generate MNIST-like images from noise.
- As in previous lectures, we will prepare our dataset and dataloader objects.
- Nothing particularly difficult or exciting here.



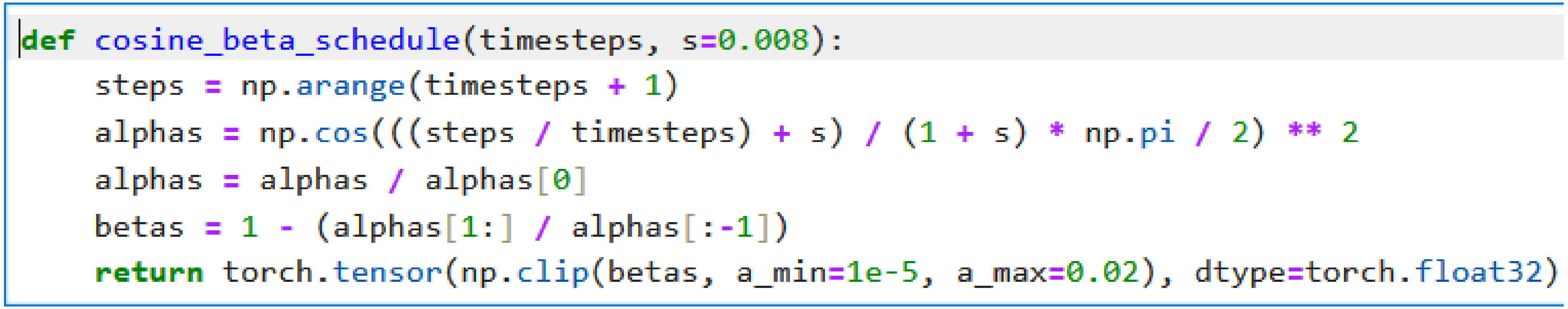
```
# Prepare the MNIST dataset
# Hyperparameters
batch_size = 64
# Transformations
transform = transforms.Compose([transforms.ToTensor(), \
                                transforms.Normalize((0.5,), (0.5,))])
# Load MNIST dataset
train_dataset = MNIST(root="./data", train = True, \
                      transform = transform, download = True)
test_dataset = MNIST(root="./data", train = False, \
                     transform = transform, download = True)
# Data Loaders
train_loader = DataLoader(train_dataset, \
                          batch_size = batch_size, \
                          shuffle = True)
test_loader = DataLoader(test_dataset, \
                         batch_size = batch_size, \
                         shuffle = False)
# Check data
data_iter = iter(train_loader)
images, labels = next(data_iter)
print(f"Batch shape: {images.shape}, Labels shape: {labels.shape}")
```

```
Batch shape: torch.Size([64, 1, 28, 28]), Labels shape: torch.Size([64])
```

A basic Diffusion Model setup

Next, we need a noising function, whose job is to progressively add noise to MNIST images

- Define a noise scheduler, have it progressively add more noise.
- Cosine beta scheduler: on early timesteps, add little noise, on late timesteps, add more noise, smooth transition.

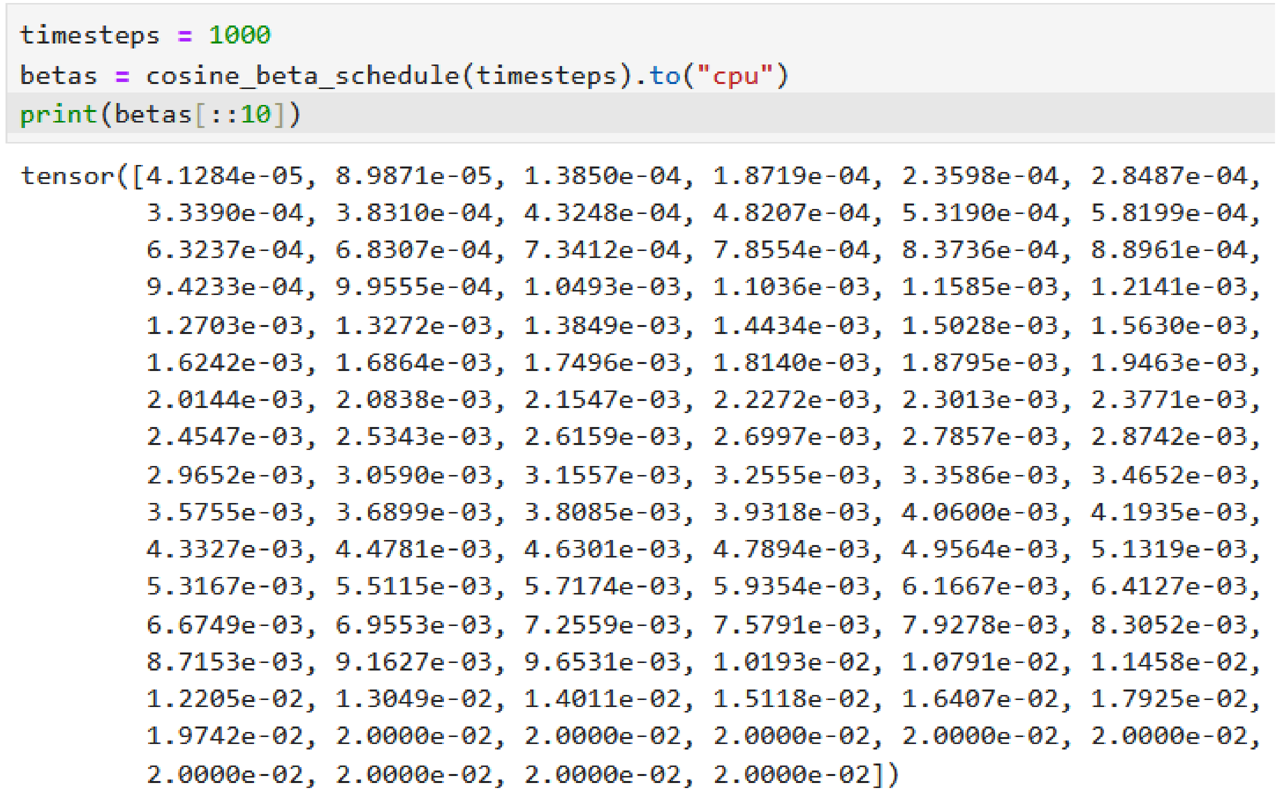


```python
def cosine_beta_schedule(timesteps, s=0.008):
    steps = np.arange(timesteps + 1)
    alphas = np.cos(((steps / timesteps) + s) / (1 + s) * np.pi / 2) ** 2
    alphas = alphas / alphas[0]
    betas = 1 - (alphas[1:] / alphas[:-1])
    return torch.tensor(np.clip(betas, a_min=1e-5, a_max=0.02), dtype=torch.float32)
```

A basic Diffusion Model setup

For a given number of timesteps, for instance $T = 1000$,

- Generate 1000 beta values using our cosine beta noise scheduler.
- As can be seen, the noise amplitude $\beta(t)$ has values that are progressively increasing with the value $t \in (0, 1000)$.



```
timesteps = 1000
betas = cosine_beta_schedule(timesteps).to("cpu")
print(betas[::10])
```

```
tensor([4.1284e-05, 8.9871e-05, 1.3850e-04, 1.8719e-04, 2.3598e-04, 2.8487e-04,
        3.3390e-04, 3.8310e-04, 4.3248e-04, 4.8207e-04, 5.3190e-04, 5.8199e-04,
        6.3237e-04, 6.8307e-04, 7.3412e-04, 7.8554e-04, 8.3736e-04, 8.8961e-04,
        9.4233e-04, 9.9555e-04, 1.0493e-03, 1.1036e-03, 1.1585e-03, 1.2141e-03,
        1.2703e-03, 1.3272e-03, 1.3849e-03, 1.4434e-03, 1.5028e-03, 1.5630e-03,
        1.6242e-03, 1.6864e-03, 1.7496e-03, 1.8140e-03, 1.8795e-03, 1.9463e-03,
        2.0144e-03, 2.0838e-03, 2.1547e-03, 2.2272e-03, 2.3013e-03, 2.3771e-03,
        2.4547e-03, 2.5343e-03, 2.6159e-03, 2.6997e-03, 2.7857e-03, 2.8742e-03,
        2.9652e-03, 3.0590e-03, 3.1557e-03, 3.2555e-03, 3.3586e-03, 3.4652e-03,
        3.5755e-03, 3.6899e-03, 3.8085e-03, 3.9318e-03, 4.0600e-03, 4.1935e-03,
        4.3327e-03, 4.4781e-03, 4.6301e-03, 4.7894e-03, 4.9564e-03, 5.1319e-03,
        5.3167e-03, 5.5115e-03, 5.7174e-03, 5.9354e-03, 6.1667e-03, 6.4127e-03,
        6.6749e-03, 6.9553e-03, 7.2559e-03, 7.5791e-03, 7.9278e-03, 8.3052e-03,
        8.7153e-03, 9.1627e-03, 9.6531e-03, 1.0193e-02, 1.0791e-02, 1.1458e-02,
        1.2205e-02, 1.3049e-02, 1.4011e-02, 1.5118e-02, 1.6407e-02, 1.7925e-02,
        1.9742e-02, 2.0000e-02, 2.0000e-02, 2.0000e-02, 2.0000e-02, 2.0000e-02,
        2.0000e-02, 2.0000e-02, 2.0000e-02, 2.0000e-02])
```

A basic Diffusion Model setup

Finally, define a forward diffusion function, whose objective is to progressively noise the MNIST images.

- Apply noise using the noise amplitude $\beta(t)$ on each time step $t \in (0, 1000)$. Return all progressively noised images and noises applied.
- More specifically, we have

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1 - \alpha_t}\, \epsilon$$

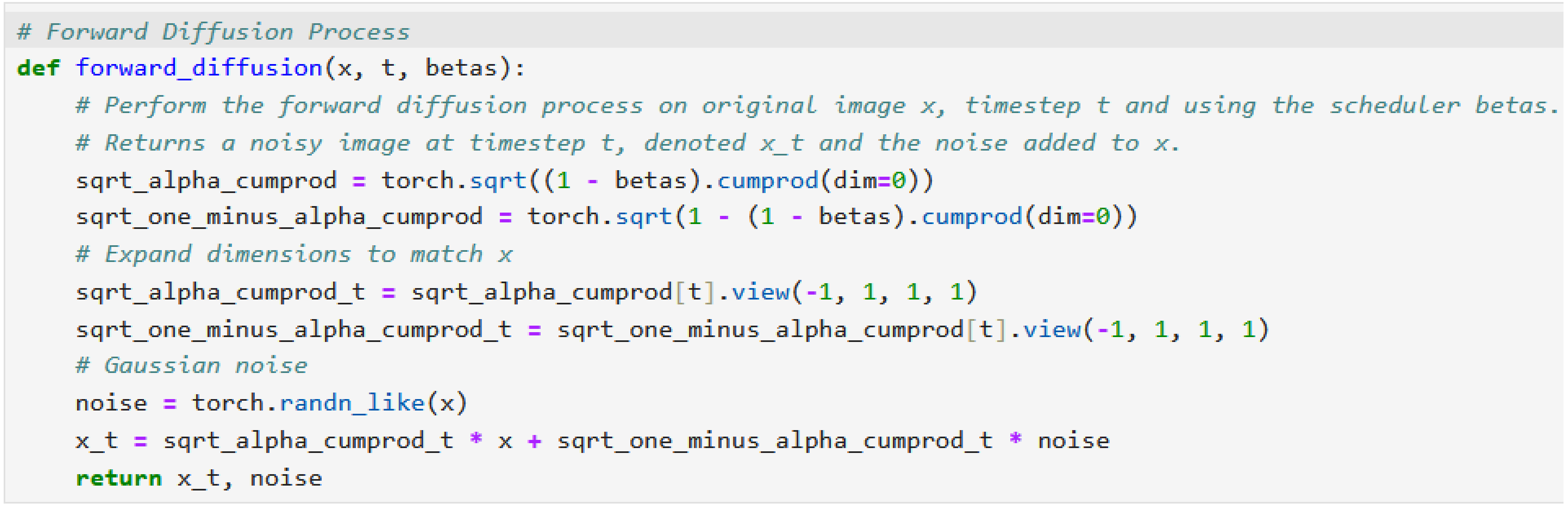
- Where ϵ is a Gaussian noise, with mean zero and variance one, to be added to the image. And α_t is defined as

$$\alpha_t = \prod_{i=0}^{t} (1 - \beta_i)$$

A basic Diffusion Model setup

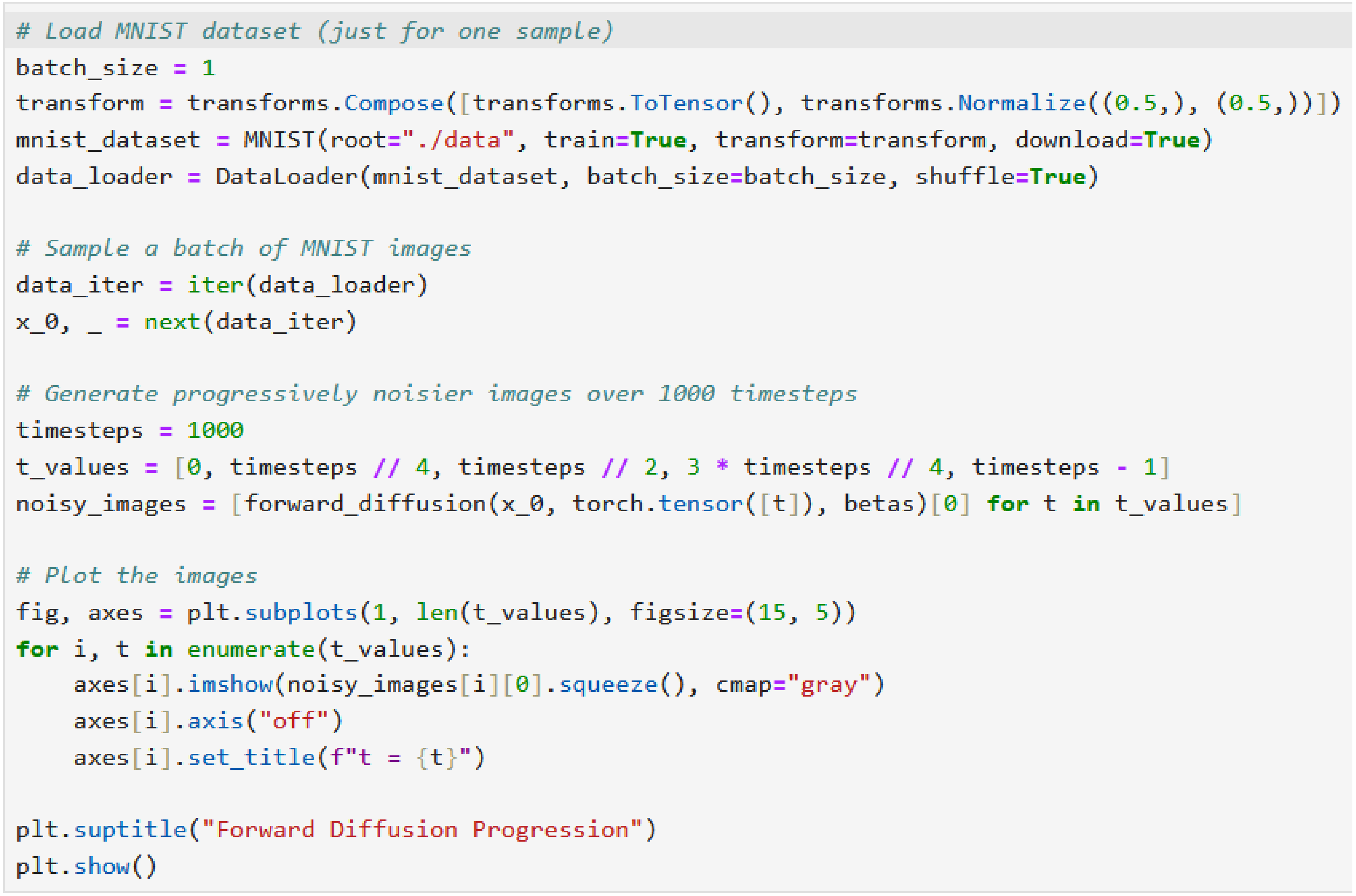
Finally, define a forward diffusion function, whose objective is to progressively noise the MNIST images.

- Apply noise using the noise amplitude $\beta(t)$ on each time step $t \in (0, 1000)$. Return all progressively noised images and noises applied.



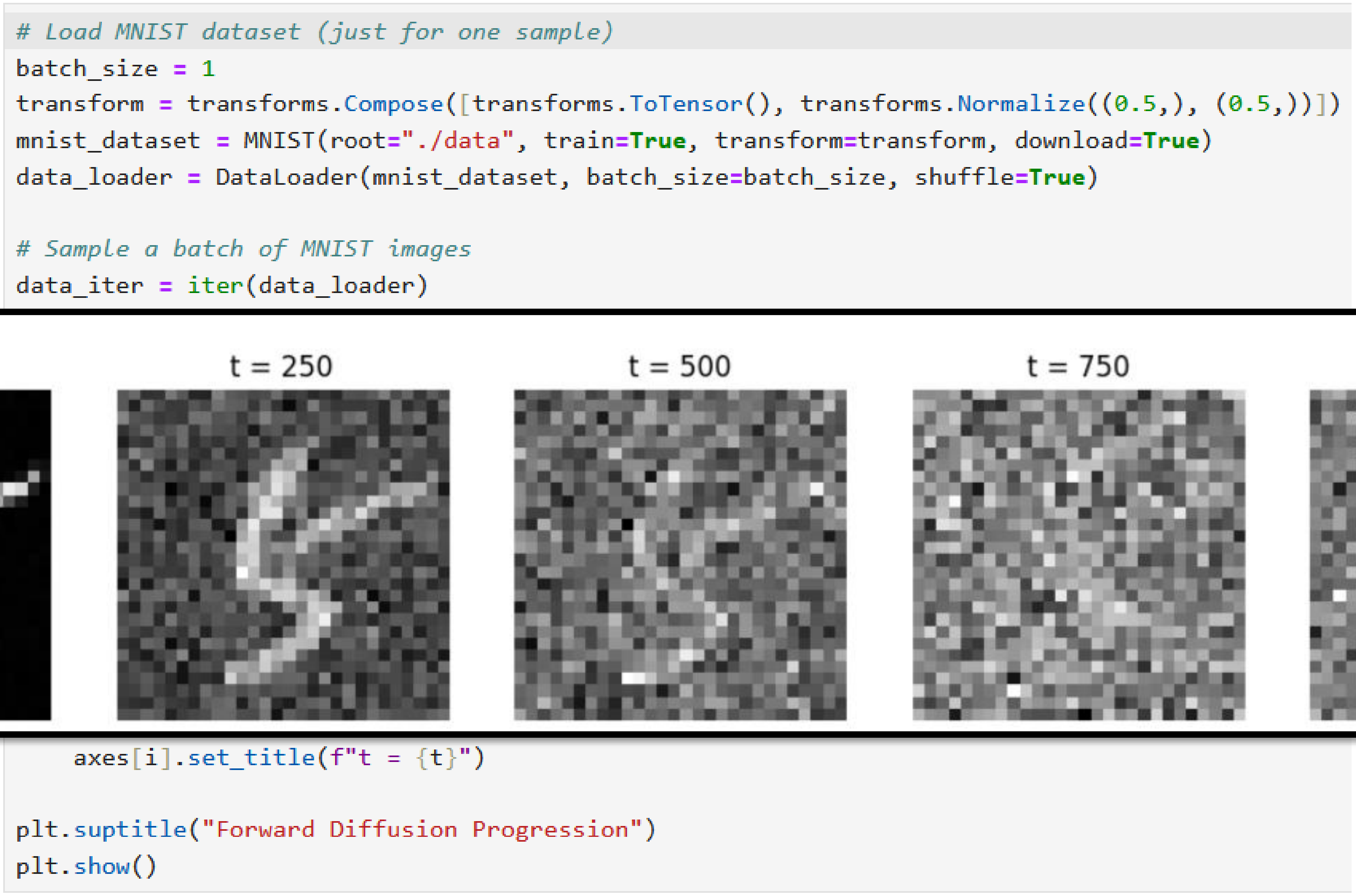
```
# Forward Diffusion Process
def forward_diffusion(x, t, betas):
    # Perform the forward diffusion process on original image x, timestep t and using the scheduler betas.
    # Returns a noisy image at timestep t, denoted x_t and the noise added to x.
    sqrt_alpha_cumprod = torch.sqrt((1 - betas).cumprod(dim=0))
    sqrt_one_minus_alpha_cumprod = torch.sqrt(1 - (1 - betas).cumprod(dim=0))
    # Expand dimensions to match x
    sqrt_alpha_cumprod_t = sqrt_alpha_cumprod[t].view(-1, 1, 1, 1)
    sqrt_one_minus_alpha_cumprod_t = sqrt_one_minus_alpha_cumprod[t].view(-1, 1, 1, 1)
    # Gaussian noise
    noise = torch.randn_like(x)
    x_t = sqrt_alpha_cumprod_t * x + sqrt_one_minus_alpha_cumprod_t * noise
    return x_t, noise
```

Try it on a single MNIST image!

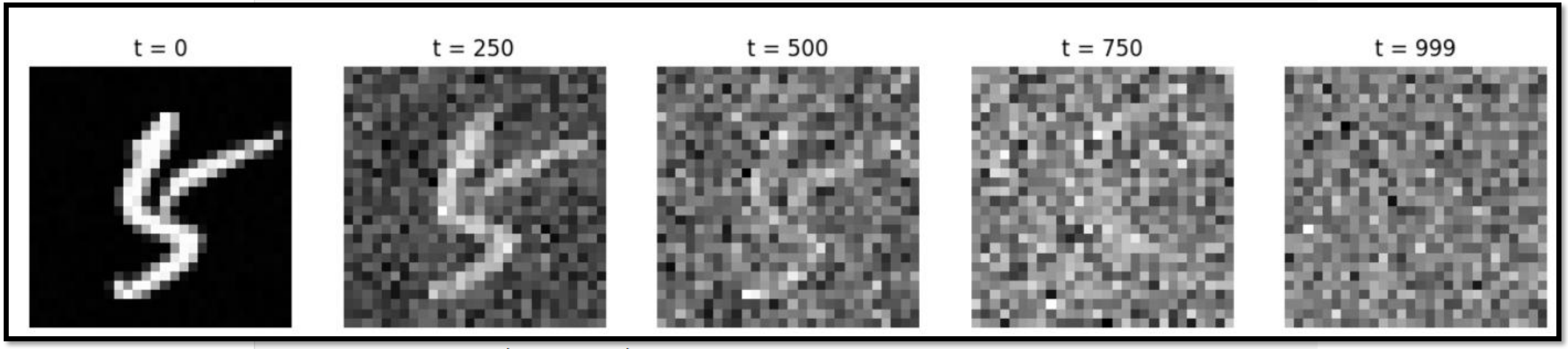


```python
# Load MNIST dataset (just for one sample)
batch_size = 1
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
mnist_dataset = MNIST(root="./data", train=True, transform=transform, download=True)
data_loader = DataLoader(mnist_dataset, batch_size=batch_size, shuffle=True)

# Sample a batch of MNIST images
data_iter = iter(data_loader)
x_0, _ = next(data_iter)

# Generate progressively noisier images over 1000 timesteps
timesteps = 1000
t_values = [0, timesteps // 4, timesteps // 2, 3 * timesteps // 4, timesteps - 1]
noisy_images = [forward_diffusion(x_0, torch.tensor([t]), betas)[0] for t in t_values]

# Plot the images
fig, axes = plt.subplots(1, len(t_values), figsize=(15, 5))
for i, t in enumerate(t_values):
    axes[i].imshow(noisy_images[i][0].squeeze(), cmap="gray")
    axes[i].axis("off")
    axes[i].set_title(f"t = {t}")

plt.suptitle("Forward Diffusion Progression")
plt.show()
```

Try it on a single MNIST image!



```
# Load MNIST dataset (just for one sample)
batch_size = 1
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
mnist_dataset = MNIST(root="./data", train=True, transform=transform, download=True)
data_loader = DataLoader(mnist_dataset, batch_size=batch_size, shuffle=True)

# Sample a batch of MNIST images
data_iter = iter(data_loader)
```



```
    axes[i].set_title(f"t = {t}")

plt.suptitle("Forward Diffusion Progression")
plt.show()
```

A quick word about our UNet model

For the task of denoising a noised image, we will use a **UNet** model.

Definition (UNet models):

A **UNet** is a special type of convolutional neural network (CNN) designed for **image-to-image tasks**, such as image segmentation, denoising, or super-resolution.

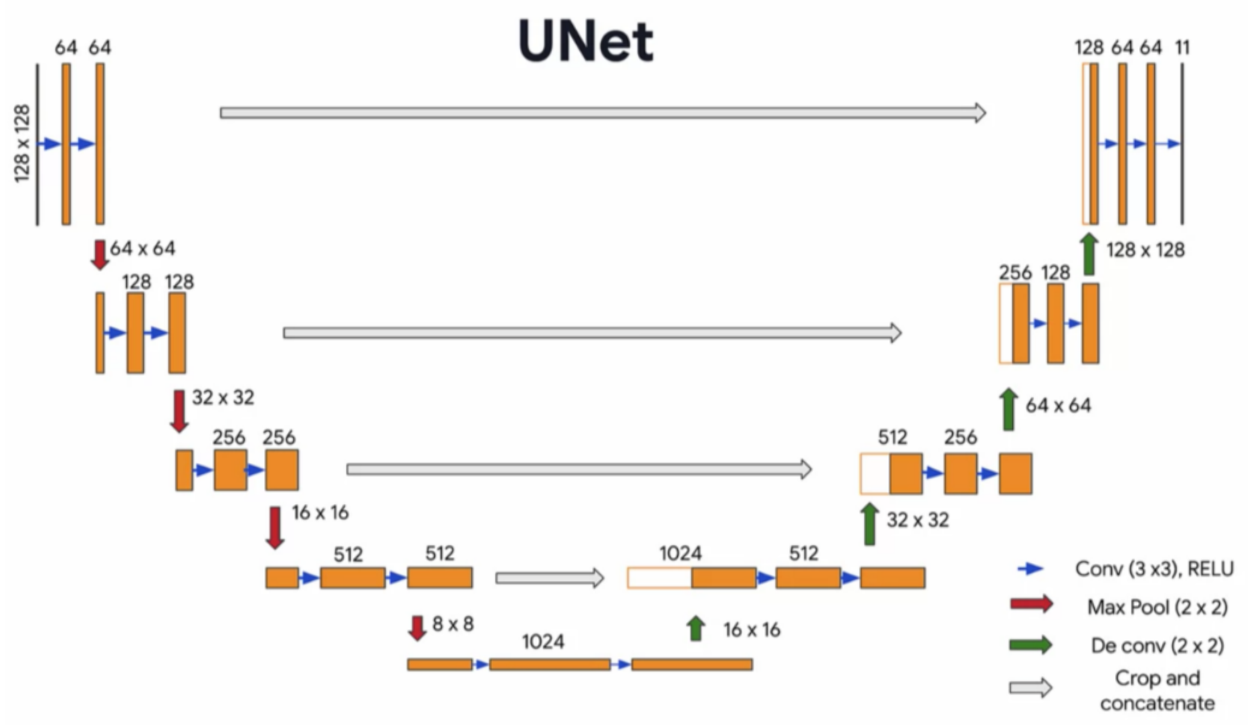


***Figure:** A visual representation of a UNet model.*

A quick word about our UNet model

For the task of denoising a noised image, we will use a **UNet** model.

Definition (UNet models):

A **UNet** is a special type of convolutional neural network (CNN) designed for **image-to-image tasks**, such as image segmentation, denoising, or super-resolution.

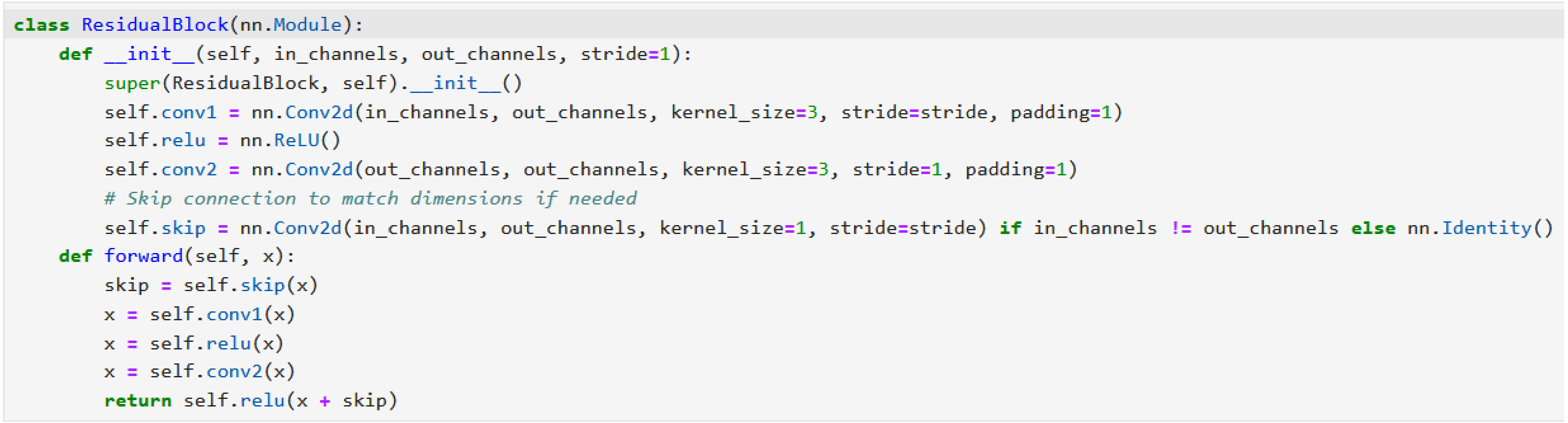
Known for their **U-shaped architecture**, which consists of:

1. **Encoder (Downsampling Path)**: Compresses the input image into a smaller representation.
2. **Bottleneck (Middle Section)**: Captures high-level features.
3. **Decoder (Upsampling Path)**: Reconstructs the image back to its original size while combining detailed features from the encoder.

Implementing a UNet

Start by designing a residual block (works as in W4).

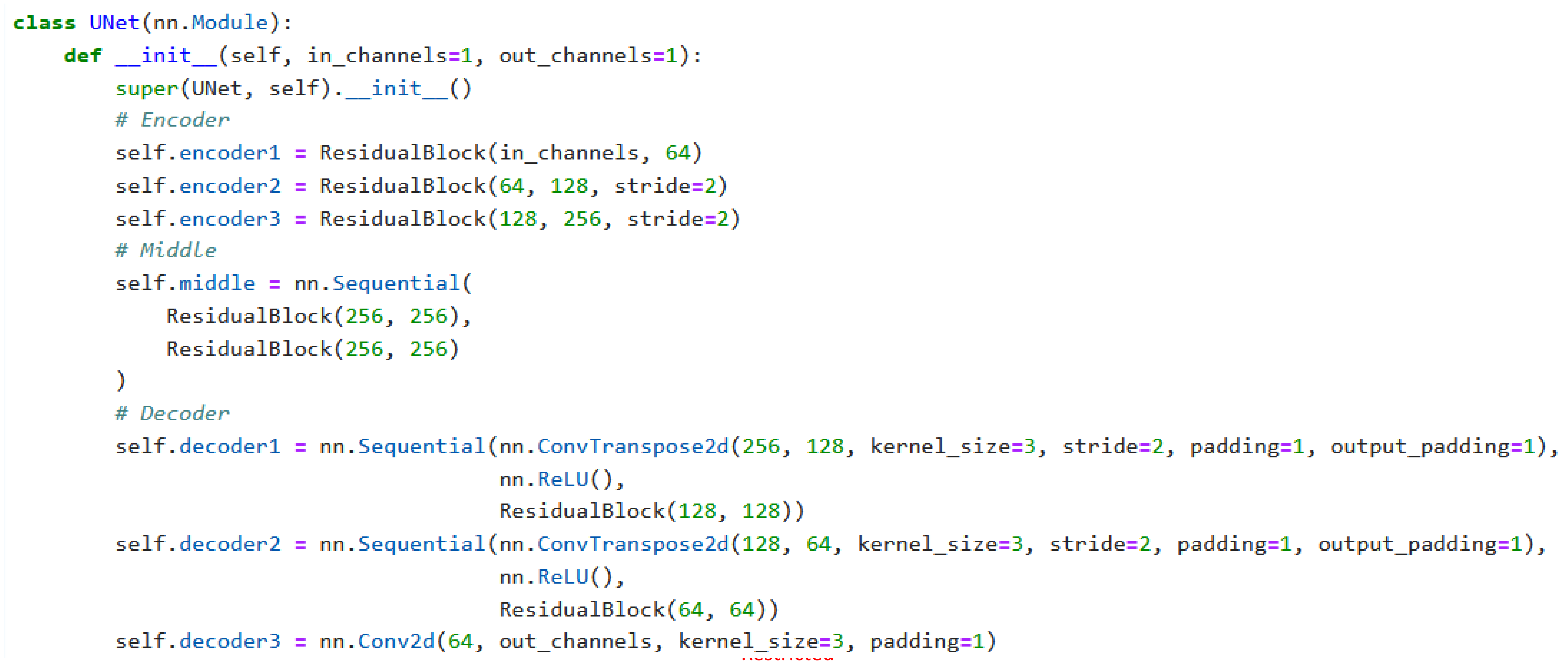
- Magic happens in the final forward operation (x + skip).



```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        # Skip connection to match dimensions if needed
        self.skip = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride) if in_channels != out_channels else nn.Identity()
    def forward(self, x):
        skip = self.skip(x)
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        return self.relu(x + skip)
```

Implementing a UNet

Then, implement the three stage UNet model.



```python
class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1):
        super(UNet, self).__init__()
        # Encoder
        self.encoder1 = ResidualBlock(in_channels, 64)
        self.encoder2 = ResidualBlock(64, 128, stride=2)
        self.encoder3 = ResidualBlock(128, 256, stride=2)
        # Middle
        self.middle = nn.Sequential(
            ResidualBlock(256, 256),
            ResidualBlock(256, 256)
        )
        # Decoder
        self.decoder1 = nn.Sequential(nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
                                      nn.ReLU(),
                                      ResidualBlock(128, 128))
        self.decoder2 = nn.Sequential(nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),
                                      nn.ReLU(),
                                      ResidualBlock(64, 64))
        self.decoder3 = nn.Conv2d(64, out_channels, kernel_size=3, padding=1)
```

Implementing a UNet

Then, implement the three stage UNet model.

```
class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1):
        super(UNet, self).__init__()
        # Encoder
        self.encoder1 = ResidualBlock(in_channels, 64)
        self.encoder2 = ResidualBlock(64, 128, stride=2)
        self.encoder3 = ResidualBlock(128, 256, stride=2)
        # Middle
        self.middle = nn.Sequential(
            ResidualBlock(256, 256),
            ResidualBlock(256, 256)
        )
        # Decoder
        self.decoder1 = nn.Sequential(nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
                                      nn.ReLU(),
                                      ResidualBlock(128, 128))
        self.decoder2 = nn.Sequential(nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),
                                      nn.ReLU(),
                                      ResidualBlock(64, 64))
        self.decoder3 = nn.Conv2d(64, out_channels, kernel_size=3, padding=1)
```

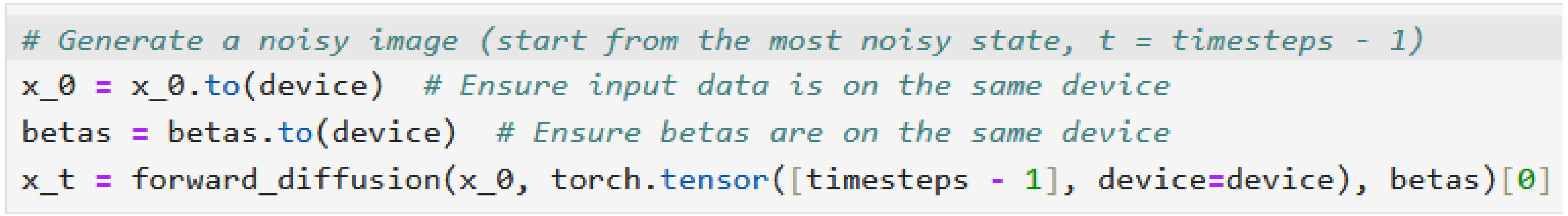
Predicting noise and denoising

Important: The purpose of the UNet model is to predict the noise added to the image, not to denoise the image directly!

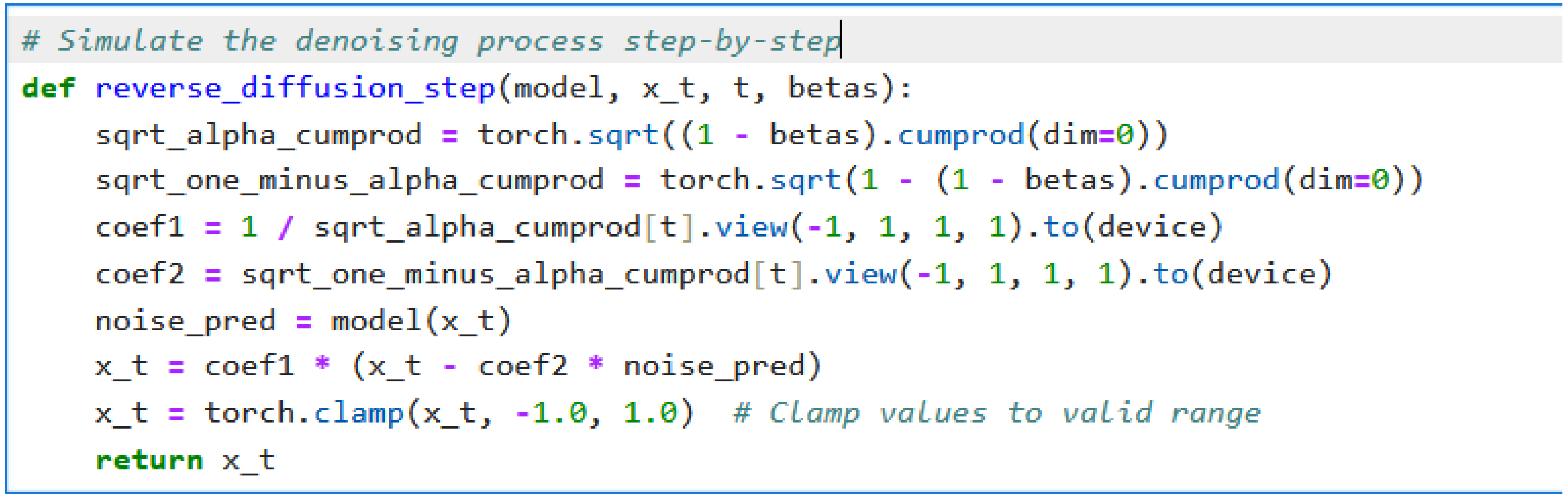
- Predicting the denoised image directly can lead to unstable training, especially when the image is heavily noised (i.e. at high timesteps).
- Noise prediction is more reliable because noise is sampled from a known Gaussian distribution, making it easier for the model to learn.
- Predicting the denoised image directly would require the model to understand the noise distribution and the underlying image structure at once. By focusing only on the noise, the model simplifies its task.
- Using the predicted noise, we calculate a "less noisy" version of the image, which is roughly equivalent.

Predicting noise and denoising

- Let us then define a denoising function that uses the model and progressively denoises the image using the model predictions.



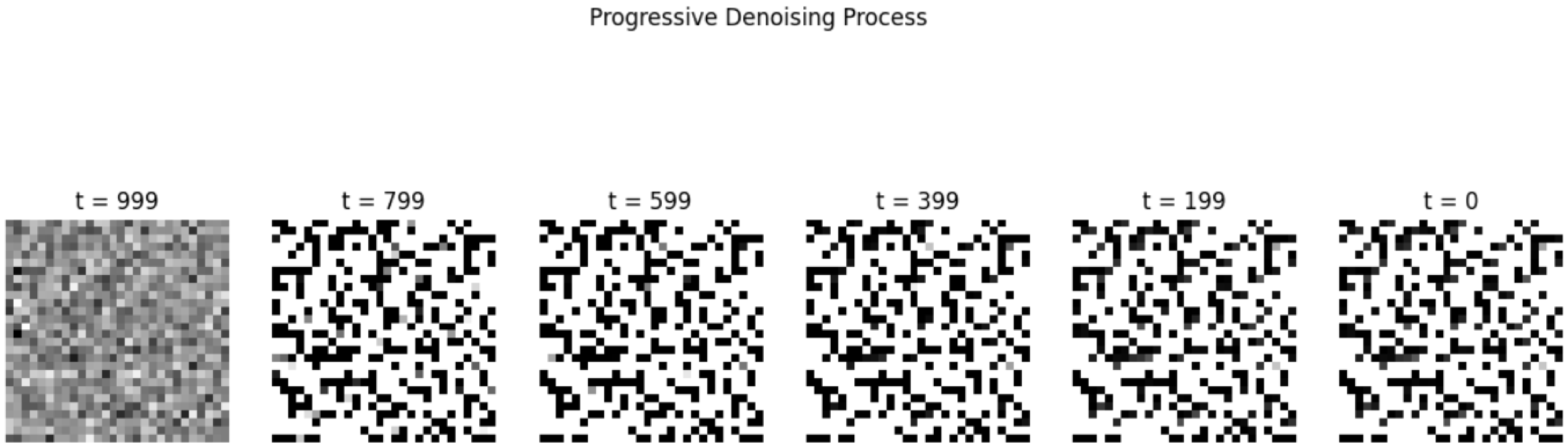
```python
# Generate a noisy image (start from the most noisy state, t = timesteps - 1)
x_0 = x_0.to(device)  # Ensure input data is on the same device
betas = betas.to(device)  # Ensure betas are on the same device
x_t = forward_diffusion(x_0, torch.tensor([timesteps - 1], device=device), betas)[0]
```



```python
# Simulate the denoising process step-by-step
def reverse_diffusion_step(model, x_t, t, betas):
    sqrt_alpha_cumprod = torch.sqrt((1 - betas).cumprod(dim=0))
    sqrt_one_minus_alpha_cumprod = torch.sqrt(1 - (1 - betas).cumprod(dim=0))
    coef1 = 1 / sqrt_alpha_cumprod[t].view(-1, 1, 1, 1).to(device)
    coef2 = sqrt_one_minus_alpha_cumprod[t].view(-1, 1, 1, 1).to(device)
    noise_pred = model(x_t)
    x_t = coef1 * (x_t - coef2 * noise_pred)
    x_t = torch.clamp(x_t, -1.0, 1.0)  # Clamp values to valid range
    return x_t
```

Predicting noise and denoising

- At the moment, the model has not been trained, so denoising does not work well (which is expected).



Training the model

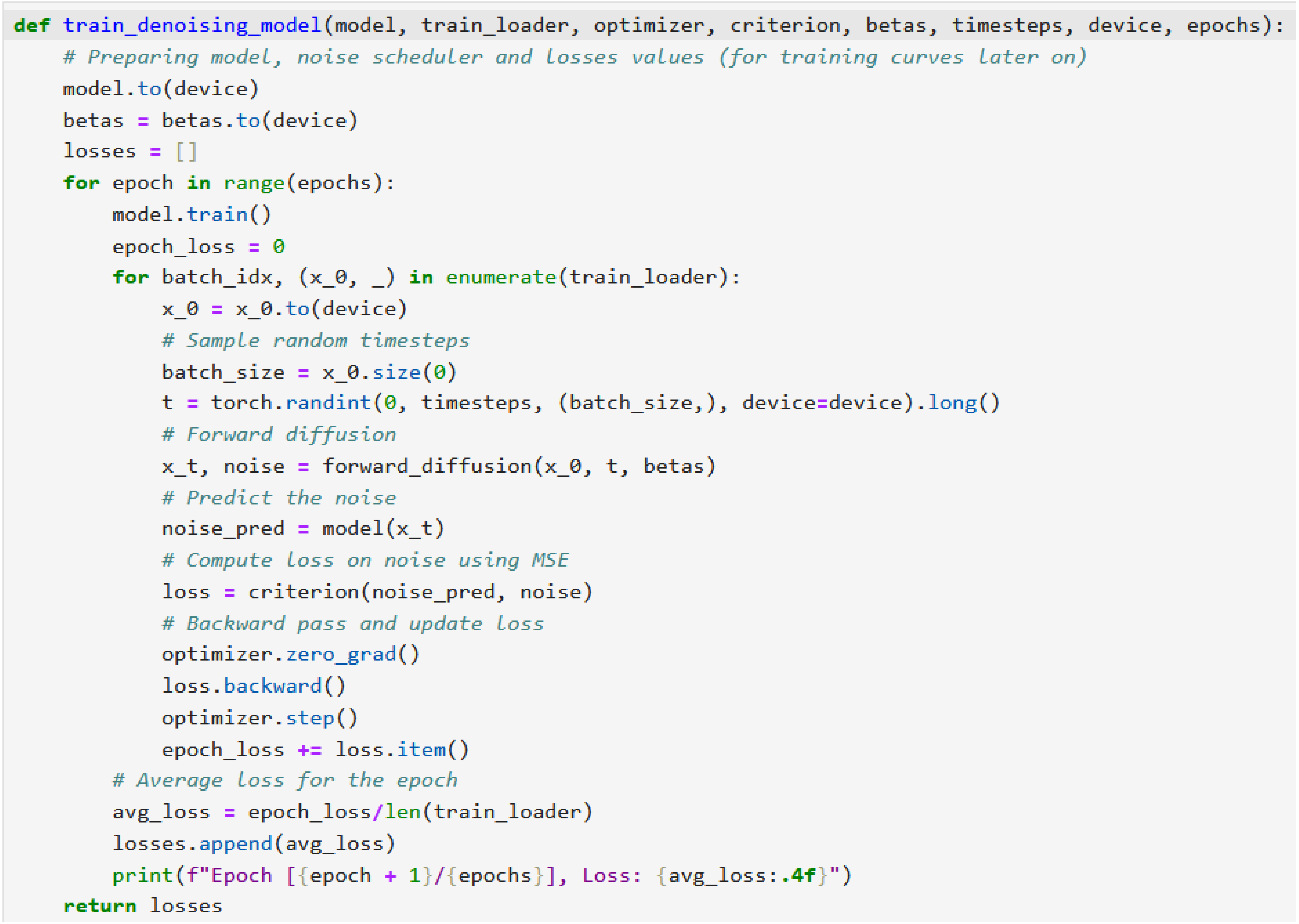
After initializing everything, train the model by

- Noising an image drawn from MNIST over the timesteps (forward diffusion) and keep track of the noise values.
- Have the model predict the noise added to each timestep.
- Use MSE to compare the predicted noise and the “real” noise.
- Backprop, rinse and repeat until convergence.
- Using Adam and default learning rate values for simplicity.
- Keep track of loss values for training curves.

Tra

After

- No
  diff
- Hav
- Use
- Bac
- Usi
- Kee

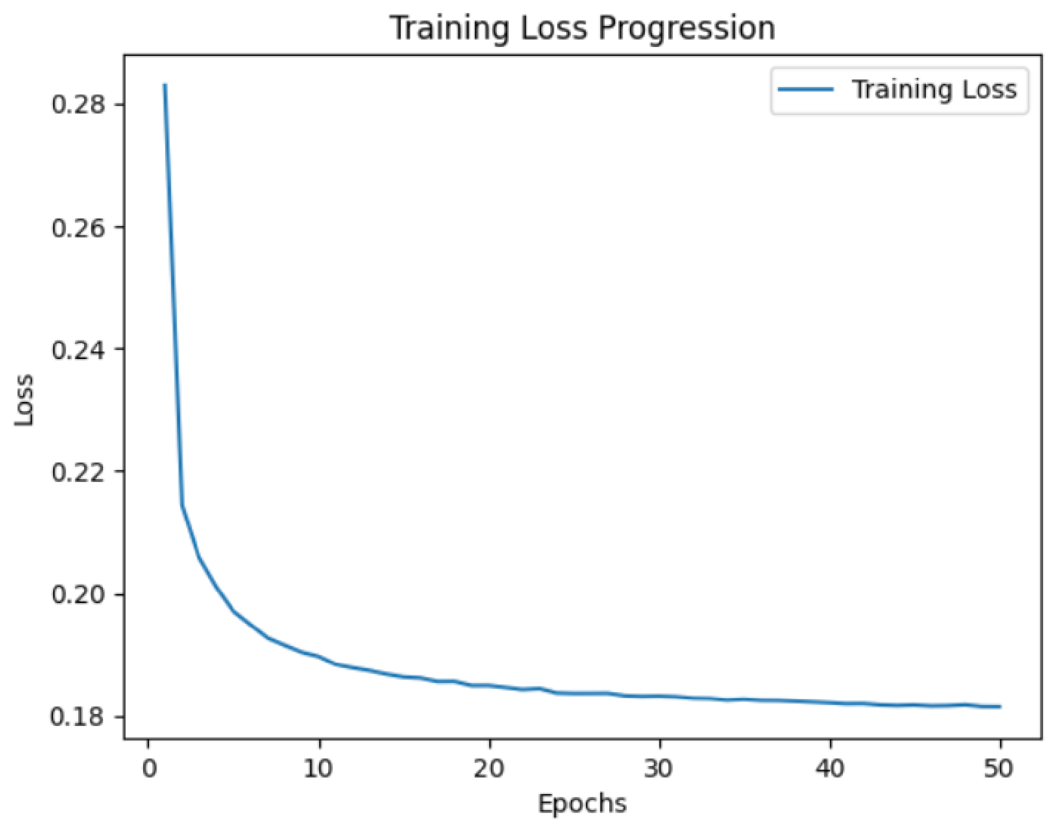


```
def train_denoising_model(model, train_loader, optimizer, criterion, betas, timesteps, device, epochs):
    # Preparing model, noise scheduler and losses values (for training curves later on)
    model.to(device)
    betas = betas.to(device)
    losses = []
    for epoch in range(epochs):
        model.train()
        epoch_loss = 0
        for batch_idx, (x_0, _) in enumerate(train_loader):
            x_0 = x_0.to(device)
            # Sample random timesteps
            batch_size = x_0.size(0)
            t = torch.randint(0, timesteps, (batch_size,), device=device).long()
            # Forward diffusion
            x_t, noise = forward_diffusion(x_0, t, betas)
            # Predict the noise
            noise_pred = model(x_t)
            # Compute loss on noise using MSE
            loss = criterion(noise_pred, noise)
            # Backward pass and update loss
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        # Average loss for the epoch
        avg_loss = epoch_loss/len(train_loader)
        losses.append(avg_loss)
        print(f"Epoch [{epoch + 1}/{epochs}], Loss: {avg_loss:.4f}")
    return losses
```

It trains?

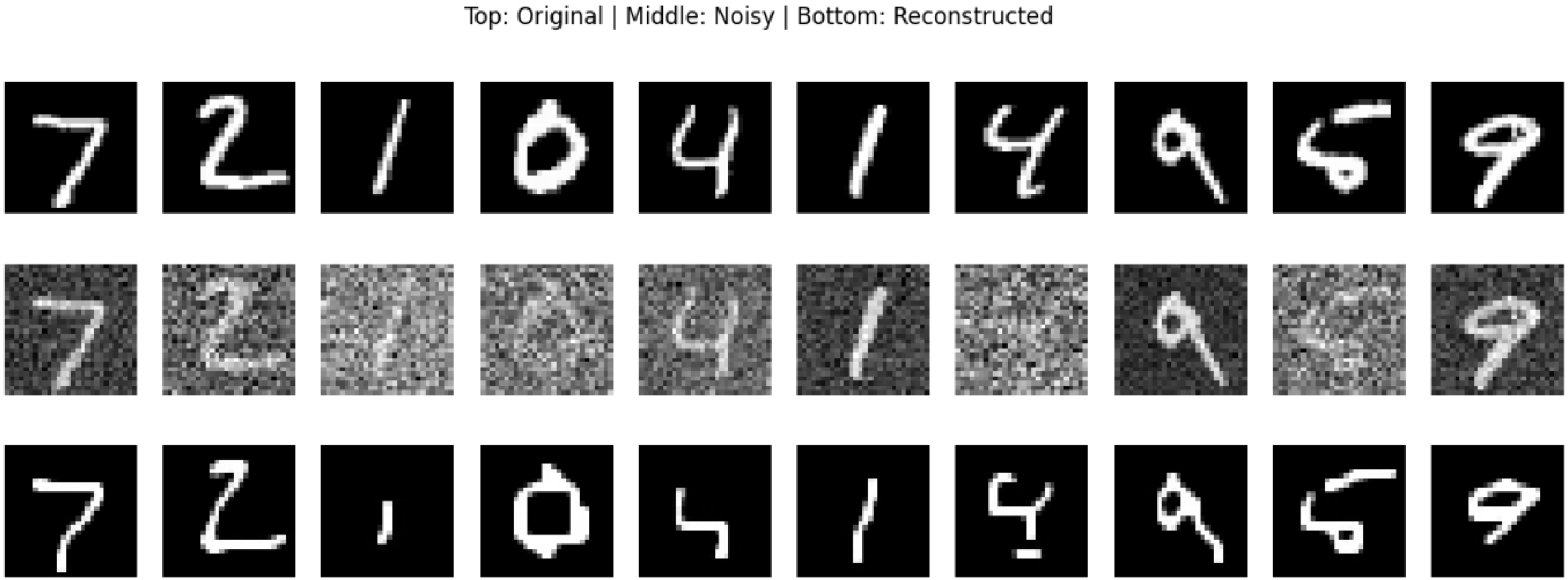
- It seems to train, although it could benefit from our arsenal of training techniques (LR decay/scheduling, using validation, etc.)

```
Epoch [1/50], Loss: 0.2830
Epoch [2/50], Loss: 0.2144
Epoch [3/50], Loss: 0.2058
Epoch [4/50], Loss: 0.2010
Epoch [5/50], Loss: 0.1971
Epoch [6/50], Loss: 0.1949
Epoch [7/50], Loss: 0.1927
Epoch [8/50], Loss: 0.1915
Epoch [9/50], Loss: 0.1904
Epoch [10/50], Loss: 0.1897
Epoch [11/50], Loss: 0.1884
Epoch [12/50], Loss: 0.1879
Epoch [13/50], Loss: 0.1874
Epoch [14/50], Loss: 0.1868
Epoch [15/50], Loss: 0.1863
Epoch [16/50], Loss: 0.1862
Epoch [17/50], Loss: 0.1856
Epoch [18/50], Loss: 0.1856
Epoch [19/50], Loss: 0.1849
Epoch [20/50], Loss: 0.1850
Epoch [21/50], Loss: 0.1846
Epoch [22/50], Loss: 0.1843
```



Let us visualize the results!

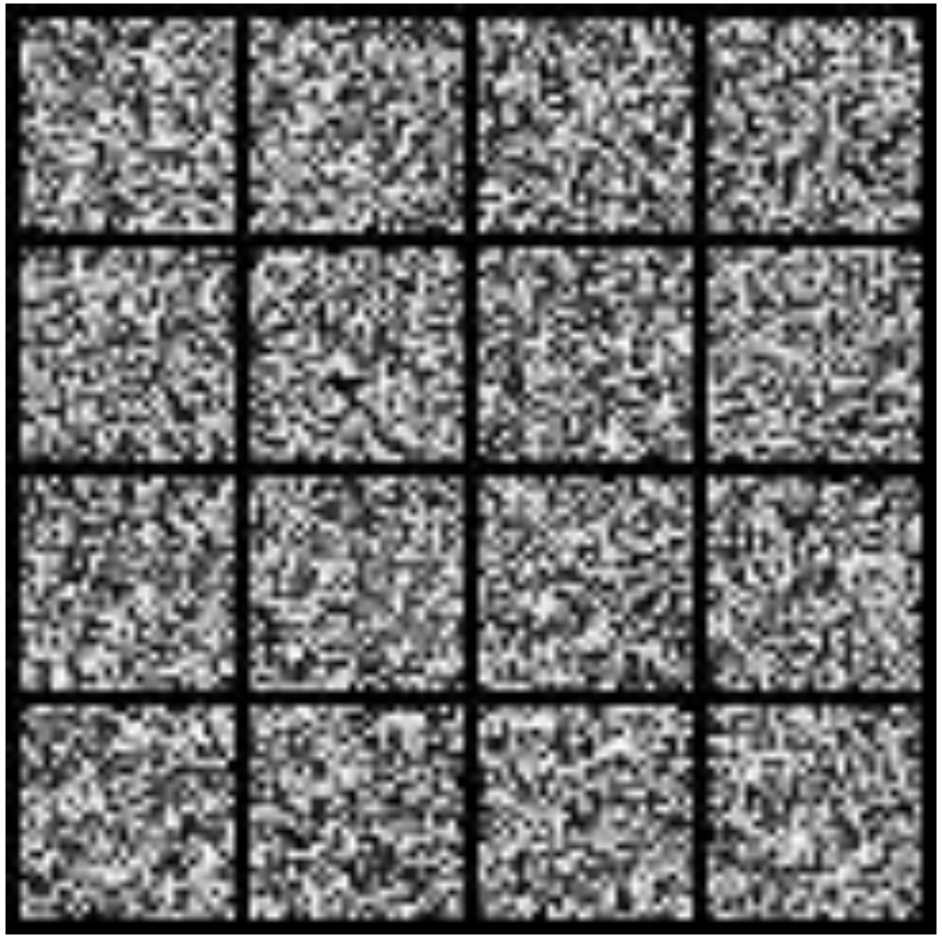
After training, we obtain a model that is “capable” of denoising images.



In practice, however…

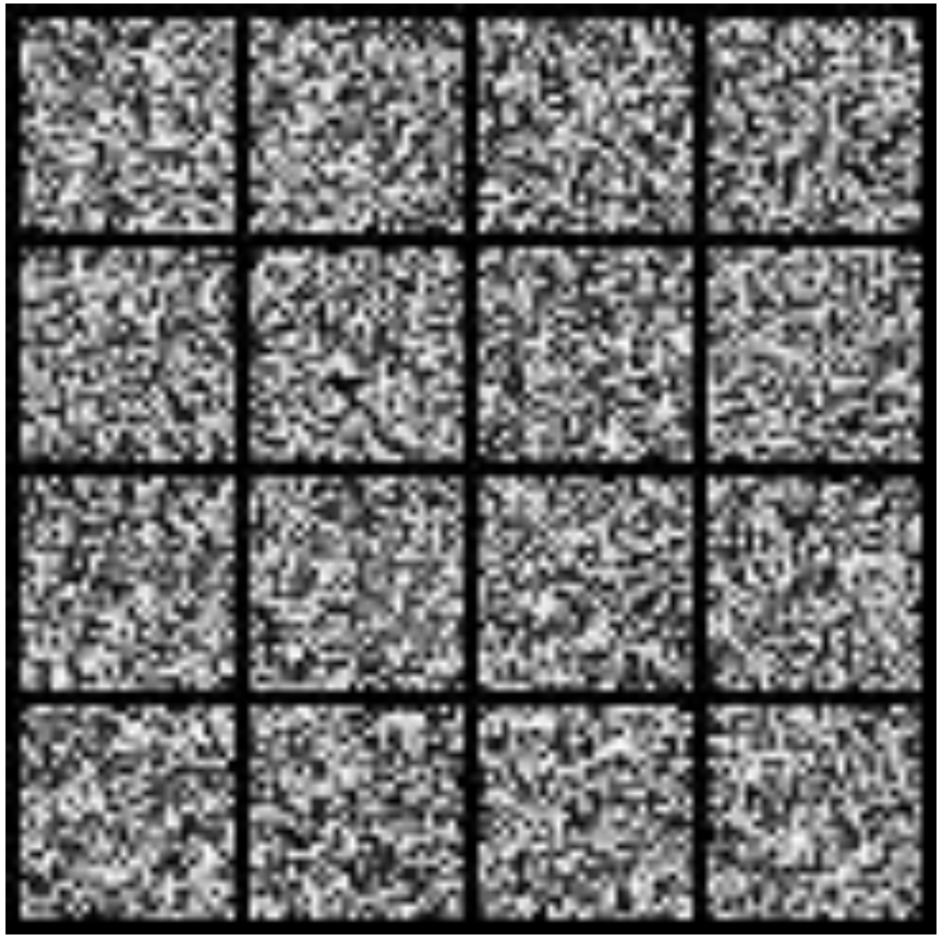
In practice, need more iterations and more steps of noising and denoising, in the thousands or so.

- Our Notebook 3 demo runs on only $T = 1000$ steps of noising/denoising and would probably work better with a larger T value and smaller noising at each step…
- And maybe a larger model…
- It is expensive to train (!), but you get the idea…



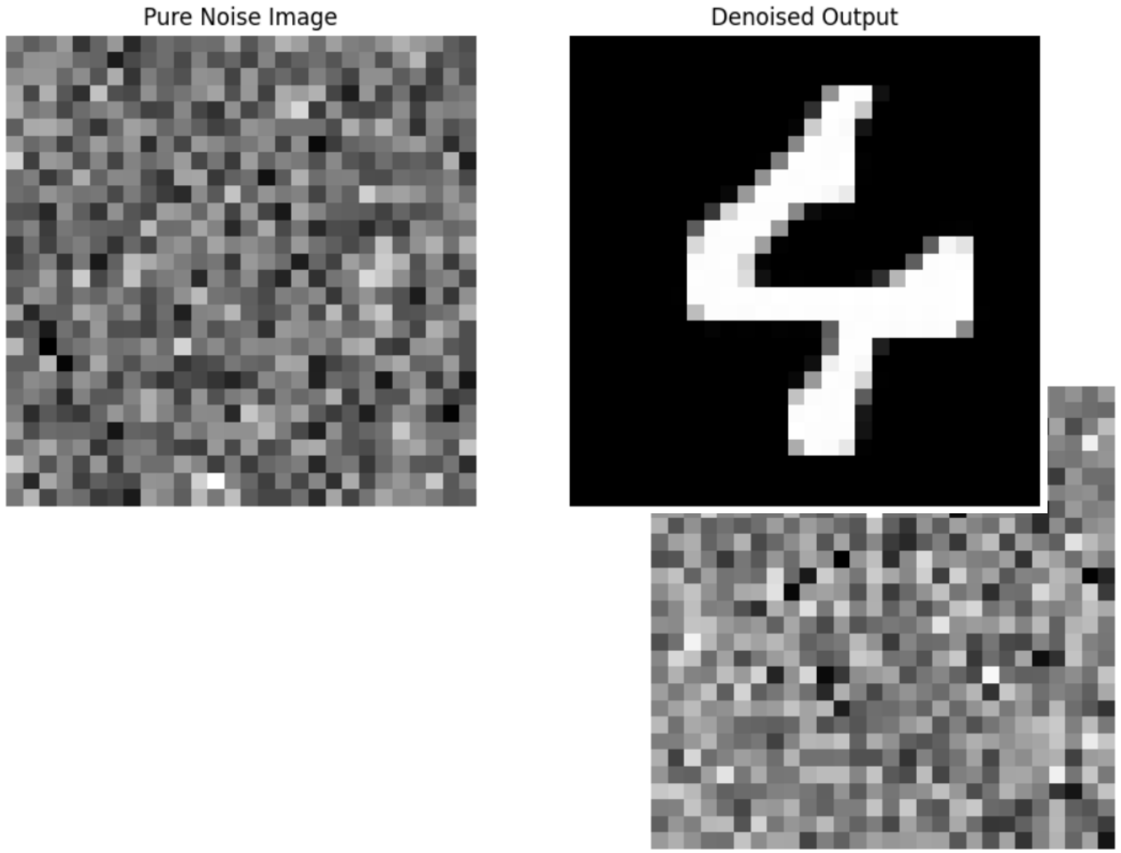
In practice, however...

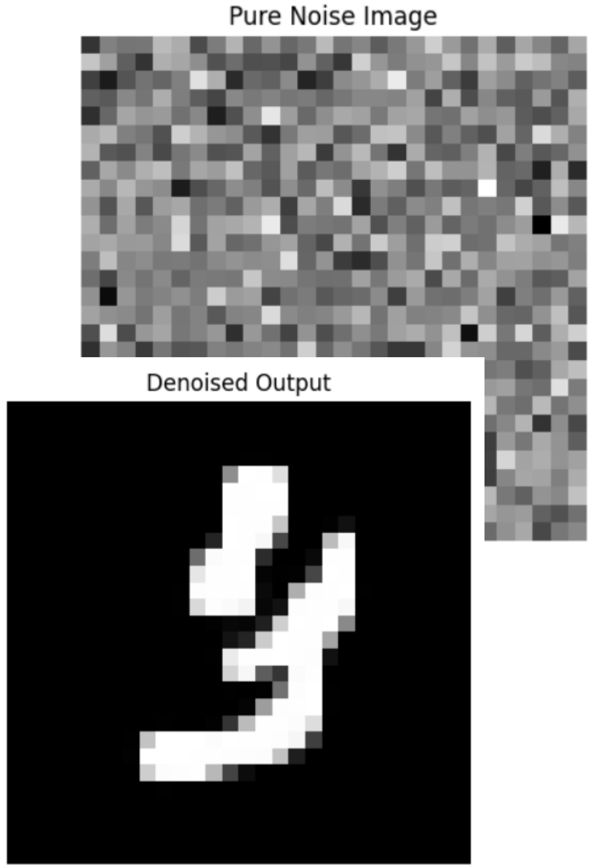
- We were somewhat able to train a model to denoise and eventually obtain a convincing MNIST picture from something that was close to pure noise!
- Another possible approach to generate images out of thin air?
- **Keep in mind:** all advanced ideas from previous weeks are open! (using advanced layers, making it conditional like conditional GANs, cyclic, progressive, etc.)

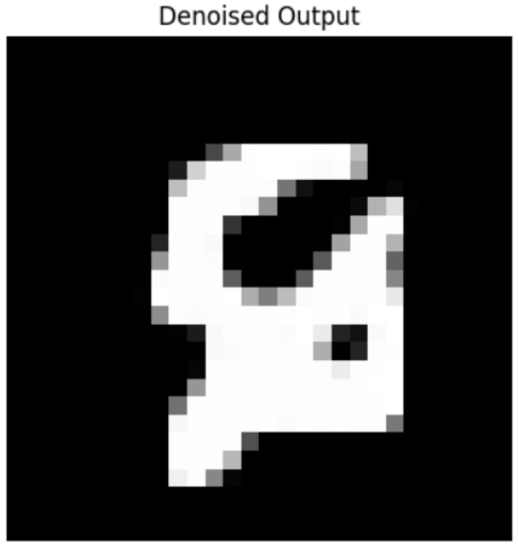


In practice, however…

- What happens if we use the reverse diffusion process and our model on an image that is pure noise? Do we obtain an MNIST-like sample?



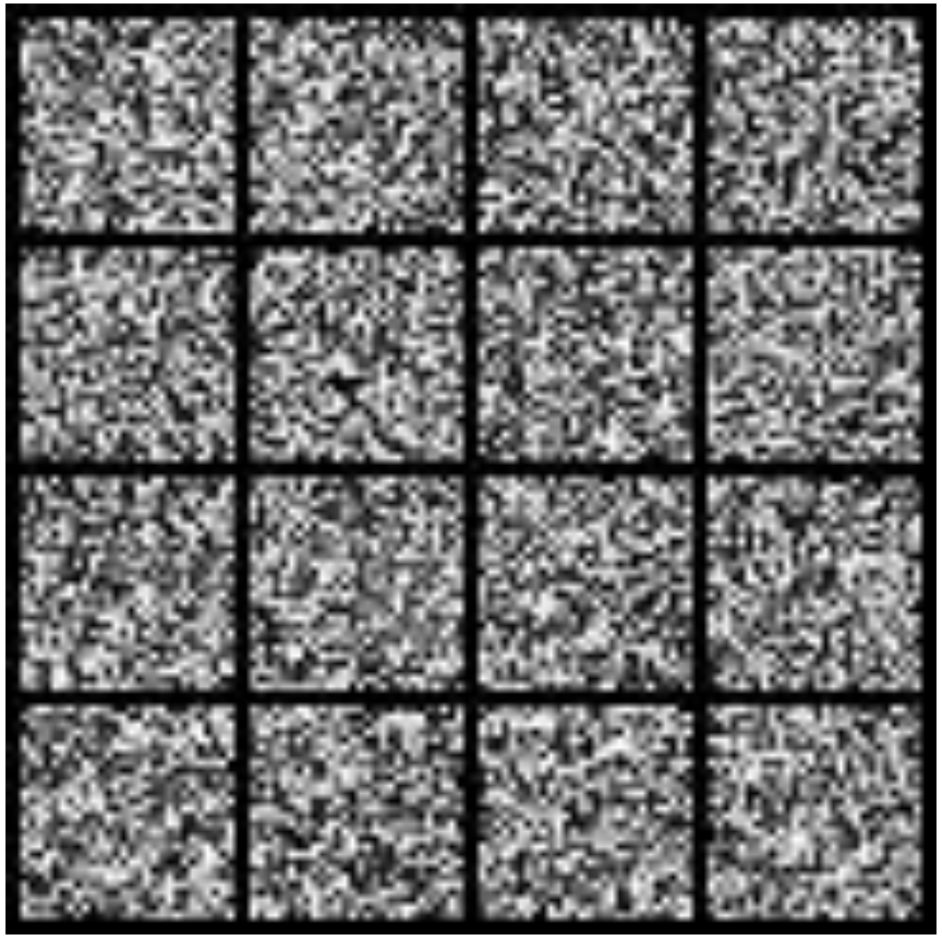




Meh… Sometimes yes, sometimes no.

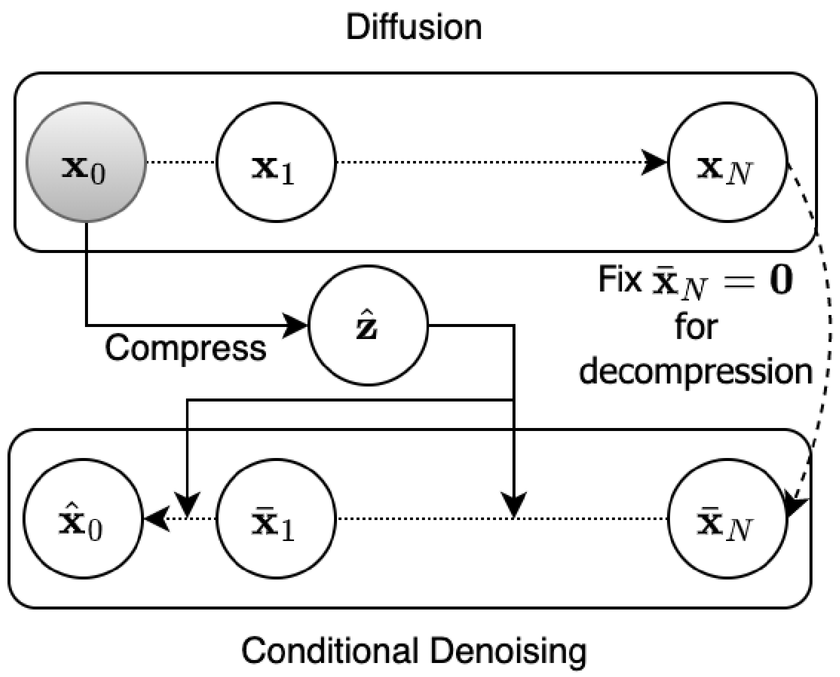
In practice, however…

- Clearly, we are not there yet!
- There is a need for more advanced practices to training diffusion models efficiently.
- Learn more in the Computer Vision course on Term 7!
- (Or if you already want a pre-trained implementation, readily available…) https://github.com/VSehwag/minimal-diffusion



Making it conditional

- If you train with a dataset of captioned images, the extra label could be used during the denoising part, to help the denoising model figure out what was the original image about.
- Encode these captions and use them during training of the denoising model in place of t.
- Helps to direct the denoising in right direction!



The man at bat readies to swing at the pitch while the umpire looks on.

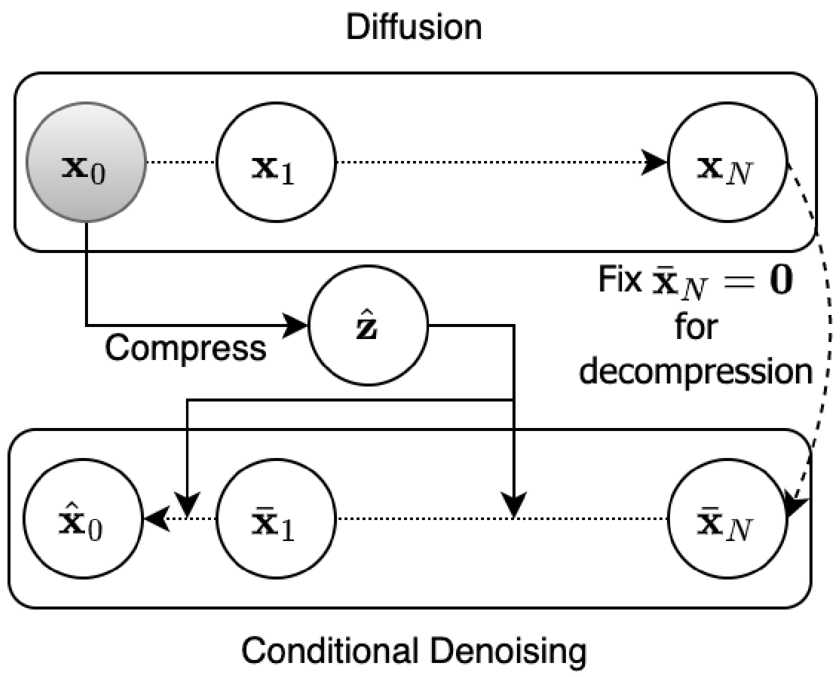
A large bus sitting next to a very tall building.

A horse carrying a large load of hay and two people sitting on it.

Bunk bed with a narrow shelf sitting underneath it.

Making it conditional

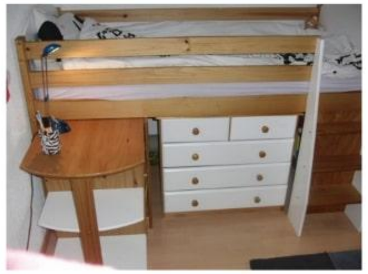
- Label $\hat{z}$ could be the class of the MNSIT image, or the embedding of a text caption.
- During testing, use pure noise as x_N and ask the user for a caption of their choice that will be compressed as $\hat{z}$.
- Use your denoiser many times in a row and obtain an image matching the caption $\hat{z}$ starting from pure noise?!



The man at bat readies to swing at the pitch while the umpire looks on.

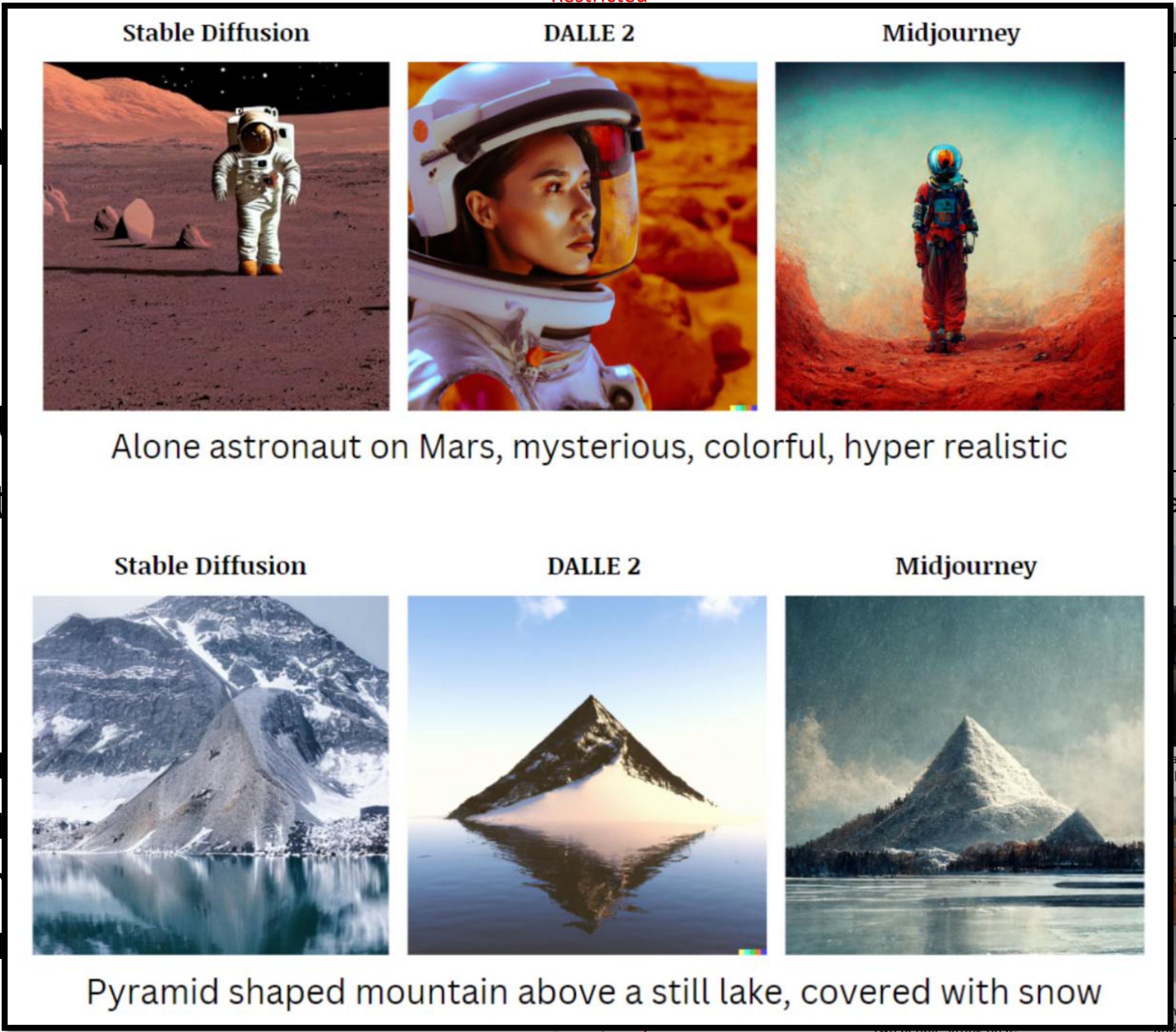
A large bus sitting next to a very tall building.

A horse carrying a large load of hay and two people sitting on it.


Bunk bed with a narrow shelf sitting underneath it.

Makin

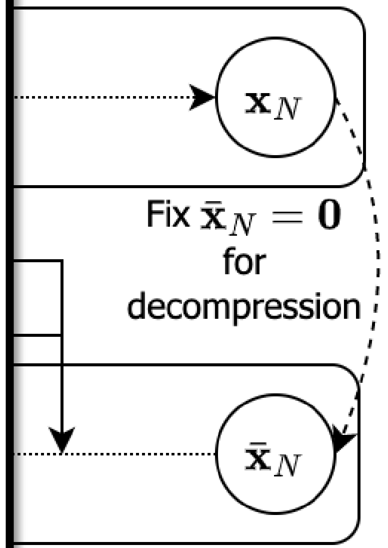
- Label $\hat{z}$
  MNSIT i
  of a text
- During t
  x_N and
  of their
  compre
- Use you
  a row a
  matchin
  from pu



Alone astronaut on Mars, mysterious, colorful, hyper realistic

Pyramid shaped mountain above a still lake, covered with snow

n



enoising

arge bus sitting next to a very tall
lding.

k bed with a narrow shelf sitting
underneath it.

two people sitting on it.

Diffusion: a good entry point to continue your study of generative models

What is “stable” diffusion then? The "stable" part of **stable diffusion** comes from optimizations that make the model more efficient and stable, particularly for large-scale text-to-image generation tasks.

- While it would be impossible to explain the issues and stable diffusion in two minutes, know that it achieves this using **latent diffusion**, a key innovation.
- **Normal Diffusion Models:** Operate directly in pixel space, which can be computationally expensive and slow for high-resolution images.
- **Stable Diffusion:** Operates in a latent space (a compressed representation of the image), significantly reducing computational costs and making the process faster and more efficient.

Diffusion: a good entry point to continue your study of generative models

What is "stable" diffusion then? The "stable" part of **stable diffusion** comes from optimizations that make the model more efficient and stable, particularly for large-scale text-to-image generation tasks.

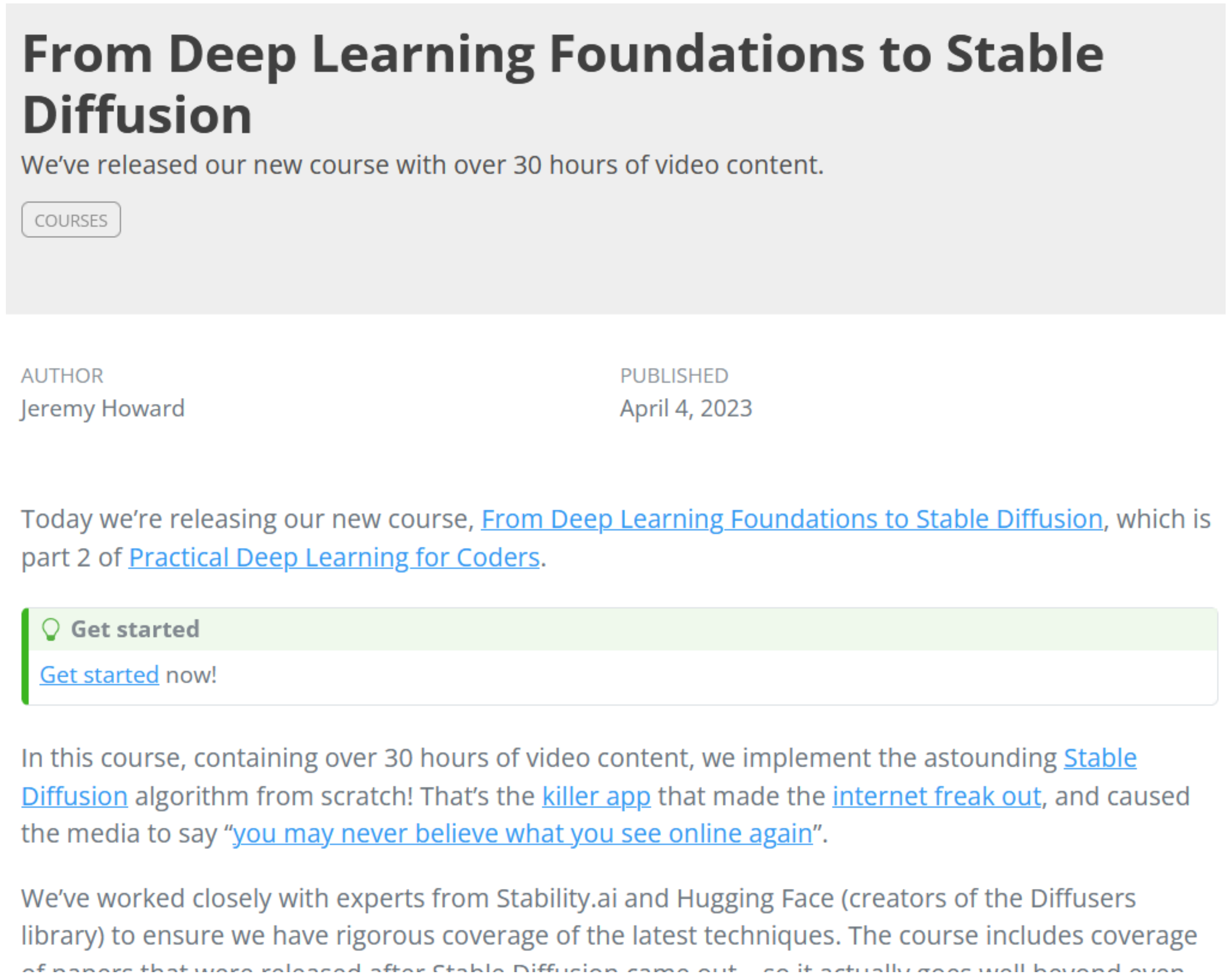
- While it would be impossible to explain the issues and stable diffusion in two minutes, know that it achieves this using **latent diffusion**, a key innovation.
- **Normal Diffusion Models:** Operate directly in pixel space, which can be computationally expensive and slow for high-resolution images.
- **Stable Diffusion:** Operates in a latent space (a compressed representation of the image), significantly reducing computational costs and making the process faster and more efficient.

Something for later in a more advanced Computer Vision course?

Diffusion: a good entry point to continue your study of generative models

- If you want online resources, I would warmly recommend this course from **Jeremy Howard (Fast.ai)**, one of the most important AI teachers out there, is probably very nice to take…

https://www.fast.ai/posts/part2-2023.html



From Deep Learning Foundations to Stable Diffusion

We've released our new course with over 30 hours of video content.

COURSES

AUTHOR
Jeremy Howard

PUBLISHED
April 4, 2023

Today we're releasing our new course, From Deep Learning Foundations to Stable Diffusion, which is part 2 of Practical Deep Learning for Coders.

Get started

Get started now!

In this course, containing over 30 hours of video content, we implement the astounding Stable Diffusion algorithm from scratch! That's the killer app that made the internet freak out, and caused the media to say "you may never believe what you see online again".

We've worked closely with experts from Stability.ai and Hugging Face (creators of the Diffusers library) to ensure we have rigorous coverage of the latest techniques. The course includes coverage of papers that were released after Stable Diffusion came out – so it actually goes well beyond even