50.039 Theory and Practice of Deep Learning

W12-S2 About Diffusion Models

Matthieu De Mari

About this week (Week 12, an informative lecture about diffusion models)

1. What is a **Stochastic Differential Equation**?
2. What are **typical uses** of SDEs?
3. Can I **noise images progressively** using SDEs?
4. Can you train a **denoising autoencoder model that removes SDE noise** from images?
5. What is the **diffusion process** and how to use it to train **diffusion models**?
6. What are the architectures of DALL-E and Midjourney models?

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**).

Stochastic Differential Equations (SDEs)

Definition (Stochastic Differential Equations):

Stochastic Differential Equations (SDEs) are **differential equations where one or more of the terms include a random component, usually in the form of a noise term.**

The randomness is often modelled using a Wiener process or Brownian motion, which is a continuous-time stochastic process characterized by a mean, a variance, and independent increments.

Diffusion models as SDEs

Mathematically, a **diffusion model can be represented as an SDE**:

$$dx(t) = a(x(t), t)dt + b(x(t), t)dW(t) \quad and \quad x(t = 0) = x_0$$

Where,

- $x(t)$ represents the **state of the system at time t**,
- $a(x(t), t)$ is the **deterministic drift term**,
- $b(x(t), t)$ is the **stochastic diffusion term**,
- and $W(t)$ is the **Wiener process**.

The drift term governs the deterministic part of the system's dynamics, while the diffusion term introduces randomness into the system's evolution.

A first SDE: the Brownian motion

As a first example, let us consider one of the simplest SDEs:

$$dx(t) = dW(t) \quad and \quad x(0) = 0$$

This SDE typically defines a random walk starting from 0 and the noise process is simply zero mean and variance $\sqrt{timestep}$.

Implemented as shown in Notebook 1 and on the right.

```python
# Parameters
num_simulations = 5
total_time = 10.0
num_steps = 1000
dt = total_time/num_steps
```

```python
# Initialize the state variable as 0, at t = 0
x = np.zeros((num_simulations, num_steps))
x[:, 0] = 0.0

# Simulate Brownian motion
for t in range(1, num_steps):
    # Definition random motion using a Normal distribution with
    # Zero mean and variance sqrt(dt)
    dW = np.random.normal(0, np.sqrt(dt), size = num_simulations)
    x[:, t] = x[:, t-1] + dW
```

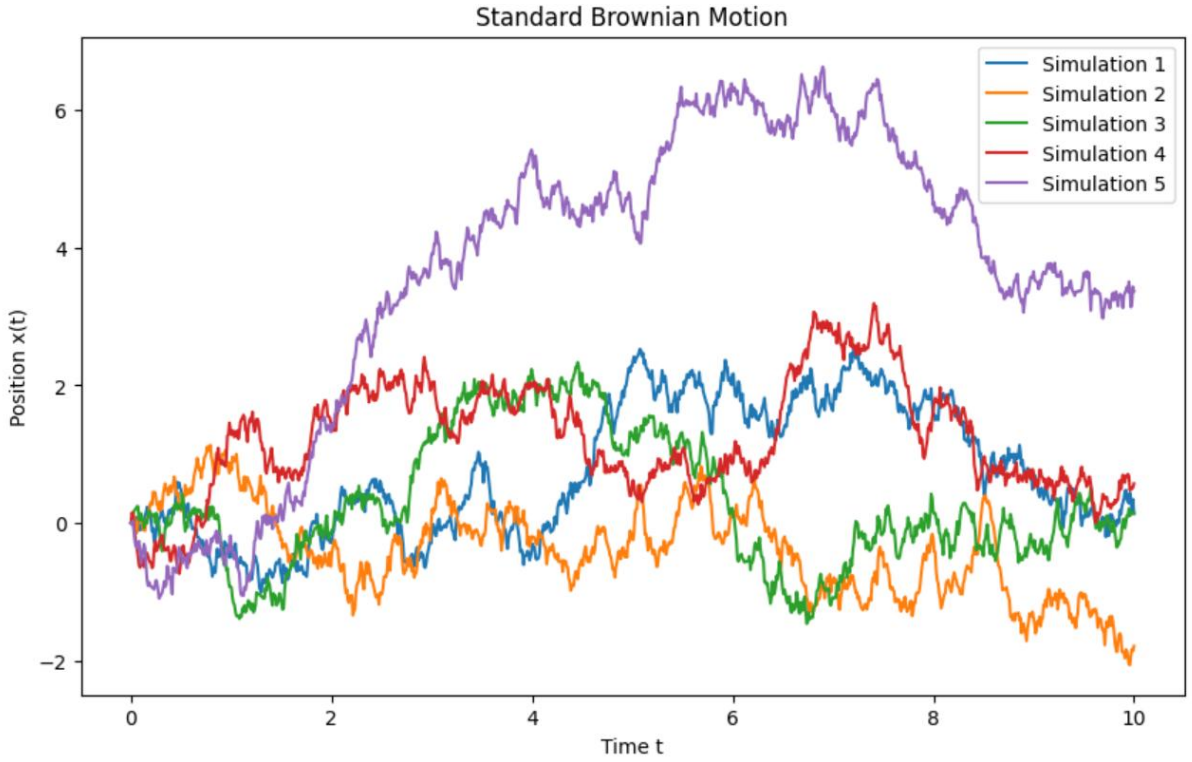
A first SDE: the Brownian motion

As a first example, let us consider one of the simplest SDEs:

$$dx(t) = dW(t) \quad and \quad x(0) = 0$$

This SDE typically defines a random walk starting from 0 and the noise process is simply zero mean and variance $\sqrt{timestep}$.

Implemented as shown in Notebook 1 and on the right.



A second SDE with drift

As a second example, let us consider an SDE with drift:

$$dx(t) = 0.1x(t)dt + dW(t) \quad and \quad x(0) = 10$$

The drift function here is $0.1x(t)$, adding deterministically $\sim 10\%$ to the current value of $x(t)$ for each unit of time t.

A second SDE with drift

This second SDE is typically implemented as shown.

- We also show as *xr*, the realisation of the SDE without any Wiener motion.

```
# Parameters
num_simulations = 5
total_time = 10.0
num_steps = 1000
dt = total_time / num_steps
```

```
# Define the drift term
def drift(x, t):
    return 0.1*x
```

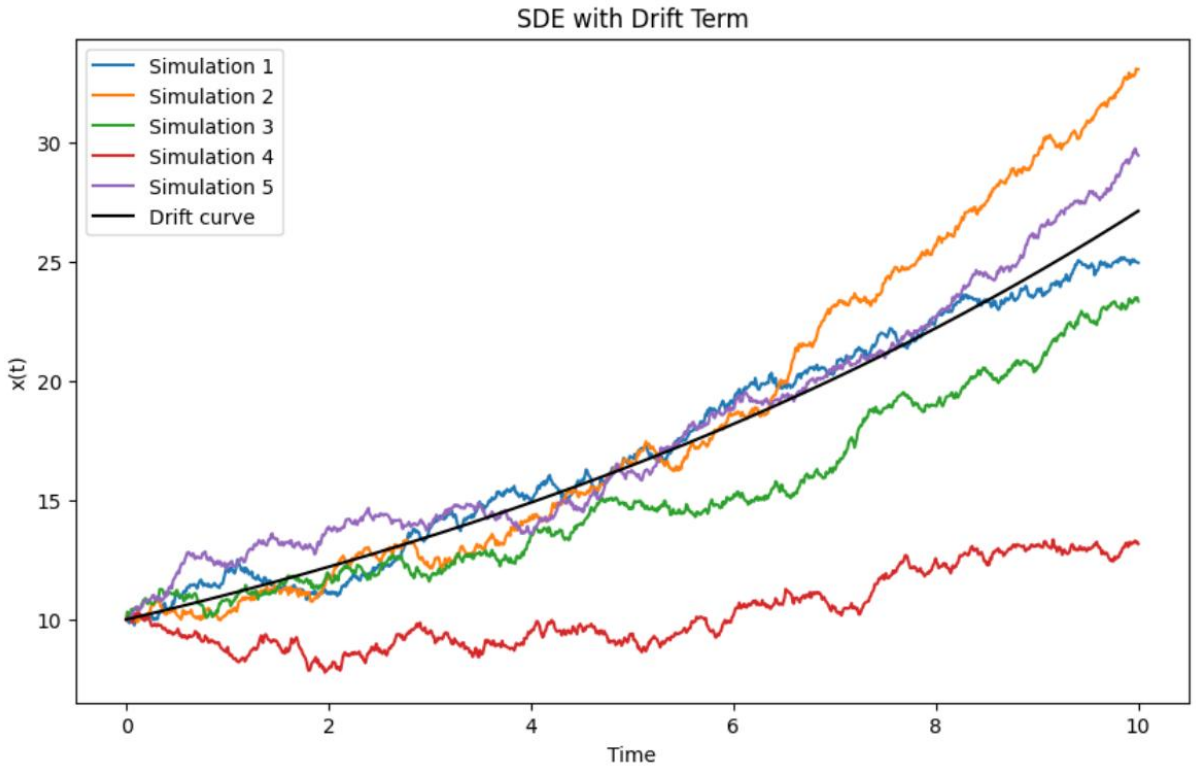
```
# Initialize the state variable
x = np.zeros((num_simulations, num_steps))
x[:, 0] = 10.0  # Initial condition

# Simulate the SDE
for t in range(1, num_steps):
    dW = np.random.normal(0, np.sqrt(dt), size = num_simulations)
    dx = drift(x[:, t-1], t*dt)*dt + dW
    x[:, t] = x[:, t-1] + dx
```

```
xr = np.zeros((1, num_steps)).reshape(num_steps)
xr[0] = 10.0
for t in range(1, num_steps):
    dx = drift(xr[t-1], t*dt)*dt
    xr[t] = xr[t-1] + dx
```

A second SDE with drift

This second SDE is typically implemented as shown.

- We also show as *xr*, the realisation of the SDE without any Wiener motion.
- Typically a 10% increase per unit of time (as is often described the US stock market), with noisy variations.
- Is the stock market an SDE of some sort then?



Typical examples of SDEs

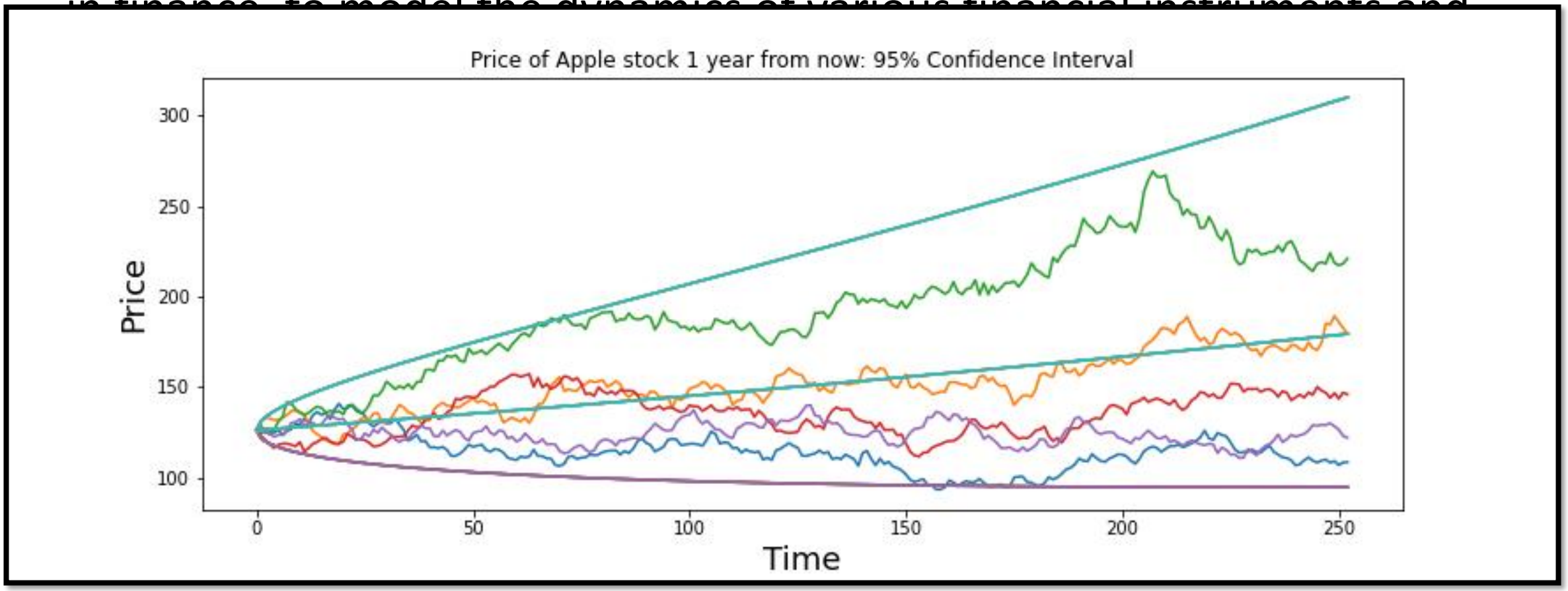
In fact, stochastic differential equations (SDEs) have been widely used, in finance, to model the dynamics of various financial instruments and quantities. For instance, the Geometric Brownian motion is a popular model for simulating the price of assets such as stocks. It is given by:

$$dS(t) = \mu S(t)dt + \sigma S(t)dW(t)$$

Where $S(t)$ is the asset price, μ is the drift of the asset (i.e. expected return of the asset over time), σ is the volatility, and $W(t)$ is a Wiener process modelling the unpredictable behaviour of the asset.

Typical examples of SDEs

In fact, stochastic differential equations (SDEs) have been widely used,



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, similar to our autoencoder in W10-NB1, running on linear layers.
- And a basic training procedure…

```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
# Define the encoder
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 400),
            nn.ReLU(),
            nn.Linear(400, 400),
            nn.ReLU()
        )

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

```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)
```

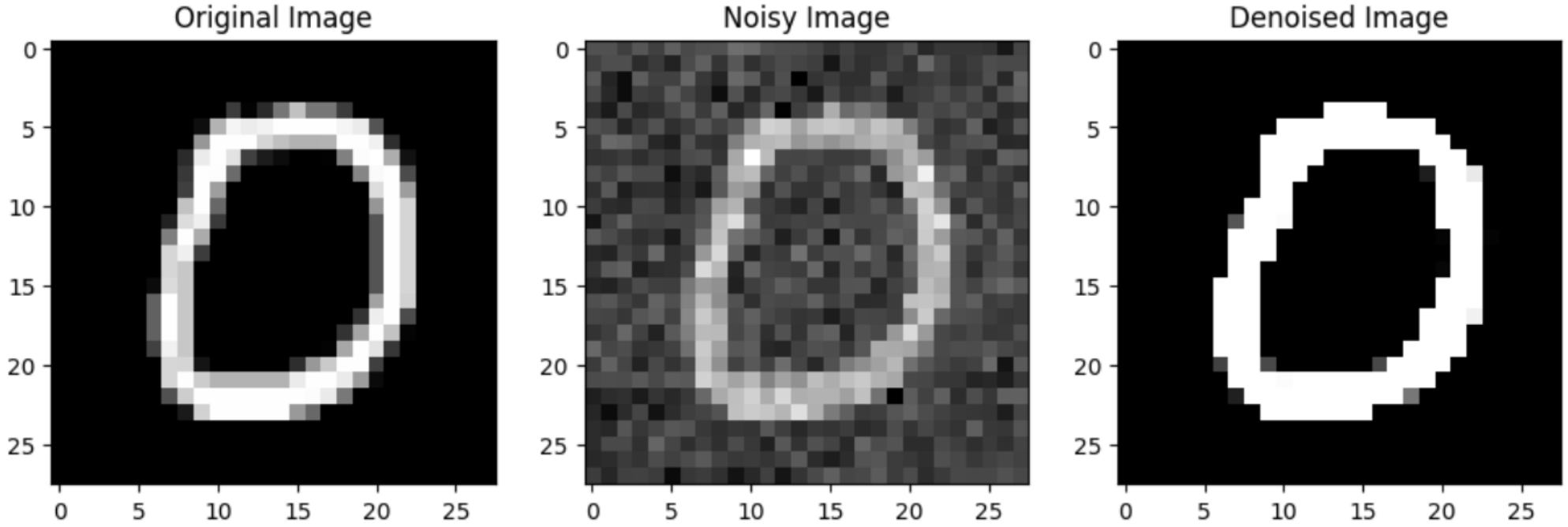
```python
# Hyperparameters
epochs = 10
learning_rate = 1e-3
noise_factor = 0.5
```

```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()}')
```

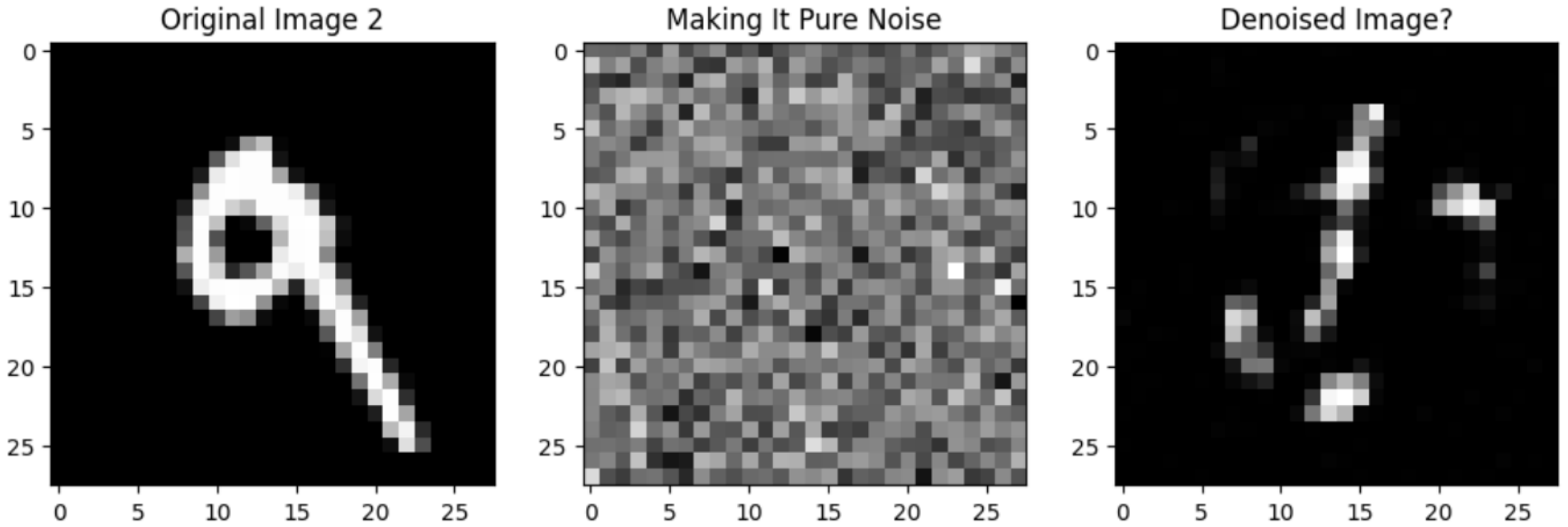
A first task, denoising images

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



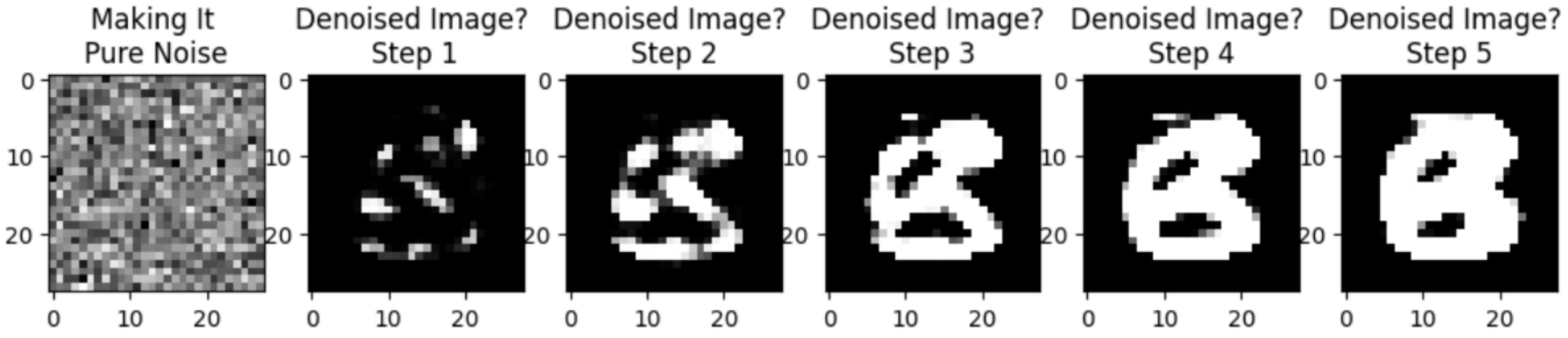
A first task, denoising images

- 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

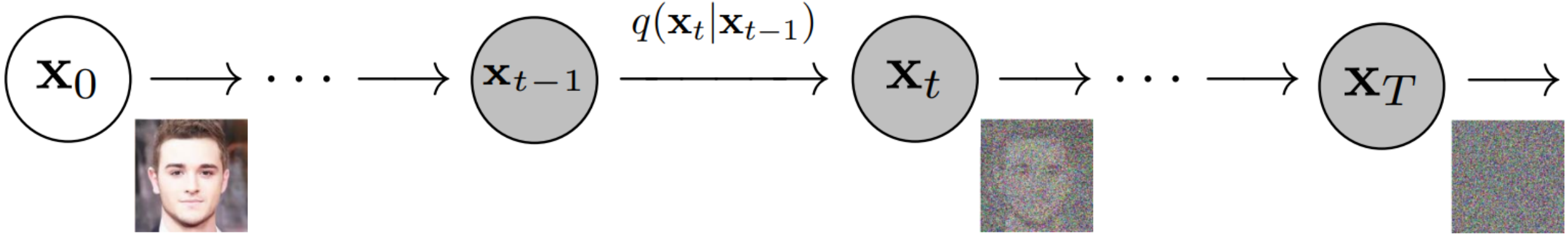
- The model trains and seems able to denoise lightly noised images.
- **Could it denoise a pure noise image, though?** Not really.
- **What if we used the model many (e.g. 5) times in a row on our full noise image?** Eventually would obtain something B&W in the likes of MNIST but somehow non-sensical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

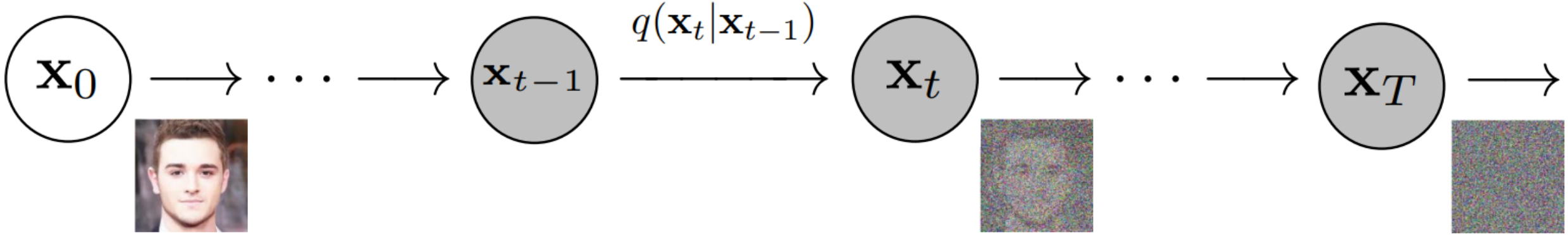
(Somehow similar to the idea of modifying the image iteratively during the iterated versions of our attacks in W5!)



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?

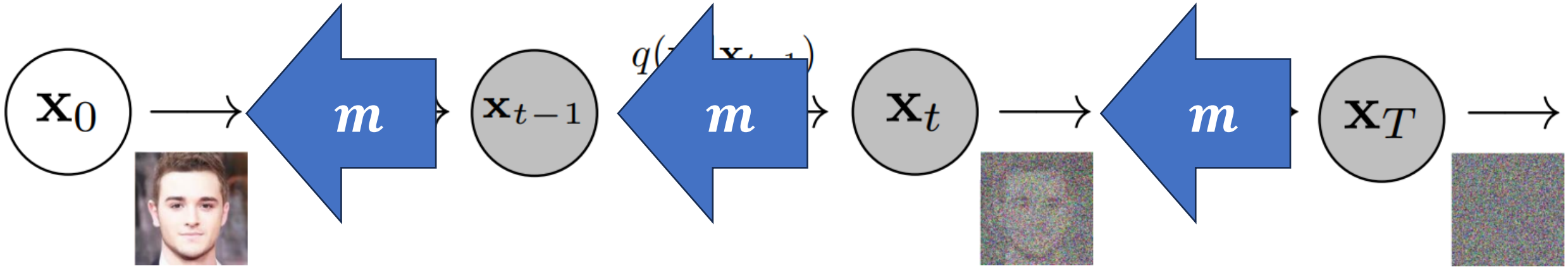
- **Step 1:** Add progressive noising T times in a row, until the image becomes pure noise (or so).
- The noising process follows some sort of an SDE equation describing the progressive noising of the original image (our Forward here!).



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?

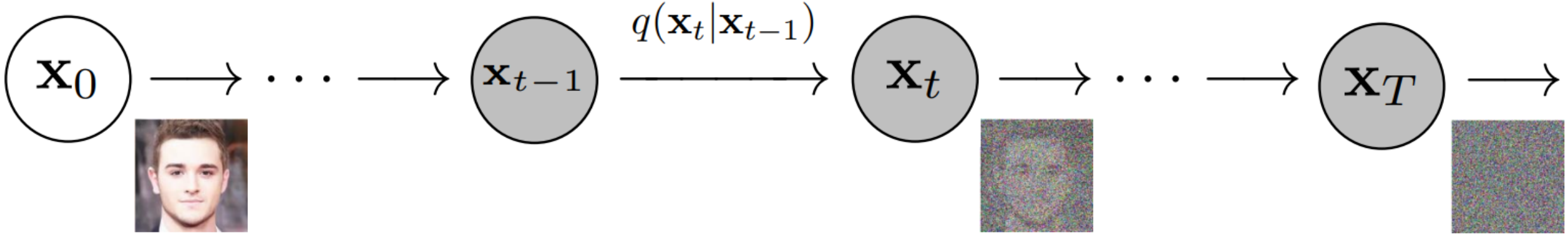
- **Step 2:** Train a single model m to try and cancel noising iteratively.
- Our model m receives image x_t and tries to produce x_{t-1}, do for each noising iteration t (going backwards here!).



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.

```
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.

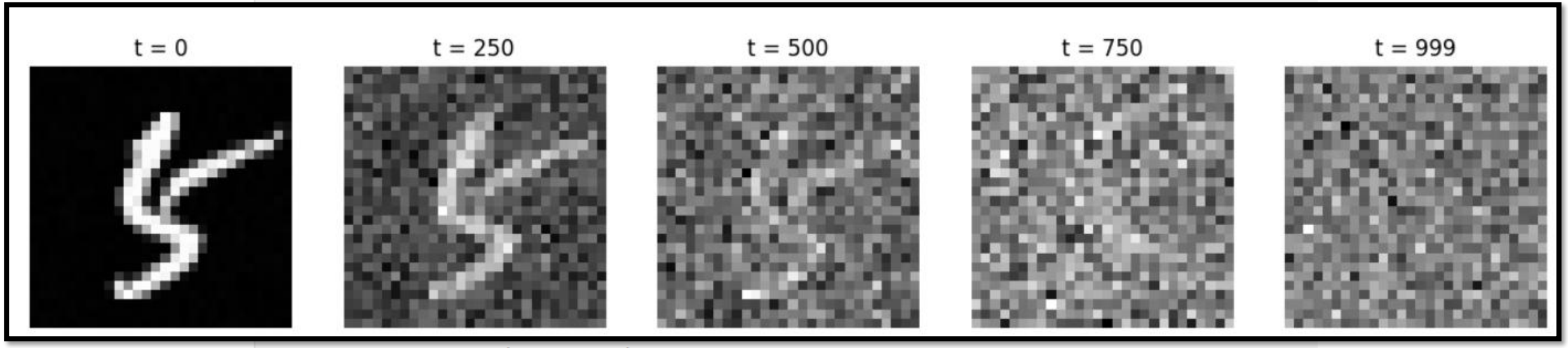
```python
# Forward Diffusion Process
def forward_diffusion(x, t, betas):
    # Perform the forward diffusion process on original image x, timestep t and using the scheduler betas.
    # Returns a noisy image at timestep t, denoted x_t and the noise added to x.
    sqrt_alpha_cumprod = torch.sqrt((1 - 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.

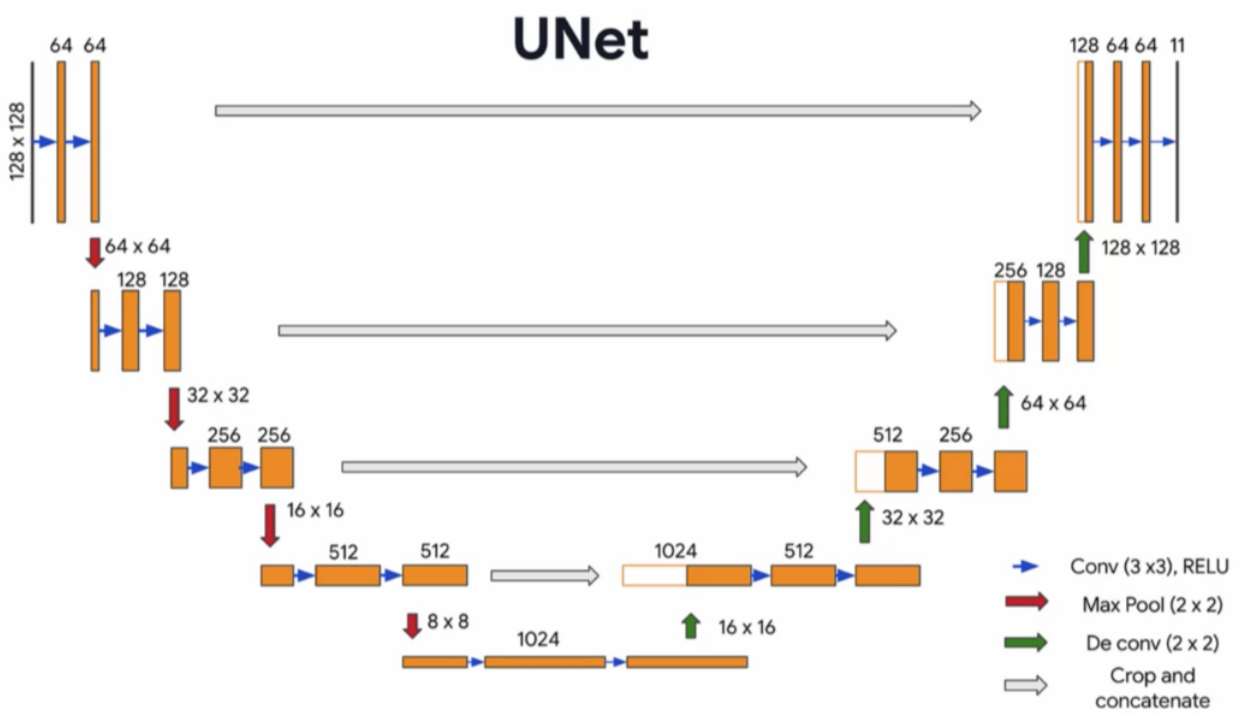


Figure: *A visual representation of a UNet model.*

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.

```
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

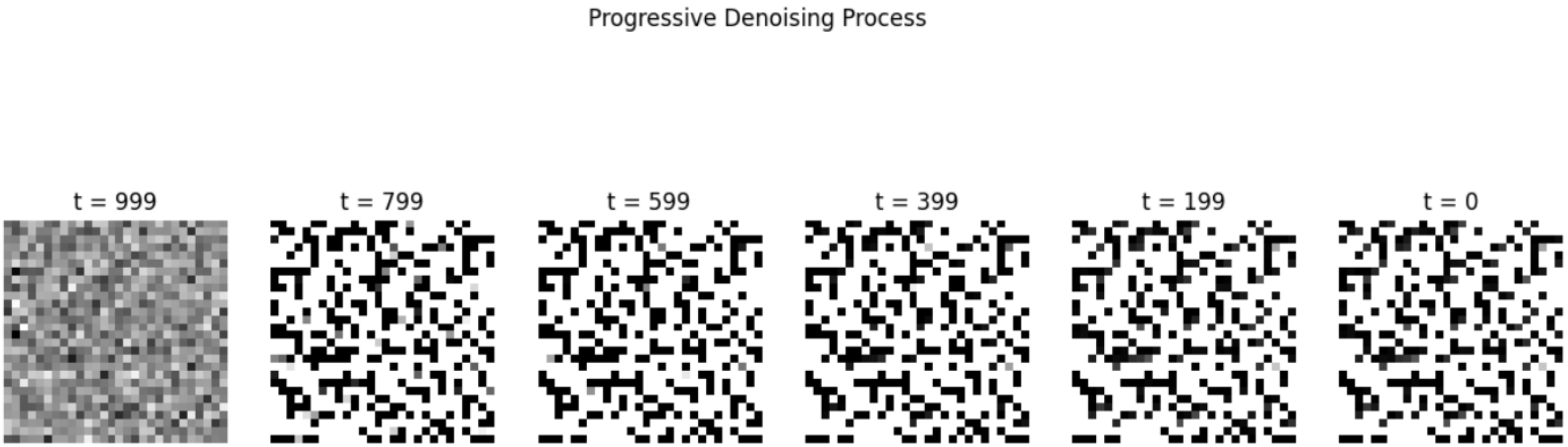
- Let us then define a denoising function that uses the model and progressively denoises the image using the model predictions.

```
# 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]
```

```
# 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

Afte

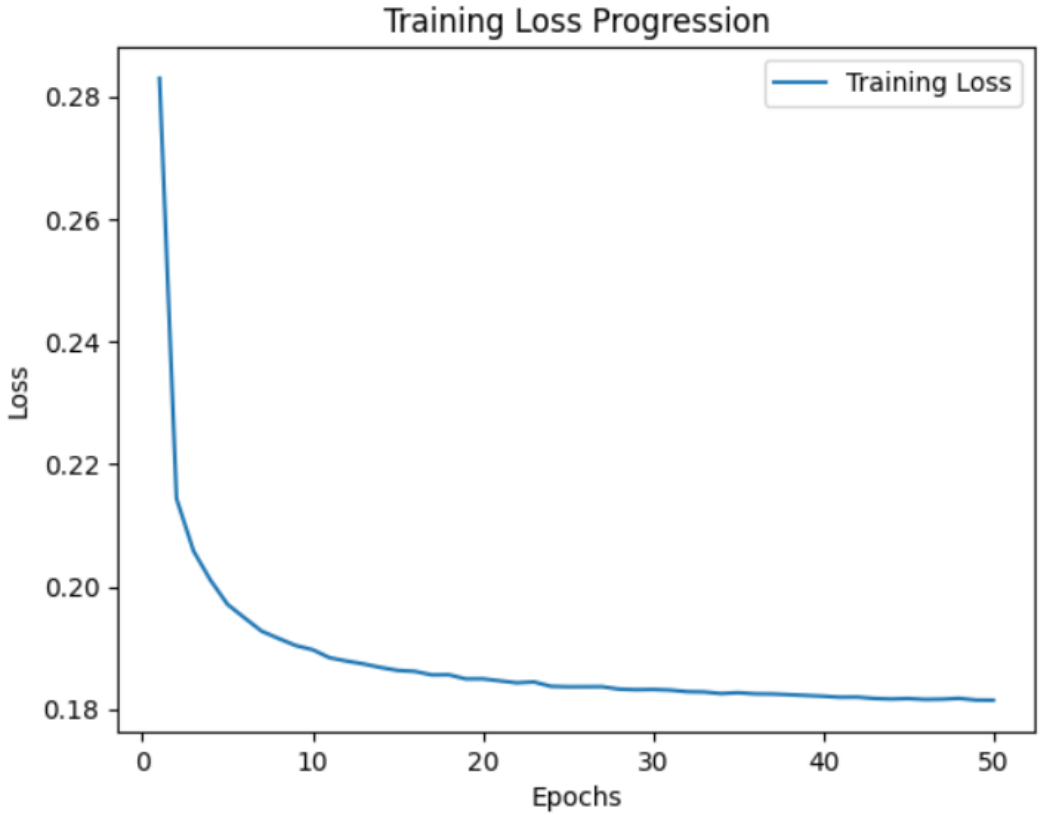
- No
  diff
- Hav
- Use
- Bac
- Usi
- Kee

```python
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?

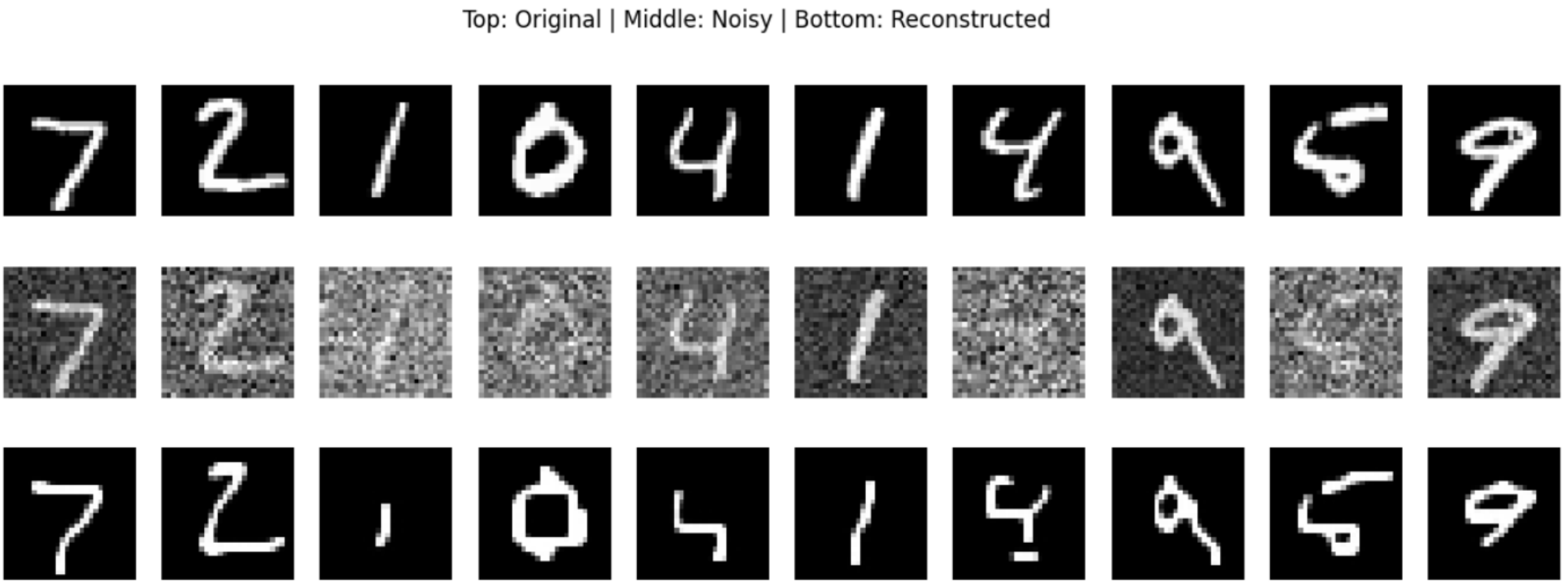
- It seems to train, although it could benefit from our arsenal of training techniques (LR decay/scheduling, using validation, etc.)

```
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!

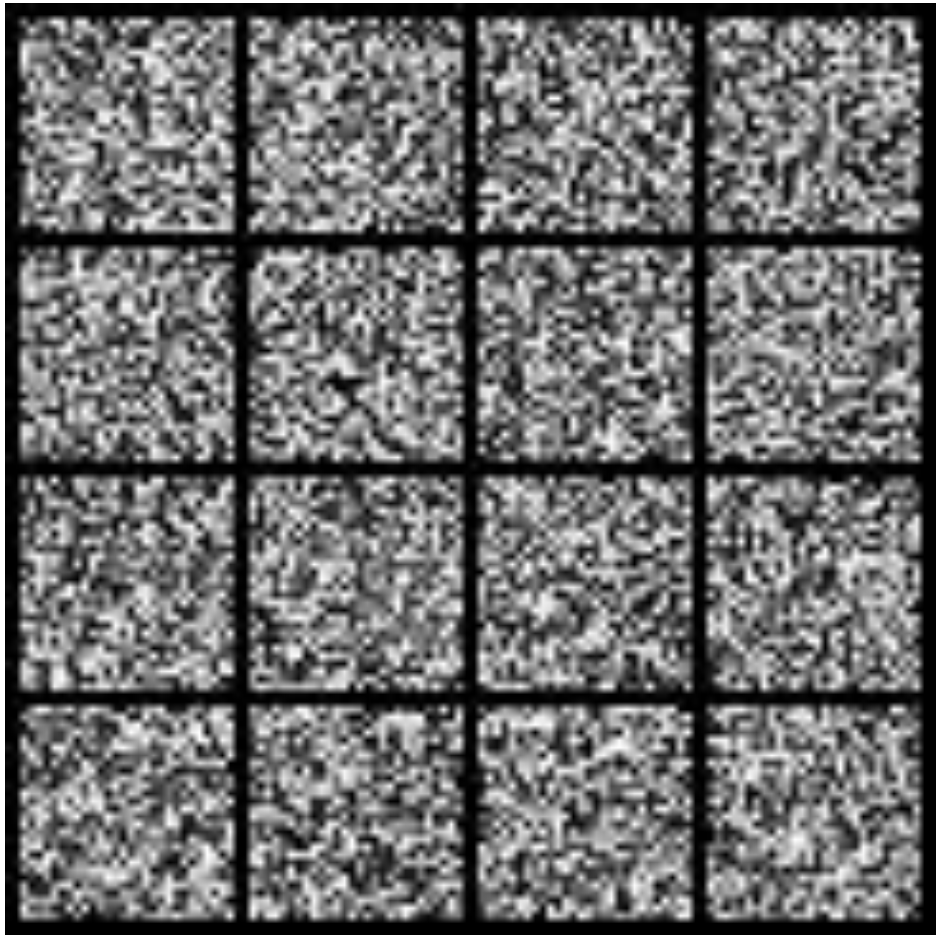
After training, we obtain a model that is “capable” of denoising images.



In practice, however…

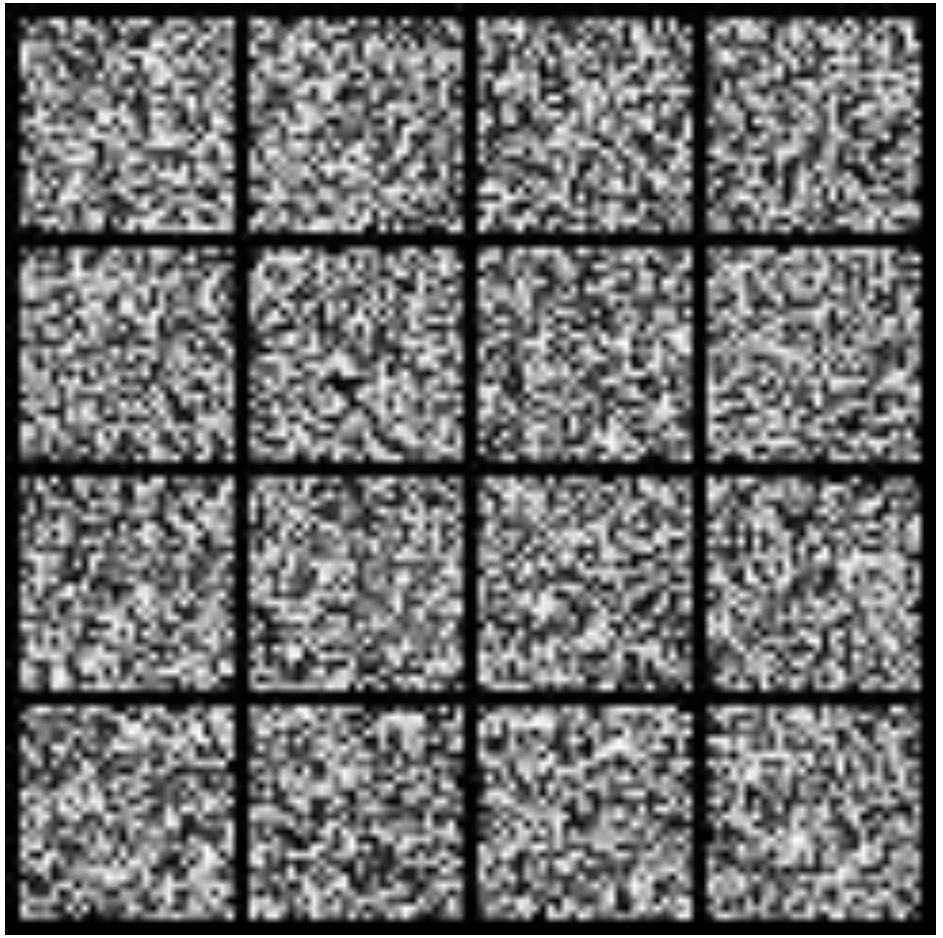
In practice, need more iterations and more steps of noising and denoising, in the thousands or so.

- 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…



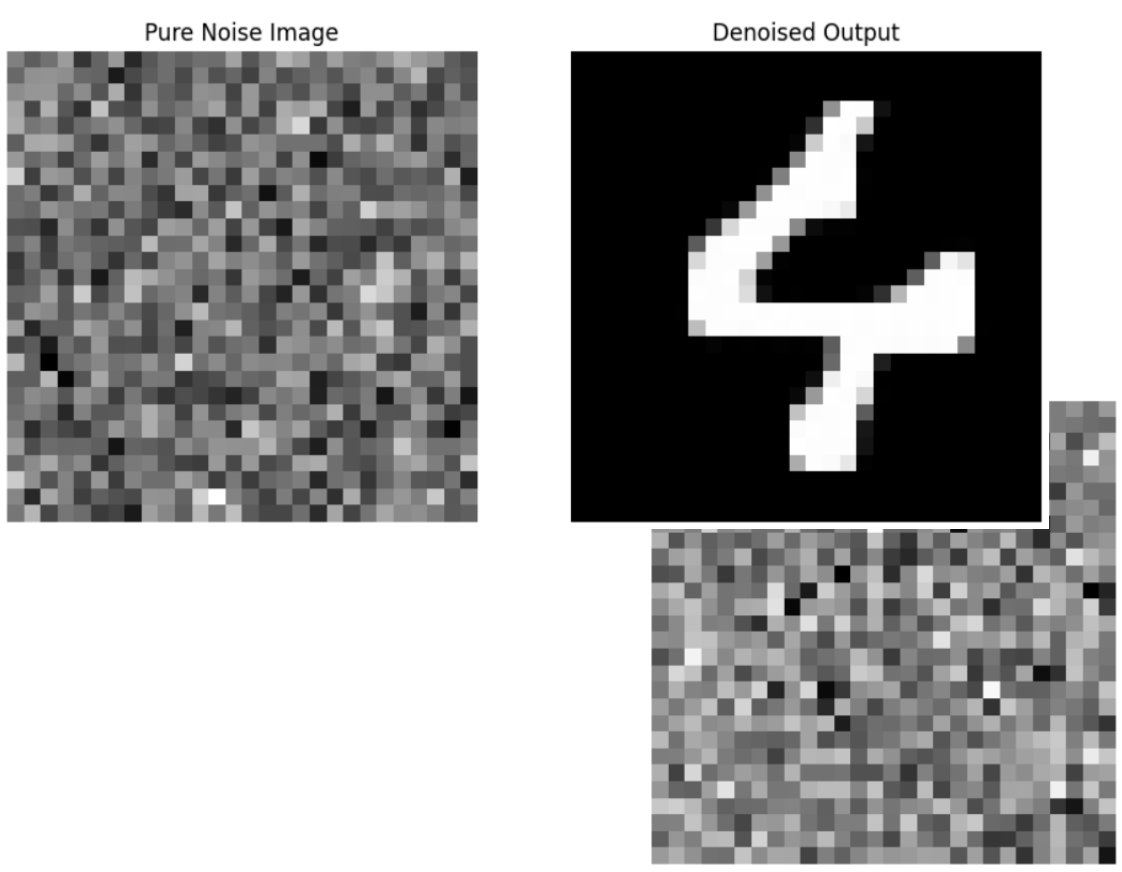
In practice, however…

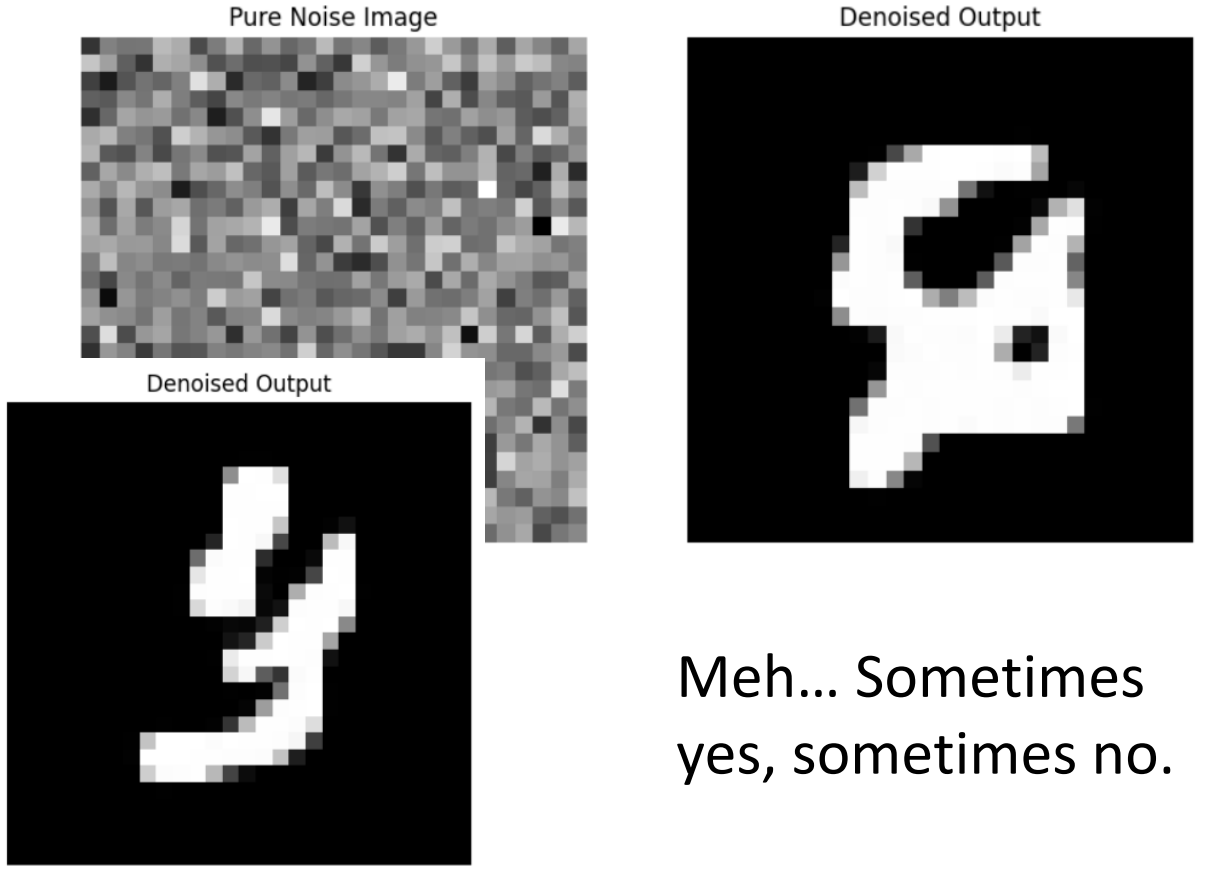
- We were somewhat able to train a model to denoise and eventually obtain a convincing MNIST picture from something that was close to pure noise!
- Another possible approach to generate images out of thin air?
- **Keep in mind:** all advanced ideas from previous weeks are open! (using advanced layers,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

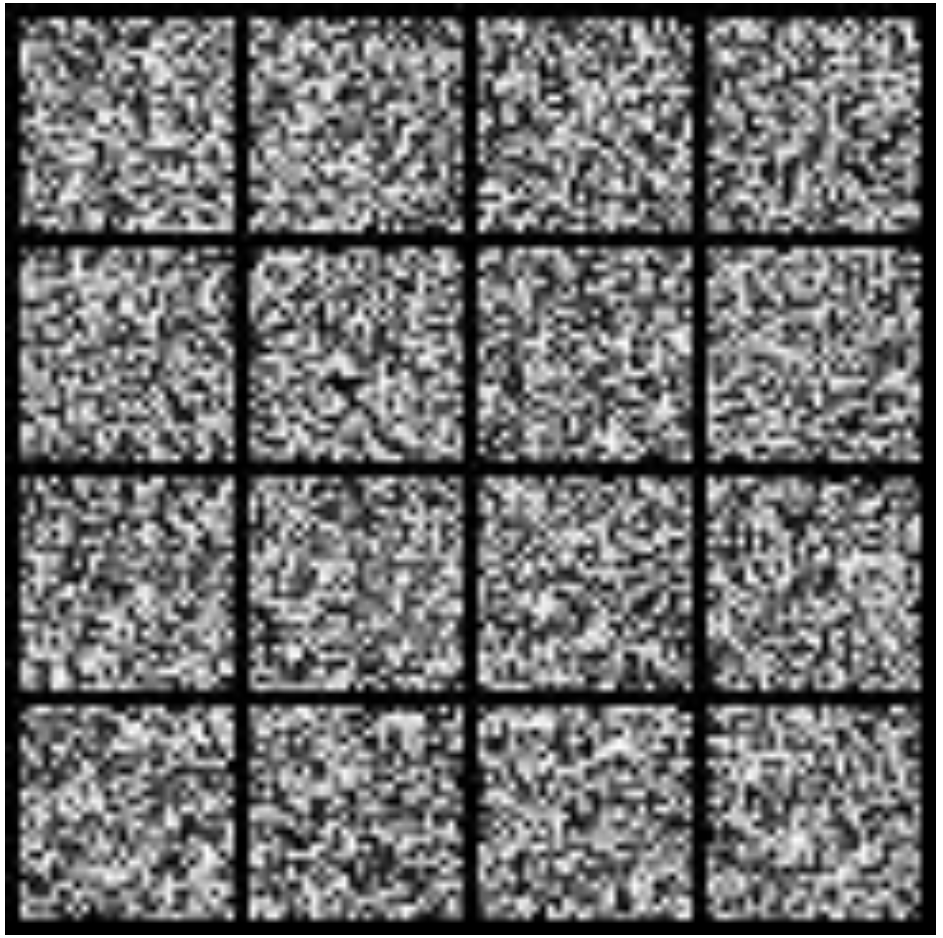




Meh… Sometimes yes, sometimes no.

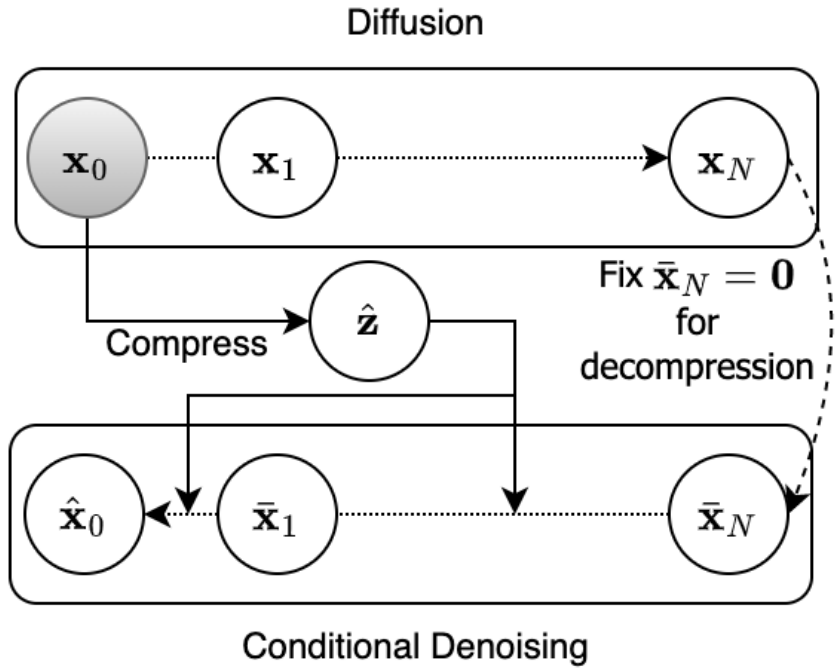
In practice, however…

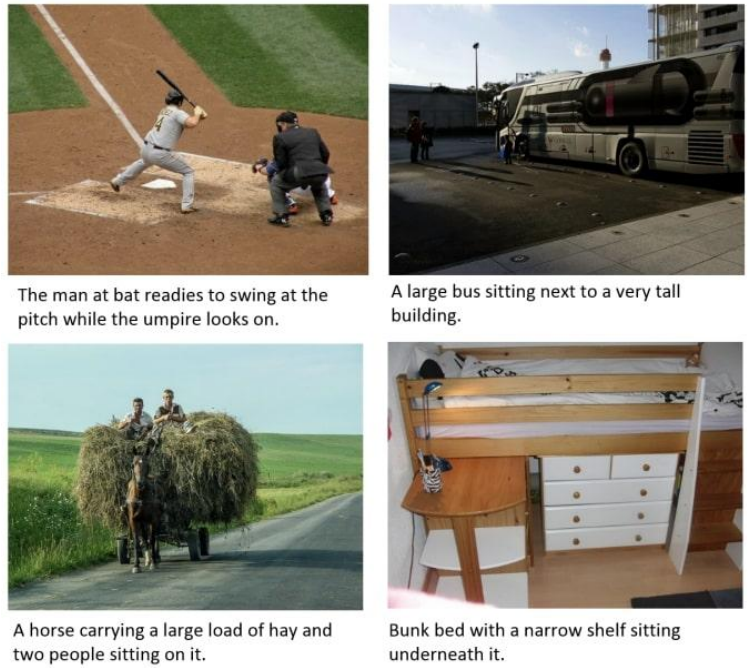
- Clearly, we are not there yet!
- There is a need for more advanced practices to training diffusion models efficiently.
- Learn more in the Computer Vision course on Term 7!
- (Or if you already want a pre-trained implementation, readily available…) https://github.com/VSehwag/minimal-diffusion



Making it conditional

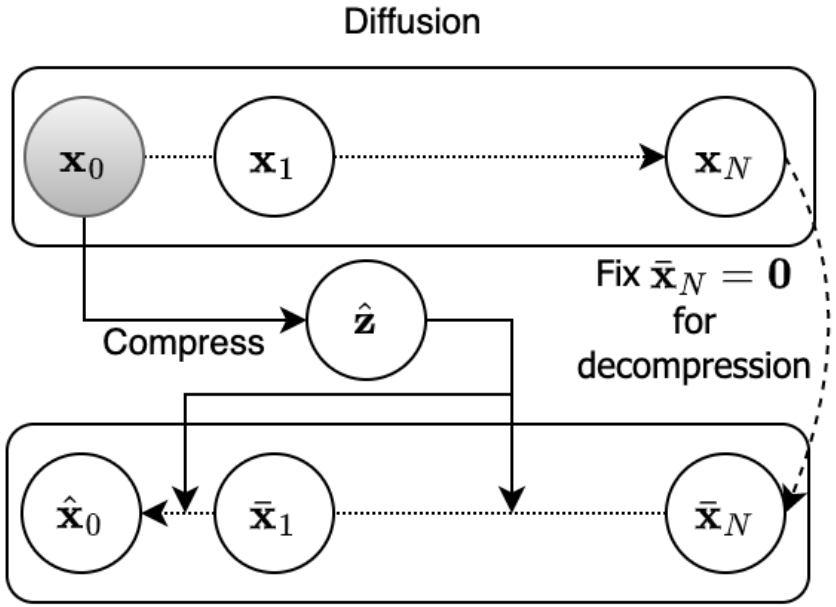
- If you train with a dataset of captioned images, the extra label could be used during the denoising part, to help the denoising model figure out what was the original image about.
- Encode these captions and use them during training of the denoising model in place of t.
- Helps to direct the denoising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

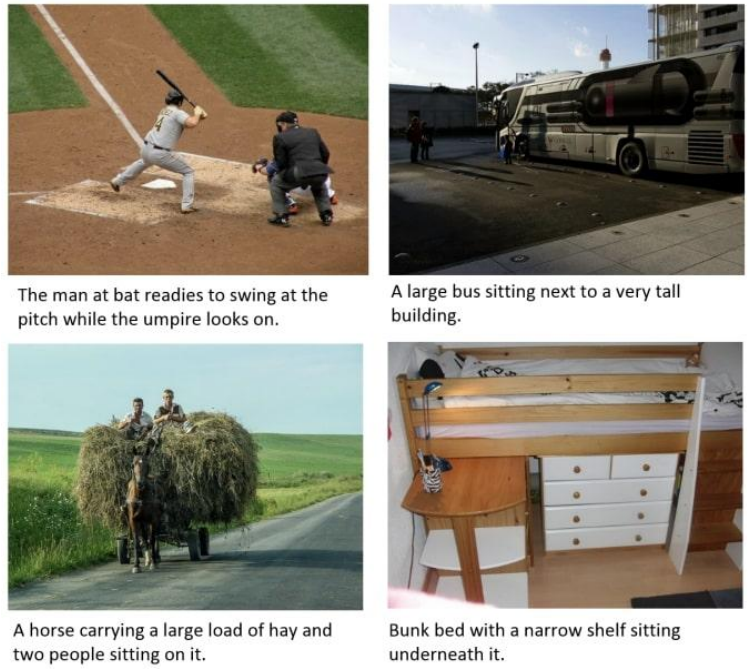




Making it conditional

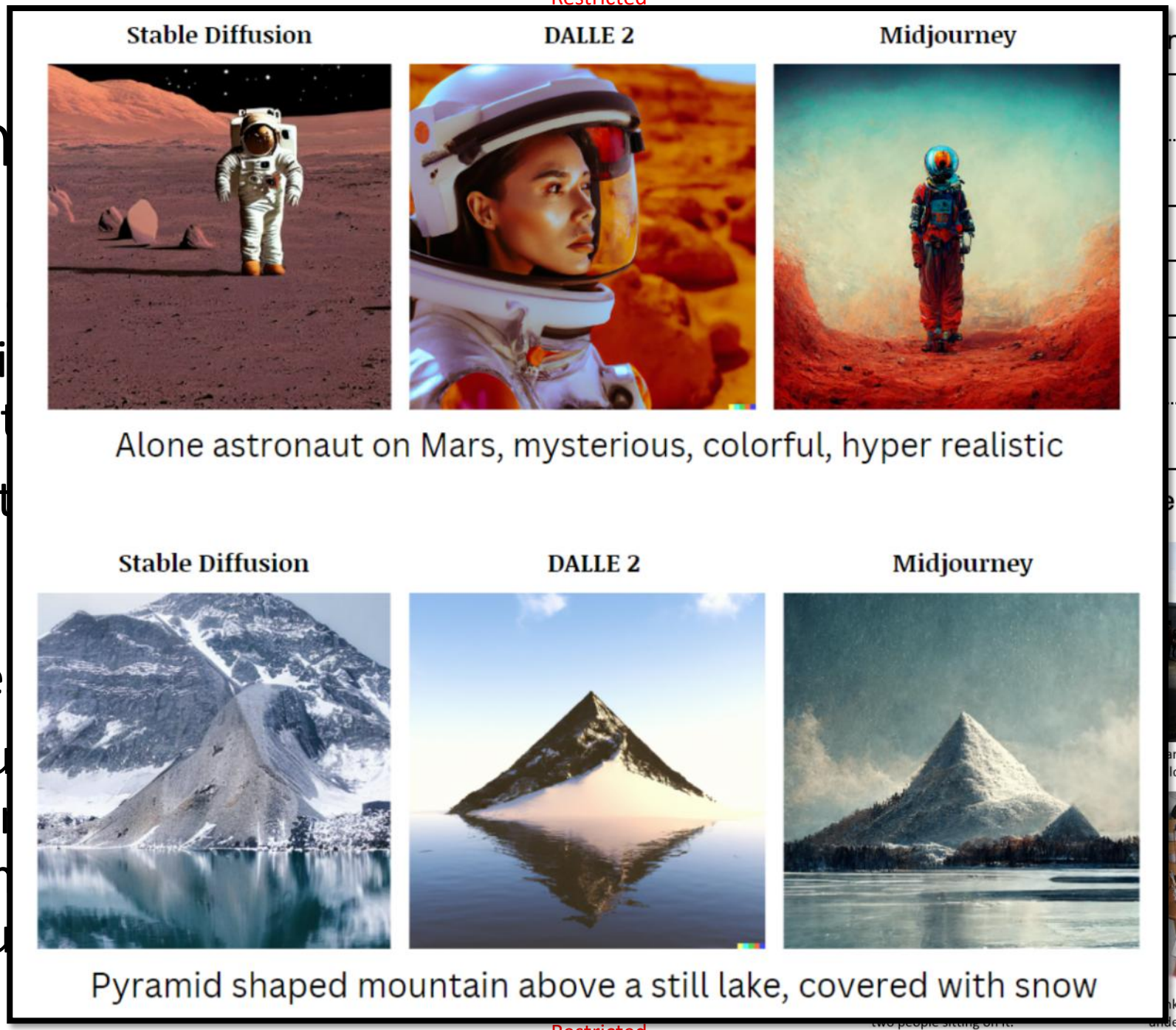
- Label $\hat{z}$ could be the class of the MNSIT image, or the embedding of a text caption.
- During testing, use pure noise as x_N and ask the user for a caption of their choice that will be compressed as $\hat{z}$.
- Use your denoiser many times in a row and obtain an image matching the caption $\hat{z}$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

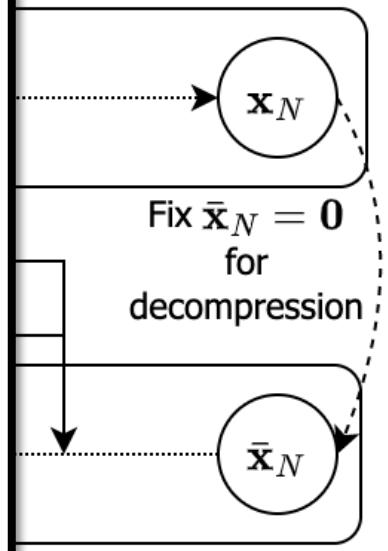




Makin

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





enoising

rge bus sitting next to a very tall
lding.

k bed with a narrow shelf sitting
derneath 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.

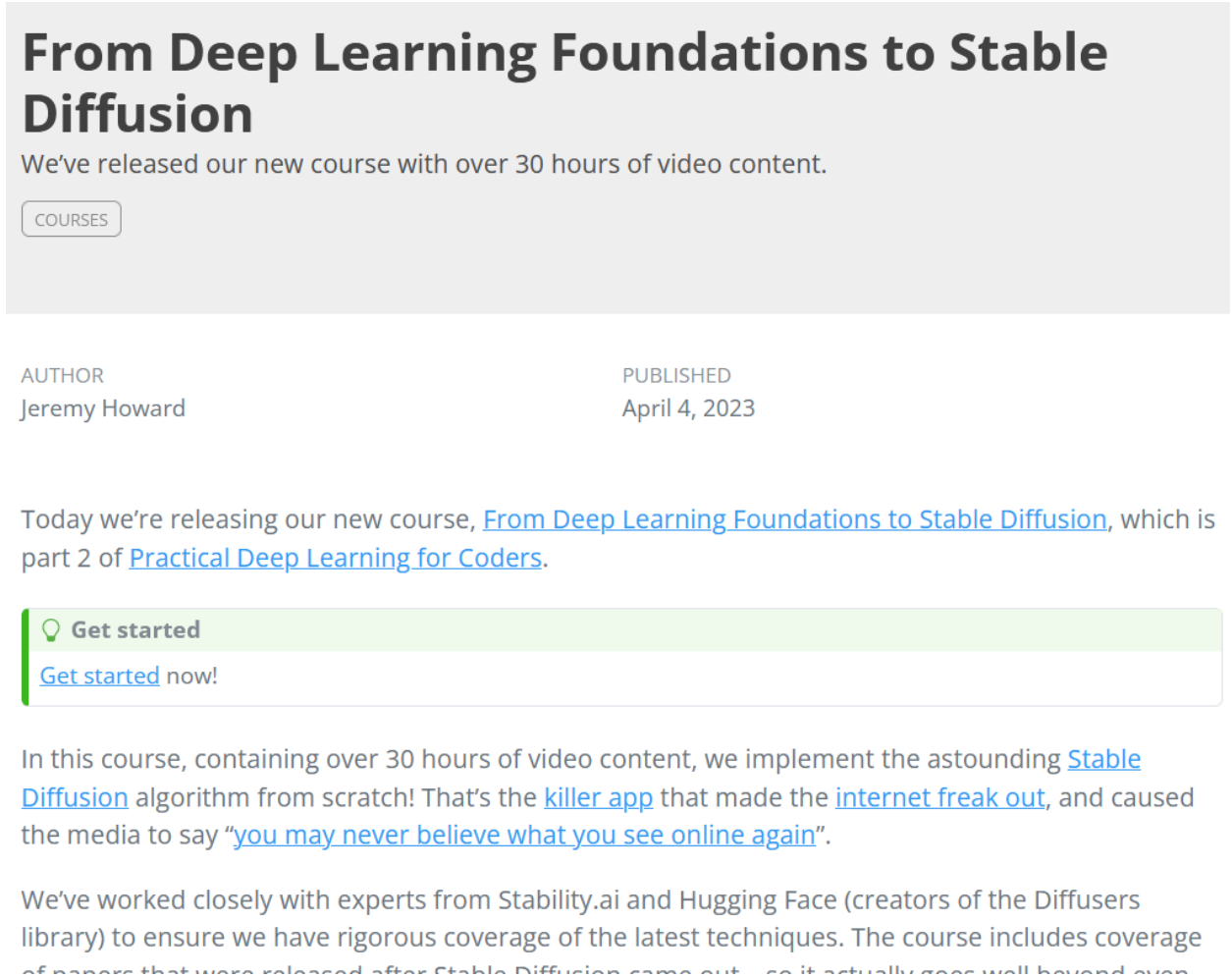
- While it would be impossible to explain the issues and stable diffusion in two minutes, know that it achieves this using **latent diffusion**, a key innovation.
- **Normal Diffusion Models:** Operate directly in pixel space, which can be computationally expensive and slow for high-resolution images.
- **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 the Term 7 Computer Vision course?

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

- If you cannot wait for the **Term 7 Computer Vision** course…
- I guess, 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

Conclusion

1. What is a **Stochastic Differential Equation**?
2. What are **typical uses** of SDEs?
3. Can I **noise images progressively** using SDEs?
4. Can you train a denoising autoencoder model that removes SDE noise from images?
5. What is the **diffusion process** and how to use it to train **diffusion models**?
6. What are the architectures of DALL-E and Midjourney models?

Learn more about these topics

Out of class, for those of you who are curious

- The one paper, usually cited for Diffusion models
  [Ho2020] J. Ho, X. Chen, A. Srinivas, Y. Duan, and P. Abbeel, "Denoising diffusion probabilistic models. Advances in Neural Information Processing Systems", 2020.
  https://arxiv.org/abs/2006.11239

- [Dhariwal2021] P. Dhariwal, and A. Nichol, "Diffusion models beat GANs on ImageNet.", 2021.
  https://arxiv.org/abs/2105.05233