Diffusion Models-2

Mustafa Hajij

Class conditioned diffusion models

```
class ClassConditionedUnet(nn.Module):
  def __init__(self, num_classes=10, class_emb_size=4):
    super().__init__()

    # The embedding layer will map the class label to a vector of size class_emb_size
    self.class_emb = nn.Embedding(num_classes, class_emb_size)

    # Self.model is an unconditional UNet with extra input channels to accept the conditioning information (the class embedding)
    self.model = UNet2DModel(
        sample_size=28,           # the target image resolution
        in_channels=1 + class_emb_size, # Additional input channels for class cond.
        out_channels=1,           # the number of output channels
        layers_per_block=2,       # how many ResNet layers to use per UNet block
        block_out_channels=(32, 64, 64),
        down_block_types=(
            "DownBlock2D",        # a regular ResNet downsampling block
            "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
            "AttnDownBlock2D",
        ),
        up_block_types=(
            "AttnUpBlock2D",
            "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
            "UpBlock2D",          # a regular ResNet upsampling block
          ),
    )

  # Our forward method now takes the class labels as an additional argument
  def forward(self, x, t, class_labels):
    # Shape of x:
    bs, ch, w, h = x.shape

    # class conditioning in right shape to add as additional input channels
    class_cond = self.class_emb(class_labels) # Map to embedding dimension
    class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
    # x is shape (bs, 1, 28, 28) and class_cond is now (bs, 4, 28, 28)

    # Net input is now x and class cond concatenated together along dimension 1
    net_input = torch.cat((x, class_cond), 1) # (bs, 5, 28, 28)

    # Feed this to the UNet alongside the timestep and return the prediction
    return self.model(net_input, t).sample # (bs, 1, 28, 28)
```

[02_class_conditioned_diffusion_model_example.ipynb - Colab (google.com)]()

Class conditioned diffusion models

```
class ClassConditionedUnet(nn.Module):
  def __init__(self, num_classes=10, class_emb_size=4):
    super().__init__()

    # The embedding layer will map the class label to a vector of size class_emb_size
    self.class_emb = nn.Embedding(num_classes, class_emb_size)

    # Self.model is an unconditional UNet with extra input channels to accept the conditioning information (the class embedding)
    self.model = UNet2DModel(
        sample_size=28,           # the target image resolution
        in_channels=1 + class_emb_size, # Additional input channels for class cond.
        out_channels=1,           # the number of output channels
        layers_per_block=2,       # how many ResNet layers to use per UNet block
        block_out_channels=(32, 64, 64),
        down_block_types=(
            "DownBlock2D",        # a regular ResNet downsampling block
            "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
            "AttnDownBlock2D",
        ),
        up_block_types=(
            "AttnUpBlock2D",
            "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
            "UpBlock2D",          # a regular ResNet upsampling block
          ),
    )

  # Our forward method now takes the class labels as an additional argument
  def forward(self, x, t, class_labels):
    # Shape of x:
    bs, ch, w, h = x.shape

    # class conditioning in right shape to add as additional input channels
    class_cond = self.class_emb(class_labels) # Map to embedding dimension
    class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
    # x is shape (bs, 1, 28, 28) and class_cond is now (bs, 4, 28, 28)

    # Net input is now x and class cond concatenated together along dimension 1
    net_input = torch.cat((x, class_cond), 1) # (bs, 5, 28, 28)

    # Feed this to the UNet alongside the timestep and return the prediction
    return self.model(net_input, t).sample # (bs, 1, 28, 28)
```

These are the changes for the conditional model

[02_class_conditioned_diffusion_model_example.ipynb - Colab (google.com)]

Class conditioned diffusion models

```
class ClassConditionedUnet(nn.Module):
  def __init__(self, num_classes=10, class_emb_size=4):
    super().__init__()

    # The embedding layer will map the class label to a vector of size class_emb_size
    self.class_emb = nn.Embedding(num_classes, class_emb_size)

    # Self.model is an unconditional UNet with extra input channels to accept the conditioning information (the class embedding)
    self.model = UNet2DModel(
        sample_size=28,           # the target image resolution
        in_channels=1 + class_emb_size, # Additional input channels for class cond.
        out_channels=1,           # the number of output channels
        layers_per_block=2,       # how many ResNet layers to use per UNet block
        block_out_channels=(32, 64, 64),
        down_block_types=(
            "DownBlock2D",        # a regular ResNet downsampling block
            "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
            "AttnDownBlock2D",
        ),
        up_block_types=(
            "AttnUpBlock2D",
            "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
            "UpBlock2D",          # a regular ResNet upsampling block
          ),
    )

  # Our forward method now takes the class labels as an additional argument
  def forward(self, x, t, class_labels):
    # Shape of x:
    bs, ch, w, h = x.shape

    # class conditioning in right shape to add as additional input channels
    class_cond = self.class_emb(class_labels) # Map to embedding dimension
    class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
    # x is shape (bs, 1, 28, 28) and class_cond is now (bs, 4, 28, 28)

    # Net input is now x and class cond concatenated together along dimension 1
    net_input = torch.cat((x, class_cond), 1) # (bs, 5, 28, 28)

    # Feed this to the UNet alongside the timestep and return the prediction
    return self.model(net_input, t).sample # (bs, 1, 28, 28)
```

Why did we make this choice ?

[02_class_conditioned_diffusion_model_example.ipynb - Colab (google.com)]

Fine-Tuning and Guidance

There are two main approaches for adapting existing diffusion models:

- With **fine-tuning**, we'll re-train existing models on new data to change the type of output they produce

- With **guidance**, we'll take an existing model and steer the generation process at inference time for additional control

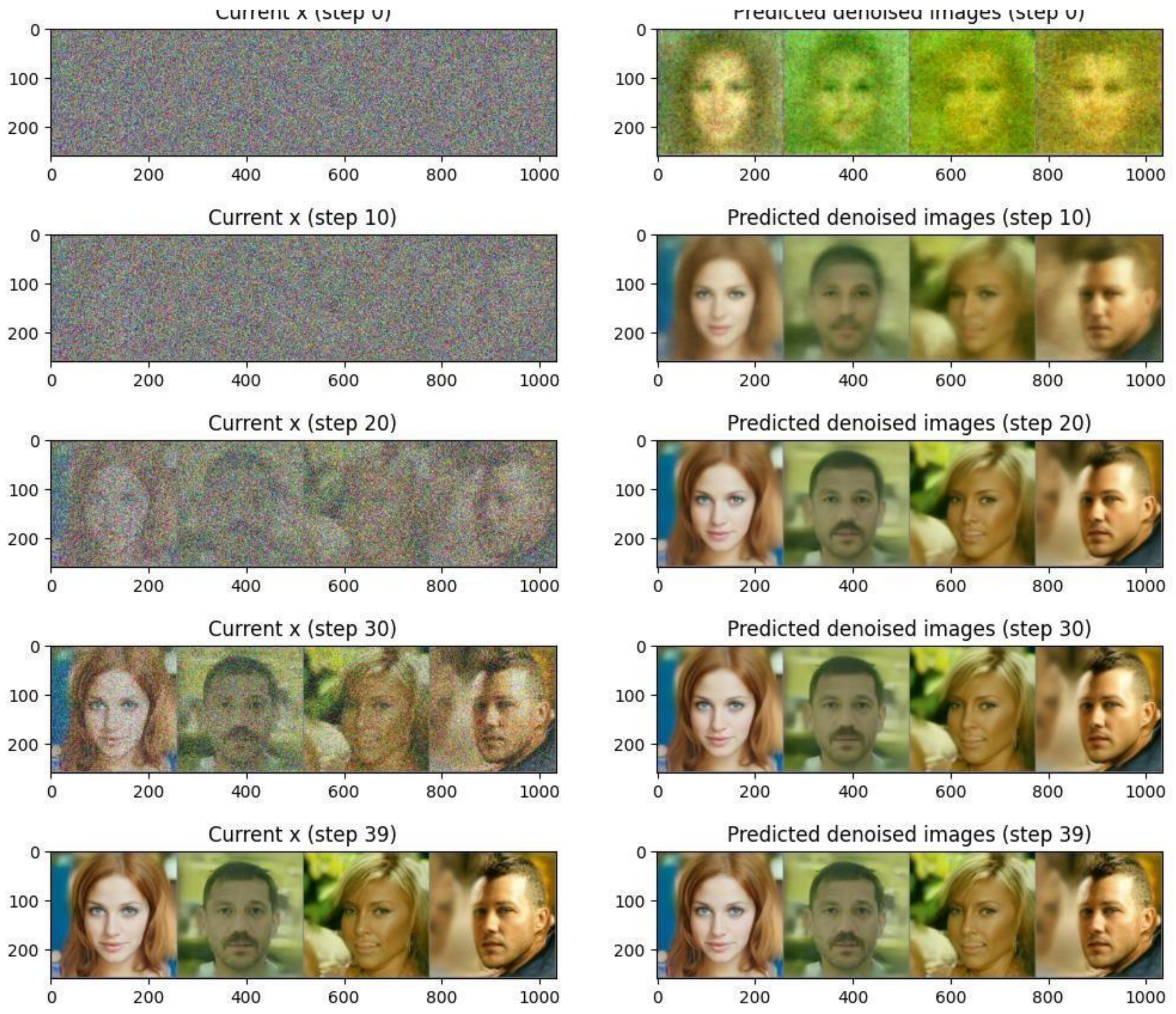
[01_finetuning_and_guidance.ipynb - Colab (google.com)]

Fine-Tuning and Guidance: image denoising overview

1. Model Process:
- The model is fed a noisy input and tasked with predicting the noise, thereby estimating the denoised image.
  - Initial predictions are inaccurate, necessitating a multi-step process.
- Recent research shows that using over 1000 steps is unnecessary; efficient sampling can be achieved with fewer steps.

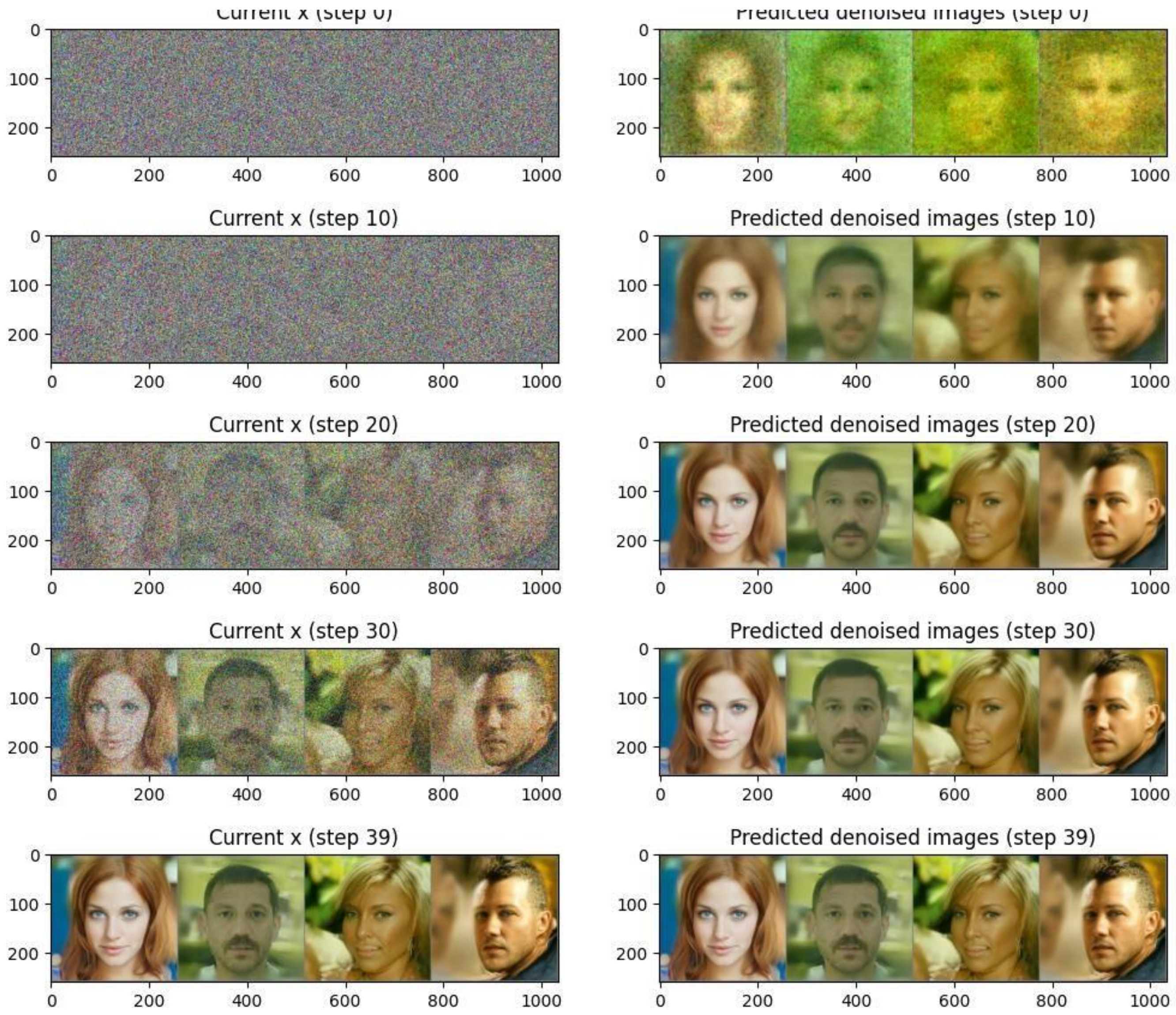
2. Sampling Methods in 🤗 Diffusers Library:
  - Scheduler: Handles sampling methods, performing updates via the `step()` function.
  - **Image Generation**:
    - Start with random noise (x).
    - For each timestep in the noise schedule, feed the noisy input (x) to the model.
    - Pass the prediction to the `step()` function.
- The function returns an output with a `prev_sample` attribute, indicating the denoised image at that step.

Fine-Tuning and Guidance: image denoising overview



Schedulers allow to sample in reasonable time

Fine-Tuning and Guidance: image denoising overview



Schedulers allow to sample in reasonable time

```
image_pipe.scheduler = scheduler
images = image_pipe(num_inference_steps=40).images
images[0]
```

```
 0%|          | 0/40 [00:00<?, ?it/s]
```

40 steps instead of 1000

Training a diffusor

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 4, figsize=(16, 4))
for i, image in enumerate(dataset[:4]["image"]):
    axs[i].imshow(image)
    axs[i].set_axis_off()
fig.show()
```

[diffusers_training_example.ipynb - Colab (google.com)](https://colab.research.google.com)

Latent Diffusion

Diffusing on pixel space is often expensive during training and inference. Earlier solutions suggested diffusing small size images then up-sampling it using, say, super-res NN. Latent diffusion model offer similar solution, but in the latent space. Main steps are :

1. Sample image $x \sim \mathcal{D}$ (dataset)
2. Compute latent $z = \mathcal{E}(x)$
3. Sample $t \sim \mathcal{U}\{1, \ldots, T\}$, noise $\epsilon \sim \mathcal{N}(0, I)$
4. Compute noisy latent $z_t = z + \sigma_t \epsilon$
5. Predict noise $\hat{\epsilon} = \epsilon_\theta(z_t, t)$
6. Update θ via gradient descent on $\|\hat{\epsilon} - \epsilon\|^2$

Latent Conditional Diffusion

Similar steps

1. Sample image $x \sim \mathcal{D}$, with condition c
2. Encode image:
   $z = \mathcal{E}(x)$
3. Sample timestep:
   $t \sim \mathcal{U}(\{1, \ldots, T\})$
4. Sample noise:
   $\epsilon \sim \mathcal{N}(0, I)$
5. Add noise:
   $z_t = z + \sigma_t \epsilon$
6. Predict noise:
   $\hat{\epsilon} = \epsilon_\theta(z_t, t, c)$
7. Minimize loss:
   $\mathcal{L} = \|\hat{\epsilon} - \epsilon\|^2$

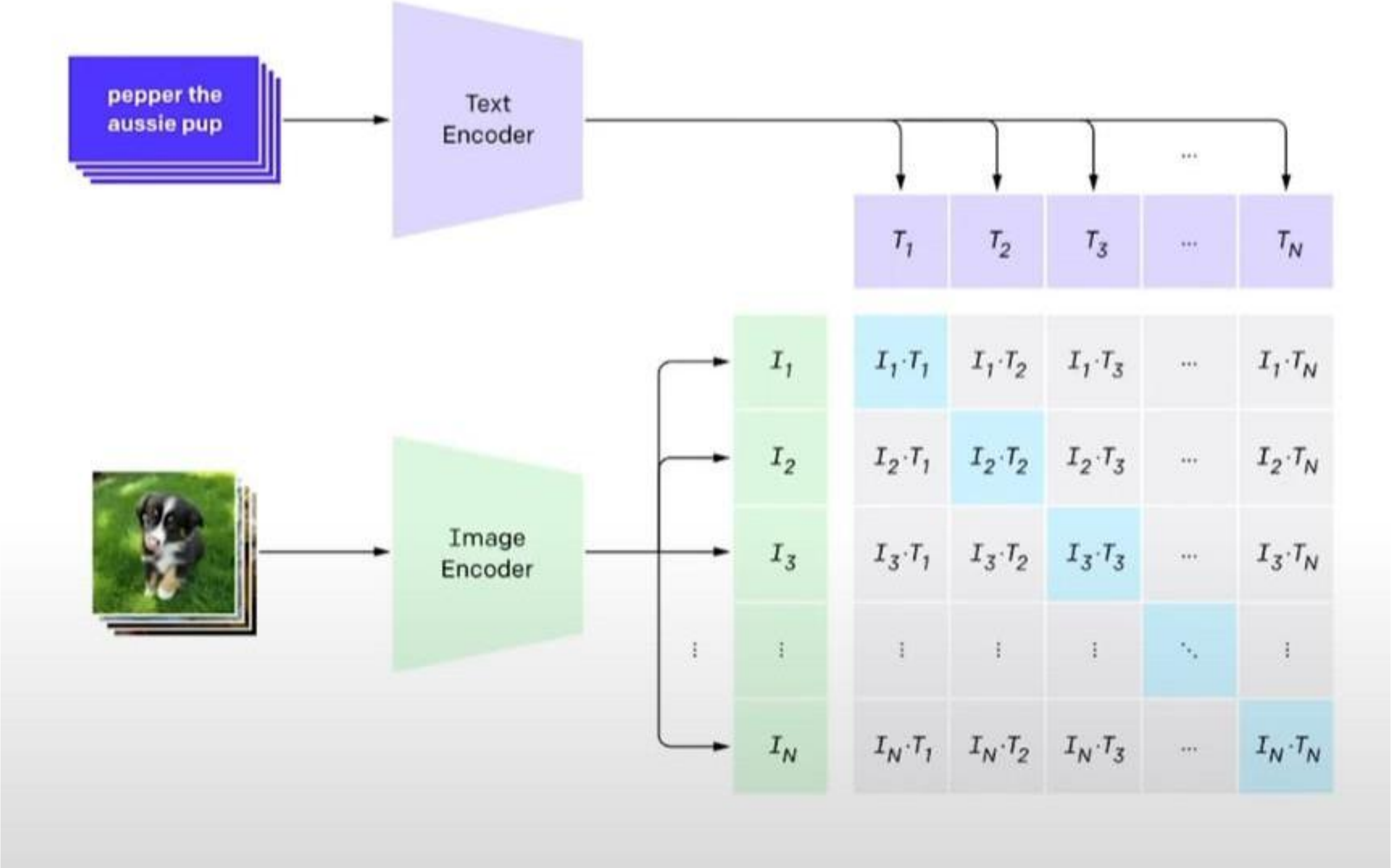
Stable Diffusion

01_stable_diffusion_introduction.ipynb - Colab (google.com)

diffusion-nbs/Stable Diffusion Deep Dive.ipynb at master · fastai/diffusion-nbs (github.com)

Grokking Stable Diffusion.ipynb - Colab (google.com)

Stable Diffusion



Stable Diffusion