Homework 3: Diffusion Models

CSC 8851 – Deep Learning - Fall 2025

Learning Objectives

- Implement a linear noise schedule and visualize the forward diffusion process.
- Use `diffusers` components: `UNet2DModel`, `DDPMScheduler`, and `DDIMScheduler`.
- Train a class-conditional denoiser that predicts x_0 and implement classifier-free guidance (CFG).
- Sample with DDPM (full steps) and DDIM (short trajectory) using CFG; compare speed/quality.

Assignment Tasks (100 pts)

Part 1: Linear Noise Schedule and Forward Noising (10 pts)

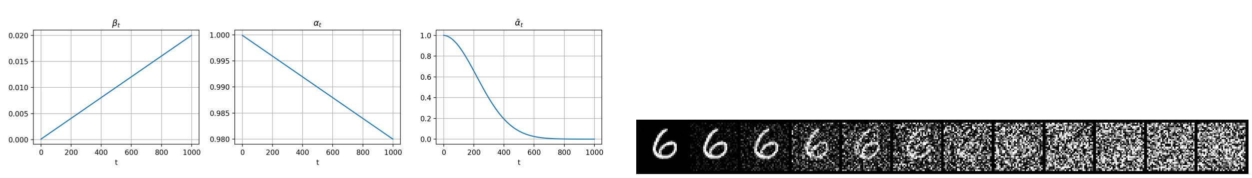
Let $T = 1000$. Define per-step quantities

$$\beta_t \in (0,1), \quad \alpha_t = 1 - \beta_t, \quad \bar{\alpha}_t = \prod_{i=0}^{t} \alpha_i, \quad t = 0, \ldots, T-1,$$

with a **linear** progression $\beta_t \in [10^{-4},\ 2 \cdot 10^{-2}]$. The forward (noising) process is

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\big(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0,\ (1-\bar{\alpha}_t)\mathbf{I}\big) \iff \mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\ \boldsymbol{\varepsilon} \sim \mathcal{N}(0, \mathbf{I}).$$

- (5 pts) Compute and plot $\{\beta_t\}$, $\{\alpha_t\}$, and $\{\bar{\alpha}_t\}$ vs. t (three plots). See Figure 1a.
- (5 pts) For one MNIST image, visualize $\mathbf{x}_t$ at `steps=[0, 50, 100, 200, 250, 300, 400, 500, 600, 700, 800, 999]`. See Figure 1b. **Note: Scale images to** $[-1, 1]$.



(a) Linear schedule: β_t, α_t, and $\bar{\alpha}_t$.

(b) Forward noising grid over the specified steps.

Figure 1: Part 1 diagnostics.

Note (not required): A *cosine* schedule designs $\bar{\alpha}_t$ via a smooth target curve that preserves more signal early and can stabilize training; we only implement **linear** here.

Part 2: Model and Schedulers (PyTorch Diffusers) (20 pts)

- `UNet2DModel`: convolutional U-Net that accepts an input tensor and a discrete timestep t. In our setup it predicts $\hat{\mathbf{x}}_0$ (`prediction_type="sample"`).

UNet2DModel configuration (Do not change the architecture).

$$\texttt{sample\_size} = 32, \quad \texttt{in\_channels} = 1 + 10, \quad \texttt{out\_channels} = 1$$

$$\texttt{layers\_per\_block} = 2, \quad \texttt{block\_out\_channels} = (32,\, 64,\, 128),$$

$$\texttt{down\_block\_types} = (\texttt{"DownBlock2D"}, \texttt{"DownBlock2D"}, \texttt{"DownBlock2D"})$$

$$\texttt{up\_block\_types} = (\texttt{"UpBlock2D"}, \texttt{"UpBlock2D"}, \texttt{"UpBlock2D"}).$$

UNet2DModel architecture. Input $\mathbf{Z} \in \mathbb{R}^{B\times 11\times 32\times 32}$ flows through:

1. **Time embedding:** sinusoidal $\to$ MLP $\to$ injected into ResNet blocks.
2. **Encoder (Down path):** three stages with two ResNet blocks each; channels $11 \to 32 \to 64 \to 128$; spatial downsample $32 \to 16 \to 8 \to 4$. Skip connections saved.
3. **Mid block:** ResNet(s) at $(B, 128, 4, 4)$.
4. **Decoder (Up path):** three stages with skip concatenations; channels $128 \to 64 \to 32$; spatial upsample $4 \to 8 \to 16 \to 32$.
5. **Output head:** 3×3 conv $\to$ $(B, 1, 32, 32)$ giving $\hat{\mathbf{x}}_0$ (no final activation).

- `DDPMScheduler`:
  - Training (forward noising): `Xt = ddpm.add_noise(X0, eps, t)` implements $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\varepsilon$.
  - Sampling (reverse): `ddpm.set_timesteps(N)` then iterate $t \in$ `ddpm.timesteps` and call `ddpm.step(model_output=`$\hat{x}\_0$`, timestep=t, sample=X)` to obtain the next sample.
  - With `prediction_type="sample"`, `model_output` must be $\hat{\mathbf{x}}_0$ (your network output).

- `DDIMScheduler`:
  - Same API as DDPM but supports a shortened, non-Markovian trajectory; set `ddim.set_timesteps(K)` with $K \ll T$.
  - The call `ddim.step(model_output=`$\hat{x}\_0$`, timestep=t, sample=X, eta=0.0)` yields the deterministic DDIM update.

DDPMScheduler and DDIMScheduler configuration

$$\texttt{num\_train\_timesteps} = \texttt{T}, \quad \texttt{beta\_schedule} = \texttt{"linear"}, \quad \texttt{prediction\_type} = \texttt{"sample"}$$

- **Class conditioning (one-hot planes).** Given labels $y \in \{0, \ldots, 9\}^B$, build one-hot class maps $\mathbf{C} \in \mathbb{R}^{B\times 10\times H\times W}$ and concatenate with the (noised) image $\mathbf{X}_t \in \mathbb{R}^{B\times 1\times H\times W}$ to form the model input:

$$\mathbf{Z}_t \;=\; [\mathbf{X}_t;\, \mathbf{C}] \;\in\; \mathbb{R}^{B\times 11\times H\times W}, \quad H = W = 32.$$

Efficient construction: `F.one_hot(y,10).float().view(B,10,1,1).expand(B,10,H,W)` (no per-pixel loops).

`make_model_input` **(training-time CFG dropout).** We randomly zero all class channels (dropout) for some samples to train the model to also handle *unconditional* inputs:

$$\texttt{make\_model\_input}(\mathbf{X}_t, y, p_{\text{uncond}}) \;=\; \big[\,\mathbf{X}_t\,;\,\mathbf{C}\odot\mathbf{M}\,\big], \quad \mathbf{M}\in\{0,1\}^{B\times1\times1\times1},\ \Pr[M{=}0]=p_{\text{uncond}}.$$

```
def make_model_input(x, y, p_uncond=0.1):
    B, _, H, W = x.shape
    C = F.one_hot(y, 10).float().view(B, 10, 1, 1).expand(B, 10, H, W)
    m = (torch.rand(B, device=x.device) < p_uncond).float().view(B, 1, 1, 1)
    C = C * (1.0 - m)                               # zero all class planes for some samples
    return torch.cat([x, C.to(x.dtype)], dim=1)  # (B, 11, H, W)
```

How to use it.

- **Training:** always feed `make_model_input(Xt, y, p_uncond)` into `UNet2DModel`, where `Xt` are noised images generated by `DDPMScheduler.add_noise`. The model predicts $\hat{\mathbf{x}}_0$; optimize $\mathrm{MSE}(\hat{\mathbf{x}}_0, \mathbf{x}_0)$.
- **Sampling with CFG (Part 5):** build two inputs at each step: *conditional* $[\mathbf{X}_t; \mathbf{C}]$ and *unconditional* $[\mathbf{X}_t; \mathbf{0}]$, run the model on both, and combine

$$\hat{\mathbf{x}}_0^{\text{cfg}} = \hat{\mathbf{x}}_0^{\text{cond}} + s\big(\hat{\mathbf{x}}_0^{\text{cond}} - \hat{\mathbf{x}}_0^{\text{uncond}}\big), \qquad s \geq 0.$$

Grading for Part 2 (20 pts).

- (10 pts) Correct instantiation of `UNet2DModel`, `DDPMScheduler`, and `DDIMScheduler` with the specified configuration; Evaluate a dummy forward: `model(Z,t).sample -> (B,1,32,32)` and verify shape.
- (10 pts) Implement `make_model_input` (correct shapes and dropout behavior).

Part 3: Training (50 epochs) and Curve (20 pts)

Train the UNet to predict $\mathbf{x}_0$:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}, \quad \mathcal{L} = \mathbb{E}\big[\|\hat{\mathbf{x}}_0(\mathbf{x}_t, y, t) - \mathbf{x}_0\|_2^2\big].$$

One-epoch training pseudocode ($\mathbf{x}_0$ prediction).

```
# Assumes: model (UNet2DModel), ddpm (DDPMScheduler), train_loader, optimizer 'opt'
model.train()
running, total, count = None, 0.0, 0
for X0, y in train_loader:                    # X0 in [-1,1], shape (B,1,32,32)
    X0 = X0.to(dev); y = y.to(dev).long()
    B  = X0.size(0)

    # Sample t and construct forward noising (DDPM)
    t   = torch.randint(0, ddpm.config.num_train_timesteps, (B,),
        device=dev, dtype=torch.long)
```

```
    eps = torch.randn_like(X0)
    Xt  = ddpm.add_noise(X0, eps, t)
    # where  x_t = sqrt(bar_alpha_t) x0 + sqrt(1-bar_alpha_t) eps

    # Build conditional/unconditional-batch input with CFG dropout
    Zt  = make_model_input(Xt, y, p_uncond=0.1)    # (B,11,32,32)

    # Predict x0 and optimize MSE(x0_hat, x0)
    x0_hat = model(Zt, t).sample
    loss   = F.mse_loss(x0_hat, X0)

    opt.zero_grad(set_to_none=True)
    loss.backward()
    opt.step()

    # (Optional) track loss
    total += float(loss.item()) * B; count += B
    running = loss.item() if running is None else 0.98*running + 0.02*loss.item()

avg_loss = total / max(count, 1)
print(f"avg_epoch_loss={avg_loss:.4f}, running={running:.4f}")
```

- (15 pts) Implement loop: sample t, form $\mathbf{x}_t$ via `ddpm.add_noise`, build inputs with `make_model_input`, predict $\hat{\mathbf{x}}_0$, optimize MSE.
- (5 pts) Plot the training loss. See Figure 2.

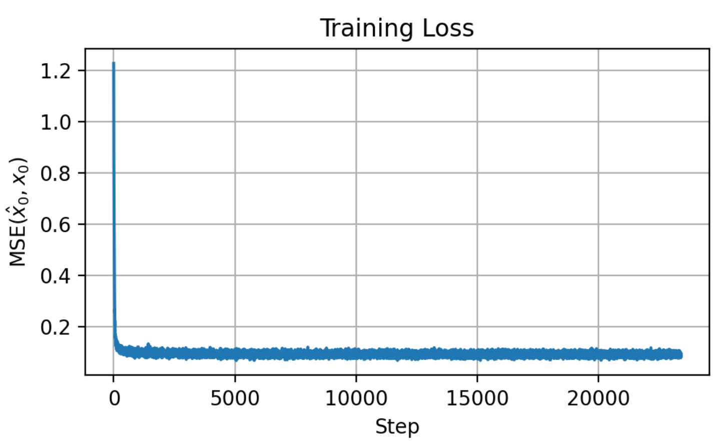
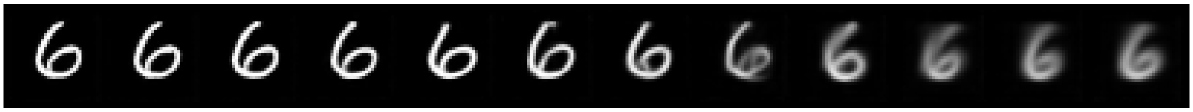


Figure 2: Training loss curve (MSE).

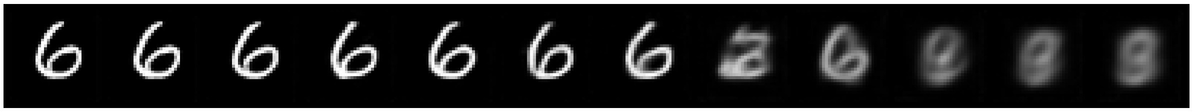
Part 4: Visualizing $\hat{x}_0$ Across Noise (20 pts)

Fix one $(\mathbf{x}_0, y)$. For the timesteps `[0, 50, 100, 200, 250, 300, 400, 500, 600, 700, 800, 999]`, form $\mathbf{x}_t$ using the forward process and visualize the model's prediction $\hat{\mathbf{x}}_0(\mathbf{x}_t, \cdot, t)$ in two settings:

- (10 pts) Conditional. Use the *conditional* input (true label; no dropout): set $p_{\text{uncond}}=0$ inside your visualization helper so the class maps are active. See Figure: 3a.
- (10 pts) Unconditional. Use the *unconditional* input by concatenating zero class maps (i.e., $[\mathbf{x}_t; \mathbf{0}]$). See Figure: 3b.



(a) Conditional $\hat{x}_0$ across timesteps (top: t=0; rightmost: t=999).



(b) Unconditional $\hat{x}_0$ across the same timesteps (zero class maps).

Figure 3: Part 5: $\hat{x}_0$ predictions across noise levels (conditional vs. unconditional).

Part 5: Sampling — DDPM and DDIM with CFG (30 pts)

At each step, form cond/uncond inputs, get $\hat{\mathbf{x}}_0^{\text{cond}}$, $\hat{\mathbf{x}}_0^{\text{uncond}}$, then $\hat{\mathbf{x}}_0^{\text{cfg}}$ with scale s.

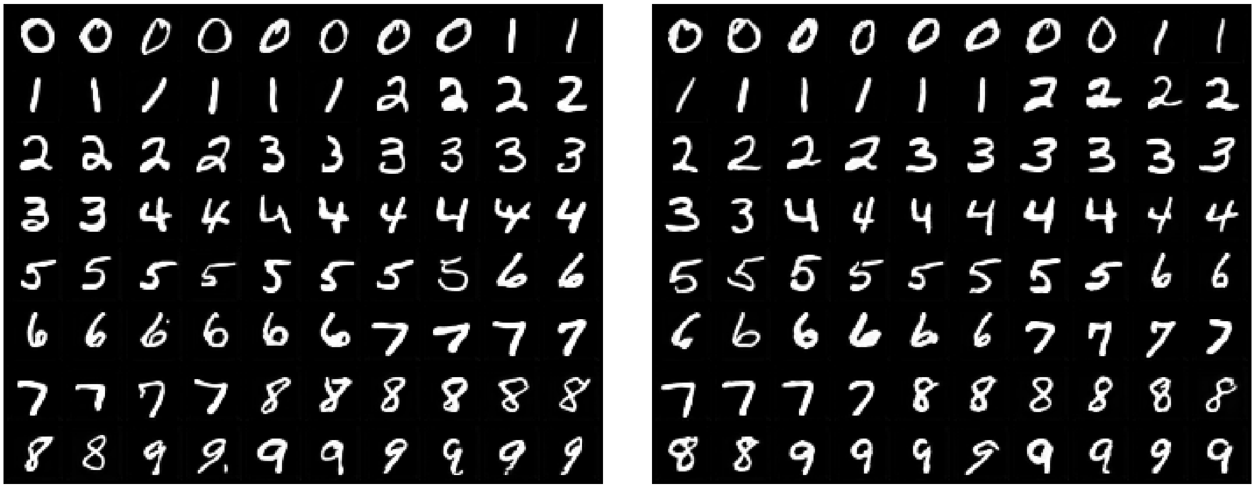
- **5.1 DDPM Sampling** (15 pts). Use `DDPMScheduler` (`prediction_type="sample"`) and its reverse-time indices to iterate and update `X <- ddpm.step(...).prev_sample`. Generate a 10×8 class grid. See Figure 4a.

- **5.2 DDIM Sampling** (15 pts). DDIM uses a shortened, often deterministic (η=0) trajectory and works with a DDPM-trained model. Use 50 steps, η=0, and the same CFG combination; generate the same class grid and compare speed/quality vs. DDPM. See Figure 4b.

Submission Instructions

1. Submit a Jupyter Notebook named `CSC8851_F2025_HW3_<YourName>.ipynb` with all code and plots (runs top-to-bottom).
2. Submit a PDF export named `CSC8851_F2025_HW3_<YourName>.pdf`. Do not paste code screenshots. **Make sure your PDF version has the figures**.
3. Upload both files (`.ipynb` and `.pdf`) to iCollege.

You may include additional qualitative results (e.g., failure cases). Cite any external sources used.



(a) DDPM + CFG ($s = 2.0$).　(b) DDIM(50, η=0) + CFG ($s = 2.0$).

Figure 4: Conditional sampling results (rows: digits 0–9; 8 samples each).

Appendix: DDIM Sampling Details and Speed

Setup. Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=0}^{t} \alpha_i$. Given a denoiser that predicts either $\hat{\mathbf{x}}_0(\mathbf{x}_t, t)$ or noise $\hat{\varepsilon}(\mathbf{x}_t, t)$, we can convert between the two via

$$\hat{\varepsilon}(\mathbf{x}_t, t) = \frac{\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\,\hat{\mathbf{x}}_0}{\sqrt{1-\bar{\alpha}_t}}, \qquad \hat{\mathbf{x}}_0(\mathbf{x}_t, t) = \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\hat{\varepsilon}}{\sqrt{\bar{\alpha}_t}}. \tag{1}$$

DDPM vs. DDIM (one-step updates)

DDPM (Markovian, stochastic). A common parameterization of the reverse step is

$$\mathbf{x}_{t-1}^{(\mathrm{DDPM})} = \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\hat{\varepsilon}(\mathbf{x}_t, t)\Big) \;+\; \sigma_t\,\mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(0, \mathbf{I}), \tag{2}$$

where σ_t^2 is chosen from the DDPM posterior variance family. This update is *Markovian* and typically run at all T steps (e.g., T=1000).

DDIM (non-Markovian, implicit). DDIM defines a family of trajectories that share the same single-step marginals as DDPM but allow skipping steps and optionally removing noise. For a chosen subsequence of indices

$$\mathcal{S} = \{\tau_1 > \tau_2 > \cdots > \tau_K\}, \qquad K \ll T,$$

the DDIM step from τ_k to τ_{k+1} is

$$\sigma_{\tau_k}^{(\mathrm{DDIM})} = \eta\sqrt{\frac{1-\bar{\alpha}_{\tau_{k+1}}}{1-\bar{\alpha}_{\tau_k}}}\sqrt{1-\frac{\bar{\alpha}_{\tau_k}}{\bar{\alpha}_{\tau_{k+1}}}}, \tag{3}$$

$$\mathbf{x}_{\tau_{k+1}}^{(\mathrm{DDIM})} = \sqrt{\bar{\alpha}_{\tau_{k+1}}}\,\hat{\mathbf{x}}_0 \;+\; \sqrt{1-\bar{\alpha}_{\tau_{k+1}} - (\sigma_{\tau_k}^{(\mathrm{DDIM})})^2}\;\hat{\varepsilon} \;+\; \sigma_{\tau_k}^{(\mathrm{DDIM})}\,\mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(0, \mathbf{I}), \tag{4}$$

with $\hat{\mathbf{x}}_0$ and $\hat{\varepsilon}$ evaluated at $(\mathbf{x}_{\tau_k}, \tau_k)$ and related by (1). The hyperparameter $\eta \in [0, 1]$ controls stochasticity.

Deterministic DDIM (fast default). Setting $\eta = 0$ yields

$$\boxed{\mathbf{x}_{\tau_{k+1}} = \sqrt{\bar{\alpha}_{\tau_{k+1}}}\,\hat{\mathbf{x}}_0 + \sqrt{1-\bar{\alpha}_{\tau_{k+1}}}\,\hat{\boldsymbol{\varepsilon}} \quad (\eta = 0).} \tag{5}$$

No per-step noise is injected; the trajectory becomes a deterministic mapping guided by the denoiser.

Where the speed-up comes from

- **Fewer model calls.** DDPM typically uses all T small steps. DDIM chooses a *coarse* timestep set $\mathcal{S}$ (e.g., K=50) and updates using (4). Complexity drops from $\mathcal{O}(T)$ to $\mathcal{O}(K)$ denoiser evaluations (e.g., $\sim 20\times$ fewer if K=50, T=1000).
- **Stable large steps.** Because the DDIM update directly re-targets the next marginal with the pair $(\bar{\alpha}_{\tau_k}, \bar{\alpha}_{\tau_{k+1}})$ and the predicted $(\hat{\mathbf{x}}_0, \hat{\boldsymbol{\varepsilon}})$, it tolerates larger jumps in t than the stochastic DDPM posterior step.
- **Optional determinism.** With η=0 (Eq. (5)), trajectories are noise-free and invertible in the ideal case, which improves consistency at small K and further reduces sampling variance.

DDPM Generation (Pseudocode)

```
Algorithm: DDPM Sampling (x0-prediction) with optional CFG

Inputs:
  - Trained model that predicts x0: x0_hat = model(Z, t).sample
  - Labels y (B,), image shape (B,1,H,W) with H=W=32 for MNIST
  - Full timestep set S_full = {T-1, ..., 0}  (usually T=1000)
  - Guidance scale s >= 0 (use s=0 to disable CFG)
  - Precomputed alpha[t], alpha_bar[t], and posterior variance tilde_beta[t]

Initialize:
  X <- Normal(0, I)         # X has shape (B, 1, H, W)

for each t in S_full:       # reverse order: T-1, T-2, ..., 0

    # ----- (A) Predict x0 (conditional + unconditional for CFG) -----
    Zc <- concat(X, class_maps(y))                 # (B, 11, H, W)
    x0_cond <- model(Zc, t).sample                 # \hat{x}_0^{cond}

    Zu <- concat(X, zeros_like(class_maps(y)))     # unconditional input
    x0_uncond <- model(Zu, t).sample               # \hat{x}_0^{uncond}

    x0_hat <- x0_cond + s * (x0_cond - x0_uncond)  # CFG combine (s=0 disables)

    # ----- (B) Convert \hat{x}_0 -> predicted noise \hat{\epsilon} (optional) -----
    eps_hat <- (X - sqrt(alpha_bar[t]) * x0_hat) / sqrt(1 - alpha_bar[t])
```

```
    # ----- (C) DDPM update (two equivalent forms) -----

    # Form 1: noise-parameterized mean + stochasticity
    mu = (1 / sqrt(alpha[t])) * (X - (1 - alpha[t]) / sqrt(1 - alpha_bar[t]) * eps_hat)
    sigma = sqrt(tilde_beta[t])                    # posterior std (DDPM original choice)
    X_prev = mu + sigma * Normal(0, I)

    # Form 2: x0/epsilon mixture (algebraically equivalent)
    # c = sqrt(1 - alpha_bar[t-1] - tilde_beta[t])   # define c for convenience
    # X_prev = sqrt(alpha_bar[t-1]) * x0_hat + c * eps_hat
    #            + sqrt(tilde_beta[t]) * Normal(0, I)

    X <- X_prev

# Output (denormalize as needed)
return X
```

Remarks.

- DDPM is *Markovian* and injects noise at every step via the posterior variance $\tilde{\beta}_t$ (or a scaled variant).
- In practice, we rarely hand-code μ and σ; the scheduler computes the reverse step given $\hat{x}_0$ (or $\hat{\epsilon}$) and the current sample.

DDIM Generation (Pseudocode)

```
Algorithm: DDIM Sampling (x0-prediction) with optional CFG

Inputs:
  - Trained model that predicts x0: x0_hat = model(Z, t).sample
  - Labels y (B,), image shape (B, 1, H, W) with H=W=32 for MNIST
  - Short timestep set S = {tau_1 > tau_2 > ... > tau_K}, K << T (e.g., K=50)
  - Guidance scale s >= 0 (use s=0 to disable CFG)
  - DDIM stochasticity eta in [0, 1] (use eta=0 for deterministic DDIM)
  - Precomputed alpha_bar[t] for all t (cumulative product of alphas)

Initialize:
  X <- Normal(0, I)           # X has shape (B, 1, H, W)

for k = 1 to K-1:
    t      <- tau_k
    t_prev <- tau_{k+1}

    # ----- (A) Predict x0 (conditional + unconditional for CFG) -----
```

```
    Zc <- concat(X, class_maps(y))                # (B, 11, H, W)
    x0_cond <- model(Zc, t).sample                # \hat{x}_0^{cond}

    Zu <- concat(X, zeros_like(class_maps(y)))    # unconditional input
    x0_uncond <- model(Zu, t).sample              # \hat{x}_0^{uncond}

    x0_hat <- x0_cond + s * (x0_cond - x0_uncond) # CFG combine (s=0 disables)

    # ----- (B) Convert \hat{x}_0 to predicted noise \hat{\epsilon} -----
    eps_hat <- (X - sqrt(alpha_bar[t]) * x0_hat) / sqrt(1 - alpha_bar[t])

    # ----- (C) DDIM variance and update -----
    # sigma controls stochasticity of the step (DDIM general form):
    sigma = eta
            * sqrt( (1 - alpha_bar[t_prev]) / (1 - alpha_bar[t]) )
            * sqrt( 1 - alpha_bar[t] / alpha_bar[t_prev] )

    c = sqrt( 1 - alpha_bar[t_prev] - sigma^2 )

    # DDIM step from t -> t_prev:
    X_prev = sqrt(alpha_bar[t_prev]) * x0_hat
             + c * eps_hat
             + sigma * Normal(0, I)              # drop this term if eta=0

    X <- X_prev

# Output (denormalize for viewing as needed):
return X
```

Using `diffusers`: DDPM vs. DDIM (x_0-prediction)

DDPM (full steps).

```
# Assume: model predicts x0 (prediction_type="sample")
ddpm.set_timesteps(ddpm.config.num_train_timesteps)   # typically T=1000
X = Normal(0, I)                                      # (B,1,32,32)
for t in ddpm.timesteps:
    Zc = concat(X, class_maps(y))
    Zu = concat(X, zeros_like(class_maps(y)))
    x0_c = model(Zc, t).sample
    x0_u = model(Zu, t).sample
    x0_cfg = x0_c + s * (x0_c - x0_u)                # CFG combine
    step = ddpm.step(model_output=x0_cfg, timestep=t, sample=X)
    X = step.prev_sample
# X is the generated image batch
```

DDIM (short trajectory, optional determinism).

```
# Use far fewer steps (e.g., K=50) and optionally set eta=0 for deterministic DDIM
ddim.set_timesteps(K)                                  # K << T, e.g., 50
X = Normal(0, I)                                       # (B,1,32,32)
for t in ddim.timesteps:
    Zc = concat(X, class_maps(y))
    Zu = concat(X, zeros_like(class_maps(y)))
    x0_c = model(Zc, t).sample
    x0_u = model(Zu, t).sample
    x0_cfg = x0_c + s * (x0_c - x0_u)                 # CFG combine
    step = ddim.step(model_output=x0_cfg, timestep=t, sample=X, eta=0.0)
    X = step.prev_sample
# X is the generated image batch
```

Key differences.

- **DDPM:** Markovian, stochastic reverse process, typically run for all T steps; higher compute cost.
- **DDIM:** Non-Markovian implicit update; can jump across a subset of timesteps $K \ll T$ and be deterministic (η=0), giving a large speed-up with comparable quality.

Notes.

- **Difference vs. DDPM.** DDPM uses a Markovian posterior update with per-step noise and is typically run for all T steps. DDIM uses a *non-Markovian* implicit update that can jump across a *subset* of steps $\mathcal{S}$ (e.g., 50) and can be *deterministic* for $\eta = 0$.
- **Speed-up.** Reducing the number of denoiser calls from T to K yields $\mathcal{O}(T/K)$ speed-up (e.g., $1000 \to 50$ gives $\sim 20\times$ fewer steps) with competitive quality.
- **Disabling CFG.** Set $s = 0$ to use unconditional sampling (or provide only the conditional branch and set $x_0^{\text{uncond}} = 0$).
- **Shapes.** For MNIST: $X \in \mathbb{R}^{B\times1\times32\times32}$, class maps $(B, 10, 32, 32)$, model input $Z \in \mathbb{R}^{B\times11\times32\times32}$.