Final Project (Written)

Jiahe (Stephen) Ling

Theory I

1

Base case $(t=1)$:

By definition, $q_{1|0}(x_1 \mid x_0) = \mathcal{N}\big(x_1;\ \sqrt{\alpha_1}\,x_0,\ (1-\alpha_1)I\big)$.

Since $\bar{\alpha}_1 = \prod_{s=1}^{1} a_s = \alpha_1, \quad \mathbb{E}[x_1 \mid x_0] = \sqrt{\alpha_1}\,x_0 = \sqrt{\bar{\alpha}_1}\,x_0, \quad \mathrm{Cov}(x_1 \mid x_0) = (1-\alpha_1)\,I = (1-\bar{\alpha}_1)\,I$

Inductive step:

Inductive Hypothesis: For some $t-1$, $q_{t-1|0}(x_{t-1} \mid x_0) = \mathcal{N}\big(x_{t-1};\ \sqrt{\bar{\alpha}_{t-1}}\,x_0,\ (1-\bar{\alpha}_{t-1})\,I\big)$, with $\bar{\alpha}_{t-1} = \prod_{s=1}^{t-1}\alpha_s$.

By definition, we have $q_{t|t-1}(x_t \mid x_{t-1}) = \mathcal{N}\big(x_t;\ \sqrt{\alpha_t}\,x_{t-1},\ (1-\alpha_t)\,I\big)$.

By inductive hypothesis, we have $q_{t-1|0}(x_{t-1} \mid x_0) = \mathcal{N}\big(x_{t-1};\ \sqrt{\bar{\alpha}_{t-1}}\,x_0,\ (1-\bar{\alpha}_{t-1})\,I\big)$.

$$
\begin{aligned}
q_{t|0}(x_t \mid x_0) &= \int q_{t|t-1}(x_t \mid x_{t-1})\ q_{t-1|0}(x_{t-1} \mid x_0)\ dx_{t-1} \\
&= \int \mathcal{N}\left(x_t; \sqrt{\alpha_t}\,x_{t-1},\, (1-\alpha_t)I\right)\, \mathcal{N}\left(x_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\,x_0,\, (1-\bar{\alpha}_{t-1})I\right)\ dx_{t-1} \\
&= \mathcal{N}\Big(x_t\ ;\ \sqrt{\alpha_t}\,\sqrt{\bar{\alpha}_{t-1}}\,x_0\ ;\ \big[\alpha_t(1-\bar{\alpha}_{t-1}) + (1-\alpha_t)\big]\ I\Big) \\
&= \mathcal{N}\left(x_t;\ \sqrt{\bar{\alpha}_t}\,x_0,\ (1-\bar{\alpha}_t)\,I\right) \text{ where } \bar{\alpha}_t = \alpha_t\,\bar{\alpha}_{t-1} = \prod_{s=1}^{t} a_s
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{E}[x_t \mid x_0] &= \sqrt{\alpha_t}\ \mathbb{E}[x_{t-1} \mid x_0] = \sqrt{\alpha_t}\ \left(\sqrt{\bar{\alpha}_{t-1}}\,x_0\right) = \sqrt{\bar{\alpha}_t}\,x_0 \\
\mathrm{Cov}(x_t \mid x_0) &= \alpha_t\,\mathrm{Cov}(x_{t-1} \mid x_0)\ +\ (1-\alpha_t)\,I \\
&= \alpha_t\,(1-\bar{\alpha}_{t-1})\,I\ +\ (1-\alpha_t)\,I \\
&= \big(1-\alpha_t\bar{\alpha}_{t-1}\big)\,I \\
&= \big(1-\bar{\alpha}_t\big)I
\end{aligned}
$$

Thus, we have $q_{t|0}(x_t \mid x_0) = \mathcal{N}\big(x_t;\ \sqrt{\bar{\alpha}_t}\,x_0,\ (1-\bar{\alpha}_t)\,I\big)$.

Proof by induction, $q_{t|0}(x_t \mid x_0)$ is Gaussian with mean $\sqrt{\bar{\alpha}_t}\,x_0$ and covariance $\big(1-\bar{\alpha}_t\big)I_d$.

2

By Markov-chain marginalization identity, we have:

$$
\begin{aligned}
\int_{x_{t+1:T}} \prod_{s=t+1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\,dx_{t+1:T} &= \int_{x_{t+1}} q_{t+1|t}(x_{t+1} \mid x_t)\Big[\int_{x_{t+2:T}} \prod_{s=t+2}^{T} q_{s|s-1}(x_s \mid x_{s-1})\,dx_{t+2:T}\Big]\,dx_{t+1} \\
&= \int_{x_{t+1}} q_{t+1|t}(x_{t+1} \mid x_t)\cdot 1\,dx_{t+1} \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
\int_{x_{1:t-2}} \prod_{s=1}^{t-1} q_{s|s-1}(x_s \mid x_{s-1})\,dx_{1:t-2} &= \int_{x_{1:t-2}} \left[\prod_{s=1}^{t-2} q_{s|s-1}(x_s \mid x_{s-1})\right] q_{\,t-1|t-2}(x_{\,t-1} \mid x_{\,t-2})\,dx_{1:t-2} \\
&= q_{\,t-1|0}(x_{\,t-1} \mid x_0)
\end{aligned}
$$

Using the results derived above,

$$
\begin{aligned}
L(\theta, x_0) &= \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log \frac{\Big[\prod_{s=1}^{T} p_{\,s-1|s}(x_{s-1} \mid x_s; \theta)\Big] \, p_T(x_T)}{\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})} \, dx_{1:T} \\
&= \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \Big[\sum_{t=1}^{T} \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \; + \; \log p_T(x_T)\Big] \, dx_{1:T} \\
&= \sum_{t=1}^{T} \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{1:T} \\
&\quad + \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log p_T(x_T) \, dx_{1:T} \\
&= \sum_{t=1}^{T} \int_{x_{1:T}} \big(\prod_{s=1}^{t-1} q_{s|s-1}(x_s \mid x_{s-1})\big) \, q_{t|t-1}(x_t \mid x_{t-1}) \, \big(\prod_{s=t+1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\big) \\
&\qquad\qquad \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{1:T} \\
&\quad + \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log p_T(x_T) \, dx_{1:T} \\
&= \sum_{t=1}^{T} \Big[\int_{x_{t+1:T}} \prod_{s=t+1}^{T} q_{s|s-1}(x_s \mid x_{s-1}) \, dx_{t+1:T}\Big] \int_{x_{1:t}} \big(\prod_{s=1}^{t-1} q_{s|s-1}(x_s \mid x_{s-1})\big) \, q_{t|t-1}(x_t \mid x_{t-1}) \\
&\qquad\qquad \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{1:t} \\
&\quad + \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log p_T(x_T) \, dx_{1:T} \\
&= \sum_{t=1}^{T} \int_{x_{1:t}} \big(\prod_{s=1}^{t-1} q_{s|s-1}(x_s \mid x_{s-1})\big) \, q_{t|t-1}(x_t \mid x_{t-1}) \; \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{1:t} \\
&\quad + \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log p_T(x_T) \, dx_{1:T} \\
&= \sum_{t=1}^{T} \int_{x_{t-1}, x_t} \Big[\int_{x_{1:t-2}} \prod_{s=1}^{t-1} q_{s|s-1}(x_s \mid x_{s-1}) \, dx_{1:t-2}\Big] \, q_{t|t-1}(x_t \mid x_{t-1}) \\
&\qquad\qquad \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{t-1} \, dx_t \\
&\quad + \int_{x_{1:T}} \Big[\prod_{s=1}^{T} q_{s|s-1}(x_s \mid x_{s-1})\Big] \, \log p_T(x_T) \, dx_{1:T} \\
&= \sum_{t=1}^{T} \int q_{t-1|0}(x_{t-1} \mid x_0) \, q_{t|t-1}(x_t \mid x_{t-1}) \; \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} \, dx_{t-1} \, dx_t \\
&\quad + \int q_T(x_T \mid x_0) \, \log p_T(x_T) \, dx_T
\end{aligned}
$$

Thus, $\boxed{L(\theta, x_0) = \sum_{t=1}^{T} \int q_{t-1|0}(x_{t-1} \mid x_0) q_{t|t-1}(x_t \mid x_{t-1}) \log \frac{p_{\,t-1|t}(x_{t-1} \mid x_t; \theta)}{q_{\,t|t-1}(x_t \mid x_{t-1})} dx_{t-1} \, dx_t + \int q_T(x_T \mid x_0) \, \log p_T(x_T) dx_T}$.

3

$$
\begin{aligned}
L(\theta, x_0) &= \sum_{t=1}^{T} \int_{x_{t-1}, x_t} - \log p_{\,t-1|t}(x_{t-1} \mid x_t; \theta) \, q_{\,t-1|0}(x_{t-1} \mid x_0) \, q_{\,t|t-1}(x_t \mid x_{t-1}) \, dx_{t-1} \, dx_t \\
&= \sum_{t=1}^{T} \int_{x_{t-1}, x_t} \Big[- \log\big((2\pi(1-\alpha_t))^{-d/2} \exp\big(-\tfrac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)}\big)\big)\Big] \\
&\qquad\qquad q_{\,t-1|0}(x_{t-1} \mid x_0) \, q_{\,t|t-1}(x_t \mid x_{t-1}) \, dx_{t-1} \, dx_t \\
&= \sum_{t=1}^{T} \int_{x_{t-1}, x_t} \Big[\tfrac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)} + \tfrac{d}{2} \log\big(2\pi(1-\alpha_t)\big)\Big] \\
&\qquad\qquad q_{\,t-1|0}(x_{t-1} \mid x_0) \, q_{\,t|t-1}(x_t \mid x_{t-1}) \, dx_{t-1} \, dx_t \\
&= \sum_{t=1}^{T} \int_{x_{t-1}, x_t} \tfrac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)} \; q_{\,t-1|0}(x_{t-1} \mid x_0) \, q_{\,t|t-1}(x_t \mid x_{t-1}) \, dx_{t-1} \, dx_t \; + \; \underbrace{\sum_{t=1}^{T} \tfrac{d}{2} \log\big(2\pi(1-\alpha_t)\big)}_{C} \\
&= \sum_{t=1}^{T} \int q_{\,t-1|0}(x_{t-1} \mid x_0) \, q_{\,t|t-1}(x_t \mid x_{t-1}) \tfrac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)} \, dx_{t-1} \, dx_t \; + \; C \\
&= \sum_{t=1}^{T} \mathbb{E}_{\substack{X_{t-1} \sim q_{\,t-1|0}(\cdot \mid x_0) \\ X_t \sim q_{\,t|t-1}(\cdot \mid X_{t-1})}} \Big[\tfrac{\|X_{t-1} - \mu_\theta(X_t, t)\|^2}{2(1-\alpha_t)} \;\Big|\; X_0 = x_0\Big] \; + \; C
\end{aligned}
$$

In the first equality, we begin with the exact ELBO written as $L(\theta, X_0) = \sum_{t=1}^{T} \int \log \frac{p_\theta(x_{t-1} \mid x_t)}{q(x_t \mid x_{t-1})} \, q_{t-1,t|0}(x_{t-1}, x_t \mid X_0) \, dx_{t-1} \, dx_t \; + \; \int q_T(x_T \mid X_0) \, \log p_T(x_T) \, dx_T$, where each reverse kernel $p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1};\, \mu_\theta(x_t, t),\, (1-\alpha_t)I)$, and each forward kernel $q(x_t \mid x_{t-1})$ is fixed. Substituting the Gaussian density into $-\log p_\theta$ yields $-\log p_\theta(x_{t-1} \mid x_t) = \frac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)} \; + \; \frac{d}{2} \log\big(2\pi(1-\alpha_t)\big)$. Because the second term is independent of θ, it can be summed over t into a single constant C, leaving exactly the mean–squared-error term in the integrand. This delivers $L(\theta, X_0) = \sum_{t=1}^{T} \int \frac{\|x_{t-1} - \mu_\theta(x_t, t)\|^2}{2(1-\alpha_t)} \, q_{t-1,t|0}(x_{t-1}, x_t \mid X_0) \, dx_{t-1} \, dx_t \; + \; C$.

In the second equality, we recognize that each such integral is simply an expectation under the known forward joint distribution $q_{t-1,t|0}$. By definition, $\int f(x_{t-1}, x_t) \, q_{t-1,t|0}(x_{t-1}, x_t \mid X_0) \, dx_{t-1} \, dx_t = \mathbb{E}_{q_{t-1,t|0}}\big[f(X_{t-1}, X_t) \mid X_0\big]$, so that the loss can be compactly written as $L(\theta, X_0) = \sum_{t=1}^{T} \mathbb{E}_{q_{t-1,t|0}}\Big[\frac{\|X_{t-1} - \mu_\theta(X_t, t)\|^2}{2(1-\alpha_t)} \;\Big|\; X_0\Big] \; + \; C$.

Intuitively, the first equality shows that training the reverse process amounts to minimizing an MSE between the network's prediction $\mu_\theta(x_t, t)$ and the true denoised sample x_{t-1}, with a fixed noise-level–dependent variance $1 - \alpha_t$. The second equality highlights that we can implement this by simply sampling (X_{t-1}, X_t) from the forward diffusion and computing its squared-error, making training both straightforward (no intractable integrals) and stable (a well-behaved regression objective).

4

Given $X_0 \approx q_0(x) = f(x)$, e would estimate each expectation in the sum by Monte Carlo method that $\hat{L} = \sum_{t=1}^{T} \frac{1}{M} \hat{L}_t \approx \sum_{t=1}^{T} \mathbb{E}_{q_{t-1,t|0}}\Big[\frac{\|X_{t-1}-\mu_\theta(X_t,t)\|^2}{2(1-\alpha_t)} \;\Big|\; X_0\Big]$. The detailed algorithm is described below.

To obtain $\mathbb{E}_{q_{t-1,t|0}}\Big[\frac{\|X_{t-1}-\mu_\theta(X_t,t)\|^2}{2(1-\alpha_t)} \;\Big|\; X_0\Big]$, we also need to derive $q_{t-1,t|0}$.

$$
\begin{aligned}
q(x_{t-1} \mid x_0) &= \mathcal{N}\big(x_{t-1};\ \sqrt{\bar\alpha_{t-1}}\,x_0,\ (1-\bar\alpha_{t-1})I\big),\\
q(x_t \mid x_{t-1}) &= \mathcal{N}\big(x_t;\ \sqrt{\alpha_t}\,x_{t-1},\ (1-\alpha_t)I\big),\\
q(x_{t-1} \mid x_t, x_0) &\propto q(x_t \mid x_{t-1})\,q(x_{t-1} \mid x_0)\\
&\propto \exp\Big(-\tfrac{\|x_{t-1}-\sqrt{\bar\alpha_{t-1}}x_0\|^2}{2(1-\bar\alpha_{t-1})} - \tfrac{\|x_t-\sqrt{\alpha_t}x_{t-1}\|^2}{2(1-\alpha_t)}\Big)\\
&\propto \exp\Big(-\tfrac12\,x_{t-1}^T\big[\tfrac{1}{1-\bar\alpha_{t-1}} + \tfrac{\alpha_t}{1-\alpha_t}\big]\,x_{t-1} + x_{t-1}^T\big[\tfrac{\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_{t-1}}x_0 + \tfrac{\sqrt{\alpha_t}}{1-\alpha_t}x_t\big]\Big)\\
&= \mathcal{N}\big(x_{t-1};\ \tilde\mu_t(x_t,x_0),\ \tilde\beta_t I\big),\\
\tilde\beta_t &= \frac{1-\bar\alpha_{t-1}}{1-\bar\alpha_t}\,(1-\alpha_t),\\
\tilde\mu_t(x_t,x_0) &= \frac{\sqrt{\bar\alpha_{t-1}}\,(1-\alpha_t)}{1-\bar\alpha_t}\,x_0 + \frac{\sqrt{\alpha_t}\,(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}\,x_t
\end{aligned}
$$

Precompute:

- $\bar\alpha_t \;=\; \prod_{s=1}^{t}\alpha_s \quad \text{for } t=1,\dots,T$
- $\tilde\beta_t = \frac{1-\bar\alpha_{t-1}}{1-\bar\alpha_t}\,(1-\alpha_t)$
- $\tilde\mu_t(x_t,x_0) = \frac{\sqrt{\bar\alpha_{t-1}}\,(1-\alpha_t)}{1-\bar\alpha_t}\;x_0 + \frac{\sqrt{\alpha_t}\,(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}\;x_t$

Compute:

1. Initialize total loss estimate: $\hat{L} \;\leftarrow\; 0$
2. **For** $t = 1,2,\dots,T$:

   A. Set $\hat{L}_t \;\leftarrow\; 0$

   B. **For** $i = 1,2,\dots,M$:

      a. Sample noise: $\varepsilon^{(i)} \sim \mathcal{N}(0,I)$
      b. Compute a noisy point: $x_t^{(i)} = \sqrt{\bar\alpha_t}\;x_0 + \sqrt{1-\bar\alpha_t}\;\varepsilon^{(i)}$, so that $x_t^{(i)} \sim q(x_t \mid x_0)$, since $q(x_t \mid x_0) = \mathcal{N}(\sqrt{\bar\alpha_t}\,x_0,\ (1-\bar\alpha_t)I)$.
      c. Sample the backward step: $x_{t-1}^{(i)} \sim q(x_{t-1} \mid x_t^{(i)}, x_0) = \mathcal{N}\big(\tilde\mu_t(x_t^{(i)},x_0),\ \tilde\beta_t I\big)$
      d. Compute the squared-error loss: $\ell_t^{(i)} \;=\; \frac{\big\|\,x_{t-1}^{(i)}-\mu_\theta(x_t^{(i)},t)\big\|^2}{2\,(1-\alpha_t)}$
      e. Accumulate: $\hat{L}_t \;\leftarrow\; \hat{L}_t + \ell_t^{(i)}$

   C. Average over the M samples and add to the total: $\hat{L} \;\leftarrow\; \hat{L} + \frac{1}{M}\,\hat{L}_t$

Return: $\hat{L} = \sum_{t=1}^{T} \frac{1}{M}\,\hat{L}_t \approx \sum_{t=1}^{T} \mathbb{E}_{q_{t-1,t|0}}\Big[\frac{\|X_{t-1}-\mu_\theta(X_t,t)\|^2}{2(1-\alpha_t)} \;\Big|\; X_0\Big]$

Theory II

1

$$
\begin{aligned}
q(x_{t-1} \mid x_t, x_0) \;&\propto\; q(x_{t-1} \mid x_0)\;q(x_t \mid x_{t-1})\\
&\propto\; \exp\Big(-\tfrac12\,(x_{t-1}-\sqrt{\bar\alpha_{t-1}}\,x_0)^\top \frac{I}{1-\bar\alpha_{t-1}}(x_{t-1}-\sqrt{\bar\alpha_{t-1}}\,x_0) - \tfrac12\,(x_t-\sqrt{\alpha_t}\,x_{t-1})^\top \frac{I}{1-\alpha_t}(x_t-\sqrt{\alpha_t}\,x_{t-1})\Big)\\
&\propto\; \exp\Big(-\tfrac12\,x_{t-1}^\top\Big[\tfrac{1}{1-\bar\alpha_{t-1}} + \tfrac{\alpha_t}{1-\alpha_t}\Big]x_{t-1} + x_{t-1}^\top\Big[\tfrac{\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_{t-1}}\,x_0 + \tfrac{\sqrt{\alpha_t}}{1-\alpha_t}\,x_t\Big]\Big)\\
&= \exp\Big(-\tfrac12\,x_{t-1}^\top A\,x_{t-1} + \,x_{t-1}^\top b\Big)\\
A &= \frac{1}{1-\bar\alpha_{t-1}}I + \frac{\alpha_t}{1-\alpha_t}I = \frac{1-\bar\alpha_t}{(1-\bar\alpha_{t-1})(1-\alpha_t)}\,I\\
b &= \frac{\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_{t-1}}\,x_0 + \frac{\sqrt{\alpha_t}}{1-\alpha_t}\,x_t\\
-\tfrac12\,x_{t-1}^\top A\,x_{t-1} + x_{t-1}^\top b &= -\tfrac12\,(x_{t-1}-A^{-1}b)^\top A\,(x_{t-1}-A^{-1}b) + \tfrac12\,b^\top A^{-1}b\\
A^{-1} &= \frac{(1-\bar\alpha_{t-1})(1-\alpha_t)}{1-\bar\alpha_t}\,I = \rho_t\,I \to \rho_t = \frac{(1-\alpha_t)(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}\\
\mu &= A^{-1}b = \rho_t\Big[\tfrac{\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_{t-1}}\,x_0 + \tfrac{\sqrt{\alpha_t}}{1-\alpha_t}\,x_t\Big]\\
&= \frac{(1-\alpha_t)\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_t}\,x_0 + \frac{(1-\bar\alpha_{t-1})\sqrt{\alpha_t}}{1-\bar\alpha_t}\,x_t = \tilde\mu_t(x_t,x_0)\\
q(x_{t-1} \mid x_t, x_0) &= \mathcal{N}\big(x_{t-1};\ \tilde\mu_t(x_t,x_0),\ \rho_t\,I\big)
\end{aligned}
$$

Thus, $\boxed{\tilde\mu_t(x_t,x_0) = \frac{(1-\alpha_t)\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_t}\,x_0 + \frac{(1-\bar\alpha_{t-1})\sqrt{\alpha_t}}{1-\bar\alpha_t}\,x_t, \quad \rho_t = \frac{(1-\alpha_t)(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}}$.

2

$$\int_{x_{t-1},x_t} \frac{\|x_{t-1}-\mu(x_t,t)\|^2}{2(1-\alpha_t)}\, q_{t-1,t|0}(x_{t-1},x_t \mid X_0)\, dx_{t-1}\, dx_t$$

$$= \int_{x_t} \frac{1}{2(1-\alpha_t)} \left(\int_{x_{t-1}} \|x_{t-1}-\mu(x_t,t)\|^2\, q_{t-1|0}(x_{t-1} \mid x_t, X_0)\, dx_{t-1} \right) q_t(x_t \mid X_0)\, dx_t$$

$$= \int_{x_t} \frac{1}{2(1-\alpha_t)} \left(\int_{x_{t-1}} \big\|(x_{t-1}-\tilde{\mu}_t)+(\tilde{\mu}_t-\mu)\big\|^2\, q_{t-1|0}(x_{t-1} \mid x_t, X_0)\, dx_{t-1} \right) q_t(x_t \mid X_0)\, dx_t$$

$$= \int_{x_t} \frac{1}{2(1-\alpha_t)} \left(\underbrace{\int \|x_{t-1}-\tilde{\mu}_t\|^2\, q_{t-1|0}\, dx_{t-1}}_{=\rho_t} \;+\; 2(\tilde{\mu}_t-\mu)^\top \underbrace{\int (x_{t-1}-\tilde{\mu}_t)\, q_{t-1|0}\, dx_{t-1}}_{=0} \;+\; \|\tilde{\mu}_t-\mu\|^2 \underbrace{\int q_{t-1|0}\, dx_{t-1}}_{=1} \right) q_t(x_t \mid X_0)\, dx_t$$

$$= \int_{x_t} \frac{\rho_t + \|\tilde{\mu}_t(x_t,X_0)-\mu(x_t,t)\|^2}{2(1-\alpha_t)}\, q_t(x_t \mid X_0)\, dx_t$$

$$= \mathbb{E}_{x_t\sim q_t(\cdot|X_0)} \left[\frac{\|\tilde{\mu}_t(X_t,X_0)-\mu(X_t,t)\|^2+\rho_t}{2(1-\alpha_t)} \;\Big|\; X_0 \right]$$

3

$$\tilde{\mu}_t(x_t,x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\,(1-\alpha_t)}{1-\bar{\alpha}_t}\, x_0 \;+\; \frac{\sqrt{\alpha_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\, x_t, \quad x_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon_t}{\sqrt{\bar{\alpha}_t}}$$

$$\begin{aligned}
\tilde{\mu}_t(x_t,x_0) &= \frac{\sqrt{\bar{\alpha}_{t-1}}\,(1-\alpha_t)}{1-\bar{\alpha}_t}\; \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon_t}{\sqrt{\bar{\alpha}_t}} + \frac{\sqrt{\alpha_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\, x_t \\
&= \frac{1}{\sqrt{\alpha_t}}\; \frac{1-\alpha_t}{1-\bar{\alpha}_t}\, x_t - \frac{1}{\sqrt{\alpha_t}} \frac{(1-\alpha_t)\sqrt{1-\bar{\alpha}_t}}{1-\bar{\alpha}_t}\, \epsilon_t + \frac{\alpha_t(1-\bar{\alpha}_{t-1})}{\sqrt{\alpha_t}(1-\bar{\alpha}_t)}\, x_t \\
&= \frac{1}{\sqrt{\alpha_t}} \Big(\frac{1-\alpha_t+\alpha_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} \Big) x_t - \frac{1-\alpha_t}{\sqrt{\alpha_t}\,\sqrt{1-\bar{\alpha}_t}}\, \epsilon_t
\end{aligned}$$

$$1-\alpha_t+\alpha_t(1-\bar{\alpha}_{t-1}) = 1-\alpha_t\bar{\alpha}_{t-1} = 1-\bar{\alpha}_t$$

$$\begin{aligned}
\tilde{\mu}_t(x_t,x_0) &= \frac{1}{\sqrt{\alpha_t}}\, x_t - \frac{1-\alpha_t}{\sqrt{\alpha_t}\,\sqrt{1-\bar{\alpha}_t}}\, \epsilon_t \\
&= \frac{1}{\sqrt{\alpha_t}} \Big(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\, \epsilon_t \Big)
\end{aligned}$$

Thus, $\boxed{\tilde{\mu}_t(x_t,x_0) = \frac{1}{\sqrt{\alpha_t}} \Big(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\, \epsilon_t \Big)}$.

4

$$\begin{aligned}
L\big(\theta,X_0\big) &= \sum_{t=1}^{T} \mathbb{E}_{\varepsilon_t\sim\mathcal{N}(0,I)} \Big[\frac{\big\|\tilde{\mu}\big(X_t,X_0\big)-\mu\big(X_t,t;\theta\big)\big\|^2}{2\big(1-\alpha_t\big)} \Big] \\
\tilde{\mu}\big(X_t,X_0\big) &= \frac{1}{\sqrt{\alpha_t}} \Big[X_t \;-\; \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\, \varepsilon_t \Big] \\
\mu\big(X_t,t;\theta\big) &= \frac{1}{\sqrt{\alpha_t}} \Big[X_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} e_t\big(X_t;\theta\big) \Big] \\
\tilde{\mu}\big(X_t,X_0\big)-\mu\big(X_t,t;\theta\big) &= \frac{1}{\sqrt{\alpha_t}} \Big[-\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \big(\varepsilon_t - e_t(X_t;\theta)\big) \Big] \\
&= -\frac{1-\alpha_t}{\sqrt{\alpha_t}\,\sqrt{1-\bar{\alpha}_t}}\, \big(\varepsilon_t - e_t(X_t;\theta)\big) \\
\big\|\tilde{\mu}(X_t,X_0)-\mu(X_t,t;\theta)\big\|^2 &= \frac{(1-\alpha_t)^2}{\alpha_t\,(1-\bar{\alpha}_t)} \big\|\varepsilon_t - e_t(X_t;\theta)\big\|^2 \\
\frac{\big\|\tilde{\mu}(X_t,X_0)-\mu(X_t,t;\theta)\big\|^2}{2(1-\alpha_t)} &= \frac{1-\alpha_t}{2\,\alpha_t\,(1-\bar{\alpha}_t)} \big\|\varepsilon_t - e_t(X_t;\theta)\big\|^2 \\
L\big(\theta,X_0\big) &= \sum_{t=1}^{T} \mathbb{E}_{\varepsilon_t\sim\mathcal{N}(0,I)} \Big[\frac{\big\|\tilde{\mu}\big(X_t,X_0\big)-\mu\big(X_t,t;\theta\big)\big\|^2}{2\big(1-\alpha_t\big)} \Big] \\
&= \sum_{t=1}^{T} \mathbb{E}_{\varepsilon_t\sim\mathcal{N}(0,I)} \Big[\tfrac{1-\alpha_t}{2\,\alpha_t\,(1-\bar{\alpha}_t)} \big\|\varepsilon_t - e_t(X_t;\theta)\big\|^2 \Big]
\end{aligned}$$

Thus, $\boxed{L\big(\theta,X_0\big) = \sum_{t=1}^{T} \mathbb{E}_{\varepsilon_t\sim\mathcal{N}(0,I)} \Big[\tfrac{1-\alpha_t}{2\,\alpha_t\,(1-\bar{\alpha}_t)} \big\|\varepsilon_t - e_t(X_t;\theta)\big\|^2 \Big]}$.

Final Project (Coding)

Jiahe (Stephen) Ling

In [1]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader, Subset, random_split
from torchvision import datasets, transforms
from tqdm.notebook import trange, tqdm
import numpy as np
from numpy.linalg import eigh
import random
import matplotlib.pyplot as plt
import sys

SEED = 37601
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)
SHOW_TQDM = sys.stdout.isatty()
```

Coding I

1

There are 4 General Forward-Pass Stages in the Network

1. **Time-Step Embedding**: Encodes the scalar noise index $t \in \mathbb{R}$ into a high-dimensional vector that conditions every convolutional layer. $\phi_F(t) = \left[\sin(2\pi W t),\ \cos(2\pi W t)\right] \ \in \mathbb{R}^d$, where $W \in \mathbb{R}^{d/2}$ is fixed at init. Then apply a learned affine map and activation: $z = \mathrm{Linear}\big(\phi_F(t)\big)$, $\mathrm{embed} = z \odot \sigma(z)$, where σ is the logistic sigmoid.

2. **Encoder (Down-sampling)**: Applies four convolutional blocks that extract features at progressively coarser scales, each shifting activations by the time embedding.
$$\begin{aligned} A_i &= \mathrm{Conv}_i(h_{i-1}) \\ B_i(t) &= \mathrm{Dense}_i\big(\mathrm{embed}\big) \\ \tilde{h}_i &= A_i + B_i(t), \\ \hat{h}_i &= \mathrm{GroupNorm}_i(\tilde{h}_i) \\ h_i &= \hat{h}_i \ \odot \ \sigma(\hat{h}_i) \end{aligned}$$

3. **Decoder (Up-sampling)**: Reconstructs full resolution via transpose-convolutions, fusing encoder skip features and again injecting the same time bias. Let $u_4 = h_4$:
$$\begin{aligned} &\text{Let } u_4 = h_4 \\ U_j &= \begin{cases} \mathrm{TConv}_4(u_4) & j = 4 \\ \mathrm{TConv}_j\big[\mathrm{concat}(u_{j+1},\ h_{j+1})\big] & j < 4 \end{cases} \\ D_j(t) &= \mathrm{Dense}_{j+4}(\mathrm{embed}) \\ \tilde{u}_j &= U_j + D_j(t) \\ \hat{u}_j &= \mathrm{TGroupNorm}_{j+4}(\tilde{u}_j) \\ u_j &= \hat{u}_j \ \odot \ \sigma(\hat{u}_j) \end{aligned}$$

4. **Final Output**: Merges the final decoder map with the first encoder feature for fine detail, then predicts the single-channel denoised image. $\mathrm{out} = \mathrm{TConv}_1\big[\mathrm{concat}(u_1,\ h_1)\big]$

At each forward pass, the scalar timestep `t` is first mapped through a fixed Gaussian-Fourier feature transform and a small linear layer into a high-dimensional vector `embed`. That same `embed` vector is then added as a channel-wise bias before normalization in every convolutional and transpose-convolutional block. By shifting each layer's activations in a learnable, time-dependent way, the network effectively knows how noisy its input is and adapts its filters and feature-statistics to remove the correct amount of noise for step `t`. This per-layer injection of `t` ensures a single U-Net can denoise all timesteps $1 \cdots T$ simply by reading off the time embedding.

In [2]:
```python
class GaussianFourierProjection(nn.Module):
  """Gaussian random features for encoding time steps."""
  def __init__(self, embed_dim, scale=30.):
    super().__init__()
    # Randomly sample weights during initialization. These weights are fixed
    # during optimization and are not trainable.
    self.W = nn.Parameter(torch.randn(embed_dim // 2) * scale, requires_grad=False)

  def forward(self, x):
    x = x.unsqueeze(-1).float() # MODIFIED TO FIX DIMENSION ERROR
    x_proj = x * self.W * 2 * np.pi
    return torch.cat([torch.sin(x_proj), torch.cos(x_proj)], dim=-1)


class Dense(nn.Module):
  """A fully connected layer that reshapes outputs to feature maps."""
  def __init__(self, input_dim, output_dim):
    super().__init__()
    self.dense = nn.Linear(input_dim, output_dim)
```

```python
  def forward(self, x):
    return self.dense(x)[..., None, None]


class ScoreNet(nn.Module):
  """A time-dependent score-based model built upon U-Net architecture."""

  def __init__(self, channels=[32, 64, 128, 256], embed_dim=256, group_num=4):

    super().__init__()

    self.embed = nn.Sequential(GaussianFourierProjection(embed_dim=embed_dim),
         nn.Linear(embed_dim, embed_dim))

    self.conv1 = nn.Conv2d(1, channels[0], 3, stride=1, bias=False)
    self.dense1 = Dense(embed_dim, channels[0])
    self.gnorm1 = nn.GroupNorm(group_num, num_channels=channels[0])
    self.conv2 = nn.Conv2d(channels[0], channels[1], 3, stride=2, bias=False)
    self.dense2 = Dense(embed_dim, channels[1])
    self.gnorm2 = nn.GroupNorm(group_num, num_channels=channels[1])
    self.conv3 = nn.Conv2d(channels[1], channels[2], 3, stride=2, bias=False)
    self.dense3 = Dense(embed_dim, channels[2])
    self.gnorm3 = nn.GroupNorm(group_num, num_channels=channels[2])
    self.conv4 = nn.Conv2d(channels[2], channels[3], 3, stride=2, bias=False)
    self.dense4 = Dense(embed_dim, channels[3])
    self.gnorm4 = nn.GroupNorm(group_num, num_channels=channels[3])

    self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], 3, stride=2, bias=False)
    self.dense5 = Dense(embed_dim, channels[2])
    self.tgnorm4 = nn.GroupNorm(group_num, num_channels=channels[2])
    self.tconv3 = nn.ConvTranspose2d(channels[2] + channels[2], channels[1], 3, stride=2, bias=False, output_padding=1)
    self.dense6 = Dense(embed_dim, channels[1])
    self.tgnorm3 = nn.GroupNorm(group_num, num_channels=channels[1])
    self.tconv2 = nn.ConvTranspose2d(channels[1] + channels[1], channels[0], 3, stride=2, bias=False, output_padding=1)
    self.dense7 = Dense(embed_dim, channels[0])
    self.tgnorm2 = nn.GroupNorm(group_num, num_channels=channels[0])
    self.tconv1 = nn.ConvTranspose2d(channels[0] + channels[0], 1, 3, stride=1)

    # The swish activation function
    self.act = lambda x: x * torch.sigmoid(x)

  def forward(self, x, t):
    # Obtain the Gaussian random feature embedding for t
    embed = self.act(self.embed(t))

    h1 = self.conv1(x) # ...
    h1 += self.dense1(embed) #...
    h1 = self.gnorm1(h1) # ...
    h1 = self.act(h1) # ...
    h2 = self.conv2(h1) # ...
    h2 += self.dense2(embed)
    h2 = self.gnorm2(h2)
    h2 = self.act(h2)
    h3 = self.conv3(h2)
    h3 += self.dense3(embed)
    h3 = self.gnorm3(h3)
    h3 = self.act(h3)
    h4 = self.conv4(h3)
    h4 += self.dense4(embed)
    h4 = self.gnorm4(h4)
    h4 = self.act(h4)

    h = self.tconv4(h4) # ...
    h += self.dense5(embed) # ...
    h = self.tgnorm4(h)
    h = self.act(h)
    h = self.tconv3(torch.cat([h, h3], dim=1)) # ...
    h += self.dense6(embed)
    h = self.tgnorm3(h)
    h = self.act(h)
    h = self.tconv2(torch.cat([h, h2], dim=1))
    h += self.dense7(embed)
    h = self.tgnorm2(h)
    h = self.act(h)
    h = self.tconv1(torch.cat([h, h1], dim=1))

    return h
```

2

In [3]:
```python
class ScoreNet(nn.Module):
  """A time-dependent score-based model built upon U-Net architecture."""

  def __init__(self, channels=[32, 64, 128, 256], embed_dim=256, group_num=4):

    super().__init__()

    self.embed = nn.Sequential(
      GaussianFourierProjection(embed_dim=embed_dim),
      nn.Linear(embed_dim, embed_dim))

    # add parameters for rho0 and rho1
    self.rho0 = nn.Parameter(torch.tensor(1.0))
    self.rho1 = nn.Parameter(torch.tensor(1.0))

    self.conv1 = nn.Conv2d(1, channels[0], 3, stride=1, bias=False)
```

```
    self.dense1 = Dense(embed_dim, channels[0])
    self.gnorm1 = nn.GroupNorm(group_num, num_channels=channels[0])
    self.conv2 = nn.Conv2d(channels[0], channels[1], 3, stride=2, bias=False)
    self.dense2 = Dense(embed_dim, channels[1])
    self.gnorm2 = nn.GroupNorm(group_num, num_channels=channels[1])
    self.conv3 = nn.Conv2d(channels[1], channels[2], 3, stride=2, bias=False)
    self.dense3 = Dense(embed_dim, channels[2])
    self.gnorm3 = nn.GroupNorm(group_num, num_channels=channels[2])
    self.conv4 = nn.Conv2d(channels[2], channels[3], 3, stride=2, bias=False)
    self.dense4 = Dense(embed_dim, channels[3])
    self.gnorm4 = nn.GroupNorm(group_num, num_channels=channels[3])

    self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], 3, stride=2, bias=False)
    self.dense5 = Dense(embed_dim, channels[2])
    self.tgnorm4 = nn.GroupNorm(group_num, num_channels=channels[2])
    self.tconv3 = nn.ConvTranspose2d(channels[2] + channels[2], channels[1], 3, stride=2, bias=False, output_padding=1)
    self.dense6 = Dense(embed_dim, channels[1])
    self.tgnorm3 = nn.GroupNorm(group_num, num_channels=channels[1])
    self.tconv2 = nn.ConvTranspose2d(channels[1] + channels[1], channels[0], 3, stride=2, bias=False, output_padding=1)
    self.dense7 = Dense(embed_dim, channels[0])
    self.tgnorm2 = nn.GroupNorm(group_num, num_channels=channels[0])
    self.tconv1 = nn.ConvTranspose2d(channels[0] + channels[0], 1, 3, stride=1)

    # Swish activation
    self.act = lambda x: x * torch.sigmoid(x)

  def forward(self, x, t):
    # Obtain the Gaussian random feature embedding for t
    embed = self.act(self.embed(t))

    # Ecoder (Simplified)
    h1 = self.act(self.gnorm1(self.conv1(x) + self.dense1(embed)))
    h2 = self.act(self.gnorm2(self.conv2(h1) + self.dense2(embed)))
    h3 = self.act(self.gnorm3(self.conv3(h2) + self.dense3(embed)))
    h4 = self.act(self.gnorm4(self.conv4(h3) + self.dense4(embed)))

    # Decoder (Simplified)
    h = self.act(self.tgnorm4(self.tconv4(h4) + self.dense5(embed)))
    h = self.act(self.tgnorm3(self.tconv3(torch.cat([h, h3], dim=1)) + self.dense6(embed)))
    h = self.act(self.tgnorm2(self.tconv2(torch.cat([h, h2], dim=1)) + self.dense7(embed)))

    delta = self.tconv1(torch.cat([h, h1], dim=1)) # original output
    mu = self.rho0 * (x - self.rho1 * delta) # modified mean

    return mu
```

3

In [4]:
```
class Diffusion(nn.Module):

    def  __init__(self, model, n_steps, device, min_beta, max_beta):
        super().__init__() # Store beta, alpha and \bar alpha
        self.model = model
        self.n_steps = n_steps
        self.device = device

        self.betas = torch.linspace(min_beta, max_beta, n_steps, device=device)
        self.alphas = 1.0 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)

        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - self.alphas_cumprod)

    def forward_process(self, x0, t): # Sample x_{t-1},x_t given x_0
        B, C, H, W = x0.shape
        alpha_bar_t = self.alphas_cumprod[t].view(B,1,1,1)
        alpha_bar_prev = self.alphas_cumprod[t-1].view(B,1,1,1)
        alpha_t = self.alphas[t].view(B,1,1,1)

        # sample x_t
        noise = torch.randn_like(x0)
        x_t = alpha_bar_t.sqrt() * x0 + (1 - alpha_bar_t).sqrt() * noise

        # compute posterior q(x_{t-1}|x_t,x0) parameters
        rho_t = (1 - alpha_t) * (1 - alpha_bar_prev) / (1 - alpha_bar_t) # posterior variance
        coef0 = alpha_bar_prev.sqrt() * (1 - alpha_t) / (1 - alpha_bar_t)
        coef1 = alpha_t.sqrt() * (1 - alpha_bar_prev) / (1 - alpha_bar_t)
        mu_tilde = coef0 * x0 + coef1 * x_t # posterior mean

        # sample x_{t-1}
        x_prev = mu_tilde + rho_t.sqrt() * torch.randn_like(x0)

        return x_prev, x_t


    def predict_next(self, xt, t):
        # Compute mu(xt,t)
        return self.model(xt, t)
```

4

In [5]:
```
def diffusion_loss(diffusion: Diffusion, x0: torch.Tensor):
    b, C, H, W = x0.shape
    T = diffusion.n_steps
    # sample a timestep for each example in the batch
```

```python
    t = torch.randint(1, T, (b,), device=x0.device)
    # run the forward noising and posterior sampling
    x_prev, x_t = diffusion.forward_process(x0, t)
    # model prediction of the mean
    mu_pred = diffusion.predict_next(x_t, t)
    # gather (1 − alpha_t) for each example and reshape to broadcast
    one_minus_alpha = (1.0 - diffusion.alphas[t]).view(b, 1, 1, 1)
    # compute per-example loss: |x_prev − mu_pred|^2 / (2 * (1 − alpha_t))
    per_example = (x_prev - mu_pred).pow(2).sum(dim=(1,2,3)) / (2.0 * one_minus_alpha.view(b))
    return per_example.mean()
```

5

We replace each per-example loss $L_{\text{sum}}(X_j,\theta)=\sum_{i=1}^{d}\frac{[X_{t_j-1,i}-\mu_i(X_{t_j},t_j;\theta)]^2}{2(1-\alpha_{t_j})}$ with $L_{\text{mean}}(X_j,\theta)=\frac{1}{d}\sum_{i=1}^{d}\frac{[X_{t_j-1,i}-\mu_i(X_{t_j},t_j;\theta)]^2}{2(1-\alpha_{t_j})}$, where $X\in\mathbb{R}^d$.

In other words, we divide the sum-of-squares by d to obtain a mean-squared-error (MSE), which reduces the estimator's variance by a factor of $1/d$.

Proof:

$$Var\big(L_{\text{sum}}(X_j,\theta)\big)=d\,\sigma^2,$$
$$Var\big(L_{\text{mean}}(X_j,\theta)\big)=Var\Big(\tfrac{1}{d}L_{\text{sum}}(X_j,\theta)\Big)=\tfrac{1}{d^2}\,Var\big(L_{\text{sum}}(X_j,\theta)\big)=\tfrac{d\,\sigma^2}{d^2}=\tfrac{\sigma^2}{d},$$
$$Var\big(L_{\text{mean}}(X_j,\theta)\big)=\tfrac{1}{d^2}\,Var\big(L_{\text{sum}}(X_j,\theta)\big)<Var\big(L_{\text{sum}}(X_j,\theta)\big)$$

In [6]:
```python
def diffusion_loss_reduced(diffusion: Diffusion, x0: torch.Tensor):
    b, C, H, W = x0.shape
    T = diffusion.n_steps
    # sample a timestep for each example in the batch
    t = torch.randint(1, T, (b,), device=x0.device)
    # run the forward noising and posterior sampling
    x_prev, x_t = diffusion.forward_process(x0, t)
    # model prediction of the mean
    mu_pred = diffusion.predict_next(x_t, t)
    # gather (1 − alpha_t) for each example and reshape to broadcast
    one_minus_alpha = (1.0 - diffusion.alphas[t]).view(b, 1, 1, 1)
    # per-example MSE loss
    per_example = (x_prev - mu_pred).pow(2).mean(dim=(1,2,3)) / (2.0 * one_minus_alpha.view(b))

    return per_example.mean()
```

6

In [7]:
```python
def load_full_mnist(datadir="./"):
    data = np.load(datadir + 'mnist/MNIST_data.npy').astype(np.float32)
    labels = np.load(datadir + 'mnist/MNIST_labels.npy').astype(np.int32)
    data /= 255.0
    data = data.reshape(-1, 1, 28, 28)
    return data, labels


def compute_variability_history(subset_hist):
    arr = np.array(subset_hist, dtype=np.float32)  # shape (epochs, 20)
    epoch_means = arr.mean(axis=1)
    epoch_stds = arr.std(axis=1)
    return epoch_means, epoch_stds
```

In [8]:
```python
# hyperparameters
T = 200
lr_main = 0.01
lr_rho  = 0.2
epochs = 60
batch_size = 100
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

data, labels = load_full_mnist()
tensor_x = torch.from_numpy(data)
tensor_y = torch.from_numpy(labels)
full_ds = TensorDataset(tensor_x, tensor_y)
perm = torch.randperm(len(full_ds))
subsets = [Subset(full_ds, perm[i*1000:(i+1)*1000]) for i in range(20)]
batch_size = 100
loaders = [
    DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=True)
    for ds in subsets
]
```

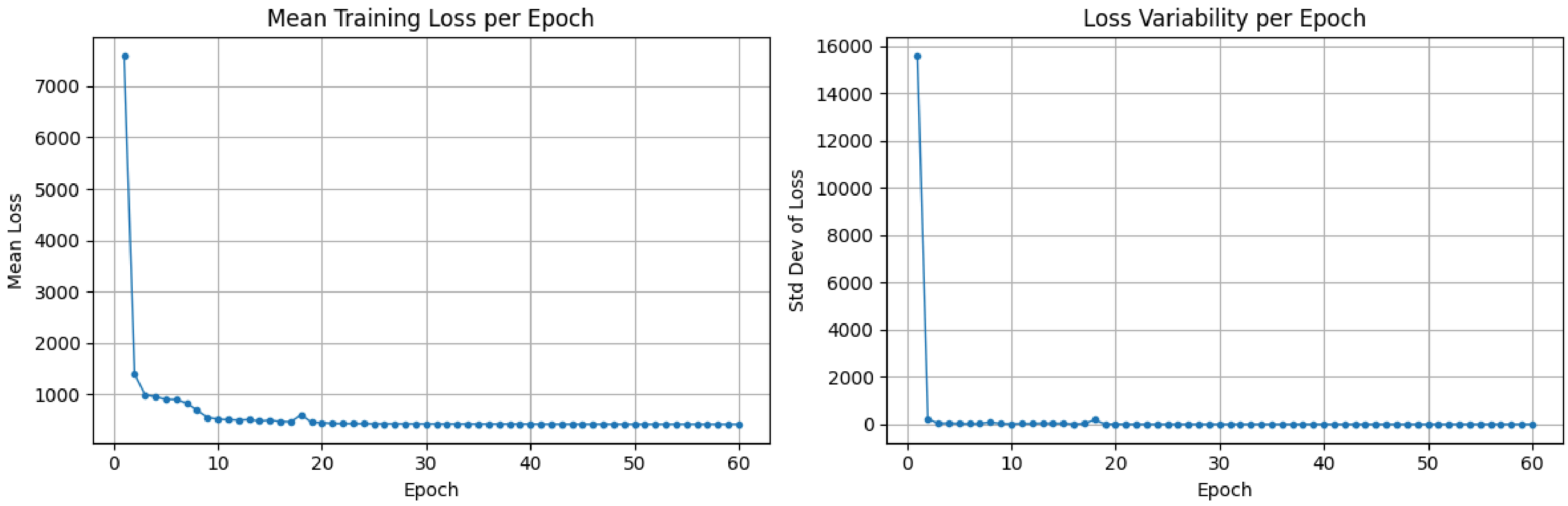
(a)

In [9]:
```python
score_net = ScoreNet().to(device)
diffusion = Diffusion(score_net, T, device, min_beta=1e-4, max_beta=0.1)

opt = optim.Adam([
    {'params': [p for n,p in score_net.named_parameters() if n not in ('rho0','rho1')],
     'lr': lr_main},
    {'params': [score_net.rho0, score_net.rho1],
     'lr': lr_rho}
])
# reduce the learning rate by a factor of 10 every 20 epochs
scheduler = optim.lr_scheduler.StepLR(opt, step_size=20, gamma=0.1)
subset_hist = [[0.0]*20 for _ in range(epochs)]
overall_hist = [0.0]*epochs
```

```python
epoch_bar = trange(epochs, desc="Epochs", position=0, leave=True, disable=not SHOW_TQDM)
for epoch in epoch_bar:
    score_net.train()

    # accumulators for this epoch
    epoch_subset_loss = [0.0]*20
    epoch_subset_count = [0]*20
    epoch_total_loss = 0.0
    epoch_total_count = 0

    for s_idx, loader in enumerate(tqdm(loaders, desc="Subsets", position=1,
                                        leave=False, disable=not SHOW_TQDM)):
        for x0, _ in loader:
            x0 = x0.to(device)
            opt.zero_grad()
            loss = diffusion_loss(diffusion, x0) # original loss
            loss.backward()
            opt.step()

            epoch_subset_loss[s_idx] += loss.item()
            epoch_subset_count[s_idx] += 1
            epoch_total_loss += loss.item()
            epoch_total_count += 1

    # compute and store per-subset averages
    for s_idx in range(20):
        subset_hist[epoch][s_idx] = (
            epoch_subset_loss[s_idx] / epoch_subset_count[s_idx]
        )

    # compute and store overall epoch average
    overall_hist[epoch] = epoch_total_loss / epoch_total_count
    avg_loss = epoch_total_loss / epoch_total_count
    epoch_bar.set_postfix_str(f"loss={avg_loss:.4f}")

    scheduler.step()

    # print(
    #     f"Epoch {epoch+1:2d} | "
    #     f"Overall Loss: {overall_hist[epoch]:.4f} | "
    #     # f"Subset Losses: {subset_hist[epoch]}"
    # )

torch.save(score_net.state_dict(), "original_loss_scorenet.pth")
```

In [10]:
```python
arr = np.array(subset_hist)
all_losses= arr.ravel()
min_, p25, med, p75, max_ = np.percentile(all_losses, [0, 25, 50, 75, 100])
epoch_vars= arr.var(axis=1)
avg_var = epoch_vars.mean()
print(f"min={min_:.4f}, 25%={p25:.4f}, median={med:.4f}, 75%={p75:.4f}, max={max_:.4f}, avg_var={avg_var:.6f}")

epoch_means, epoch_stds = compute_variability_history(subset_hist)
epochs_plot = np.arange(1, len(epoch_means) + 1)
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
# Mean loss plot
axs[0].plot(epochs_plot, epoch_means, marker='o', markersize=3, linewidth=1)
axs[0].set_xlabel("Epoch")
axs[0].set_ylabel("Mean Loss")
axs[0].set_title("Mean Training Loss per Epoch")
axs[0].grid(True)
# Variability plot
axs[1].plot(epochs_plot, epoch_stds, marker='o', markersize=3, linewidth=1)
axs[1].set_xlabel("Epoch")
axs[1].set_ylabel("Std Dev of Loss")
axs[1].set_title("Loss Variability per Epoch")
axs[1].grid(True)
plt.tight_layout()
plt.show()
```

```
min=416.1397, 25%=420.4990, median=426.1324, 75%=477.6914, max=73708.5737, avg_var=4041606.432964
```



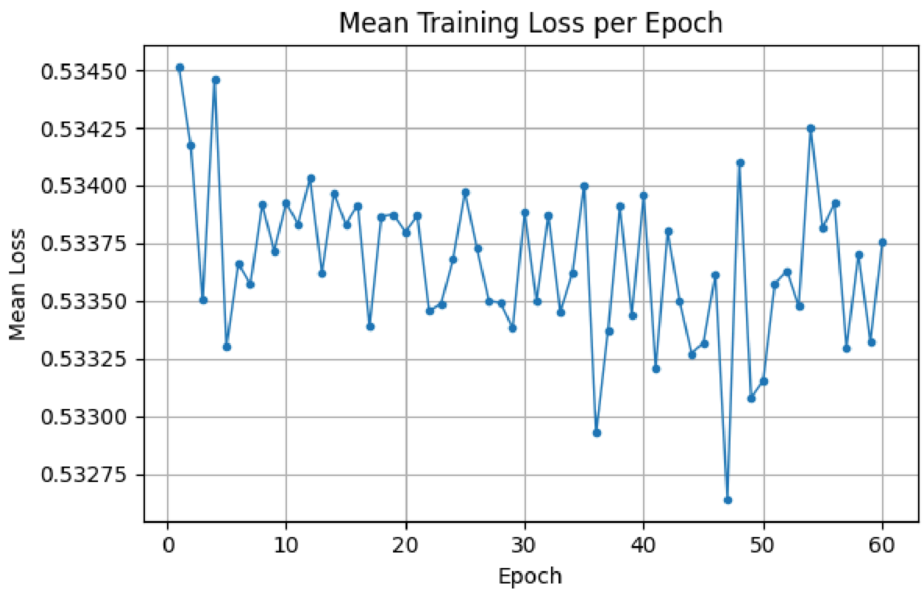
(b)

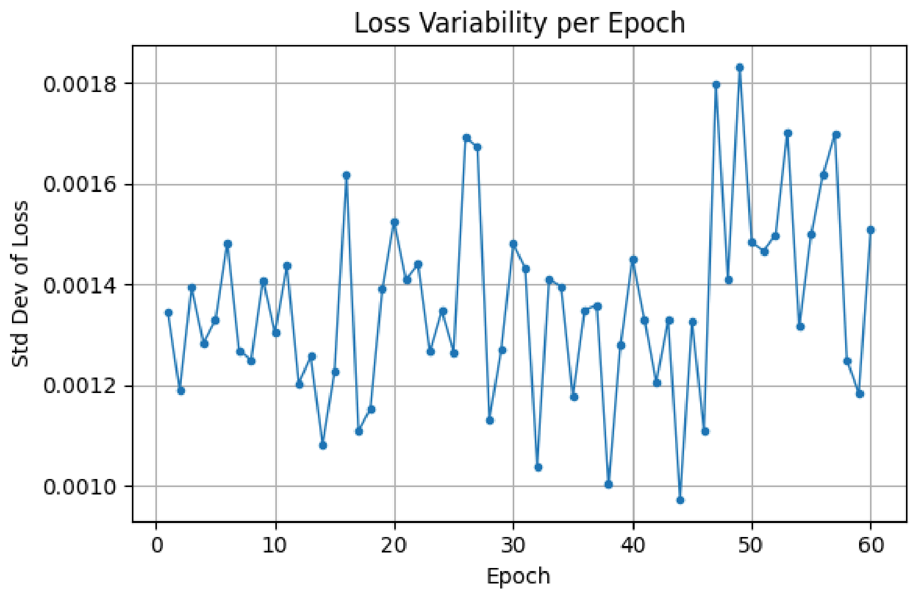
In [11]:
```python
subset_hist_reduced = [[0.0]*20 for _ in range(epochs)]
overall_hist_reduced = [0.0]*epochs
```

```python
epoch_bar = trange(epochs, desc="Epochs", position=0, leave=True, disable=not SHOW_TQDM)
for epoch in epoch_bar:
    score_net.train()

    # per-epoch accumulators
    epoch_subset_loss = [0.0]*20
    epoch_subset_count = [0]*20
    epoch_total_loss = 0.0
    epoch_total_count = 0

    for s_idx, loader in enumerate(tqdm(loaders, desc="Subsets", position=1,
                                        leave=False, disable=not SHOW_TQDM)):
        for x0, _ in loader:
            x0 = x0.to(device)
            opt.zero_grad()

            loss = diffusion_loss_reduced(diffusion, x0) # reduced-variance loss
            loss.backward()
            opt.step()

            epoch_subset_loss[s_idx] += loss.item()
            epoch_subset_count[s_idx] += 1
            epoch_total_loss += loss.item()
            epoch_total_count += 1

    # store per-subset averages
    for s_idx in range(20):
        subset_hist_reduced[epoch][s_idx] = (
            epoch_subset_loss[s_idx] / epoch_subset_count[s_idx]
        )

    # store overall epoch average
    overall_hist_reduced[epoch] = epoch_total_loss / epoch_total_count
    avg_loss = epoch_total_loss / epoch_total_count
    epoch_bar.set_postfix_str(f"loss={avg_loss:.4f}")
    scheduler.step()

    # print(
    #     f"Epoch {epoch+1:2d} | "
    #     f"Reduced Overall Loss: {overall_hist_reduced[epoch]:.4f}"
    # )

torch.save(score_net.state_dict(), "reduced_loss_scorenet.pth")
```

In [12]:
```python
arr = np.array(subset_hist_reduced)
all_losses= arr.ravel()
min_, p25, med, p75, max_ = np.percentile(all_losses, [0, 25, 50, 75, 100])
epoch_vars= arr.var(axis=1)
avg_var = epoch_vars.mean()
print(f"min={min_:.4f}, 25%={p25:.4f}, median={med:.4f}, 75%={p75:.4f}, max={max_:.4f}, avg_var={avg_var:.6f}")

epoch_means, epoch_stds = compute_variability_history(subset_hist_reduced)
epochs_plot = np.arange(1, len(epoch_means) + 1)
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
# Mean loss plot
axs[0].plot(epochs_plot, epoch_means, marker='o', markersize=3, linewidth=1)
axs[0].set_xlabel("Epoch")
axs[0].set_ylabel("Mean Loss")
axs[0].set_title("Mean Training Loss per Epoch")
axs[0].grid(True)
# Variability plot
axs[1].plot(epochs_plot, epoch_stds, marker='o', markersize=3, linewidth=1)
axs[1].set_xlabel("Epoch")
axs[1].set_ylabel("Std Dev of Loss")
axs[1].set_title("Loss Variability per Epoch")
axs[1].grid(True)
plt.tight_layout()
plt.show()
```

```
min=0.5291, 25%=0.5327, median=0.5337, 75%=0.5347, max=0.5378, avg_var=0.000002
```





Ans:

There is a difference in the variability of the loss over the 20 training sets between (a) and (b). When using the original ELBO loss, where we sum squared errors over all d pixels, the early-training variability across the 20 subsets is enormous, only settling after two learning-rate drops but still significant. In contrast, the per-pixel MSE formulation immediately stabilizes the loss that from epoch 1 onward the subset-to-subset standard deviation is very small comapring to the original loss function, and remains in that narrow band throughout training. In other words, switching to the mean-squared-error reduces the estimator's variance by $1/d$, yielding dramatically more consistent losses across different data splits.

7

I would use the `per-pixel mean-squared-error loss` (MSE) rather than the summed-squares ELBO. It immediately cuts variance by $1/d$, giving far more stable training curves across different subsets and making hyperparameter tuning and convergence monitoring far more reliable.

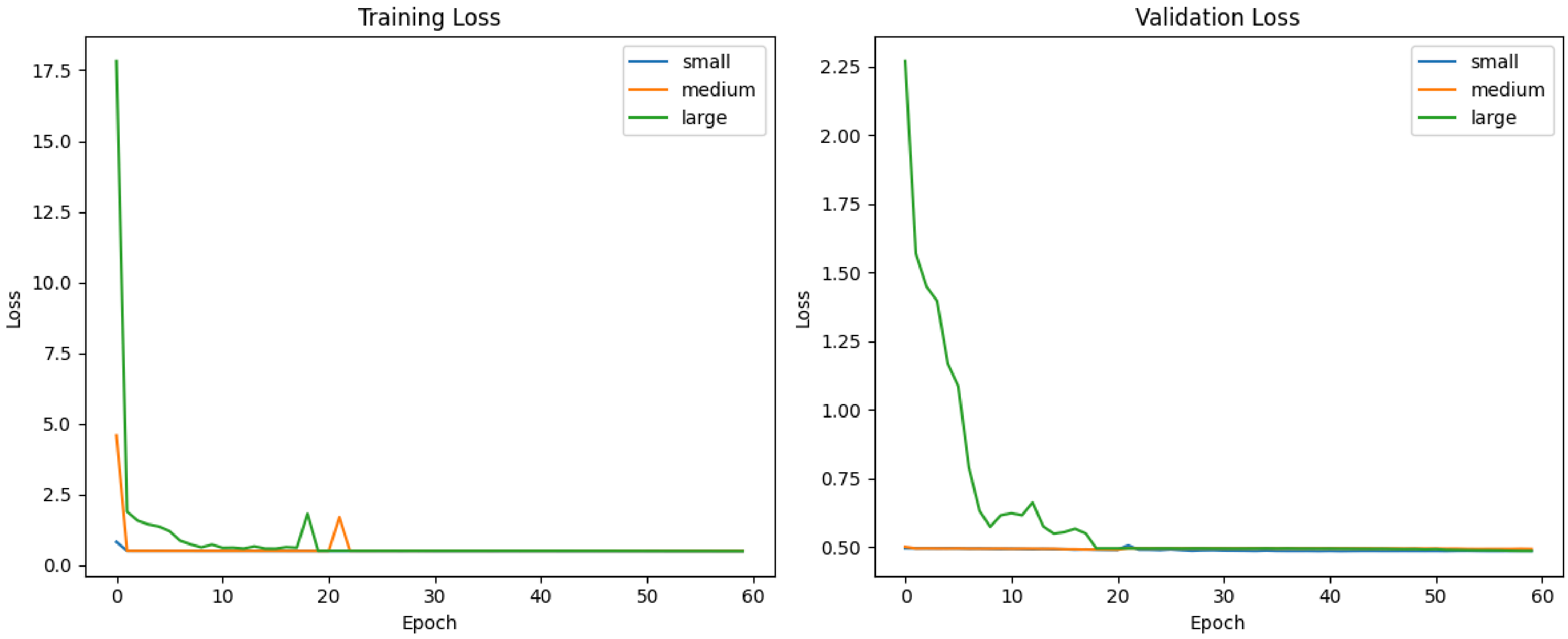
In [13]:
```
# hyperparameters
T = 200
lr_main = 0.01
lr_rho  = 0.2
epochs = 60
batch_size = 100
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# data
data, labels = load_full_mnist()
x = torch.from_numpy(data)
y = torch.from_numpy(labels)
full_ds = TensorDataset(x, y)
train_ds, val_ds = random_split(full_ds, [50000, 20000])
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=batch_size, shuffle=False)
```

In [14]:
```
archs = {
    'small' : [8,  16,  32, 64],
    'medium': [32,  64, 128, 256],
    'large': [64, 128, 256, 512]
}

results = {}
for name, channels in archs.items():
    # instantiate model & diffusion
    model = ScoreNet(channels=channels).to(device)
    diffusion = Diffusion(model, T, device, min_beta=1e-4, max_beta=0.1)
    opt = optim.Adam([
        {'params': [p for n,p in model.named_parameters() if n not in ('rho0','rho1')], 'lr': lr_main},
        {'params': [model.rho0, model.rho1], 'lr': lr_rho}
    ])
    sched = optim.lr_scheduler.StepLR(opt, step_size=30, gamma=0.1)

    train_hist, val_hist = [], []

    epoch_bar = trange(1, epochs+1, desc=f"[{name}] Epochs", position=1,
                       leave=True, disable=not SHOW_TQDM)

    for epoch in epoch_bar:
        model.train()
        running_loss = 0.0

        for x0, _ in tqdm(train_loader, desc="  Train batches", position=0,
                          leave=False, disable=not SHOW_TQDM):
            x0 = x0.to(device)
            opt.zero_grad()
            loss = diffusion_loss_reduced(diffusion, x0)
            loss.backward()
            opt.step()
            running_loss += loss.item()

        train_hist.append(running_loss / len(train_loader))
        avg = running_loss / len(train_loader)
        epoch_bar.set_postfix_str(f"loss={avg:.4f}")

        # validation pass
        model.eval()
        val_loss = 0.0
        for x0, _ in tqdm(val_loader, desc="  Val batches", position=0,
                          leave=False, disable=not SHOW_TQDM):
            x0 = x0.to(device)
            with torch.no_grad():
                val_loss += diffusion_loss_reduced(diffusion, x0).item()
        val_hist.append(val_loss / len(val_loader))
        sched.step()
    results[name] = {'train': train_hist, 'val': val_hist}
    torch.save(model.state_dict(), f"{name}_scorenet.pth")
```

In [15]:
```
for name, hist in results.items():
    val_losses = np.array(hist['val'])
    min_, q25, q50, q75, max_ = np.percentile(val_losses, [0, 25, 50, 75, 100])
    var = np.var(val_losses, ddof=0)
    print(f"{name:6s} | min={min_:.4f}, 25%={q25:.4f}, median={q50:.4f}, "
          f"75%={q75:.4f}, max={max_:.4f}, variance={var:.6f}")

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)

for name, hist in results.items():
    plt.plot(hist['train'], label=name)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

```
plt.subplot(1, 2, 2)
for name, hist in results.items():
    plt.plot(hist['val'], label=name)
plt.title('Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

```
small  | min=0.4851, 25%=0.4857, median=0.4870, 75%=0.4921, max=0.5066, variance=0.000018
medium | min=0.4902, 25%=0.4939, median=0.4948, 75%=0.4951, max=0.5009, variance=0.000002
large  | min=0.4856, 25%=0.4939, median=0.4951, 75%=0.5578, max=2.2699, variance=0.102686
```



Ans:

I chose three channel-width configurations (small $[8, 16, 32, 64]$, medium $[32, 64, 128, 256]$ and large $[64, 128, 256, 512]$) to explore how network capacity affects training speed, stability, and final performance. Exploring different channel-widths could explore the trade-off between model capacity and computational cost. Wider layers (more channels) increase the number of feature detectors at each spatial resolution, which can improve expressiveness and reduce bias, but also greatly inflate the number of parameters, slow down each training iteration, and increase the risk of overfitting. Narrower layers, by contrast, drastically cut parameter count and runtime and tend to regularize the network, at the expense of limiting representational power. By comparing small, medium, and large configurations, we could identify the point at which adding channels no longer yields meaningful gains.

Over 60 epochs, the small model achieved a median per-epoch loss of 0.4870 with an interquartile range of (0.4857, 0.4921), a loss variance of 0.000018, and required only about 26.5s per epoch. By contrast, the medium model's median loss was 0.4948 with IQR (0.4939, 0.4951) with variance 0.000002 but took 67.2s per epoch, while the large model's median loss was 0.4951 with IQR (0.4939, 0.5578) with much higher variance 0.102686 and an epoch time of 245.4s. Not only reaching the lowest 0%, 25%, 50%, and 75% quantile loss, the **small network** offers dramatically faster training and relatively low variability, making it the most efficient and reliable choice. Thus, the small model achieves the same or better stability and final loss with only a fraction of the compute, demonstrating that it already has sufficient capacity for MNIST denoising.

8

In [16]:
```python
@torch.no_grad()
def sample_from_model(diffusion: Diffusion, n_samples: int):
    device = diffusion.device
    T = diffusion.n_steps

    # Draw x_T ~ N(0, I)
    x_t = torch.randn(n_samples, 1, 28, 28, device=device)

    # Step backwards
    for t in range(T, 0, -1):
        # convert integer timestep
        t_tensor = torch.full((n_samples,), t, device=device, dtype=torch.long)
        # predict mean mu
        mu_t = diffusion.predict_next(x_t, t_tensor)
        # Compute sqrt(1 - alpha)
        sqrt_var = torch.sqrt(1.0 - diffusion.alphas[t-1])
        # sample fresh noise
        z = torch.randn_like(x_t) if t > 1 else 0.0
        # form next sample
        x_t = mu_t + sqrt_var * z
    return x_t # x_t is now x_0
```

9

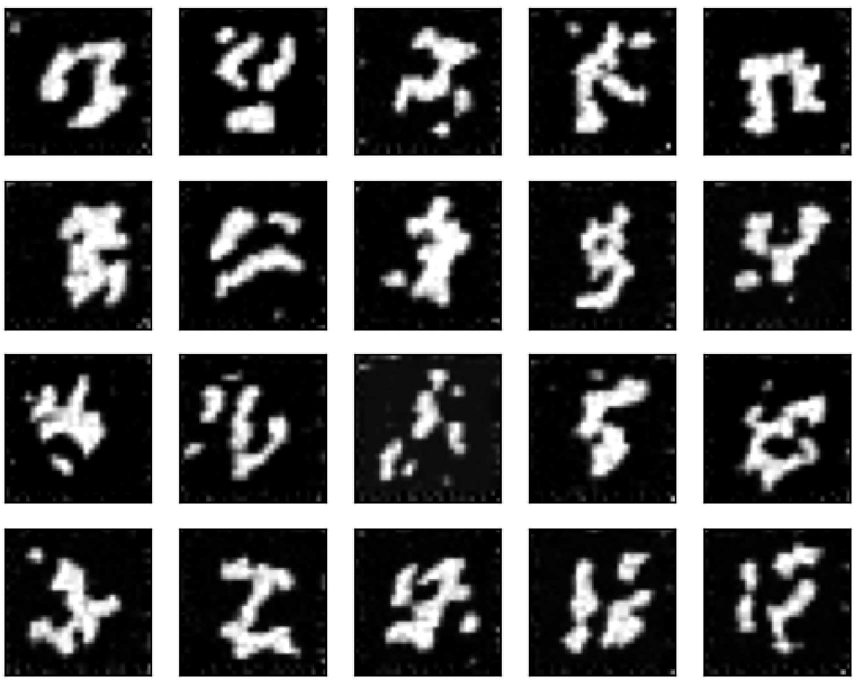
In [17]:
```python
def fid(features1: np.ndarray, features2: np.ndarray) -> float:
    mu1, mu2 = features1.mean(axis=0), features2.mean(axis=0)
    C1 = np.cov(features1, rowvar=False)
    C2 = np.cov(features2, rowvar=False)

    diff = mu1 - mu2
    mean_term = diff.dot(diff)

    w1, V1 = eigh(C1)
    w1 = np.clip(w1, 0, None)
```

```
    C1_sqrt = V1 @ np.diag(np.sqrt(w1)) @ V1.T
    S = C1_sqrt @ C2 @ C1_sqrt
    wS, VS = eigh(S)
    wS = np.clip(wS, 0, None)
    covmean = VS @ np.diag(np.sqrt(wS)) @ VS.T
    trace_term = np.trace(C1 + C2 - 2 * covmean)

    return mean_term + trace_term
```

In [18]:
```
# load model (small)
model = ScoreNet(channels=[8,16,32,64]).to(device)
diffusion = Diffusion(model, T, device, min_beta=1e-4, max_beta=0.1)
state = torch.load("small_scorenet.pth", map_location=device)
model.load_state_dict(state)
model.eval();

# load model (medium)
# model = ScoreNet(channels=[32,64,128,256]).to(device)
# diffusion = Diffusion(model, T, device, min_beta=1e-4, max_beta=0.1)
# state = torch.load("medium_scorenet.pth", map_location=device)
# model.load_state_dict(state)
# model.eval();

# load model (large)
# model = ScoreNet(channels=[64,128,256,512]).to(device)
# diffusion = Diffusion(model, T, device, min_beta=1e-4, max_beta=0.1)
# state = torch.load("large_scorenet.pth", map_location=device)
# model.load_state_dict(state)
# model.eval();
```

In [19]:
```
# load real data
data, labels = load_full_mnist()
x_tensor = torch.from_numpy(data)
real_ds = TensorDataset(x_tensor, torch.from_numpy(labels))
real_subset  = Subset(real_ds, list(range(1000)))
real_loader = DataLoader(real_subset, batch_size=1000, shuffle=False)
real_imgs, _ = next(iter(real_loader))
real_feats = real_imgs.view(1000, -1).numpy()

# generate fake data
with torch.no_grad():
    fake_imgs = sample_from_model(diffusion, 1000).cpu()
fake_feats = fake_imgs.view(1000, -1).numpy()
real_feats = real_imgs.view(1000, -1).numpy()

# compute FID score
fid_score = fid(real_feats, fake_feats)
print(f"FID: {fid_score:.4f}")

# plot 20 samples
fig, axes = plt.subplots(4, 5, figsize=(5, 4), subplot_kw={'xticks': [], 'yticks': []})
for i, ax in enumerate(axes.flatten()):
    ax.imshow(fake_imgs[i, 0], cmap='gray', vmin=0, vmax=1)
plt.tight_layout()
plt.show()
```

```
FID: 12.6141
```



Coding II

1

Define new Loss Function `diffusion_loss_new()` and update `ScoreNet`.

In [20]:
```
def diffusion_loss_new(diffusion, x0):
    B = x0.size(0)
    T = diffusion.n_steps
    # sample a random integer t_j
    t = torch.randint(1, T+1, (B,), device=x0.device)
    # look up alpha_bar_t = prod_{s=1}^t alpha_s
```

```python
    alpha_bar = diffusion.alphas_cumprod[t-1].view(B,1,1,1)
    alpha = diffusion.alphas[t-1].view(B,1,1,1)
    # ample noise eps ~ N(0,I)
    eps = torch.randn_like(x0)
    # form the noised input x_t
    xt = torch.sqrt(alpha_bar) * x0 + torch.sqrt(1 - alpha_bar) * eps
    # predict the noise with your network e_theta(xt, t)
    eps_pred = diffusion.predict_next(xt, t)
    # compute the coefficient factor
    weight = (1 - alpha) / (2 * alpha * (1 - alpha_bar))
    # MSE loss on the noise, averaged over pixels and batch
    per_example = weight * (eps - eps_pred).pow(2).flatten(1).sum(dim=1, keepdim=True)

    return per_example.mean()
```

In [21]:
```python
class ScoreNet(nn.Module):
    """U-Net that predicts ϵ(x,t). All kernels=3."""
    def __init__(self, channels=[32, 64, 128, 256], embed_dim=256, group_num=4):
        super().__init__()
        self.embed = nn.Sequential(
            GaussianFourierProjection(embed_dim=embed_dim),
            nn.Linear(embed_dim, embed_dim))
        self.act = lambda x: x * torch.sigmoid(x)

        # Encoder
        self.conv1 = nn.Conv2d(1, channels[0], kernel_size=3, stride=1, padding=1, bias=False)
        self.dense1 = Dense(embed_dim, channels[0])
        self.norm1  = nn.GroupNorm(group_num, channels[0])
        self.conv2 = nn.Conv2d(channels[0], channels[1], kernel_size=3, stride=2, padding=1, bias=False)
        self.dense2 = Dense(embed_dim, channels[1])
        self.norm2  = nn.GroupNorm(group_num, channels[1])
        self.conv3 = nn.Conv2d(channels[1], channels[2], kernel_size=3, stride=2, padding=1, bias=False)
        self.dense3 = Dense(embed_dim, channels[2])
        self.norm3  = nn.GroupNorm(group_num, channels[2])
        self.conv4 = nn.Conv2d(channels[2], channels[3], kernel_size=3, stride=2, padding=1, bias=False)
        self.dense4 = Dense(embed_dim, channels[3])
        self.norm4  = nn.GroupNorm(group_num, channels[3])

        # Decoder
        self.tconv4 = nn.ConvTranspose2d(channels[3], channels[2], kernel_size=3, stride=2, padding=1, output_padding=0, bias=Fals
        self.dense5 = Dense(embed_dim, channels[2])
        self.tnorm4 = nn.GroupNorm(group_num, channels[2])
        self.tconv3 = nn.ConvTranspose2d(channels[2]*2, channels[1], kernel_size=3, stride=2, padding=1, output_padding=1,bias=Fal
        self.dense6 = Dense(embed_dim, channels[1])
        self.tnorm3 = nn.GroupNorm(group_num, channels[1])
        self.tconv2 = nn.ConvTranspose2d(channels[1]*2, channels[0], kernel_size=3, stride=2, padding=1, output_padding=1, bias=Fa
        self.dense7 = Dense(embed_dim, channels[0])
        self.tnorm2 = nn.GroupNorm(group_num, channels[0])
        self.tconv1 = nn.ConvTranspose2d(channels[0]*2, 1, kernel_size=3, stride=1, padding=1)

    def forward(self, x, t):
        emb = self.act(self.embed(t))
        # encode
        h1 = self.act(self.norm1(self.conv1(x) + self.dense1(emb)))
        h2 = self.act(self.norm2(self.conv2(h1) + self.dense2(emb)))
        h3 = self.act(self.norm3(self.conv3(h2) + self.dense3(emb)))
        h4 = self.act(self.norm4(self.conv4(h3) + self.dense4(emb)))

        # decode
        u = self.act(self.tnorm4(self.tconv4(h4) + self.dense5(emb)))
        u = self.act(self.tnorm3(self.tconv3(torch.cat([u, h3], dim=1)) + self.dense6(emb)))
        u = self.act(self.tnorm2(self.tconv2(torch.cat([u, h2], dim=1)) + self.dense7(emb)))
        eps_pred = self.tconv1(torch.cat([u, h1], dim=1))
        return eps_pred
```

Trainning the Model

In [22]:
```python
T = 200
lr_main = 0.01
lr_rho  = 0.2
epochs = 60
batch_size = 100
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# data
data, labels = load_full_mnist()
x = torch.from_numpy(data)
y = torch.from_numpy(labels)
full_ds = TensorDataset(x, y)
train_ds, val_ds = random_split(full_ds, [50000, 20000])
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=batch_size, shuffle=False)
```

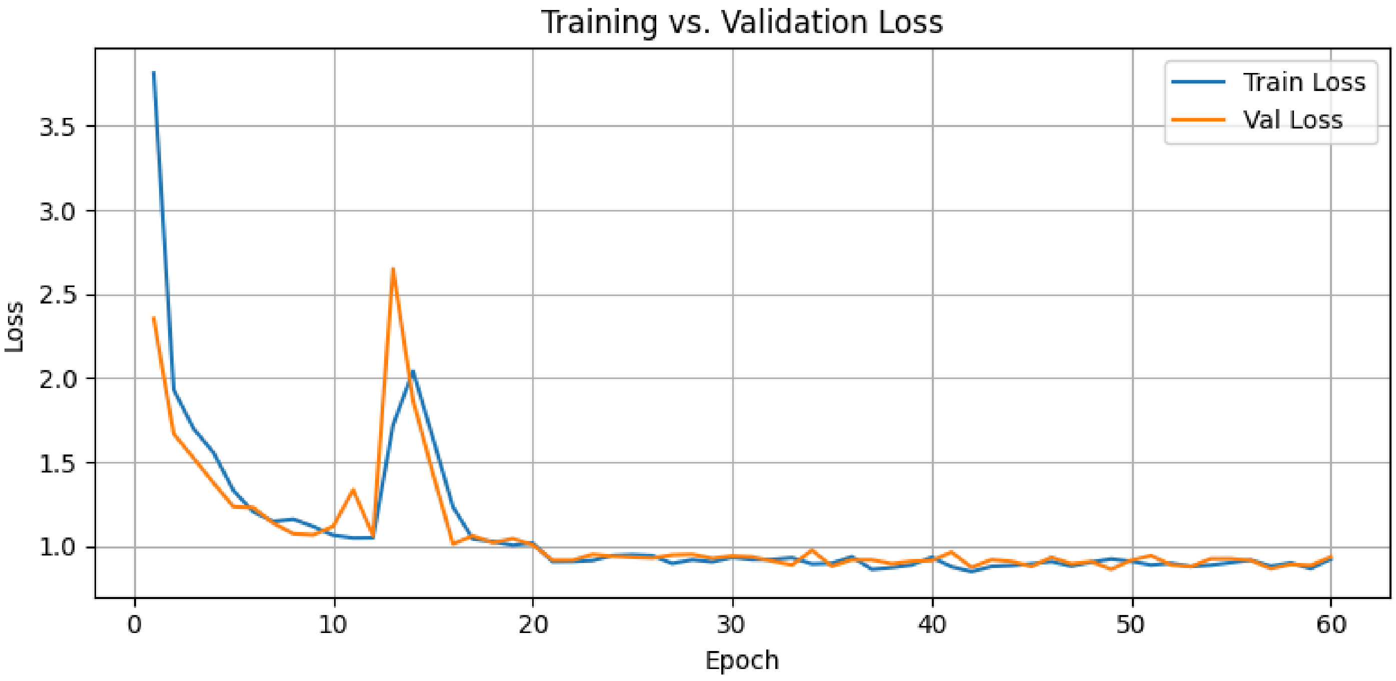
In [23]:
```python
score_net = ScoreNet(channels=[8,16,32,64]).to(device)
diffusion = Diffusion(score_net, n_steps=T, device=device, min_beta=1e-4, max_beta=0.1)
opt   = optim.Adam(score_net.parameters(), lr=lr_main)
sched = optim.lr_scheduler.StepLR(opt, step_size=20, gamma=0.1)

train_hist, val_hist = [], []

epoch_bar = trange(1, epochs+1, desc="Epochs", position=1, leave=True, disable=not SHOW_TQDM)
for epoch in epoch_bar:
    score_net.train()
    running_loss = 0.0
    for x0, _ in tqdm(train_loader, desc="  Train batches", position=0,
                      leave=False, disable=not SHOW_TQDM):
```

```
        x0 = x0.to(device)
        opt.zero_grad()
        loss = diffusion_loss_new(diffusion, x0)
        loss.backward()
        opt.step()
        running_loss += loss.item()
    train_loss = running_loss / len(train_loader)
    train_hist.append(train_loss)
    epoch_bar.set_postfix_str(f"train_loss={train_loss:.4f}")

    model.eval()
    v_loss = 0.0
    for x0, _ in tqdm(val_loader, desc="  Val batches", position=1,
                      leave=False, disable=not SHOW_TQDM):
        x0 = x0.to(device)
        with torch.no_grad():
            v_loss += diffusion_loss_new(diffusion, x0).item()
    val_loss = v_loss / len(val_loader)
    val_hist.append(val_loss)
    sched.step()

torch.save(score_net.state_dict(), "new_scorenet.pth")
```

In [32]:

```
val_arr = np.array(val_hist)
q0, q25, q50, q75, q100 = np.percentile(val_arr, [0, 25, 50, 75, 100])
var = np.var(val_arr)
print(f"Validation loss | min: {q0:.4f}, 25%: {q25:.4f} | median: {q50:.4f} | 75%: {q75:.4f} | max: {q100:.4f} | variance: {var:.6

epochs = np.arange(1, len(train_hist) + 1)
plt.figure(figsize=(8,4))
plt.plot(epochs, train_hist, label='Train Loss')
plt.plot(epochs, val_hist,   label='Val Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training vs. Validation Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Validation loss | min: 0.8635, 25%: 0.9120 | median: 0.9361 | 75%: 1.0626 | max: 2.6501 | variance: 0.111346
```



2

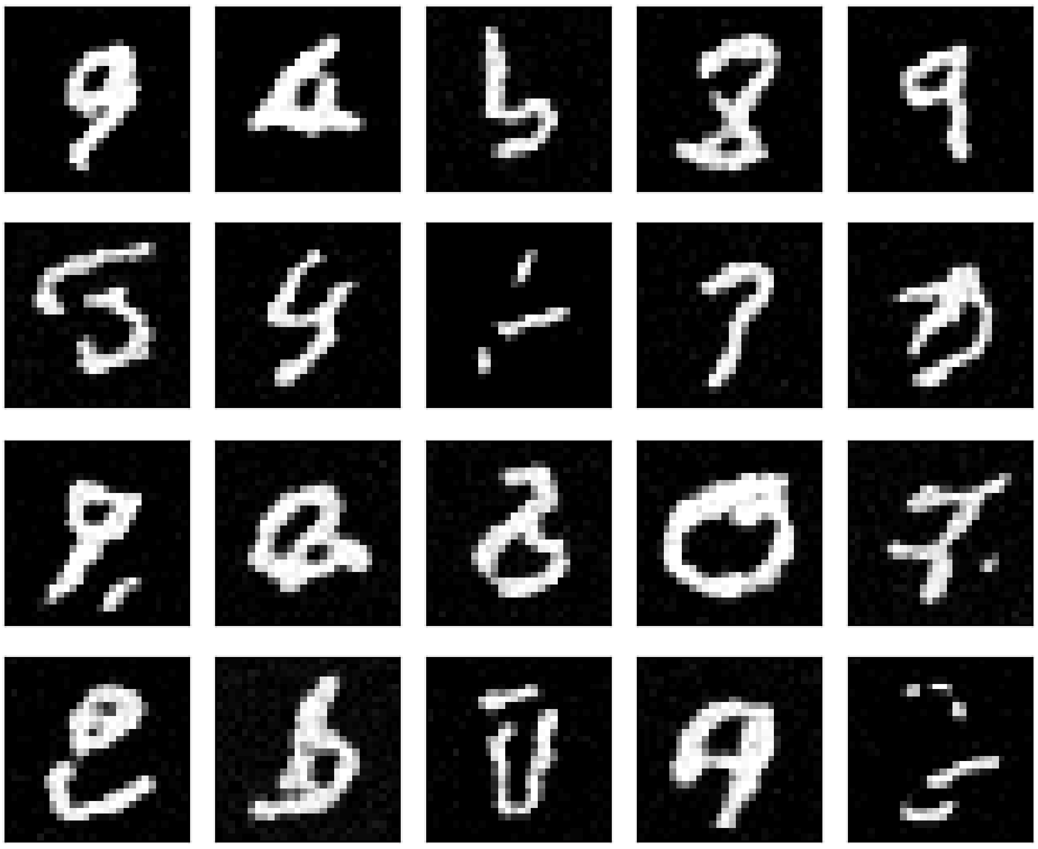
In [24]:

```
@torch.no_grad()
def sample_from_model_new(diffusion: Diffusion, n_samples: int):
    """
    Sample x₀ by running the reverse diffusion using an ε-prediction network.
    """
    device = diffusion.device
    T = diffusion.n_steps

    # Gaussian at time T
    x_t = torch.randn(n_samples, 1, 28, 28, device=device)

    for t in range(T, 0, -1):
        # make a (n_samples,) tensor of the current timestep
        t_tensor = torch.full((n_samples,), t, device=device, dtype=torch.long)
        # predict e(X,t)
        eps_pred = diffusion.predict_next(x_t, t_tensor)
        beta_t  = diffusion.betas[t-1] # beta_t = 1 - alpha_t
        alpha_t = diffusion.alphas[t-1] # alpha_t
        alpha_bar_t = diffusion.alphas_cumprod[t-1] # alpha_bar_t
        # compute mu_t
        coef = beta_t / torch.sqrt(1.0 - alpha_bar_t)
        mu_t = (x_t - coef * eps_pred) / torch.sqrt(alpha_t)
        # add scaled noise
        if t > 1:
            z = torch.randn_like(x_t)
            x_t = mu_t + torch.sqrt(beta_t) * z
        else:
            x_t = mu_t
    return x_t
```

In [25]:
```
model = ScoreNet(channels=[8, 16, 32, 64]).to(device)
diffusion = Diffusion(model, n_steps=T, device=device, min_beta=1e-4, max_beta=0.1)
state = torch.load("new_scorenet.pth", map_location=device)
model.load_state_dict(state)
model.eval();
```

In [26]:
```
gen_samples = sample_from_model_new(diffusion, n_samples=1000).cpu().numpy()
real_data, _ = load_full_mnist()
idx = np.random.choice(len(real_data), size=1000, replace=False)
real_samples = real_data[idx]
gen_feats  = gen_samples.reshape(1000, -1)
real_feats = real_samples.reshape(1000, -1)

# FID score
fid_score  = fid(real_feats, gen_feats)
print(f"FID score (pixel-space) = {fid_score:.4f}")

# plot 20 samples
fig, axes = plt.subplots(4, 5, figsize=(6, 5),
                         subplot_kw={'xticks':[], 'yticks':[]})
for i, ax in enumerate(axes.flatten()):
    ax.imshow(gen_samples[i, 0], cmap='gray', vmin=0, vmax=1)
plt.tight_layout()
plt.show()
```

```
FID score (pixel-space) = 5.0627
```



3

Ans:

When we replace $L(\theta) = \mathbb{E}_{t,X_0,\epsilon}\left[\frac{\|X_{t-1}-\mu_\theta(X_t,t)\|^2}{2(1-\alpha_t)}\right]$ with the $L_\epsilon(\theta) = \mathbb{E}_{t,X_0,\epsilon}\left[\frac{1-\alpha_t}{2\,\alpha_t\,(1-\bar{\alpha}_t)}\ \|\epsilon - e_\theta(X_t,t)\|^2\right]$, we see a dramatic drop in FID (from around 20 to around 5) and visibly crisper digits, while the training time is essentially unchanged. This improvement stems from two key effects. First, by parameterizing the reverse mean as $\mu_\theta(X_t,t) = \frac{1}{\sqrt{\alpha_t}}\left[X_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\, e_\theta(X_t,t)\right]$, the noise-prediction loss automatically normalizes the learning signal at every t, and each residual $\epsilon - e_\theta$ has unit variance and enters the gradient with a single scalar weight $\frac{1-\alpha_t}{\alpha_t(1-\bar{\alpha}_t)}$, avoiding the extreme amplification or attenuation of gradients that the mean-matching loss suffers when α_t is near 0 or 1. Second, targeting $\epsilon \sim \mathcal{N}(0,I)$ yields lower-variance Monte Carlo estimates since ϵ is zero-mean Gaussian with fixed scale, whereas the original loss targets X_{t-1}, whose distribution's variance itself depends on $\bar{\alpha}_t$. Together, these factors give more stable, uniformly scaled gradients across timesteps, leading to faster convergence to a higher-quality model.