CV-2025: **Optimal Transport–Guided Diffusion on MNIST Passive & Active Phases**

Improving generative quality with Sinkhorn OT in training and sampling

Lian Natour:207300443
Mohammad Mhneha:315649814

Code & instructions: [https://github.com/lian99/cv_project_final]

Passive Phase (Baseline)

1.1 Objective

Establish a baseline DDPM on MNIST and a metrics toolkit to quantify:

- **Fidelity/Precision:** How close generated samples are to real test images.
- **Coverage/Diversity:** How broadly generated samples cover the training distribution.

We use two complementary distances:

- **Sinkhorn OT** (Optimal Transport; sensitive to distribution geometry)
- **FID** (Fréchet distance between Gaussian fits to features; stable and widely used)

1.2 Data & Preprocessing

- **Dataset:** MNIST (60k train, 10k test).
- **Preprocessing:** images resized to 28×28, converted to tensors, scaled to [-1, 1] for diffusion training; converted back to [0, 1] for the feature network.
- **Splits:** deterministic split of the original train into train and val (5k). The untouched MNIST test set is used for evaluation experiments.

Snippet—transforms & loaders:

dataset splits and their loaders

```python
from torchvision import datasets
import torchvision.transforms as T
import torch
from torch.utils.data import DataLoader, random_split

torch.manual_seed(42)  # ensure reproducibility across runs

# --- Config
BATCH_SIZE = 128
IMAGE_SIZE = 28
TIMESTEPS = 1000
SAMPLING_TIMESTEPS = 50

# --- Transform
transform = T.Compose([
    T.Resize(IMAGE_SIZE),
    T.ToTensor(),
    lambda x: x * 2 - 1
])

# --- Full original training set (60K)
full_train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

# --- Deterministic split into train/val
VAL_SIZE = 5000
TRAIN_SIZE = len(full_train_dataset) - VAL_SIZE
train_dataset, val_dataset = random_split(full_train_dataset, [TRAIN_SIZE, VAL_SIZE])

# --- True test set (untouched)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# --- DataLoaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
val_loader   = DataLoader(val_dataset,   batch_size=BATCH_SIZE, shuffle=False)
test_loader  = DataLoader(test_dataset,  batch_size=BATCH_SIZE, shuffle=False)
```

1.3 Baseline Diffusion Model

We use lucidrains/denoising-diffusion-pytorch:

- **UNet:** dim=64, dim_mults=(1,2,4), grayscale channels=1.
- **DDPM training timesteps:** 1000; DDIM sampling steps: 50.
- **Loss:** standard noise-prediction MSE with a quadratic timestep sampler (biasing toward late timesteps to stabilize learning).

Snippet—model & loss:

```python
def compute_loss(model, diffusion, x0, device):
    b = x0.size(0)
    T_full = diffusion.num_timesteps

    # Quadratic timestep sampling
    u = torch.rand(b, device=device)
    t = (u ** 2 * T_full).long().clamp(0, T_full - 1).view(-1)

    # Forward noising
    noise = torch.randn_like(x0)
    x_t = diffusion.q_sample(x_start=x0, t=t, noise=noise)

    # Noise prediction
    pred_noise = model(x_t, t)

    # Loss
    return F.mse_loss(pred_noise, noise)
```

Training details: 35 epochs, Adam (2e-4). We log train/val loss and reconstructions via Weights & Biases.

1.4 Feature Extractor (for OT/FID)

We train a small CNN classifier (Conv→ReLU→Pool×2; 128-d penultimate features) on MNIST (inputs in [0,1]) and freeze it.

- For **OT**, we use raw 128-d features (more sensitive to distributional shifts).
- For **FID**, we L2-normalize features for stable covariance estimates.

Snippet—feature extraction:

Feature Extraction Pipeline

```
# Cell 7: Dual Feature Extraction Pipeline
"""
Specialized feature extraction for OT and FID analysis.
OT uses raw features for sensitivity, FID uses normalized features for stability.
"""

def convert_range_to_01(tensor_minus1_plus1):
    """Convert tensor from [-1,1] to [0,1] range."""
    return (tensor_minus1_plus1.clamp(-1, 1) + 1.0) / 2.0

@torch.no_grad()
def extract_features_for_ot(model, dataset, device, batch_size=256):
    """Extract raw features optimized for OT computation."""
    model.eval()
    data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=False)

    feature_collections = []
    label_collections = []

    for image_batch, label_batch in tqdm(data_loader, desc="Extracting OT features"):
        image_batch = image_batch.to(device)
        image_batch = convert_range_to_01(image_batch)

        _, feature_vectors = model(image_batch)
        # Raw features - no normalization for OT

        feature_collections.append(feature_vectors.cpu())
        label_collections.append(label_batch)

    return torch.cat(feature_collections).numpy(), torch.cat(label_collections).numpy()

@torch.no_grad()
def extract_features_for_fid(model, dataset, device, batch_size=256):
    """Extract normalized features optimized for FID computation."""
    model.eval()
    data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=False)

    feature_collections = []
    label_collections = []

    for image_batch, label_batch in tqdm(data_loader, desc="Extracting FID features"):
        image_batch = image_batch.to(device)
        image_batch = convert_range_to_01(image_batch)

        _, feature_vectors = model(image_batch)
        # L2 normalize for FID stability
        feature_vectors = feature_vectors / (feature_vectors.norm(dim=1, keepdim=True) + 1e-8)

        feature_collections.append(feature_vectors.cpu())
        label_collections.append(label_batch)

    return torch.cat(feature_collections).numpy(), torch.cat(label_collections).numpy()
```

1.5 Metrics

- **Sinkhorn OT (GeomLoss):** compares two empirical feature distributions with entropic regularization. Used for both precision/fidelity stress test (noise) and coverage/diversity stress test (missing classes).
- **FID**: Gaussian approximation in feature space (means/covariances + matrix square root).

Snippet—OT & FID:

Optimal Transport Distance Computation

```
# Cell 5: Enhanced Sinkhorn Distance Calculator
"""
Implement optimized Sinkhorn optimal transport distance computation
using GeomLoss with proper feature normalization.
"""

def compute_normalized_sinkhorn_distance(feature_set_x, feature_set_y, blur_param=0.05,
                                         power=2, scaling_factor=0.9, backend="tensorized"):
    """Compute Sinkhorn distance with proper normalization."""
    distance_function = geomloss.SamplesLoss(loss="sinkhorn", p=power, blur=float(blur_param),
                                             scaling=scaling_factor, backend=backend)
    return distance_function(feature_set_x, feature_set_y)

def calculate_transport_distance(data_x, data_y, blur_param=0.05):
    """Calculate optimal transport distance between two feature sets."""
    # Convert to tensors if needed
    if isinstance(data_x, np.ndarray):
        data_x = torch.from_numpy(data_x).float()
    if isinstance(data_y, np.ndarray):
        data_y = torch.from_numpy(data_y).float()

    # L2 normalize features to unit vectors
    data_x = data_x / (data_x.norm(dim=1, keepdim=True) + 1e-8)
    data_y = data_y / (data_y.norm(dim=1, keepdim=True) + 1e-8)

    # Compute distance
    with torch.no_grad():
        distance = float(compute_normalized_sinkhorn_distance(
            data_x, data_y, blur_param=blur_param
        ).item())

    return distance
```

Fréchet Inception Distance (FID) Computation

```
# Cell 6: Fréchet Inception Distance Calculator
"""
Implement FID score computation for comparing feature distributions.
FID measures the distance between two multivariate Gaussian distributions.
"""

def calculate_frechet_distance(feature_set_x, feature_set_y, stability_epsilon=1e-6):
    """
    Calculate Fréchet Inception Distance between two feature sets.

    Args:
        feature_set_x, feature_set_y: Arrays of shape (n_samples, n_features)
        stability_epsilon: Small constant for numerical stability

    Returns:
        float: FID score (lower indicates better similarity)
    """
    def compute_covariance_and_mean(feature_array, epsilon=1e-6):
        """Calculate covariance matrix and mean with regularization."""
        feature_data = np.asarray(feature_array, dtype=np.float64)
        mean_vector = np.mean(feature_data, axis=0)
        covariance_matrix = np.cov(feature_data, rowvar=False)
        dimensionality = covariance_matrix.shape[0]
        # Add diagonal regularization for numerical stability
        regularized_cov = covariance_matrix + epsilon * np.eye(dimensionality, dtype=np.float64)
        return regularized_cov, mean_vector

    # Calculate statistics for both feature distributions
    covariance_x, mean_x = compute_covariance_and_mean(feature_set_x, stability_epsilon)
    covariance_y, mean_y = compute_covariance_and_mean(feature_set_y, stability_epsilon)
    mean_difference = mean_x - mean_y

    # Calculate trace term: tr(Σ₁) + tr(Σ₂) - 2*tr(√(Σ₁Σ₂))
    try:
        covariance_product_sqrt = sqrtm(covariance_x.dot(covariance_y))
        if np.iscomplexobj(covariance_product_sqrt):
            covariance_product_sqrt = covariance_product_sqrt.real
        trace_component = (np.trace(covariance_x) + np.trace(covariance_y) -
                           2.0 * np.trace(covariance_product_sqrt))
    except:
        # Fallback calculation if matrix square root fails
        trace_component = np.trace(covariance_x) + np.trace(covariance_y)

    # FID = ||μ₁ - μ₂||² + tr(Σ₁ + Σ₂ - 2√(Σ₁Σ₂))
    frechet_distance = float(mean_difference @ mean_difference + trace_component)
    return max(frechet_distance, 0.0)
```

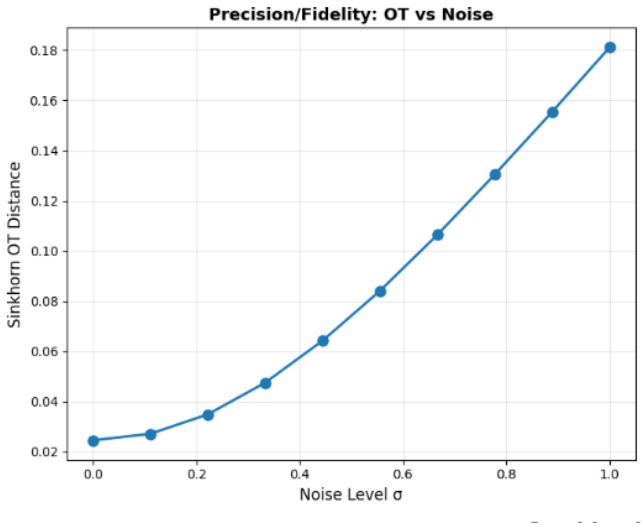
1.6 Tests (Passive Phase)

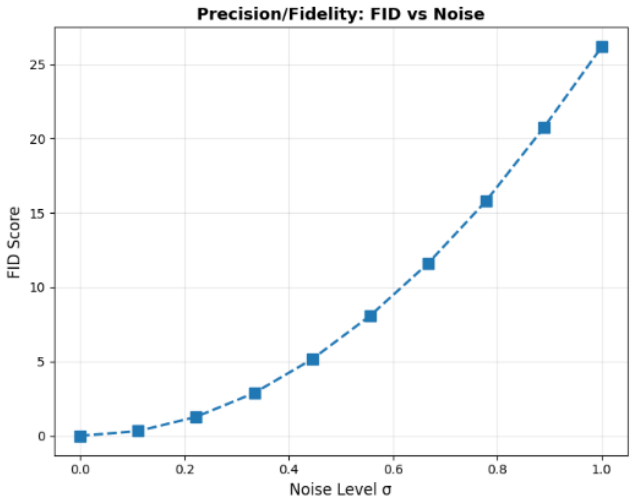
1.6.1 Precision / Fidelity vs. Noise

We perturb test features with additive Gaussian noise of increasing σ and measure distances to train features.

- What it tests: fidelity/precision—if generated (or noised) samples drift from the real manifold, distances should grow monotonically.
- Observation: OT distance increases smoothly and roughly convex with σ; FID shows a similar monotonic trend.

Here is what we got :





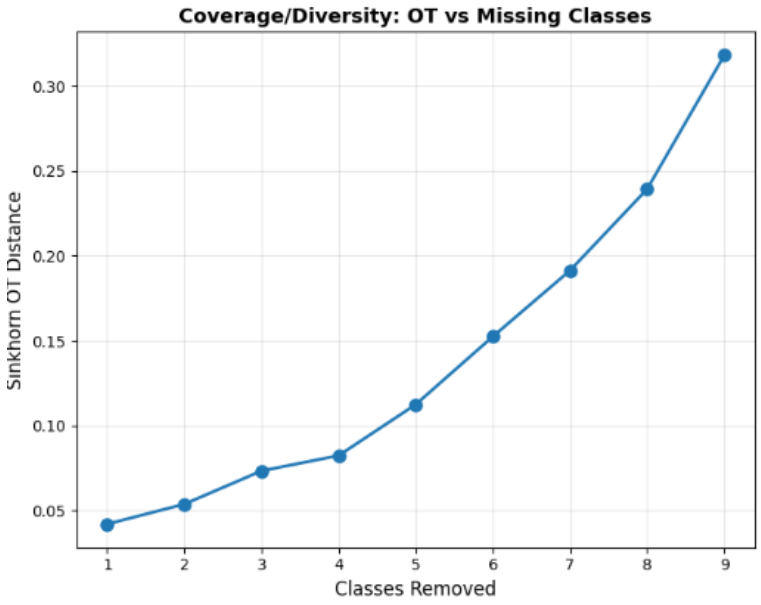
*lower is better; curves increasing with noise confirm metric sensitivity.

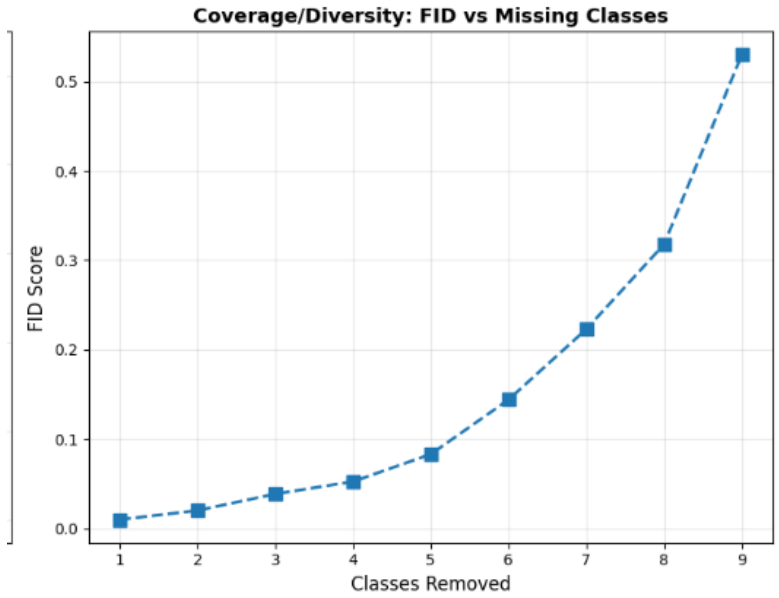
1.6.2 Coverage / Diversity vs. Missing Classes

We remove k classes from the reference (train) pool and compare to the (unaltered) test features.

- What it tests: coverage/diversity—when the reference lacks classes, distance should rise.
- Observation: both OT and FID rise with the number of removed classes; OT shows stronger curvature at large drops.

Here is what we got :



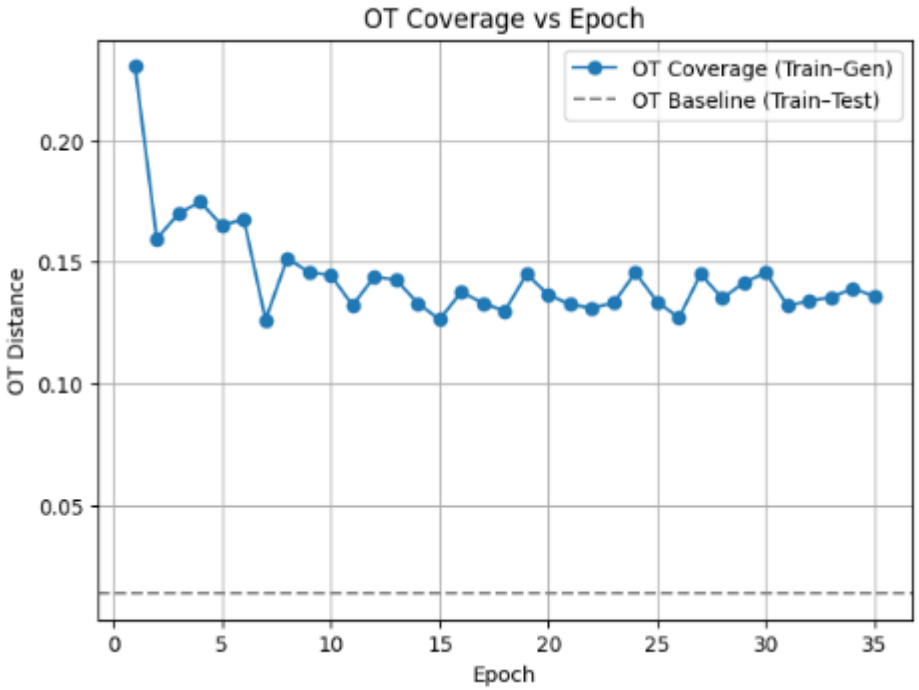


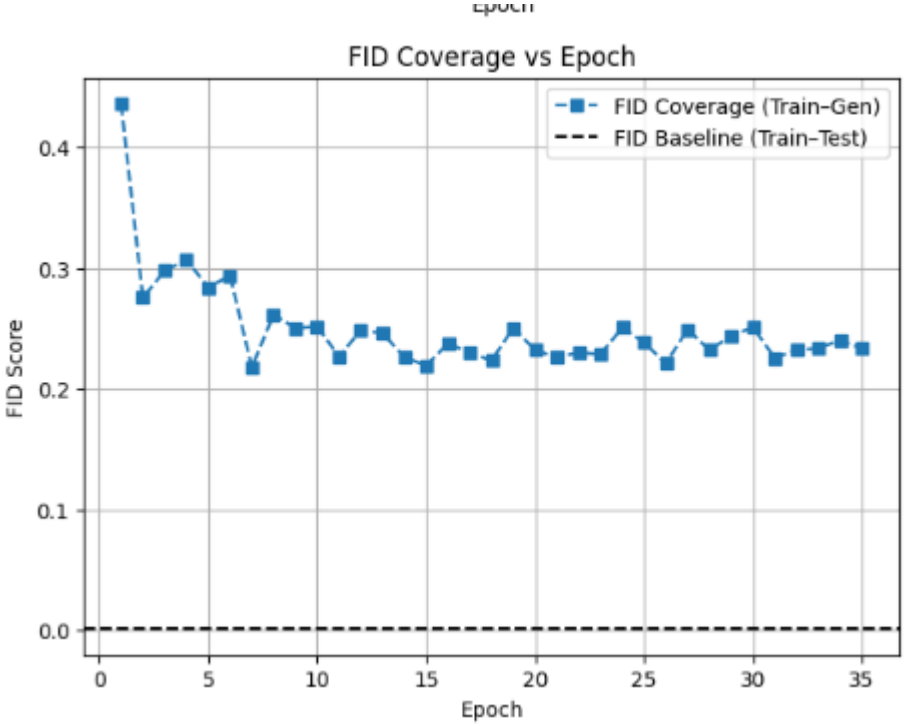
1.8 Epoch-wise Baseline Tracking

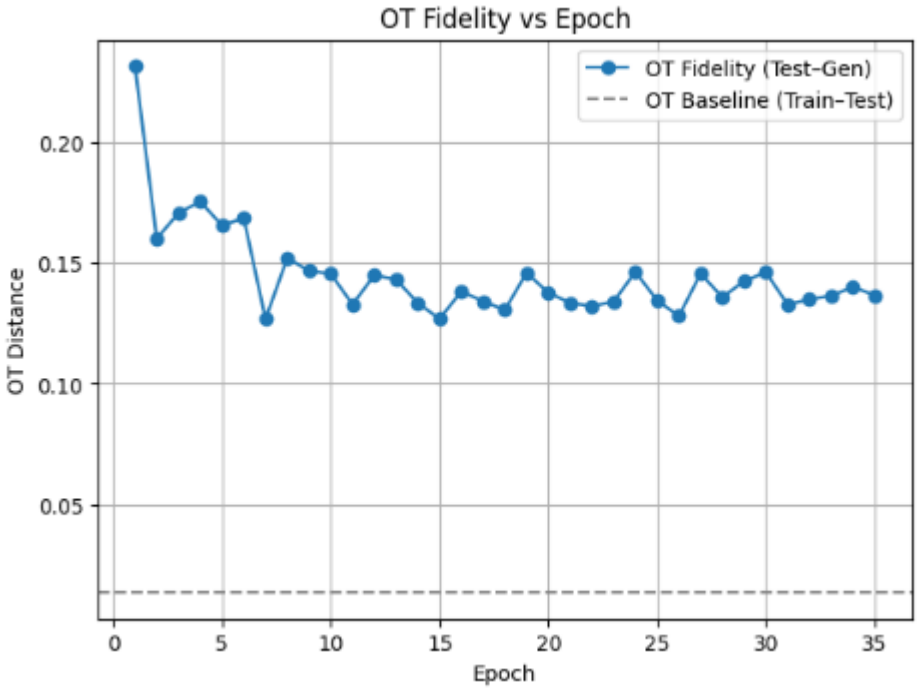
Using samples from each baseline checkpoint (1…35), we compute:

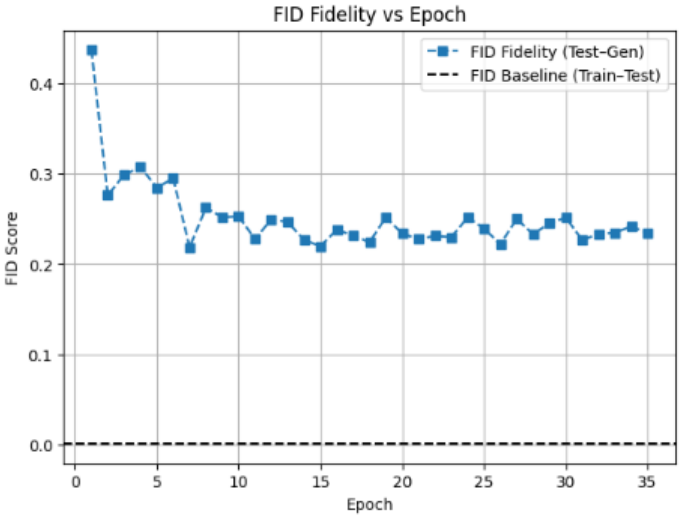
- Coverage (Train–Gen): distance train↔generated
- Fidelity (Test–Gen): distance test↔generated
- Baseline (Train–Test): horizontal reference line
- Generalization Gap: (Test–Gen) − (Train–Gen)

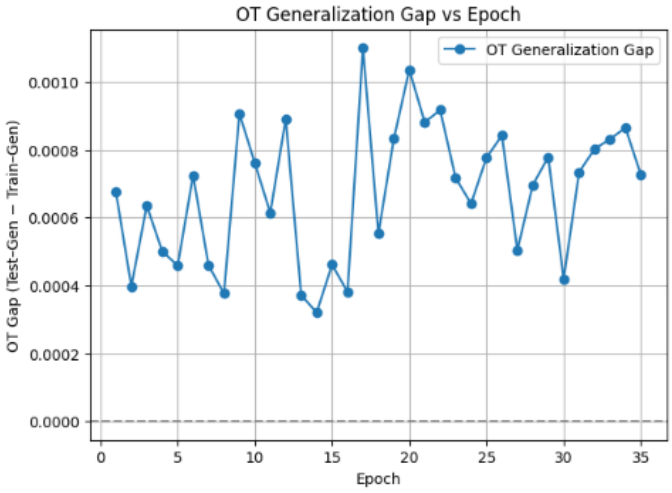
Here is what we got :

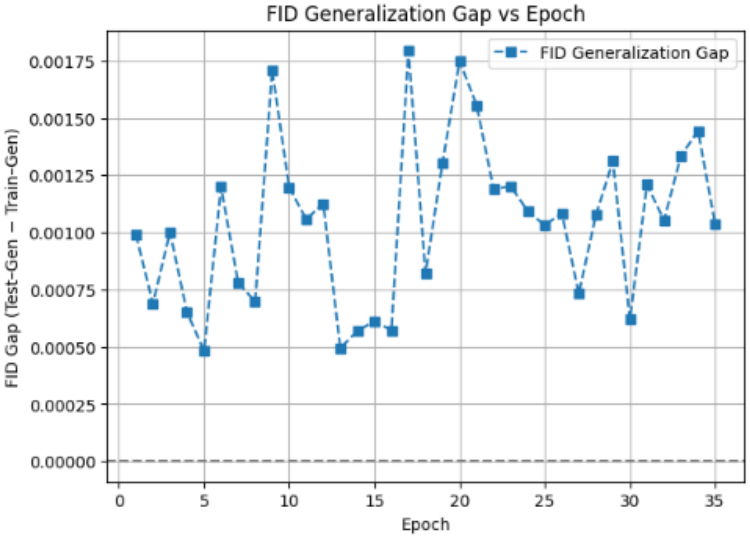



FID Coverage vs Epoch
FID Coverage (Train–Gen)
FID Baseline (Train–Test)
FID Score
Epoch
0.0
0.1
0.2
0.3
0.4
0
5
10
15
20
25
30
35


OT Fidelity vs Epoch
OT Fidelity (Test–Gen)
OT Baseline (Train–Test)
OT Distance
Epoch
0.05
0.10
0.15
0.20
0
5
10
15
20
25
30
35


FID Fidelity vs Epoch
FID Fidelity (Test–Gen)
FID Baseline (Train–Test)
FID Score
Epoch


OT Generalization Gap vs Epoch
OT Generalization Gap
OT Gap (Test–Gen − Train–Gen)
Epoch


FID Generalization Gap vs Epoch
FID Generalization Gap
FID Gap (Test–Gen − Train–Gen)
Epoch

- Early epochs show higher distances, then stabilize around epoch ~8–10.
- Coverage (Train–Gen) stays above the Train–Test baseline, as expected (generated ≠ real), but trends downward as training improves.
- Fidelity (Test–Gen) mirrors the coverage trend; gaps remain small and stable across later epochs, suggesting no severe overfitting.

2. Active Phase: OT-Enhanced Diffusion Training

2.1 Goal

Improve a baseline DDPM on MNIST by adding an Optimal Transport (OT) term during training and OT guidance during sampling. We want the generator's feature distribution to move closer to real data, which should reduce both OT distance and FID and make digits look cleaner.

2.2 Baseline diffusion (what we start from)

What the baseline learns.
A diffusion model receives a noised image X_t at a random timestep t and predicts the noise εθ(xt,t)that created it. We train it with a plain mean-squared error (MSE) between the predicted and true noise.

Forward noising.
Let βt be the variance schedule (set by the GaussianDiffusion object). Define:

$$\alpha_t = 1 - \beta_t, \qquad \bar{\alpha}_t = \prod_{s=0}^{t} \alpha_s.$$

The forward process is:

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I).$$

Training loss.
For a batch x0 we pick random t, make xt with the formula above, run the network, and compute:

$$\mathcal{L}_{\text{MSE}} = \|\varepsilon_\theta(x_t, t) - \varepsilon\|_2^2.$$

```python
def compute_baseline_loss(model, diffusion, images, device):
    """Standard diffusion loss"""
    # Batch size
    b = images.size(0)

    # Sample a (possibly different) timestep t for each image in the batch, uniform in [0, T-1]
    t = torch.randint(0, diffusion.num_timesteps, (b,), device=device).long()

    # Draw Gaussian noise with the same shape as images
    noise = torch.randn_like(images)

    # Forward diffuse clean images x0 to xt using the known noising process q(xt|x0, t)
    noisy_images = diffusion.q_sample(x_start=images, t=t, noise=noise)

    # The model tries to predict the noise that was added at timestep t
    predicted_noise = model(noisy_images, t)

    # Standard DDPM objective: MSE between predicted noise and true noise
    mse_loss = F.mse_loss(predicted_noise, noise)

    # Return:
    #  - total loss (here just mse_loss)
    #  - a detached copy of mse_loss for logging (no gradients)
    #  - a zero OT term (kept for consistent return signature)
    return mse_loss, mse_loss.detach(), torch.tensor(0.0, device=device)
```

*αt̄(tag) is the running product of (1−βt); it controls the blend between signal and noise across timesteps

Training setup.

- Epochs: 35
- Batch size: 128
- LR: 2e-4
- Optimizer: Adam
- Checkpoints: saved every epoch to _ddpm_mnist_checkpoints_baseline/

2.3 OT-enhanced training (what we add)

We keep the MSE term but add an OT distance in feature space between the model's predicted clean images and a disjoint reference batch of real images.

Frozen feature extractor.
We load a small CNN trained on MNIST classification (from the passive phase), freeze it, and use its 128-D penultimate layer as features. Images are mapped from [−1,1to [0,1] before the feature net.

How we form the OT term.

1. From xt and the model's predicted noise, reconstruct x^0(x0 hat) by predict_start_from_noise.
2. Smooth with tanh for stability.
3. Extract features for x^0 (with gradients) and for a separate real batch (no gradients).
4. Compute Sinkhorn OT distance with GeomLoss (p=2, blur=SINKHORN_BLUR).
5. Combine with a weight λ:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{MSE}} + \lambda \cdot \mathcal{L}_{\text{OT}}.$$

Key hyperparameters:

- **λ (TRANSPORT_WEIGHT) = 0.001** — strength of the OT term.
- **Sinkhorn blur (SINKHORN_BLUR) = 0.2** — entropic regularization; larger = smoother/softer match.
- **Reference pool size = 2000** — how many real images we compare to per step (drawn disjoint from the current training batch).

```python
def compute_ot_enhanced_loss(model, diffusion, images, device, feature_extractor,
                             reference_batch=None, transport_weight=TRANSPORT_WEIGHT,
                             blur=SINKHORN_BLUR):
    """OT-enhanced loss"""
    # Batch size
    b = images.size(0)

    # Sample per-example timestep
    t = torch.randint(0, diffusion.num_timesteps, (b,), device=device).long()

    # Gaussian noise and forward diffusion to build xt
    noise = torch.randn_like(images)
    noisy_images = diffusion.q_sample(x_start=images, t=t, noise=noise)

    # Predict the noise at timestep t and compute the usual MSE loss
    predicted_noise = model(noisy_images, t)
    mse_loss = F.mse_loss(predicted_noise, noise)

    # If we don't have a feature extractor or a reference batch, fall back to baseline loss
    if feature_extractor is None or reference_batch is None:
        return mse_loss, mse_loss.detach(), torch.tensor(0.0, device=device)

    # Reconstruct the model's estimate of the clean image x0 from (xt, t, predicted_noise)
    predicted_x0 = diffusion.predict_start_from_noise(noisy_images, t, predicted_noise)

    # Light squashing to keep values bounded; also promotes numerical stability for features
    pred_smooth = torch.tanh(predicted_x0)
    ref_smooth = torch.tanh(reference_batch)

    # Extract features for the predicted images WITH gradients
    # (so that the OT term can backpropagate into the generator)
    pred_features = extract_features_with_gradients(
        feature_extractor, pred_smooth, enable_gradients=True
    )

    # Extract features for the reference real images WITHOUT gradients
    # (we don't want/need to update the feature net or real data)
    with torch.no_grad():
        ref_features = extract_features_with_gradients(
            feature_extractor, ref_smooth, enable_gradients=False
        )

    # Define Sinkhorn OT loss in feature space:
    #   - loss="sinkhorn": entropic regularization (fast/stable)
    #   - p=2: 2-Wasserstein geometry (squared Euclidean ground cost)
    #   - blur: regularization/smoothing strength
    ot_loss_fn = SamplesLoss(loss="sinkhorn", p=2, blur=blur, backend="auto")

    # Compute OT distance between distributions of predicted vs reference features
    ot_distance = ot_loss_fn(pred_features.float(), ref_features.float())

    # Combine objectives: standard noise MSE + λ * OT(feature_pred, feature_ref)
    total_loss = mse_loss + transport_weight * ot_distance

    # Return:
    #  - total loss (drives training)
    #  - detached MSE (for logging)
    #  - detached, weighted OT term (for logging)
    return total_loss, mse_loss.detach(), (transport_weight * ot_distance).detach()
```

Why disjoint reference sampling?

We sample the reference batch from indices not in the current training batch. This prevents trivial matching and encourages the generator to align with the overall data distribution, not just the same images.

```python
# ============================================================================
# Disjoint Reference Sampling
# ============================================================================

def get_disjoint_reference_indices(total_size, current_indices, reference_size):
    """Sample disjoint reference indices"""
    # Build the set of ALL possible indices in the dataset: {0, 1, ..., total_size-1}
    all_indices = set(range(total_size))
    # Convert the currently-used batch indices to a set for fast subtraction
    forbidden = set(current_indices)
    # Compute indices that are NOT in the current batch (the "disjoint" pool)
    available = list(all_indices - forbidden)

    # If the available pool is smaller than we want, sample WITH replacement
    # (keeps the reference_size fixed but may repeat some indices)
    if len(available) < reference_size:
        return random.choices(available, k=reference_size)
    else:
        # Otherwise, sample WITHOUT replacement to avoid duplicates
        return random.sample(available, reference_size)
```

2.4 OT-guided sampling

After training, we also nudge samples during the first DDIM steps. We keep the model frozen and optimize the image tensor itself for a few iterations to reduce OT distance to a set of reference features:

- **guidance_scale** = 0.5
- **num_guide_steps** = 10 DDIM steps where we allow guidance
- **OT_CORRECTION_STEPS** = 3 Adam steps on x0 per guided DDIM step

```python
def apply_ot_guidance(x0, feature_extractor, reference_features, guidance_scale, blur, steps):
    """Apply OT guidance to x0"""
    # Create a leaf tensor we can optimize directly
    x0_corrected = x0.clone().detach().requires_grad_(True)

    # Optimize the image tensor itself using Adam (small learning rate)
    optimizer = torch.optim.Adam([x0_corrected], lr=0.01)

    # Sinkhorn OT distance in feature space (p=2 Wasserstein, entropic smoothing via 'blur')
    ot_loss_fn = SamplesLoss(loss="sinkhorn", p=2, blur=blur, backend="auto")

    for _ in range(steps):
        optimizer.zero_grad()

        # Extract features with gradients so OT loss can update x0_corrected
        current_features = extract_features_with_gradients(
            feature_extractor, x0_corrected, enable_gradients=True
        )

        # Compute OT distance to the target (precomputed) reference feature distribution
        ot_loss = ot_loss_fn(current_features, reference_features)

        # Scale the guidance effect and backprop through x0_corrected (not the model)
        (guidance_scale * ot_loss).backward()
        optimizer.step()

        # Keep images in [-1,1] after each small correction step
        x0_corrected.data.clamp_(-1, 1)

    # Return the corrected images, detached from graph
    return x0_corrected.detach()
```

2.5 Training loop & logging

We train two runs:

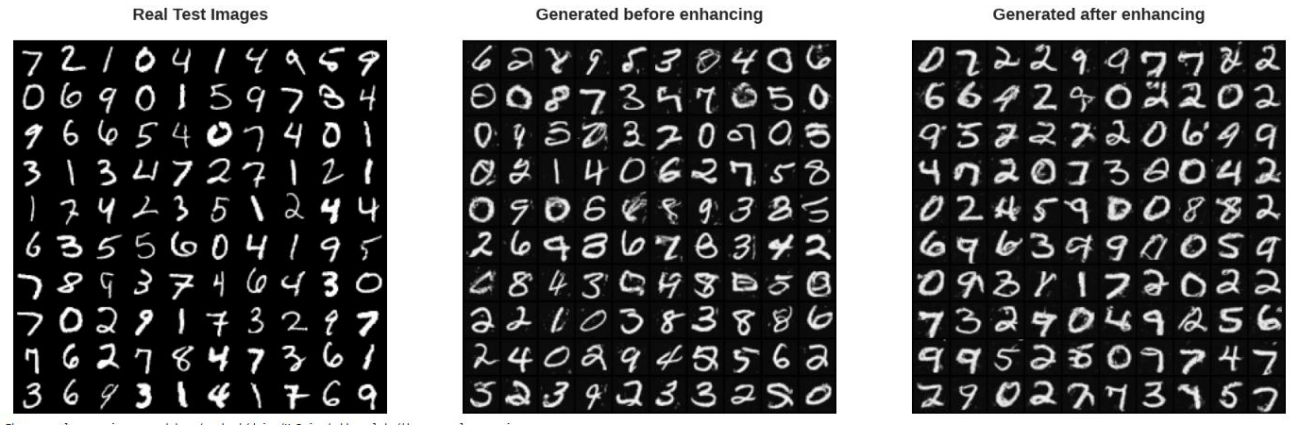
1. **Baseline**: only MSE.
2. **OT-Enhanced**: MSE + λ·OT.

We log per-epoch:

- total train loss
- MSE part
- weighted OT part
- validation loss

2.6 Results:

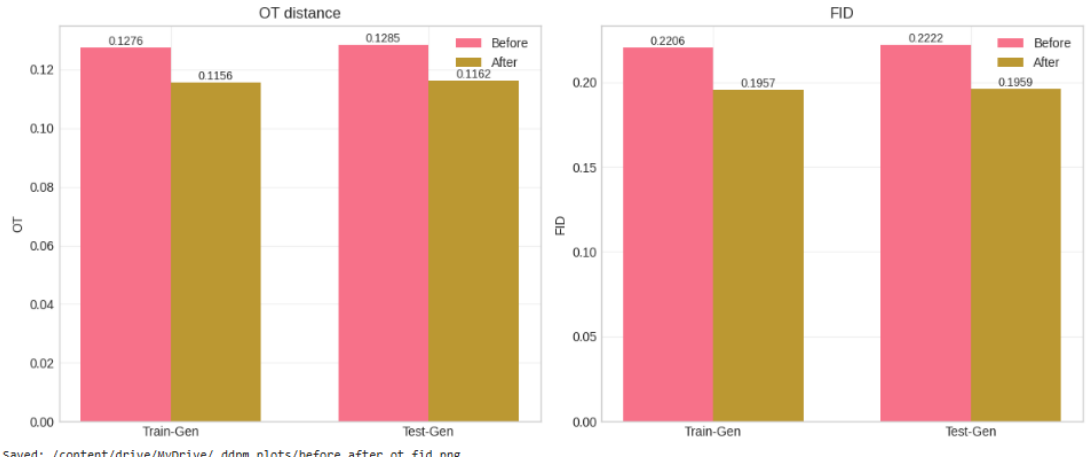
Qualitative comparison:



- Before enhancing, some digits are fuzzy or broken (e.g., loops not closed, strokes uneven).
- After adding OT, digits look cleaner and more consistent with real ones.
- Background becomes more uniform; fewer gray smudges.
- Variety is preserved (we still see a spread of digits, not mode collapse).

Before vs After: distances:

```
============================================================
EVALUATION COMPLETE - SUMMARY
============================================================
OT Distance Improvements:
  Train-Gen: +0.011940
  Test-Gen:  +0.012332
FID Score Improvements:
  Train-Gen: +0.024928
  Test-Gen:  +0.026257
```

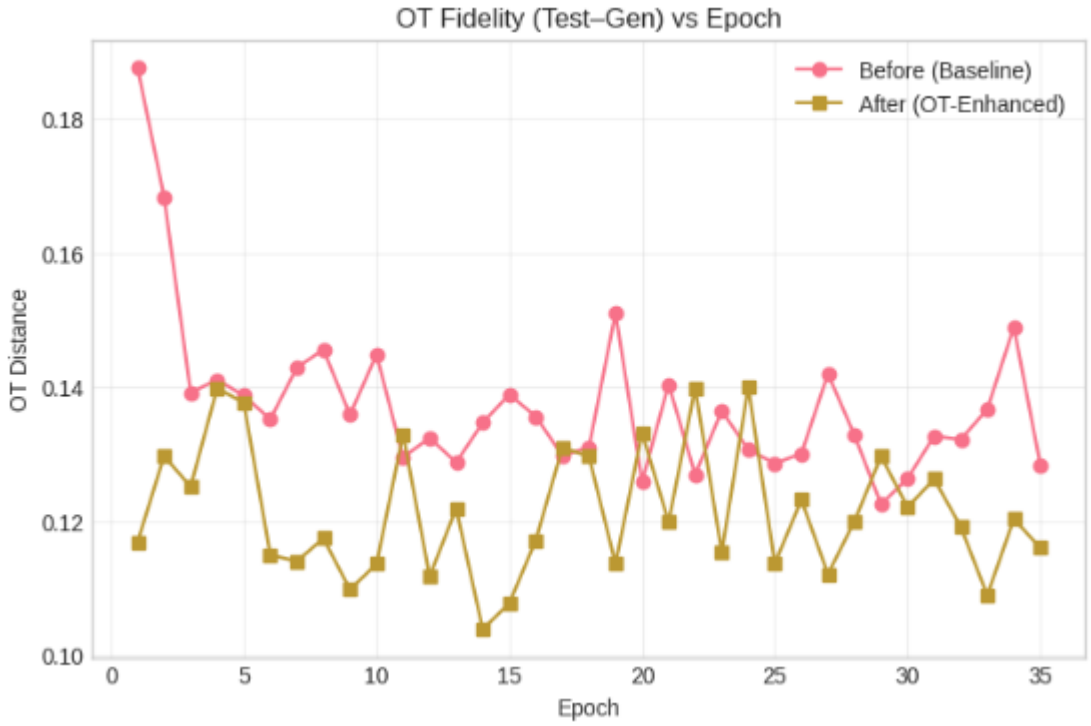


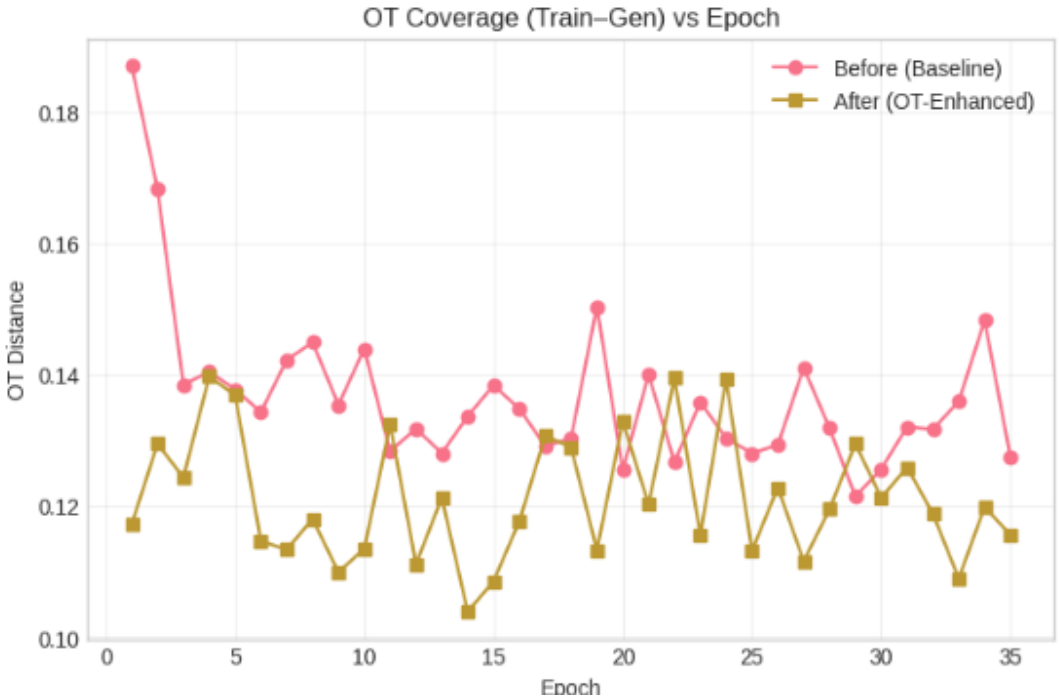
Saved: /content/drive/MyDrive/_ddpm_plots/before_after_ot_fid.png

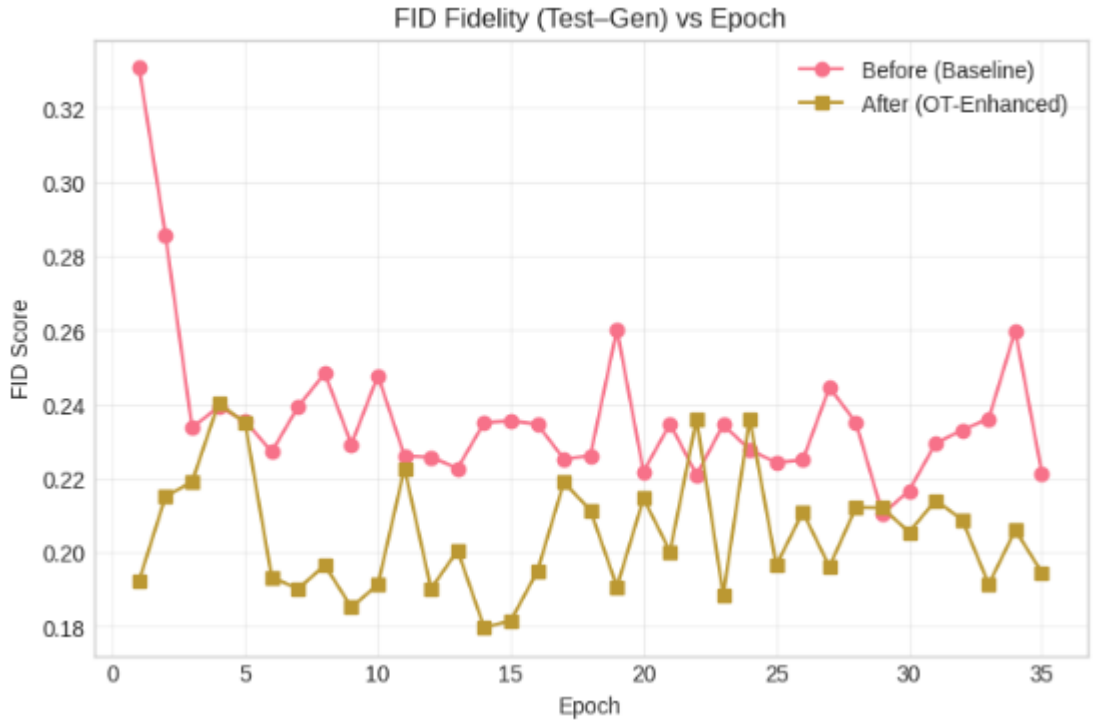
- OT (Test–Gen) drops from ~0.1285 → ~0.1162.
- FID (Test–Gen) drops from ~0.2222 → ~0.1959.

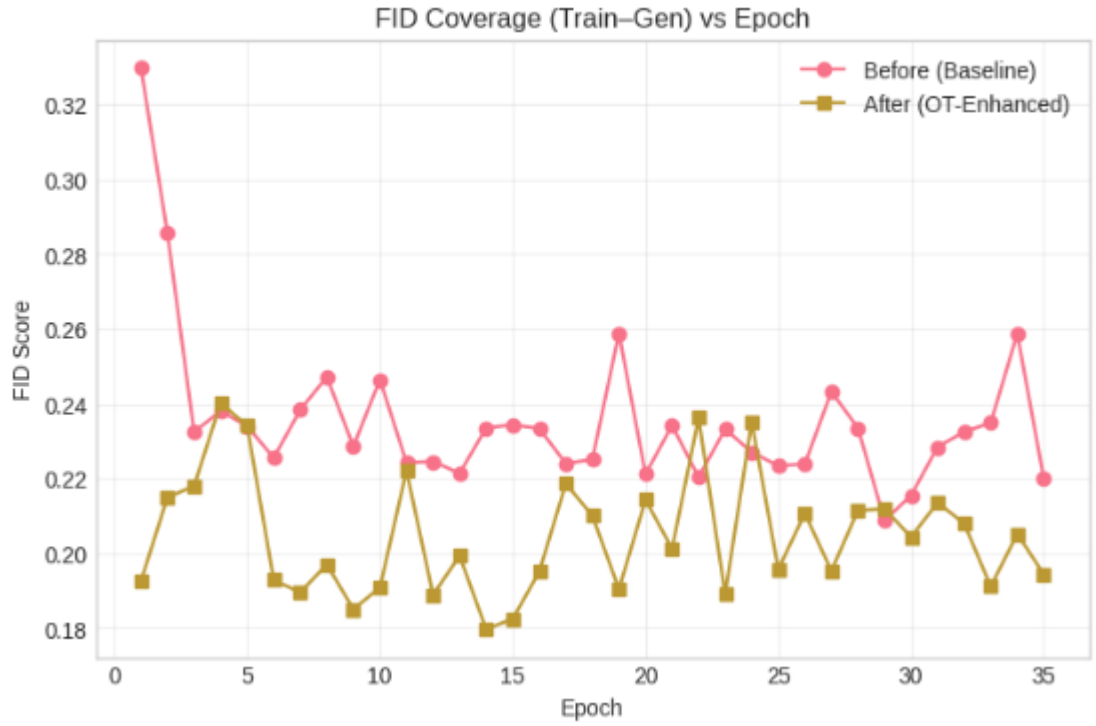
  What it means: smaller is better. Both scores improve, so the generator is closer to real data distributions and keeps coverage.

Epoch-wise curves (Before vs After):









- The gold curve (OT-Enhanced) stays below the pink curve (Baseline) across most epochs → consistent improvement.
- Both coverage (Train–Gen) and fidelity (Test–Gen) improve, so we're not trading one for the other.
- The curves are not perfectly monotonic.

Conclusions & What We Learned

Summary

- We built a two-phase pipeline for evaluating and improving a diffusion model on MNIST.
    - Passive phase (Section 1): trained a small CNN feature extractor and used it to measure Optimal Transport (OT) and FID under controlled stress tests (added noise, removed classes). These tests verified that both metrics respond in the expected directions (↑ noise → ↑ distance; ↓ diversity → ↑ distance).
    - Active phase (Section 2): trained a baseline DDPM and an OT-enhanced DDPM that adds a Sinkhorn OT term in feature space to the standard MSE objective, plus OT guidance during sampling.

What improved:

- Quantitative: both coverage (Train–Gen) and fidelity (Test–Gen) improved after adding the OT term.
- Qualitative: side-by-side grids show cleaner strokes, fewer faint artifacts, and more uniform backgrounds after enhancement, while variety is preserved (no obvious mode collapse).

Why the approach helped

- The MSE term teaches the model to predict the diffusion noise correctly for each timestep.
- The OT term shapes the distribution of predicted clean images x^0 so that, in a frozen feature space trained on MNIST, the generated features are closer to real features. This adds a global distribution constraint that the pixel-space MSE alone does not provide.
- OT-guided sampling adds a small, direct correction on the images during the first DDIM steps, further nudging samples toward the reference feature distribution without changing network weights.

Practical lessons

- Choosing λ (transport weight) matters.
    - Too small → little effect; too large → can fight MSE and hurt training.
    - λ = 0.001 worked reliably on MNIST.
- Sinkhorn blur controls stability vs. sharpness of the feature matching.
    - 0.2 was a good middle ground; lower blur gave noisier gradients, higher blur made the signal too soft.
- Disjoint reference sampling (do not reuse the same batch for references) prevents trivial matching and encourages alignment with the full data distribution.

Limitations:

- Feature dependence: OT is computed in the learned feature space. If the feature extractor is weak or biased, the OT signal may not reflect true image quality.
- Compute cost: computing Sinkhorn OT every step adds overhead (feature passes + Sinkhorn). We mitigated it with batch-level reference pools and moderate blur.
- Scope: Results are on MNIST. Gains on harder datasets may require stronger features, tuning λ/blur, and longer training.