Lab 06: Generative Models

112550127 Pin-Kuan Chiang

August 21, 2025

1 Introduction

In this lab we explore **Denoising Diffusion Probabilistic Models (DDPMs)**, a recent generative modeling approach that builds images by gradually denoising random noise. The idea is simple: during training, we learn how to add noise step by step using a predefined schedule, and train a UNet-based model to predict the noise at each step. During sampling, the process is reversed, starting from pure Gaussian noise and progressively recovering a clean image.

Our goal is to train a DDPM to synthesize CLEVR-style scenes, either conditioned on object labels or in an unconditional setting. We rely on Hugging Face's `diffusers` library for the noise scheduler, which supports multiple β-schedules (`linear`, `scaled_linear`, `squaredcos_cap`). The dataset comes with splits (`train.json`, `test.json`, `new_test.json`), allowing us to evaluate both in-distribution and slightly shifted data.

The evaluation is twofold: (i) qualitatively, by looking at generated image grids to see if the model produces coherent shapes and colors, and (ii) quantitatively, by checking classification accuracy on generated samples using a pretrained evaluator. Overall, this lab helps us get hands-on practice with diffusion models, experiment with design choices like the noise schedule or number of timesteps, and better understand how they affect the quality of the generated images.

2 Implementation details

2.1 Model architecture

2.1.1 Sinusoidal Embedding

```python
class SinusoidalEmbedding(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.dim = dim
        self.proj = nn.Sequential(
            nn.Linear(dim, dim),
            nn.SiLU(),
            nn.Linear(dim, dim),
        )

    def forward(self, t: torch.Tensor) -> torch.Tensor:
        t = t.view(-1).float()
        half = self.dim // 2
        device = t.device
        freqs = torch.exp(
            -math.log(10000) * torch.arange(0, half, device=device, dtype=torch.float32) /
                float(half)
        )
        args = t[:, None] * freqs[None, :]
        emb = torch.cat([torch.sin(args), torch.cos(args)], dim=-1)
        if self.dim % 2 == 1:
            emb = F.pad(emb, (0, 1), value=0.0)
        emb = self.proj(emb)
        return emb
```

This module converts a scalar time step t into a k-dimensional embedding. It builds fixed sinusoidal Fourier features by applying sine and cosine at exponentially spaced frequencies, then uses a small two-layer MLP to learn a task-specific mixing of these features. The result is a tensor of shape $(B, \texttt{dim})$.

2.1.2 AdaNorm

```python
class AdaptiveGroupNorm(nn.Module):
    def __init__(self, num_groups: int, num_channels: int, time_emb_dim: int,
                 cond_dim: Optional[int] = None, eps: float = 1e-6):
        super().__init__()
        self.num_channels = num_channels
        self.base_norm = nn.GroupNorm(num_groups=num_groups,
                                      num_channels=num_channels, eps=eps)
        in_dim = time_emb_dim + (cond_dim if cond_dim is not None and cond_dim > 0 else 0)
        self.proj = nn.Linear(in_dim, num_channels * 2)
        self.act = nn.SiLU()
        self.cond_dim = cond_dim

    def forward(self, x: torch.Tensor, t_emb: torch.Tensor,
                cond_emb: Optional[torch.Tensor] = None) -> torch.Tensor:
        B = x.shape[0]
        inp = torch.cat([t_emb, cond_emb], dim=-1) if (self.cond_dim is not None and cond_emb
            is not None) else t_emb
        params = self.proj(self.act(inp))                       # (B, 2*C)
        params = params.view(B, 2, self.num_channels)           # (B, 2, C)
        gamma = params[:, 0].view(B, self.num_channels, 1, 1)  # (B, C, 1, 1)
        beta  = params[:, 1].view(B, self.num_channels, 1, 1)  # (B, C, 1, 1)
        out = self.base_norm(x)
        out = out * (1.0 + gamma) + beta
        return out
```

This layer first normalizes `x` with GroupNorm and then conditions the result by applying per-channel scale and shift:

$$\text{out} = \text{GN}(x) \cdot (1 + \gamma) + \beta.$$

We concatenate `t_emb` (and optionally `cond_emb`), apply `SiLU` then a linear layer to predict $2C$ values per batch, reshape into $\gamma, \beta \in \mathbb{R}^{(B,C,1,1)}$ and broadcast to modulate the normalized features.

2.1.3 Residual Block

```python
class ResidualBlock(nn.Module):
    def __init__(self, in_ch: int, out_ch: int, time_emb_dim: int,
                 groups: int = 8, dropout: float = 0.0, cond_dim: Optional[int] = None):
        super().__init__()
        self.in_ch = in_ch
        self.out_ch = out_ch before conv1
        self.norm1 = AdaptiveGroupNorm(num_groups=groups, num_channels=in_ch, time_emb_dim=
            time_emb_dim, cond_dim=cond_dim)
        self.act = nn.SiLU()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.time_proj = nn.Linear(time_emb_dim, out_ch)
        self.cond_dim = cond_dim
        if cond_dim is not None and cond_dim > 0:
            self.cond_proj = nn.Linear(cond_dim, out_ch)
        else:
            self.cond_proj = None
        self.norm2 = AdaptiveGroupNorm(num_groups=groups, num_channels=out_ch, time_emb_dim=
            time_emb_dim, cond_dim=cond_dim)
        self.dropout = nn.Dropout(dropout) if dropout > 0.0 else nn.Identity()
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.shortcut = nn.Conv2d(in_ch, out_ch, kernel_size=1) if in_ch != out_ch else nn.
            Identity()

    def forward(self, x: torch.Tensor, t_emb: torch.Tensor, cond_emb: Optional[torch.Tensor] =
         None) -> torch.Tensor:
        h = self.conv1(self.act(self.norm1(x, t_emb, cond_emb)))
        t = self.time_proj(self.act(t_emb))  # (B, out_ch)
        h = h + t[:, :, None, None]
        if cond_emb is not None and self.cond_proj is not None:
            c = self.cond_proj(self.act(cond_emb))  # (B, out_ch)
            h = h + c[:, :, None, None]
        h = self.conv2(self.dropout(self.act(self.norm2(h, t_emb, cond_emb))))
        return h + self.shortcut(x)
```

This residual block first applies an `AdaptiveGroupNorm` + SiLU and a 3×3 conv, then adds time conditioning (and optionally an extra conditioning vector) projected to out_ch and broadcast over spatial dims. After a second adaptive normalization, activation and dropout, a second 3×3 conv is applied and the result is added to a residual shortcut (a 1×1 conv if channels differ). In short: normalization $\rightarrow$ conv $\rightarrow$ add time/cond $\rightarrow$ normalization $\rightarrow$ conv $\rightarrow$ residual add, producing an output of the same spatial size with out_ch channels.

2.1.4 Attention Block

```
class AttentionBlock(nn.Module):
    def __init__(self, n_channels: int, n_heads: int = 4, d_k: Optional[int] = None,
                 norm_groups: int = 8, time_emb_dim: Optional[int] = None,
                 cond_dim: Optional[int] = None):
        super().__init__()
        self.n_channels = n_channels
        self.n_heads = n_heads
        if d_k is None:
            assert n_channels % n_heads == 0,
            d_k = n_channels // n_heads
        self.d_k = d_k
        if time_emb_dim is not None:
            self.norm = AdaptiveGroupNorm(num_groups=norm_groups,
                                          num_channels=n_channels,
                                          time_emb_dim=time_emb_dim,
                                          cond_dim=cond_dim)
        else:
            self.norm = nn.GroupNorm(num_groups=norm_groups, num_channels=n_channels, eps=1e
                -6)
        self.qkv = nn.Linear(n_channels, n_heads * d_k * 3)
        self.out = nn.Linear(n_heads * d_k, n_channels)

    def forward(self, x: torch.Tensor, t_emb: Optional[torch.Tensor] = None,
                cond_emb: Optional[torch.Tensor] = None) -> torch.Tensor:
        B, C, H, W = x.shape
        seq_len = H * W
        if isinstance(self.norm, AdaptiveGroupNorm):
            xn = self.norm(x, t_emb, cond_emb)  # (B, C, H, W)
        else:
            xn = self.norm(x)
        xn = xn.view(B, C, seq_len).permute(0, 2, 1)  # (B, seq, C)
        qkv = self.qkv(xn)  # (B, seq, 3 * n_heads * d_k)
        qkv = qkv.reshape(B, seq_len, 3, self.n_heads, self.d_k).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]  # each: (B, n_heads, seq, d_k)

        scale = 1.0 / math.sqrt(self.d_k)
        attn = torch.matmul(q, k.transpose(-2, -1)) * scale  # (B, n_heads, seq, seq)
        attn = torch.softmax(attn, dim=-1)
        out = torch.matmul(attn, v)  # (B, n_heads, seq, d_k)
        out = out.permute(0, 2, 1, 3).reshape(B, seq_len, self.n_heads * self.d_k)
        out = self.out(out)  # (B, seq, C)
        out = out.permute(0, 2, 1).view(B, C, H, W)
        return out + x  # residual
```

Performs multi-head self-attention over spatial positions of an image tensor. The block first normalizes channels (AdaptiveGroupNorm if time/cond are provided, otherwise GroupNorm), flattens $H \times W$ into a sequence, and applies a single linear to produce concatenated Q/K/V. Q/K/V are split into heads, scaled dot-product attention is computed, heads are recombined and projected back to channel dimension, reshaped to (B, C, H, W), and added residually to the input. Input $(B, C, H, W) \rightarrow$ sequence length $S = H * W$; q/k/v shapes become $(B, \text{n\_heads}, S, d_k)$. The normalization being adaptive lets time/conditioning influence the attention inputs. (Default behavior sets d_k so that $\text{n\_heads} \cdot d_k = C$.)

2.1.5 Up / down sampling

```
class Downsample(nn.Module):
    def __init__(self, channels: int):
        super().__init__()
        self.op = nn.Conv2d(channels, channels, kernel_size=3, stride=2, padding=1)
```

```
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.op(x)


class Upsample(nn.Module):
    def __init__(self, channels: int):
        super().__init__()
        self.op = nn.ConvTranspose2d(channels, channels, kernel_size=4, stride=2, padding=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.op(x)
```

Downsample halves the spatial resolution using a learned 3×3 convolution with stride 2 and padding 1, mapping $(B, C, H, W) \to (B, C, H/2, W/2)$.
Upsample doubles the spatial resolution with a transposed convolution (4×4 kernel, stride 2, padding 1), mapping $(B, C, H, W) \to (B, C, 2H, 2W)$.

2.1.6 Unet Architecture

```
class UNet(nn.Module):
    def __init__(self, ..... ):
        super().__init__()
        # project image + time/cond embeddings
        self.time_dim = base_channels * time_emb_factor
        self.time_emb = SinusoidalEmbedding(self.time_dim)
        self.image_proj = nn.Conv2d(image_channels, base_channels, 3, padding=1)
        # build down path
        self.downs = nn.ModuleList([... ResidualBlocks, Attention, Downsample ...])
        # middle (bottleneck)
        self.mid = nn.ModuleList([
            ResidualBlock(...), AttentionBlock(...), ResidualBlock(...)
        ])
        # build up path (with skip connections)
        self.ups = nn.ModuleList([... ResidualBlocks, Attention, Upsample ...])
        # final projection
        self.final_norm = AdaptiveGroupNorm(...) or GroupNorm(...)
        self.final_act = nn.SiLU()
        self.final_conv = nn.Conv2d(base_channels, image_channels, 3, padding=1)
    def forward(self, x, t, cond=None):
        # 1. embed time (and condition if provided)
        # 2. pass through down path, saving skip connections
        # 3. process in middle block
        # 4. pass through up path, merging skips
        # 5. final normalization, activation, projection
        return out
```

This UNet follows an encoder–decoder design with skip connections and is adapted for diffusion. It ingests an image (B, C, H, W), embeds the timestep via `SinusoidalEmbedding`, and optionally uses conditional vectors to modulate normalization (`AdaptiveGroupNorm`) and residual blocks.

- **Down path.** At resolution stage i, a stack of `ResidualBlocks` (time/cond-conditioned) extracts features; optional `AttentionBlock` improves long-range dependencies. A `Downsample` halves spatial size between stages. We push feature maps to a skip list before each downsample.
- **Middle.** A bottleneck of `ResidualBlock`–`AttentionBlock`–`ResidualBlock` mixes local and global information.
- **Up path.** Each stage begins by concatenating the matching skip feature (after resizing if needed), then applies `ResidualBlocks` (and optional attention). A `Upsample` doubles spatial size between stages.
- **Final.** An adaptive normalization (conditioned on time/cond), `SiLU`, and a 3×3 conv map features back to the image channels.

2.1.7 Architecture Overview (base_channel = 32)

- **Encoder**

  - Layer 0: 3 → 32 channels 2 ResNet blocks
  - Layer 1: 32 → 64 channels 2 ResNet blocks
  - Layer 2: 64 → 128 channels 2 ResNet blocks + Self-Attention
  - Layer 3: 128 → 256 channels 2 ResNet blocks

- **Bottleneck**: 256 channels 2 ResNet blocks + Self-Attention
- **Decoder**
  - Layer 3: 256+256 → 256 channels 2 ResNet blocks
  - Layer 2: 256+128 → 128 channels 2 ResNet blocks + Self-Attention
  - Layer 1: 128+64 → 64 channels 2 ResNet blocks
  - Layer 0: 64+32 → 32 channels 2 ResNet blocks
- **Output**: 32 → 3 channels (RGB)

2.1.8 Diffusion

```python
class DiffusionDDPM:
    """Minimal DDPM wrapper around a UNet-like model + diffusers.DDPMScheduler."""
    def __init__(self, model, num_train_timesteps=1000, device=None, scheduler=None):
        self.model = model
        self.device = device or next(model.parameters()).device
        self.scheduler = scheduler or DDPMScheduler(
            num_train_timesteps=num_train_timesteps, beta_schedule="linear"
        )

    def add_noise(self, clean_images, timesteps, noise=None):
        noise = torch.randn_like(clean_images) if noise is None else noise
        return self.scheduler.add_noise(clean_images, noise, timesteps), noise
```

This class provides a wrapper around Hugging Face's `DDPMScheduler` to implement both the forward diffusion process (adding noise) and the reverse process (sampling). The implementation supports three β-schedules commonly used in DDPM research: `linear`, `scaled_linear`, and `squaredcos_cap`. By default, the scheduler is initialized with a linear schedule, but the design allows easy substitution.

Noise Addition. The forward diffusion step is delegated entirely to the scheduler, ensuring consistency with the underlying β-schedule. Given clean images x_0 and sampled timesteps t, Gaussian noise $\epsilon \sim \mathcal{N}(0, I)$ is injected to produce

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon,$$

where $\bar{\alpha}_t$ is precomputed by the scheduler according to the chosen β-schedule. Importantly, the method returns both the noisy image and the exact ϵ, since the latter is required as the supervised target for training.

2.1.9 training

```python
for batch in dataloader:
    imgs = batch['image'].to(device)
    cond = batch['cond'].to(device)
    B = imgs.shape[0]

    # sample random timesteps
    timesteps = torch.randint(0, ddpm.num_train_timesteps, (B,), device=device).long()

    # add Gaussian noise to images
    noise = torch.randn_like(imgs)
    noisy = ddpm.scheduler.add_noise(imgs, noise, timesteps)

    # predict noise with UNet
    pred = unet(noisy, timesteps, cond)

    # optimization
    loss = loss_fn(pred, noise)
```

```
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

In our implementation, each mini-batch is corrupted by Gaussian noise at randomly sampled diffusion timesteps. The scheduler applies the forward diffusion process by injecting noise into the clean images, simulating the progressive degradation of data. The conditional UNet model then predicts this noise, and training minimizes the mean squared error between predicted and true noise. By repeatedly applying this denoising objective across epochs, the model learns to reverse the diffusion process, enabling conditional image generation during sampling.

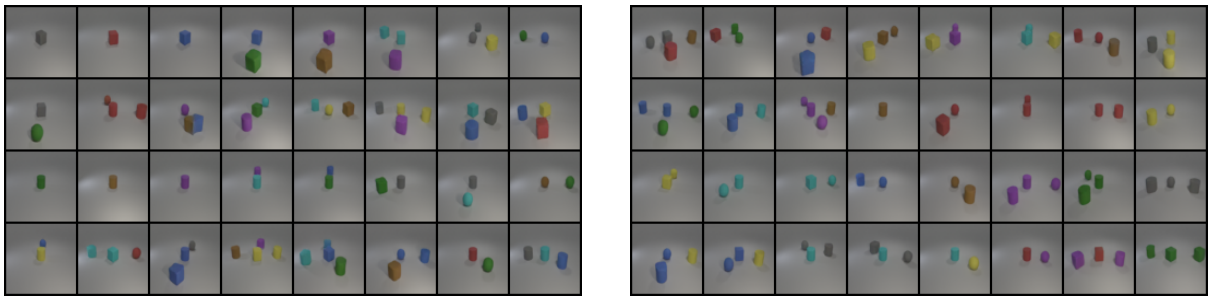
2.1.10 Sampling

```python
def sample_conditional(model, scheduler, cond, shape, device):
    model = model.to(device)
    model.eval()
    num_timesteps = scheduler.num_train_timesteps

    with torch.no_grad():
        # start from Gaussian noise
        sample = torch.randn(shape, device=device)
        B = shape[0]
        for t in range(num_timesteps - 1, -1, -1):
            ts = torch.full((B,), t, device=device, dtype=torch.long)
            pred_noise = model(sample, ts, cond)
            step_out = scheduler.step(pred_noise, t, sample)

            # scheduler returns previous sample
            if hasattr(step_out, 'prev_sample'):
                sample = step_out.prev_sample
            elif isinstance(step_out, dict):
                sample = step_out['prev_sample']
            else:
                sample = step_out
    model.train()
    return sample
```

During inference, the sampling procedure begins from pure Gaussian noise of the target shape. At each timestep, the conditional UNet predicts the noise component, and the scheduler updates the latent sample one step closer to the data distribution. This iterative denoising continues until timestep $t = 0$, producing a clean image that is consistent with the given conditioning vector. Thus, reverse diffusion enables controlled image synthesis, guided by the learned denoising model.

3 Results and discussion

3.1 Synthetic image grids



(a) Synthetic grid for `test.json` (b) Synthetic grid for `new_test.json`

Figure 1: Generated synthetic image grids on the two test sets.

From Figure 1, we present the **baseline model** results. The baseline is able to generate coherent object arrangements conditioned on the textual labels. However, We can still observe some errors such as blurred boundaries or incorrect colors.

The corresponding classification performance is 84.7% on `test.json` and 80.9% on `new_test.json`.

Parameter	Value
Base channels	32
Batch size	64
Beta schedule	Linear
Epochs	100
Image size	64
Learning rate	2×10^{-4}
Timesteps	1000
Optimizer	AdamW
Accuracy (`test.json`)	84.7%
Accuracy (`new_test.json`)	86.9%

Table 1: Baseline model hyperparameters and performance.

3.2 Denoising process image

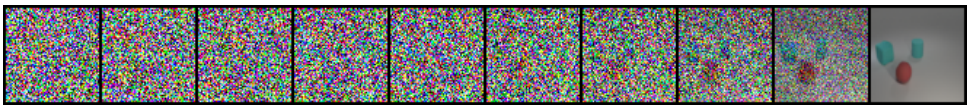
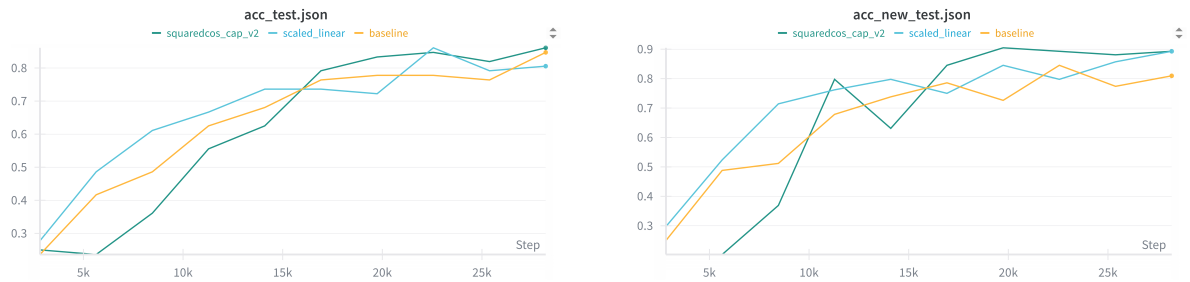


Figure 2: Denoising trajectory for label set [“red sphere”, “cyan cylinder”, “cyan cube”]. The figure illustrates the progressive refinement from Gaussian noise ($t = T$) to a clean image ($t = 0$).

Figure 2 shows the iterative denoising process. Early steps ($t \approx T$) are dominated by Gaussian noise, but as t decreases, shapes become recognizable. At intermediate timesteps, object outlines appear, though color assignments may still be ambiguous. The final timestep ($t = 0$) produces distinct objects matching the labels.

3.3 Discussion of extra implementations or experiments

Noise schedule We compared several schedulers, including linear, scaled_linear, and squaredcos_cap_v2. As shown in Figure 3, , the `squaredcos_cap_v2` scheduler consistently outperforms the others, particularly in mid-to-late training. It not only stabilizes convergence but also yields higher perceptual quality on both `test.json` and `new_test.json`, making it the most suitable choice overall.



(a) Schedulers on `test.json`

(b) Schedulers on `new_test.json`

Figure 3: Effect of different noise schedules. The `squaredcos_cap_v2` scheduler provides the best stability and quality.

Base channel configuration. Figure 4 compares different base channel sizes. Larger settings such as 64 and 128 channels indeed improve fidelity and accuracy, but they require significantly longer training time. With 32 channels, the model already achieves above 0.8 classification accuracy, indicating that this lighter configuration is sufficient for stable training while being more computationally efficient. Thus, higher channels are beneficial but may be unnecessary unless peak fidelity is the priority.

(a) Base channel on `test.json` (b) Base channel on `new_test.json`

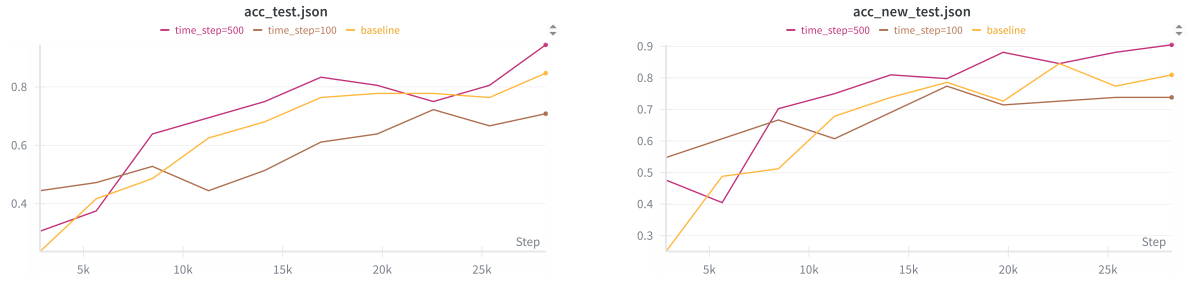
Figure 4: Effect of different base channel sizes.

Image resolution (input size). Motivated by the fact that the evaluator was trained using samples downscaled to 32×32, we also experimented with training our diffusion model at the same resolution. However, as seen in Figure 5, the 32×32 setting performed extremely poorly, leading to blurry, indistinguishable objects and very low classification accuracy. In contrast, larger input sizes preserved clearer structures and yielded far superior results. This indicates that while the evaluator can operate at 32×32, the generative model itself requires higher resolution training to learn meaningful structures.

(a) Different size on `test.json` (b) Different size on `new_test.json`

Figure 5: Effect of input resolution. Training at 32×32 leads to severe quality degradation.

Number of timesteps. We evaluated different diffusion lengths, illustrated in Figure 6. At $T = 100$, the model failed to generate a picture with right color, shown as Figure 7. Interestingly, the best performance was observed at $T = 500$, while $T = 1000$ did not yet outperform it, likely because the current training epochs were insufficient for the model to fully learn such long denoising trajectories. This suggests that while higher T has theoretical benefits, in practice $T = 500$ strikes the best balance between accuracy and training cost under limited epochs.



(a) Different timesteps on `test.json` (b) Different timesteps on `new_test.json`

Figure 6: Effect of timestep number. $T = 500$ gives the best trade-off under current training.

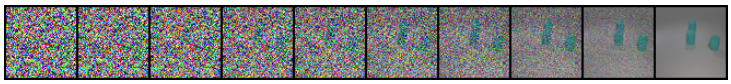


Figure 7: Failure case at 100 timesteps: objects remain noisy and the model often fails to learn the conditioning properly, resulting in wrong color assignments.

These observations suggest that insufficient timesteps prevent the model from effectively incorporating conditional information, leading to frequent mistakes such as incorrect object colors. Potential remedies include improving dataset balance, adopting more robust schedulers (e.g., squared cosine), or extending training epochs to allow the model to better utilize longer denoising schedules.

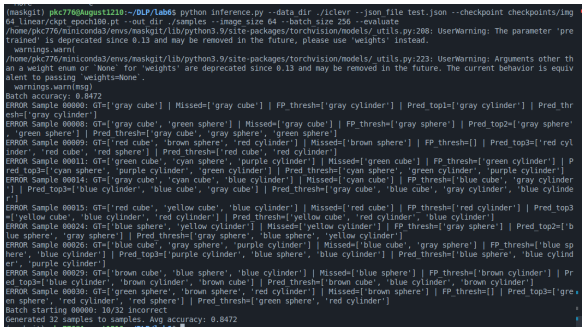
4 Experimental results

4.1 Accuracy

Dataset	Accuracy
`test.json`	0.8472
`new_test.json`	0.8690

Table 2: Classification accuracy on the two test sets.

4.2 Accuracy screenshots



(a) Accuracy screenshot for `test.json`

(b) Accuracy screenshot for `new_test.json`

Figure 8: Screenshots showing classifier accuracy on both test sets.

References

[1] Zou, B. *Denoising Diffusion Probability Model (DDPM)*. GitHub repository, 2021. `https://github.com/zoubohao/DenoisingDiffusionProbabilityModel-ddpm-`

[2] Hugging Face. *Diffusers Library*. GitHub repository, 2022. `https://github.com/huggingface/diffusers`