Deep Leanring HW6

109208063 NCCU 陳詠鑫

May 2025

1 Introduction

This work aims to implement and evaluate a multi-label conditional diffusion model (Conditional DDPM) capable of generating synthetic scene images according to specified color–shape combinations (e.g., "red sphere", "cyan cylinder"). We employ the i-CLEVR dataset—which comprises eight colors and three shapes (24 object classes in total)—in scenarios where each image contains up to three objects. A pre-trained ResNet18 classifier is used to assess the accuracy of the generated images. Unlike unconditional diffusion models, our approach integrates a cross-attention mechanism to inject multi-label one-hot embeddings into the UNet backbone, enabling precise control over the multiple attribute conditions. Furthermore, we provide a full experiment pipeline featuring checkpoint resume, training progress visualization, composite grid generation, and denoising process visualization to facilitate reproducibility and report preparation.

2 Implementation Details

2.1 Data Loading and Multi-Label Encoding

- **Data structure:**
  - `objects.json` maps the 24 color–shape combinations to integer indices.
  - `train.json`, `test.json`, and `new_test.json` list image filenames and their multi-label annotations.
- **ICLEVRDataset** (in `dataset.py`):
  - On initialization, reads JSON files and builds `label2idx` and `idx2label` mappings.
  - In `__getitem__`, loads each image by filename, applies `Resize`→`ToTensor`→`Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))` to map pixel values into $[-1, 1]$.

– Converts the list of labels into a length-24 one-hot tensor for conditioning.

2.2 Model Architecture: Conditional UNet with Cross-Attention

- **Base model:** We use `UNet2DConditionModel` from Hugging Face Diffusers with the following key configuration:

```
sample_size=64
in_channels=3
out_channels=3
layers_per_block=2
block_out_channels=(64,128,256)
down_block_types=("DownBlock2D","DownBlock2D","AttnDownBlock2D")
up_block_types=("AttnUpBlock2D","UpBlock2D","UpBlock2D")
cross_attention_dim=24
```

- **Cross-attention conditioning:**
  - Expand the one-hot vector cond to shape $(B, 1, 24)$ and pass it as `encoder_hidden_states`.
  - In each attention block, the model attends jointly to timestep embeddings and the conditional embedding, guiding the denoising towards the specified multi-label attributes.

2.3 Noise Schedule and Time-Step Sampling

- **Scheduler:** We employ `DDPMScheduler` with a linear noise schedule over 1000 timesteps.
- **Forward noising (training):**

$$x_t = \text{scheduler.add\_noise}(x_0, \epsilon, t), \quad t \sim \mathcal{U}[0, 999].$$

- **Reverse denoising (inference):**

```
for t in 999,998,...,0:
    noise_pred = model(x_t, timestep=t, encoder_hidden_states=cond_seq).sample
    x_{t-1}   = scheduler.step(noise_pred, t, x_t).prev_sample
```

2.4 Loss Function and Optimizer

- **Loss:** Mean squared error between predicted noise and true noise:

$$\mathcal{L} = E_{t,x_0,} \left\| -_\theta(x_t, t, c) \right\|_2^2.$$

- **Optimizer:** Adam with learning rate 2×10^{-4}.

2.5 Training Pipeline and Checkpointing

- **Hyperparameters:** Parsed via `argparse` (e.g., `--epochs`, `--batch-size`, `--resume`). We choose epochs=50,batch_size=64 as our hyperparametera for training.
- **Loop:**
  1. Set `model.train()` and iterate over DataLoader.
  2. Forward noise addition, noise prediction, loss computation, and backward update.
  3. Display progress and average loss with `tqdm`.
  4. Save checkpoint `model_epoch_001.pt` including model and optimizer states.(I have saved the checkpoints from epoch 1 to 50, and all subsequent analyses are based on the results of model_epoch_050.)
- **Resume:** If `--resume` is provided, load saved states and continue from the saved epoch. (Easily for trainging if some bad thing happened lead to training process is suspended.)

2.6 Image Generation and Evaluation

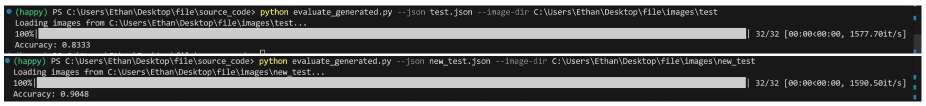
- **Generation:** `generate.py` reads a JSON (`test.json` or `new_test.json`), generates 32 images per split, and saves them in a user-specified directory.
- **Evaluation:** `evaluate_generated.py` loads generated images, normalizes to $[-1, 1]$, and uses a pretrained ResNet18 classifier (`evaluator.py`) to compute top-k accuracy, where k equals the number of ground-truth labels per image.

2.7 Visualization Scripts

- **Grid generation (`grid.py`):** Given any JSON and its corresponding output directory, automatically compose an 8×4 grid of 32 images.
- **Denoising process (`denoise.py`):** For fixed labels {"red sphere","cyan cylinder","cyan cube"}, sample every ≈ 125 timesteps (8 frames total) and display them in a single-row grid.

3 Results & Discussion

3.1 Training Result



```
(happy) PS C:\Users\Ethan\Desktop\file\source_code> python evaluate_generated.py --json test.json --image-dir C:\Users\Ethan\Desktop\file\images\test
Loading images from C:\Users\Ethan\Desktop\file\images\test...
100%|████████████████████████████████████████| 32/32 [00:00<00:00, 1577.70it/s]
Accuracy: 0.8333
```

```
(happy) PS C:\Users\Ethan\Desktop\file\source_code> python evaluate_generated.py --json new_test.json --image-dir C:\Users\Ethan\Desktop\file\images\new_test
Loading images from C:\Users\Ethan\Desktop\file\images\new_test...
100%|████████████████████████████████████████| 32/32 [00:00<00:00, 1590.50it/s]
Accuracy: 0.9048
```

Figure 1: The top image shows the training result of model_epoch_050.pt on test.json, while the bottom image shows the training result of _epoch_050.pt on new_test.json. The accuracy are 0.8333 and 0.9048,respecrively.

3.2 Synthetic Grids

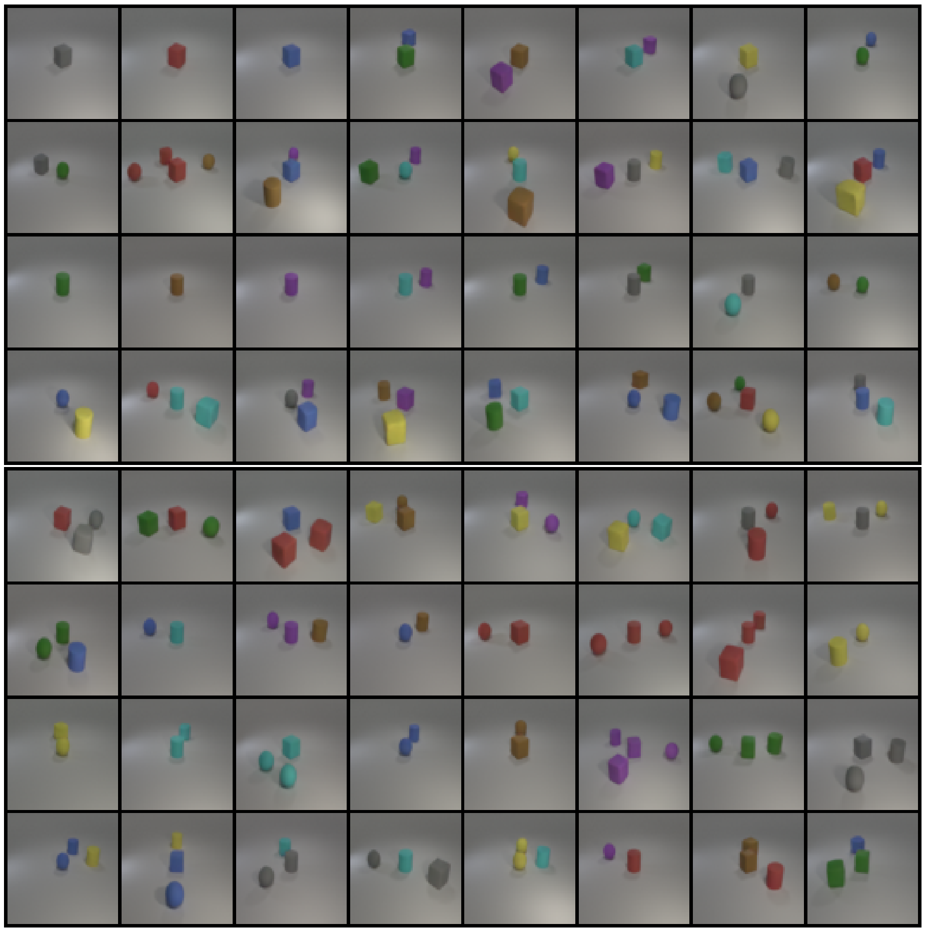


Figure 2: The top image shows the result of test.json, while the bottom image shows the result of new_test.json.

3.3 Denoising Process

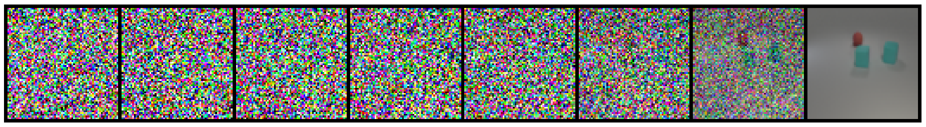


Figure 3: Using the labels [**"red sphere", "cyan cylinder", "cyan cube"**] as an example, show at least 8 frames illustrating the evolution from pure noise to the final image.

3.4 extra implementations or experiments

Although we did not employ explicit overfitting prevention measures—such as data augmentation (random flips, rotations, color jitter) or L2 regularization (weight decay)—we nonetheless did not observe any clear signs of overfitting. Several factors may explain this:

- **Highly Aligned Train–Test Distributions**
  Both the training set (18 000 images) and the test set (32 images) are procedurally generated under the same color–shape combinations and scene rules, ensuring that the mappings the model learns transfer almost seamlessly to the held-out samples.

- **Balanced Model Capacity** Our UNet2DConditionModel uses moderate channel widths (64, 128, 256) and only three down/up blocks, which matches the dataset size. This “just-right”capacity inherently limits its ability to memorize every detail, acting as a form of structural regularization.

- **Intrinsic Stochasticity of Diffusion Training** At every training step, we add random noise and predict it—this built-in randomness acts like a natural “noise regularizer,”encouraging the model to learn robust features rather than over-specializing to fixed patterns.

Together, these factors—distribution alignment, capacity control, stochastic noise injection, metric/sample-size effects, and implicit stopping—explain why our model maintains strong generalization without explicit overfitting safeguards.