Lab1-DDPM

李睿彬
Student ID: 314552050

Task 1 (Code)

1. TODO1-network.py:

a. **init**: 實作模型架構, 利用 time embedded dimension 的大小來當作 MLP 的輸入與輸出 channel 數量, 接著由 dim_hids 陣列的長度看要加入多少 layer, 這$\boxed{F}$只有 [128, 128, 128], 因此加入三層 Linear+ReLU

```
class SimpleNet(nn.Module):
    def __init__(
        self, dim_in: int, dim_out: int, dim_hids: List[int], num_timesteps: int
    ):
        super().__init__()
        # (TODO) Build a noise estimating network.
        # Args:
        #   dim_in: dimension of input
        #   dim_out: dimension of output
        #   dim_hids: dimensions of hidden features
        #   num_timesteps: number of timesteps
        ######## TODO ########
        # DO NOT change the code outside this part.
        time_embed_dim = dim_hids[0] # get time_emb
        self.MLP = nn.Sequential(
            TimeEmbedding(hidden_size=time_embed_dim),
            nn.Linear(time_embed_dim, time_embed_dim),
            nn.ReLU(),
        )
        layers = []
        current_dim = dim_in + time_embed_dim
        for dim_hid in dim_hids:
            layers.append(nn.Linear(current_dim, dim_hid))
            layers.append(nn.ReLU())
            current_dim = dim_hid
        layers.append(nn.Linear(current_dim, dim_out))
        self.main_net = nn.Sequential(*layers)
        ######################
```

Listing 1: SimpleNet Class Definition

b. **forward**: 透過 forward 呼叫 SimpleNet model, 最後在進入 main_net 前, 要先 concat x 和 time_emb

```python
def forward(self, x: torch.Tensor, t: torch.Tensor):
    # (TODO) Implement the forward pass. This should output
    # the noise prediction of the noisy input x at timestep t.
    # Args:
    #   x: the noisy data after t period diffusion
    #   t: the time that the forward diffusion has been running
    ######## TODO ########
    # DO NOT change the code outside this part.
    # x is (batch_size, 2)
    # t is (batch_size,)
    time_emb = self.MLP(t)
    x_and_t = torch.cat((x, time_emb), dim=-1)
    x = self.main_net(x_and_t)
    return x
    ######################
```

Listing 2: Forward Pass Implementation

2. TODO2-ddpm.py:

a. **q_sample**: 先算出 $\bar{\alpha}_t$, 對它開根號算出 $\sqrt{\bar{\alpha}_t}$, 接著利用 $\bar{\alpha}_t$ 算出 $\sqrt{1-\bar{\alpha}_t}$, 透過前面算出的值和 noise, 計算出 x_t

```python
def q_sample(self, x0, t, noise=None):
    # sample x_t from q(x_t|x_0) of DDPM.
    if noise is None:
        noise = torch.randn_like(x0)

    alphas_prod_t = extract(self.var_scheduler.alphas_cumprod, t, x0)
    sqrt_alphas_prod_t = torch.sqrt(alphas_prod_t)
    sqrt_one_minus_alphas_prod_t = torch.sqrt(1.0 - alphas_prod_t)
    xt = sqrt_alphas_prod_t * x0 + sqrt_one_minus_alphas_prod_t * noise
    return xt
```

Listing 3: q_sample Implementation

b. **TODO3-p_sample**: 先算出需要用到參數，接著

   i. 先使用 network.py 中實作的內容預測出 noise

   ii. 計算出 posterior mean

   iii. 和 iv. 一起, 若 t 大於 0, 利用 posterior variance 和 noise 計算出 x_{t-1}, 否則 x_0 = post_mean

```
@torch.no_grad()
    def p_sample(self, xt, t):
        """
        One step denoising function of DDPM: x_t -> x_{t-1}.
        Input:
            xt (`torch.Tensor`): samples at arbitrary timestep t.
            t (`torch.Tensor`): current timestep in a reverse process.
        Ouptut:
            x_t_prev (`torch.Tensor`): one step denoised sample. (= x_{t-1})
        """
        ######## TODO ########
        # DO NOT change the code outside this part.
        # compute x_t_prev.
        if isinstance(t, int):
            t = torch.full((xt.shape[0],), t, device=self.device, dtype=torch.long)
        eps_factor = (1 - extract(self.var_scheduler.alphas, t, xt)) / (
            1 - extract(self.var_scheduler.alphas_cumprod, t, xt)
        ).sqrt()

        beta_t      = extract(self.var_scheduler.betas,           t, xt)         #  _t
        alpha_t     = extract(self.var_scheduler.alphas,          t, xt)         #  _t = 1 -  _t
        alpha_bar_t = extract(self.var_scheduler.alphas_cumprod,  t, xt)         # \bar{ }_t
        t_prev      = (t - 1).clamp(min=0)
        alpha_bar_t_prev = extract(self.var_scheduler.alphas_cumprod, t_prev, xt) # \bar{ }_{t-1}

        # 1. predict noise
        predicted_noise = self.network(xt, t)
        # 2. Posterior mean
        post_mean = 1 / torch.sqrt(alpha_t) * (xt - eps_factor * predicted_noise)
        # 3. Posterior variance
        # 4. Reverse step
        if t[0].item() > 0:
            post_var = (1 - alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
            noise = torch.randn_like(xt)
            x_t_prev = post_mean + torch.sqrt(post_var) * noise
        else:
            x_t_prev = post_mean

        #######################
        return x_t_prev
```

Listing 4: p_sample Implementation

d. **TODO4-p_sample_loop**: iterate p_sample 來得到預測的 x_0

```python
@torch.no_grad()
    def p_sample_loop(self, shape):
        """
        The loop of the reverse process of DDPM.

        Input:
            shape (`Tuple`): The shape of output. e.g., (num particles, 2)
        Output:
            x0_pred (`torch.Tensor`): The final denoised output through the DDPM
                reverse process.
        """
        ######## TODO ########
        # DO NOT change the code outside this part.
        # sample x0 based on Algorithm 2 of DDPM paper.
        xt = torch.randn(shape).to(self.device)
        T = self.var_scheduler.num_train_timesteps

        from tqdm import tqdm
        for i in tqdm(reversed(range(0, T))):
            batch_size = xt.shape[0]
            t = torch.full((batch_size,), i, device=self.device, dtype=torch.long)
            xt = self.p_sample(xt, t)
        x0_pred = xt
        ######################
        return x0_pred
```

Listing 5: p_sample_loop Implementation

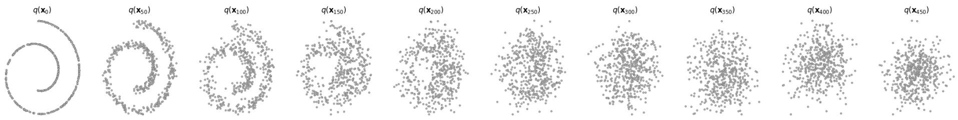
e. **TODO5-compute_loss**: 先利用 x_0 和 t sample 出 x_t, 接著利用 network 預測在時間 t 時的 noise, 利用預測到的 noise, 和先前 random 的 noise 在 MSE, 算出 loss

```
def compute_loss(self, x0):
    """
    The simplified noise matching loss corresponding Equation 14 in DDPM paper.
    Input:
        x0 (`torch.Tensor`): clean data
    Output:
        loss: the computed loss to be backpropagated.
    """
    ######## TODO ########
    # DO NOT change the code outside this part.
    # compute noise matching loss.
    batch_size = x0.shape[0]

    # 1) random choose timestep
    t = (
        torch.randint(0, self.var_scheduler.num_train_timesteps, size=(batch_size,)
            )
        .to(x0.device)
        .long()
    )
    # 2) get GT noise, and use q_sample to get x_t
    noise = torch.randn_like(x0)
    x_t = self.q_sample(x0=x0, t=t, noise=noise)

    # 3) predict noise
    predicted_noise = self.network(x_t, t)

    # 4) MSE loss (eps, eps_pred)
    loss = F.mse_loss(noise, predicted_noise)

    ######################
    return loss
```

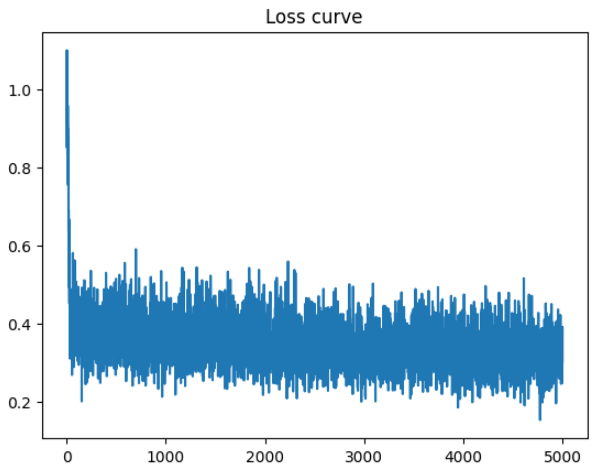
Listing 6: compute_loss Implementation

Task 1 (Result)

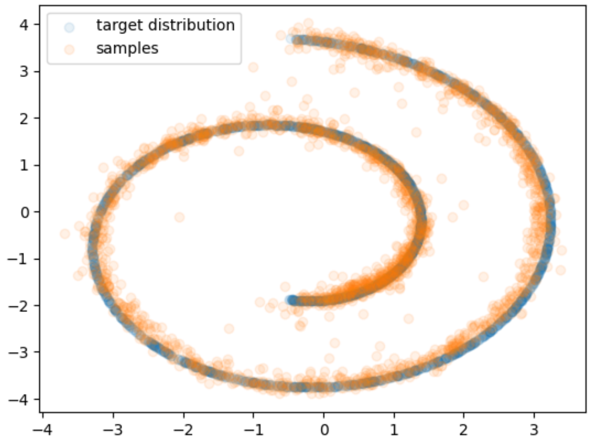
3. Loss Curve:



2. Loss Curve:



3. evaluation result:



Task 2 (Code)

1. TODO1-add_noise:

跟 task1 的 q_sample 一模一樣

```
def add_noise(
        self,
        x_0: torch.Tensor,
        t: torch.IntTensor,
        eps: Optional[torch.Tensor] = None,
    ):
        """
        A forward pass of a Markov chain, i.e., q(x_t | x_0).

        Input:
            x_0 (`torch.Tensor [B,C,H,W]`): samples from a real data distribution q
                (x_0).
            t: (`torch.IntTensor [B]`)
            eps: (`torch.Tensor [B,C,H,W]`, optional): if None, randomly sample
                Gaussian noise in the function.
        Output:
            x_t: (`torch.Tensor [B,C,H,W]`): noisy samples at timestep t.
            eps: (`torch.Tensor [B,C,H,W]`): injected noise.
        """

        if eps is None:
            eps       = torch.randn(x_0.shape, device='cuda')

        ######## TODO ########
        # DO NOT change the code outside this part.
        # Assignment 1. Implement the DDPM forward step.
        alphas_prod_t = extract(self.alphas_cumprod, t, x_0) # select the timesteps
             t and reshape them to match x0's dim
        sqrt_alphas_prod_t = torch.sqrt(alphas_prod_t)
        sqrt_one_minus_alphas_prod_t = torch.sqrt(1.0 - alphas_prod_t)
        x_t = sqrt_alphas_prod_t * x_0 + sqrt_one_minus_alphas_prod_t * eps
        #######################

        return x_t, eps
```

Listing 7: add_noise Implementation

2. TODO4-beta scheduling:

實作 Cosine Scheduler

i. 先依照公式算出 $\bar{\alpha}_t$， s 設成 0.008

ii. $\beta_t = 1 - \bar{\alpha}_t / \bar{\alpha}_{\text{t-1}}$

iii. 回傳 betas

```python
def __init__(
        self, num_train_timesteps: int, beta_1: float, beta_T: float, mode="linear"
    ):
        super().__init__()
        self.num_train_timesteps = num_train_timesteps
        self.num_inference_timesteps = num_train_timesteps
        self.timesteps = torch.from_numpy(
            np.arange(0, self.num_train_timesteps)[::-1].copy().astype(np.int64)
        )

        if mode == "linear":
            betas = torch.linspace(beta_1, beta_T, steps=num_train_timesteps)
        elif mode == "quad":
            betas = (
                torch.linspace(beta_1**0.5, beta_T**0.5, num_train_timesteps) ** 2
            )
        elif mode == "cosine":
            ######## TODO ########
            # Implement the cosine beta schedule (Nichol & Dhariwal, 2021).
            # Hint:
            # 1. Define alphā_t = f(t/T) where f is a cosine schedule:
            #       alphā_t = cos^2( ( (t/T + s) / (1+s) ) * ( /2) )
            #    with s = 0.008 (a small constant for stability).
            # 2. Convert alphā_t into betas using:
            #       beta_t = 1 - alphā_t / alphā_{t-1}
            # 3. Return betas as a tensor of shape [num_train_timesteps].
            s = 0.008
            t = torch.linspace(0, 1, steps=num_train_timesteps+1)
            f_t = torch.cos(((t / num_train_timesteps + s) / (1 + s)) * (torch.pi / 2)) ** 2
            alpha_cumprod_t = f_t / f_t[0]
            betas = 1 - alpha_cumprod_t[1:] / alpha_cumprod_t[:-1] # [1:T] / [1:T-1] Noted: alpht_cumprod_t[0] = 1
            betas = torch.clip(betas, 0.0001, 0.9999)
        else:
            raise NotImplementedError(f"{mode} is not implemented.")

        alphas = 1 - betas
        alphas_cumprod = torch.cumprod(alphas, dim=0)

        self.register_buffer("betas", betas)
```

```
        self.register_buffer("alphas", alphas)
        self.register_buffer("alphas_cumprod", alphas_cumprod)
```

Listing 8: Cosine Beta Schedule

2. TODO5-predictor:

a. step_predict_noise:

i. 一樣先 extract 出需要用到的參數

ii. 利用參數計算出 mean

iii. 和 iv. 一起，如果 t 大於 0，利用 posterior variance 計算出 $x_{t\text{-}1}$

```
def step_predict_noise(self, x_t: torch.Tensor, t: int, eps_theta: torch.Tensor):
    """
    Noise prediction version (the standard DDPM formulation).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        eps_theta: predicted noise ˆ_ (x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########
    # 1. Extract beta_t, alpha_t, and alpha_bar_t from the scheduler.
    # 2. Compute the predicted mean _ (x_t, t) = 1/√ _t * (x_t - ( _t/√(1-¯_t)) * ˆ_ ).
    # 3. Compute the posterior variance \tilde{ }_t = ((1-¯_{t-1})/(1-¯_t)) * _t.
    # 4. Add Gaussian noise scaled by √(\tilde{ }_t) unless t == 0.
    # 5. Return the final sample at t-1.
    if isinstance(t, int):
        t = torch.full((x_t.shape[0],), t, device=self.device, dtype=torch.long)
    eps_factor = (1 - extract(self.alphas, t, x_t)) / (
        1 - extract(self.alphas_cumprod, t, x_t)
    ).sqrt()

    beta_t      = extract(self.betas,          t, x_t)         # _t
    alpha_t     = extract(self.alphas,         t, x_t)         # _t = 1 - _t
    alpha_bar_t = extract(self.alphas_cumprod, t, x_t)         # \bar{ }_t
    t_prev      = (t - 1).clamp(min=0)
    alpha_bar_t_prev = extract(self.alphas_cumprod, t_prev, x_t) # \bar{ }_{t-1}

    # 1. predict noise
```

```
    predicted_noise = eps_theta
    # 2. Posterior mean
    post_mean = 1 / torch.sqrt(alpha_t) * (x_t - eps_factor * predicted_noise)
    # 3. Posterior variance
    # 4. Reverse step
    if t.item() > 0:
        post_var = (1 - alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
        noise = torch.randn_like(x_t)
        sample_prev = post_mean + torch.sqrt(post_var) * noise
    else:
        sample_prev = post_mean
    #######################
    return sample_prev
```

Listing 9: Noise Predictor

b. step_predict_x0:

i. 利用預測出來的 x_0_pred 來計算出 posterior mean

ii. 若 t 大於 0，則使用 posterior variance 和 noise 計算出 $x_{t\text{-}1}$，否則 $x_{t\text{-}1}$ = post_mean

```
def step_predict_x0(self, x_t: torch.Tensor, t: int, x0_pred: torch.Tensor):
    """
    x0 prediction version (alternative DDPM objective).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        x0_pred: predicted clean image x̂ (x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########
    if isinstance(t, int):
        t = torch.full((x_t.shape[0],), t, device=self.device, dtype=torch.long)

    beta_t      = extract(self.betas,          t, x_t)         #  _t
    alpha_t     = extract(self.alphas,         t, x_t)         #  _t = 1 -  _t
    alpha_bar_t = extract(self.alphas_cumprod,  t, x_t)         # \bar{ }_t
    t_prev      = (t - 1).clamp(min=0)
    alpha_bar_t_prev = extract(self.alphas_cumprod, t_prev, x_t) # \bar{ }_{t-1}

    # 1. posterior mean
    x0_coef = torch.sqrt(alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
    x_t_coef = torch.sqrt(alpha_t) * (1 - alpha_bar_t_prev) / (1 - alpha_bar_t)
    post_mean = x0_coef * x0_pred + x_t_coef * x_t
```

```python
    # 2. Reverse step
    if t.item() > 0:
        # 3. Posterior variance
        post_var = (1 - alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
        noise = torch.randn_like(x_t)
        sample_prev = post_mean + torch.sqrt(post_var) * noise
    else:
        sample_prev = post_mean
    #######################
    return sample_prev
```

Listing 10: x_0 Predictor

c. step_predict_mean

i. t 大於 0 的話，使用 posterior mean, mean_θ 和 noise 來計算 $x_{\text{t-1}}$，否則 $x_{\text{t-1}} = \text{mean}_\theta$

```python
def step_predict_mean(self, x_t: torch.Tensor, t: int, mean_theta: torch.Tensor):
    """
    Mean prediction version (directly outputting the posterior mean).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        mean_theta: network-predicted posterior mean ^_ (x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########
    if isinstance(t, int):
        t = torch.full((x_t.shape[0],), t, device=self.device, dtype=torch.long)

    beta_t      = extract(self.betas,          t, x_t)        #  _t
    alpha_t     = extract(self.alphas,         t, x_t)        #  _t = 1 - _t
    alpha_bar_t = extract(self.alphas_cumprod, t, x_t)        # \bar{ }_t
    t_prev      = (t - 1).clamp(min=0)
    alpha_bar_t_prev = extract(self.alphas_cumprod, t_prev, x_t) # \bar{ }_{t-1}

    if t.item() > 0:
        post_var = (1 - alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
        noise = torch.randn_like(x_t)
        sample_prev = mean_theta + torch.sqrt(post_var) * noise
    else:
        sample_prev = mean_theta
    #######################
    return sample_prev
```

Listing 11: Mean Predictor

d. get_loss_x0 sample a t，然後使用這個 t 在 x_0 上加上 noise 得到 x_t，利用 x_t 和 t 透過 network 預測出 x_0_pred，接著使用 x_0_pred 和 x_0 算出 loss

```python
def get_loss_x0(self, x0, class_label=None, noise=None):
    ######## TODO ########
    # Here we implement the "predict x0" version.
    # 1. Sample a timestep and add noise to get (x_t, noise).
    # 2. Pass (x_t, timestep) into self.network, where the output should
        represent the clean sample x0_pred.
    # 3. Compute the loss as MSE(predicted x0_pred, ground-truth x0).
    ######################
    B = x0.shape[0]
    t = self.var_scheduler.uniform_sample_t(B, x0.device)
    x_t, eps = self.var_scheduler.add_noise(x0, t, eps=noise)
    x0_pred = self.network(x_t, t, class_label) if class_label is not None
        else self.network(x_t, t)
    loss = F.mse_loss(x0_pred, x0)
    return loss
```

Listing 12: Loss Function of x0

e. get_loss_mean

i. sample a t，得到加噪過的 x_t，使用 network 預測出 mean_pred

ii. extract 出要使用的參數

iii. 利用參數算出真實的 mean

iv. 使用 mean_pred 和真實的 mean 計算出 loss

```python
def get_loss_mean(self, x0, class_label=None, noise=None):
    ######## TODO ########
    # Here we implement the "predict mean" version.
    # 1. Sample a timestep and add noise to get (x_t, noise).
    # 2. Pass (x_t, timestep) into self.network, where the output should
        represent the posterior mean   (x_t, t).
    # 3. Compute the *true* posterior mean from the closed-form DDPM formula (
        using x0, x_t, noise, and scheduler terms).
    # 4. Compute the loss as MSE(predicted mean, true mean).
    ######################
    B = x0.shape[0]
    t = self.var_scheduler.uniform_sample_t(B, x0.device)
    x_t, eps = self.var_scheduler.add_noise(x0, t, eps=noise)
    mean_pred = self.network(x_t, t, class_label) if class_label is not None
        else self.network(x_t, t)

    beta_t          = extract(self.var_scheduler.betas,          t, x0)
    alpha_t         = extract(self.var_scheduler.alphas,         t, x0)
```

```python
    alpha_bar_t      = extract(self.var_scheduler.alphas_cumprod, t, x0)
    t_prev           = (t - 1).clamp(min=0)
    alpha_bar_t_prev = extract(self.var_scheduler.alphas_cumprod, t_prev, x0)

    x0_coef = torch.sqrt(alpha_bar_t_prev) * beta_t / (1 - alpha_bar_t)
    x_t_coef = torch.sqrt(alpha_t) * (1 - alpha_bar_t_prev) / (1 - alpha_bar_t)
    mean = x0_coef * x0 + x_t_coef * x_t
    loss = F.mse_loss(mean_pred, mean)
    return loss
```

Listing 13: Loss Function of x0

3. Training:

我修改–train_num_steps=50000，–log_interval=2500，來 reduce 訓練的時間

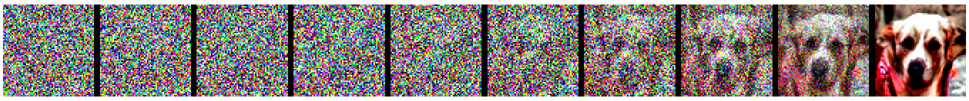
```python
parser.add_argument(
    "--train_num_steps",
    type=int,
    default=50000, #50000, #100000 # 100000 -> 50000
    help="the number of model training steps.",
)

parser.add_argument("--log_interval", type=int, default=2500) # 200 -> 2500
```

Listing 14: Training Arguments

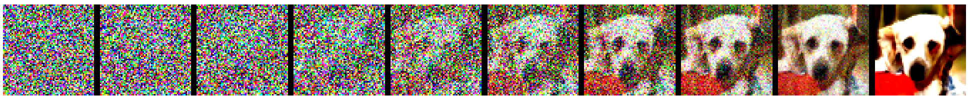
Task 2 (Result)

1. trajectory fig

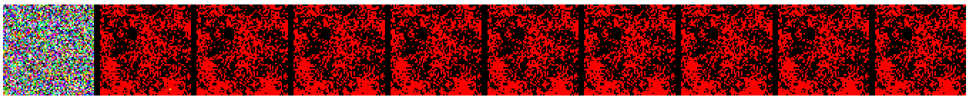
i. –mode = linear, –predictor = noise



ii. –mode = quad, –predictor = noise

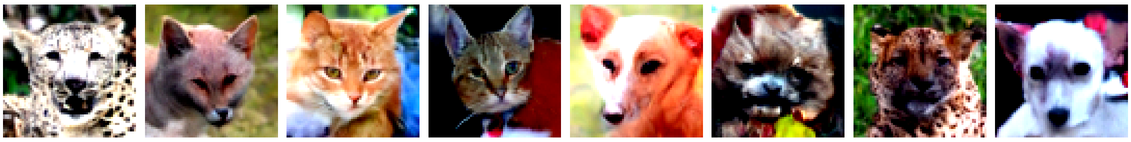


iii. –mode = cosine, –predictor = noise



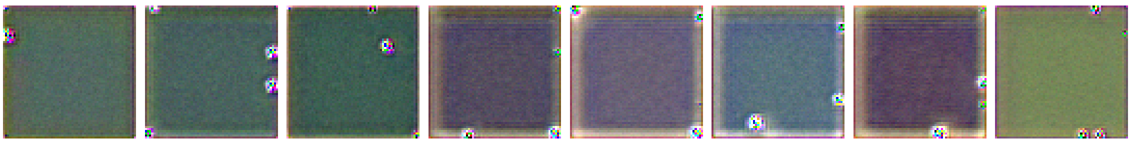
2. results fig of different predictor

i. –mode = linear, –predictor = noise



ii. –mode = linear, –predictor = x0

iii. –mode = linear, –predictor = mean



3. fid

My best fid is from –mode linear –predictor noise

```
(ddpm) he001295863@ddpm-linear-noise:~/Generative_Model/Lab1-DDPM/image_diffusion_todo/fid$ python measure_fid.py ../data/afhq/eval/ ../linear-noise/
Calculating FID given paths ../data/afhq/eval/ and ../linear-noise/...
/home/he001295863/miniconda3/envs/ddpm/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/he001295863/miniconda3/envs/ddpm/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The curre
nt behavior is equivalent to passing `weights=None`.
  warnings.warn(msg)
/home/he001295863/miniconda3/envs/ddpm/lib/python3.9/site-packages/torchvision/models/inception.py:43: FutureWarning: The default weight initialization of inception_v3 will be changed in future releases of torchvision. If you wish to keep the
 old behavior (which leads to long initialization times due to scipy/scipy#11299), please set init_weights=True.
  warnings.warn(
100%|| 24/24 [00:02<00:00,  9.88it/s]
100%|| 8/8 [00:00<00:00,  9.02it/s]
FID: 4.535422049409975
```