CS771 Homework Assignment 4 Write-up

Moniek Smink
smink2@wisc.edu

Venkata Saikiranpatnaik
vbalivada@wisc.edu

Sourav Suresh
sourav.suresh@wisc.edu

1. Team Member Contributions

Name	Contribution
Moniek Smink	Understanding DDPM
Venkata Saikiranpatnaik	Implementation
Sourav Suresh	Experiment with MNIST

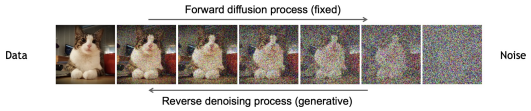
2. Overview

The main objectives of the assignment are three-fold. First, we understand the DDPM model, see Section 3. Next, we fill in the required code sections for training DDPMs in Section 4. Finally, we implement the multi-task loss and train the model in Section 5.

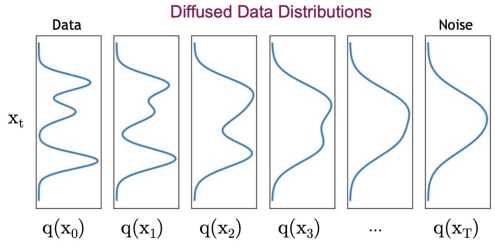
3. Understanding DDPM

3.1. DDPM latent space and sampling

DDPMs are a form of latent variable models [1]. The latent space of DDPMs is the same dimension as the original input image. The last latent is close to random noise distributed as a Gaussian with a mean of 0 and a unit variance. This occurs because the original signal is lost when adding random noise for a large number of steps, see 1.



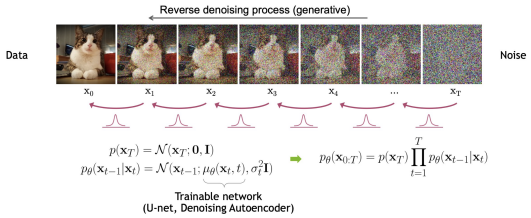
(a) Noising and Denoising



(b) Original image lost after a large number of steps.

Figure 1. Diffusion Process

In order to draw samples from a DDPM, random noise is converted into a sample using a finite number of denoising steps. We use the trained UNet networks to denoise intermediate latents starting from random Gaussian noise as shown in figure 2.



(a) Reverse denoising process

Algorithm 2 Sampling

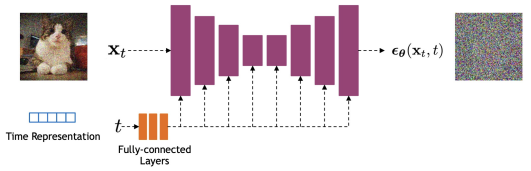
$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

(b) Algorithm for sampling an Image

Figure 2. Trained reverse denoising networks used for sampling

3.2. Injecting time and condition info into UNet

At a high level, the time and condition information is embedded and injected into each layer of UNet, see 3a. First, we embed both time and label/condition information using embedding layers specified in figure 3b and then inject the embeddings into the UNet architecture. When we zoom in, the UNet layer consists of a Residual block and a Transformer block. The time information is injected into the residual block, while the label information is injected into the transformer block, see figure 4a. For time embedding, we choose a sinusoidal positional encoding similar to positional embeddings in the original transformers paper, see 3b. After obtaining the embedding, we add the same value to each pixel position in the corresponding UNet layer similar to bias in a Convolutional Network, see broadcasting in 4b. However, different layers each have a different number



(a) High-level overview of injection

```python
# mlp for embedding time steps (we will use sinusoidal PE here
self.time_embd = nn.Sequential(
    SinusoidalPE(dim),
    nn.Linear(dim, time_dim),
    nn.SiLU(),
    nn.Linear(time_dim, time_dim),
)

# mlp for embedding labels, similar to time steps
self.label_embd = nn.Sequential(
    LabelEmbedding(num_classes, dim),
    nn.Linear(dim, context_dim),
    nn.SiLU(),
    nn.Linear(context_dim, context_dim),
)
```

(b) Embeddings

Figure 3. Information Injection

of channels. To handle this non-uniformity, we first transform the embedding using a Linear layer to match the number of channels. Next, we employ the usual embedding to embed the label information. Finally, we use cross attention to attend to this label information, see `context` in 4c.

```python
# first conv
x = self.conv_in(x)
# time embedding
t = self.time_embd(time)
# label embedding
c = self.label_embd(label).unsqueeze(1)
# cache
h = []

# encoder
for resblock, transformer, downsample in self.encoder:
    x = resblock(x, t)
    x = transformer(x, c)
    x = downsample(x)
    h.append(x)
```

(a) Demonstrates how time and label/condition information is integrated into UNet

```python
# Time modulation (with shift only, similar to stable diffusion)
if t_emb is not None:
    t_hidden = self.time_emb_proj(self.act(t_emb))
    h = h + t_hidden[:, :, None, None]
```

(b) Time embeddings are added to the convolutional layer similar to the bias term

```python
def forward(self, x, context):
    """
    Args:
        x (tensor): Input feature map of size B x C x H x W (used for q)
        context (tensor): Input context feature of size B x T x C (used for k, v)
    """
    shortcut = x
    x = self.proj_in(self.group_norm(x))
    x = x.permute(0, 2, 3, 1)  # B C H W -> B H W C
    x = self.self_attn(self.norm1(x)) + x
    x = self.cross_attn(self.norm2(x), context) + x
    x = self.mlp(self.norm3(x)) + x
    x = x.permute(0, 3, 1, 2) # B H W C -> B C H W
    return self.proj_out(x) + shortcut
```

(c) Labels are attended to using the cross attention mechanism

Figure 4. Embedding Integration

4. Implementation

4.1. Implementing forward diffusion process

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{4}$$

Figure 5. Forward diffusion process

See figure 5 for forward diffusion process formula after t steps of adding Gaussian random noise. To sample, we first scale the original image by $\sqrt{\bar{\alpha}_t}$ and add noise with stddev $\sqrt{1-\bar{\alpha}_t}$.

4.2. Implementing reverse diffusion process

The sampling process is described in figure 2b. We first denoise the image x_t by subtracting the model output scaled by $\frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}$, then scale it using $\frac{1}{\sqrt{\alpha_t}}$, finally add random noise using posterior stddev $\sqrt{\sigma_t}$.

4.3. Implementing DDPM loss

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Figure 6. Training diffusion model

We use mean squared loss on noise predictions during training as shown in figure 6. To compute the predicted noise, we first compute the noised image of tth step forward diffusion using `q_sample`, then we pass this image through the UNet model.

5. Experiment with MNIST

We train the model and run to completion for 30 epochs. The training curve is shown in figure 7. We observe that the loss plateaus after a while.

Some sample images are shown in figure 8. As expected, the first sample is not very good and looks very noisy. However, the sample after 10th epoch is very decent. Clearly, it is taking the label into account when generating images. However, some images are not very sharp such as some 2s and 6s. Finally, the sampled image after full training has sharp images overall. Moreover, the images look to be coming from a diverse style of "handwritings".

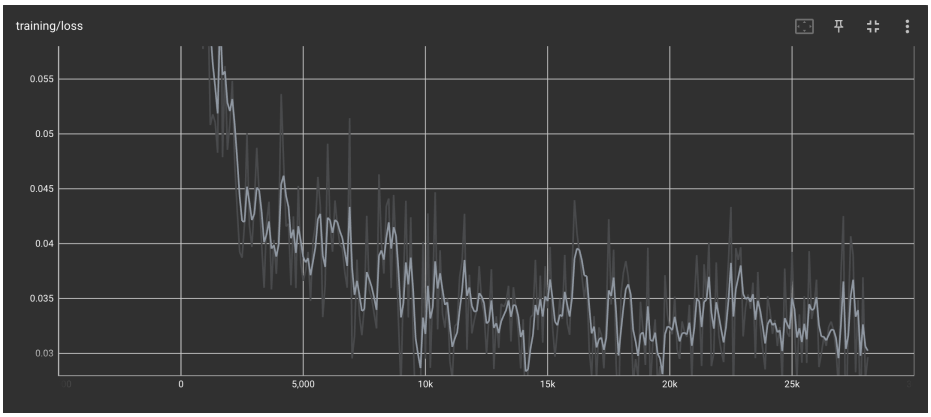
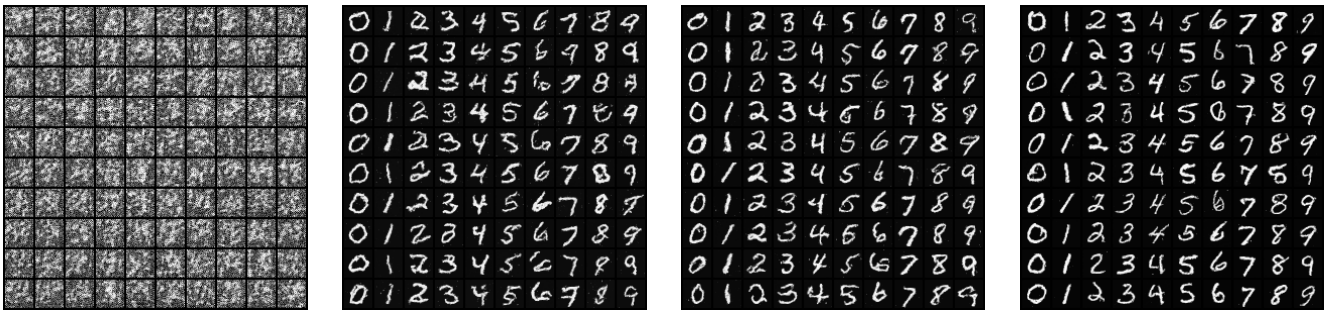


Figure 7. Learning curve during training of DDPM for 30 epochs on MNIST dataset.



(a) Sample images after 1 epoch (b) Sample images after 10 epochs (c) Sample images after 15 epochs (d) Sample images after 30 epochs

Figure 8. Sample Images during training

References

[1] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. 1

[2] Kashif Rasul Niels Rogge. The annotated diffusion model. https://huggingface.co/blog/annotated-diffusion, 2022. Accessed: 2023-12-06.