CS 771 - Computer Vision
Assignment - 4

Team Members: Sujay Chandra Shekara Sharma, Jiangyi Liu, Venkata Abhijeeth Balabhadruni
Wisc Emails: schandrashe5@wisc.edu, jiangyi.liu@wisc.edu, balabhadruni@wisc.edu

4.1 UNet

Understanding UNet

1. **Can you describe the architecture of the encoder and decoder within the UNet?**

Encoder Architecture

The encoder consists of multiple blocks corresponding to different resolution levels. Each block includes:

- *Residual Module (ResBlock):* Processes the input features while incorporating time embeddings.
- *Transformer (SpatialTransformer)*: Enhances the representation by focusing on relevant features in a self-attention mechanism. Transformers are only included in specified attention levels.
- *Downsampling (Downsample):* Reduces spatial resolution to aggregate contextual information. The last encoder block does not perform downsampling.

At each resolution level, the encoder extracts hierarchical features by progressively downsampling the spatial dimensions while preserving important features using the ResBlock and optional attention mechanism.

Middle Block Architecture

The middle block bridges the encoder and decoder and includes:

- Two ResBlocks to process the latent features.

- One Transformer in between to refine features using attention.

Decoder Architecture

The decoder mirrors the encoder's architecture but in reverse order. Each block includes:

- *Residual Module (ResBlock):* Processes the concatenated features from the encoder's output and the decoder's previous state. It integrates time embeddings into this processing.
- *Transformer (SpatialTransformer)*: Similar to the encoder, it is conditionally applied to refine features using attention.
- *Upsampling (Upsample)*: Increases spatial resolution, reconstructing the image progressively.

The decoder takes inputs from the latent features generated by the encoder and middle block. At each step, it upsamples and combines these features with skip connections from the encoder, allowing the network to retain high-resolution details while generating the output.

2. How does the current implementation inject time t and condition c (image labels) into the UNet?

Time t Embedding:

A Sinusoidal Positional Encoding (SinusoidalPE) encodes the scalar time t into a feature vector. This feature is then passed through a multi-layer perceptron (MLP) composed of linear layers and activation functions (SiLU) to generate a time embedding of size time_dim.
The time embedding is integrated into each ResBlock in the encoder, middle block, and decoder to condition the processing of features on the diffusion process's time step.

Condition c Embedding:

The class label (condition c) is embedded using a LabelEmbedding layer, which maps the label to a high-dimensional space. This embedding is further passed through a multi-layer perceptron (MLP) composed of linear layers and activation functions (SiLU) to generate an embedding of size context_dim.

The encoder sequentially processes the input image, time embedding (t), and label embedding (c), storing the intermediate results in a list called encoder_output.

The output from the encoder undergoes further processing through a middle block.

In the decoder, the current processed image is concatenated with a tensor popped from the encoder_output list. This combined tensor is then processed by the decoder. Finally, a convolutional layer is applied to the decoder's output to generate the final result.

Together, time t and condition c embeddings guide the UNet to focus on denoising in a time- and class-aware manner, ensuring the outputs align with the desired class and noise levels at each diffusion step.

Completing the implementation of UNet

The implementation of decoder in the UNet processes features by combining information from the encoder with its current feature maps. For each decoding block, it performs the following steps:

1. *Concatenation*: Combines the current feature map with the corresponding encoder output, preserving spatial details.
2. *Residual Module*: Processes the combined features with temporal context **(t)** to refine them.
3. *Transformer Module*: Further enhances the features using label embedding **(c)** for contextual understanding if a transformer module is present.
4. *Upsampling Module***:** Increases the spatial resolution for gradual reconstruction of the output, if an upsampling module is present.

4.2 DDPM

Forward diffusion process

`q_sample`

The `q_sample` function implements the forward diffusion process by adding noise to a given starting image (`x_start`) at a specific time step (`t`) to produce its noisy version (`x_t`).

If no noise is provided, the function generates Gaussian noise with the same shape as the input image.

The function extracts time-dependent parameters, `sqrt_alphas_cumprod_t` and `sqrt_one_minus_alphas_cumprod_t`, based on precomputed cumulative product terms of the diffusion process.

From Equation 4, using these parameters, the noisy image `x_t` is computed as a weighted combination of the original image and the noise, following the formula:

x_t = sqrt_alphas_cumprod_t * x_start + sqrt_one_minus_alphas_cumprod_t * noise

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

Figure 1: Equation 4 from DDPM paper.

Reverse denoising process

The implementation consists of two functions: `p_sample` and `p_sample_loop`, which together perform the reverse diffusion process to generate denoised images.

`p_sample`

Denoises a given noisy image `x` at a specific time step `t` by predicting its mean and optionally adding noise for stochastic sampling.

Retrieves time-dependent values like `betas`, `sqrt_one_minus_alphas_cumprod`, and `sqrt_recip_alphas`.

Calculates the predicted denoised image mean using Equation 11 from the DDPM paper, which involves subtracting the scaled model output from the input.

$$\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) = \tilde{\boldsymbol{\mu}}_t\left(\mathbf{x}_t, \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t))\right) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) \tag{11}$$

Figure 2: Equation 11 from DDPM paper.

If `t_index` is 0, the predicted mean is returned directly. Otherwise, noise is added, scaled by the posterior variance, to allow stochastic sampling.

`p_sample_loop`

Iteratively applies `p_sample` across all time steps in reverse to generate an image from pure noise.

`p_sample` denoises the input at a single time step, and `p_sample_loop` chains these steps to generate a full image by iteratively refining random noise.

Simplified DDPM loss

The `compute_loss` function calculates the training loss for the diffusion model by comparing the predicted noise to the actual noise added during the forward diffusion process.

If no noise is provided, Gaussian noise with the same shape as the input is generated.

The noisy version of the input image is computed using the `q_sample` function, which simulates the forward diffusion process by adding the generated noise at a given time step `t`.

The model predicts the noise component present in the noisy image using the current state of the model.

Then the Mean Squared Error (MSE) loss is computed between the actual noise and the predicted noise. This simplified objective directly trains the model to accurately denoise at each time step.

This implementation aligns with Algorithm 1 of the DDPM paper

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

Experimenting with MNIST dataset

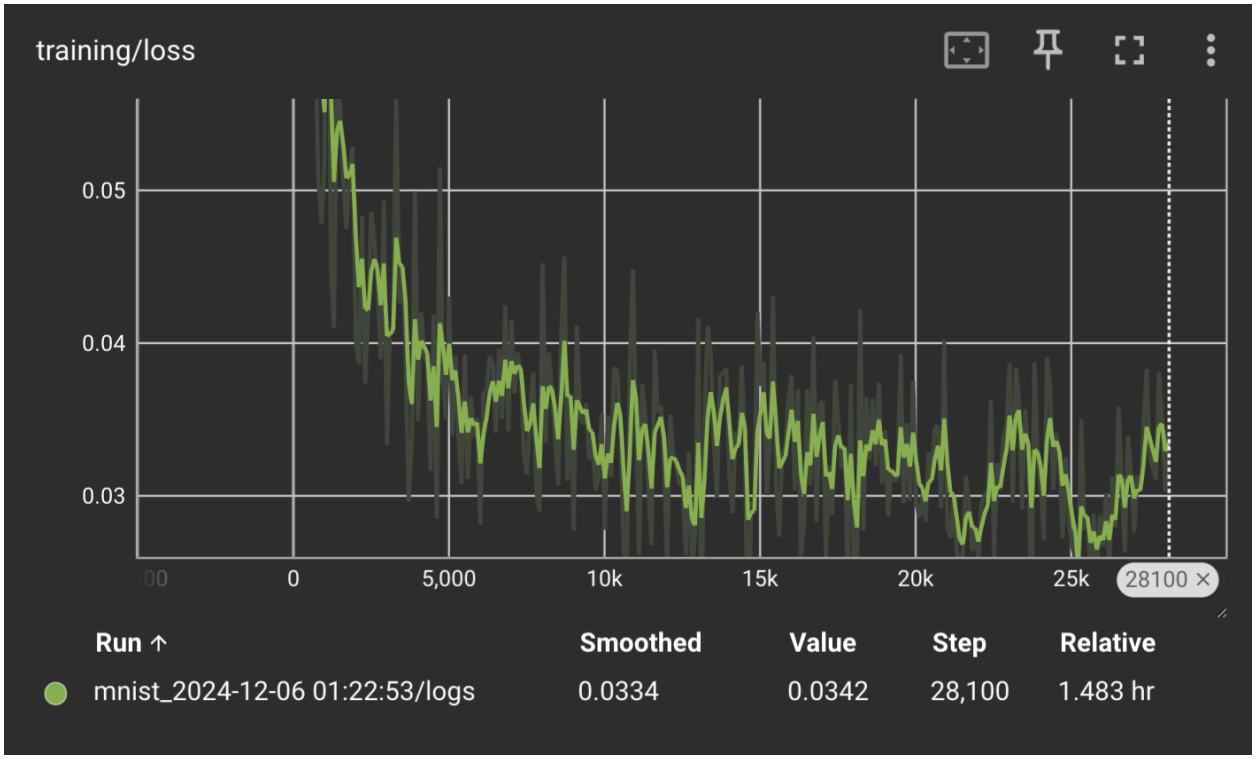

Figure 4: Training Curve for MNIST Dataset

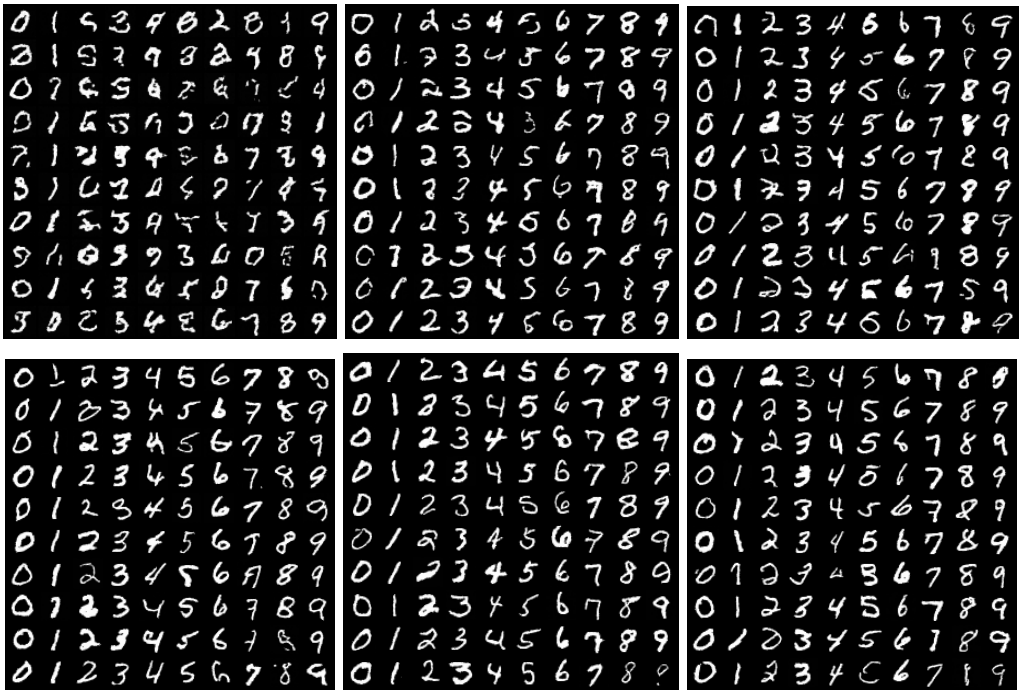

Figure 5: Sample Images

The above sample images are generated from epochs 5, 10, 15, 20, 25, 30.

We ran the code for 30 epochs and it took around 1.48hrs as observed from Figure 4. We started seeing digits at epoch 5, but they became clearer at epoch 10. By epoch 30 all the digits were very clear.

Since MNIST is a fairly simple dataset, we are able to see clear digits with just 30 epochs using vanilla DDPM.

4.3 Latent Diffusion Models

LDMs

For implementing this part, we have added an additional encoding step in the compute_loss function and an additional decoding step in p_sample_loop function.

Experimenting with AFHQ dataset

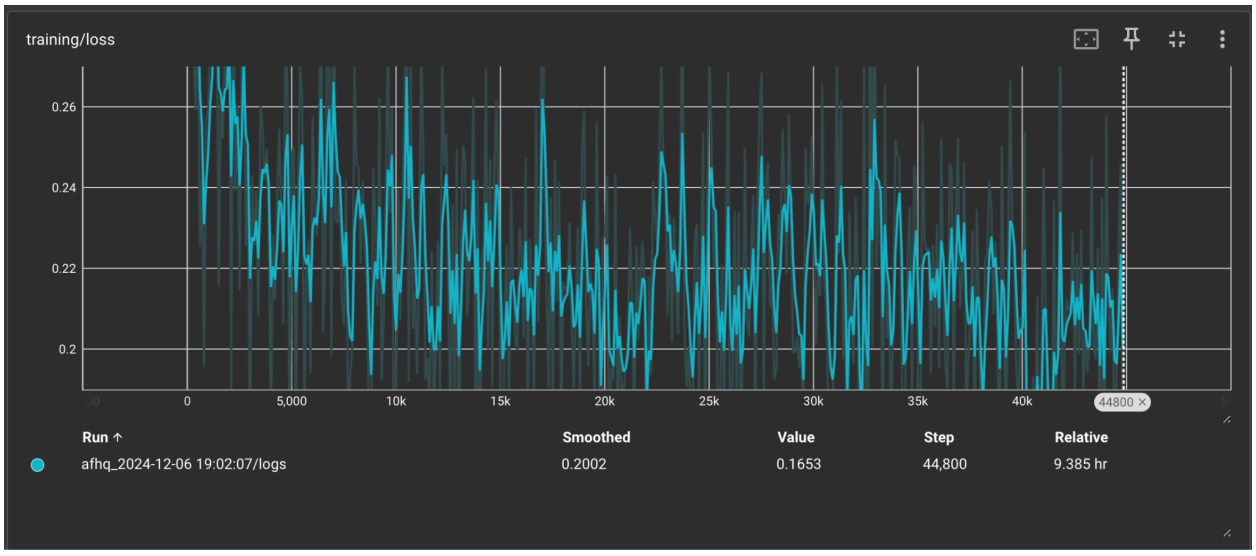

Figure 6: Training Curve for AFHQ Dataset

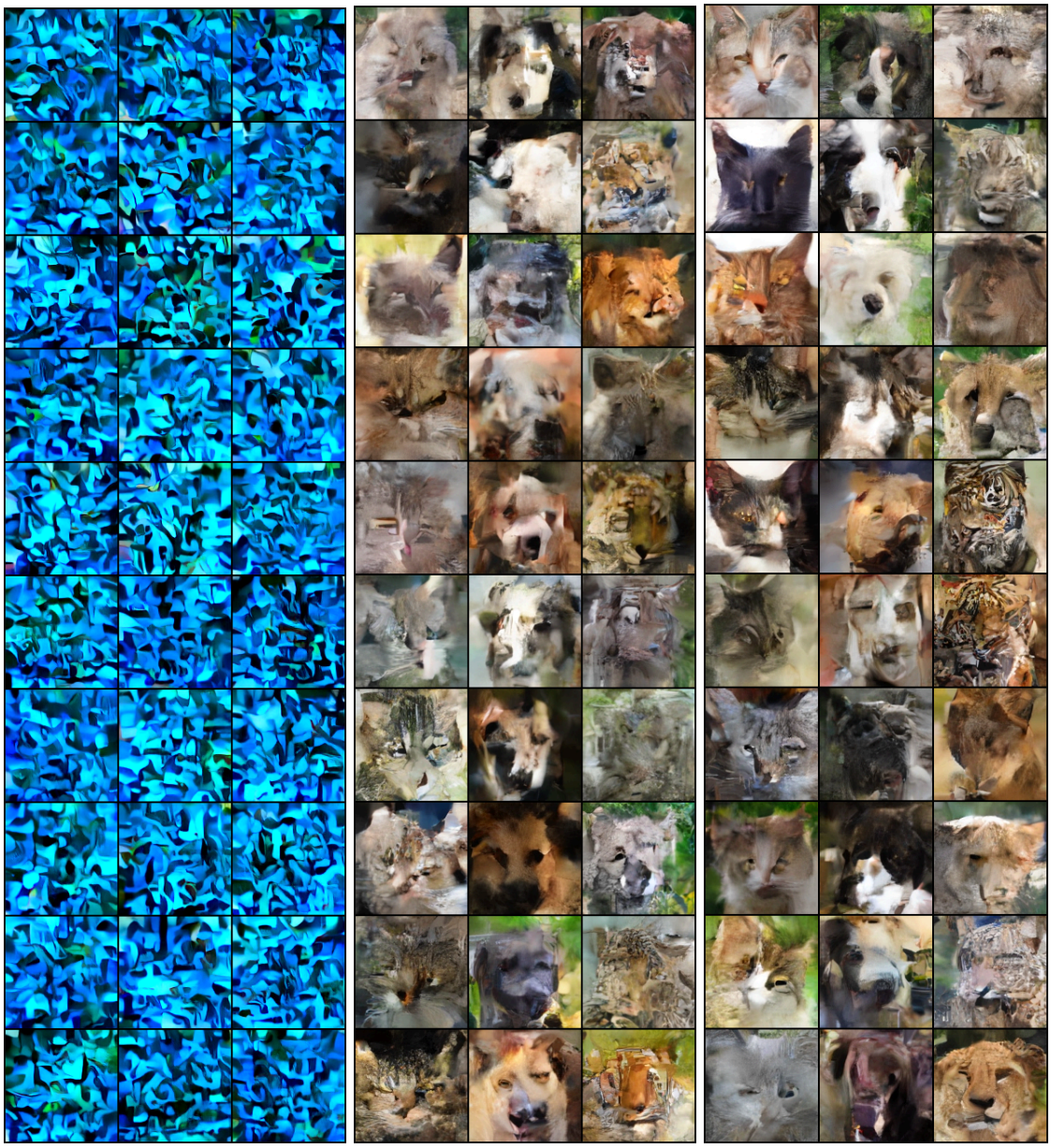

Figure 7: Sample Images

The above sample images are generated from epochs 5, 55, 105, 155, 200.

We ran the code for 200 epochs and it took around 9.38hrs as observed from Figure 6.

As we can see from the sample images, epoch 5 is almost just noise. We can see animal faces by epoch 105, they became a bit clearer by epoch 200. However, the generated images are still distorted as we only trained for 200 epochs and AFHQ is a complex dataset.

Contributions :

Jiangyi Liu -> Understanding Code, 4.1
Venkata Abhijeeth Balabhadruni -> Understanding Code, 4.2
Sujay Chandra Shekara Sharma -> Understanding Code, 4.3

Acknowledgements :

ChatGPT - used chatgpt to rephrase some sentences.