Pipeline - Final Draft

CS156 - Finding Patterns in Data with Machine Learning
Minerva University
December 18, 2025
Prof. Watson

Introduction

In the previous assignment, I explored my personal food photo archive using Convolutional and Variational Autoencoders to learn compact image representations. These models were useful for compressing the images and analyzing their latent spaces, but their reconstructions were often blurry and missed important visual details. At times, the reconstructed images looked less like food and more like abstract paintings, which was interesting but not exactly the goal. While this was expected to some extent, it made it clear that autoencoders alone are limited when it comes to generating high-quality images. Based on this observation and the feedback from the earlier project, I decided to extend the pipeline by experimenting with a more powerful generative approach.

In this project, I use a denoising diffusion probabilistic model (DDPM) trained in the latent space of a pre-trained Stable Diffusion VAE. Instead of generating images directly at the pixel level, the diffusion model learns to generate compressed latent representations with shape $4\times32\times32$, which are then decoded back into images using the Stable Diffusion decoder. Working in latent space makes the problem more manageable while still preserving meaningful structure from the original dataset. To model the diffusion process, I implement a UNet architecture with residual connections and timestep embeddings so the model can learn how to gradually remove noise across diffusion steps.

The main goal of this assignment is to see whether combining diffusion models with a structured latent space leads to better and more expressive image generation than the autoencoder-based approaches I used before. By applying latent diffusion to my food photo dataset, I want to evaluate both the visual quality of the generated images and how well the model captures patterns in the data.

Meet My Data

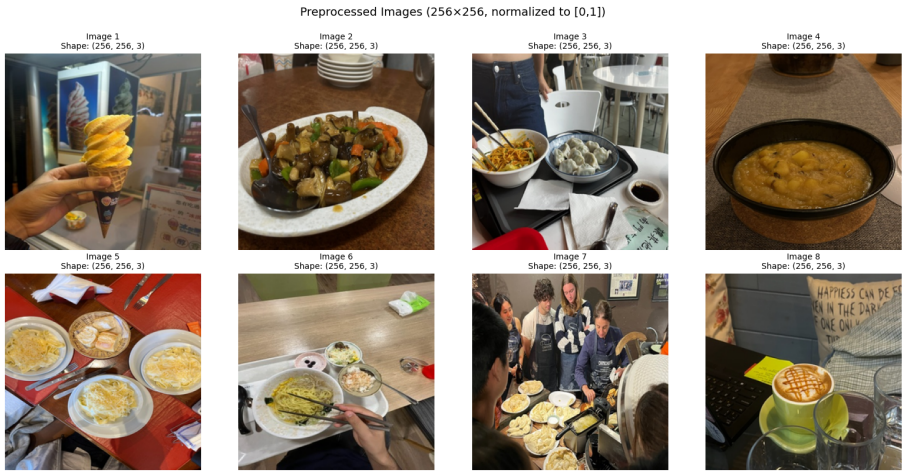


Figure 1: Sample of preprocessed food images (256×256 RGB, normalized to [0,1])

This project uses the same personal food photo dataset as in the previous assignment, but looking at it again makes it clear just how complex it actually is. The images vary a lot in lighting, color, composition, and context: some photos are close-ups of a single dish, others include multiple plates, people, hands, or cluttered backgrounds, and many are taken in very different environments like restaurants, cafes, or home kitchens. On top of that, textures range from smooth soups to highly detailed dishes, and lighting conditions go from dim indoor scenes to bright daylight. Because of this variability, the dataset doesn't have one simple visual pattern that a model can easily learn. This level of complexity is exactly why the earlier autoencoder reconstructions tended to blur everything together.

Exploratory Data Analysis

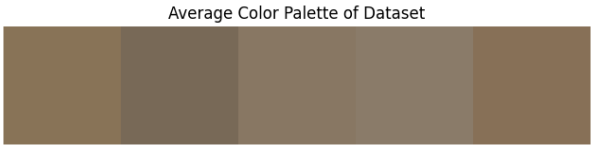


Figure 2: Average color palette of the food image dataset

To get a high-level sense of the color distribution in the dataset, I extract a small color palette from a random subset of images using k-means clustering in RGB space. For each image, pixels are grouped into five dominant colors, and these palettes are then averaged across 50 randomly selected images to produce an overall "average" color palette. This visualization gives a rough summary of the typical colors present in the dataset, without focusing on any single image.

The average palette ends up being mostly warm, muted colors like browns, beiges, and soft yellows, which makes sense given what's actually in the dataset. A lot of the photos are taken indoors with warm lighting, often on wooden tables or backgrounds that have similar tones, and the food itself usually falls into the same color range. Because of this, it seems unlikely that color by itself would be enough to cleanly separate the images or form strong clusters. This also helps explain why simpler models tend to blur things together as they're mostly seeing very similar color information. Seeing this makes it clearer why I need models that can focus more on spatial structure and texture instead of relying mainly on average color values.

PCA and Clustering

Before training or evaluating generative models in latent space, it is important to understand what that space actually looks like and how structured it is. Dimensionality reduction and unsupervised analysis help reveal whether the SD-VAE latent space is smooth, noisy, or clustered, and provide intuition about how suitable it is for tasks like diffusion, interpolation, and similarity search.

First, all precomputed SD-VAE latents are loaded and concatenated into a single dataset for both training and validation. Each latent tensor is then flattened from its original spatial shape into a vector so that standard linear methods can be applied. Principal Component Analysis (PCA) is used to reduce the dimensionality from 4096 latent features down to 50 components, providing a compact representation while preserving as much variance as possible. Then, the flattened latent vectors are used as input to two unsupervised clustering methods: KMeans, with the number of clusters fixed to eight, and DBSCAN, which attempts to discover clusters based on density without pre-specifying their number. To evaluate cluster quality, the silhouette score is computed for KMeans (see the Appendix).

The PCA results show that even with 50 components, only about 37.5% of the variance is explained, which suggests that the latent space has some structure but is still very spread out and complex. The clustering results point in the same direction: DBSCAN does not find any clear groups, and KMeans gives a very low silhouette score, meaning the clusters overlap a lot. Overall, this suggests that the SD-VAE latent space is smooth and continuous rather than broken into clear categories, which makes sense for diffusion models that rely on gradual, step-by-step changes instead of sharp boundaries.

I visualize a subset of the latents using t-SNE, which projects high-dimensional representations into two dimensions while preserving local neighborhood relationships.

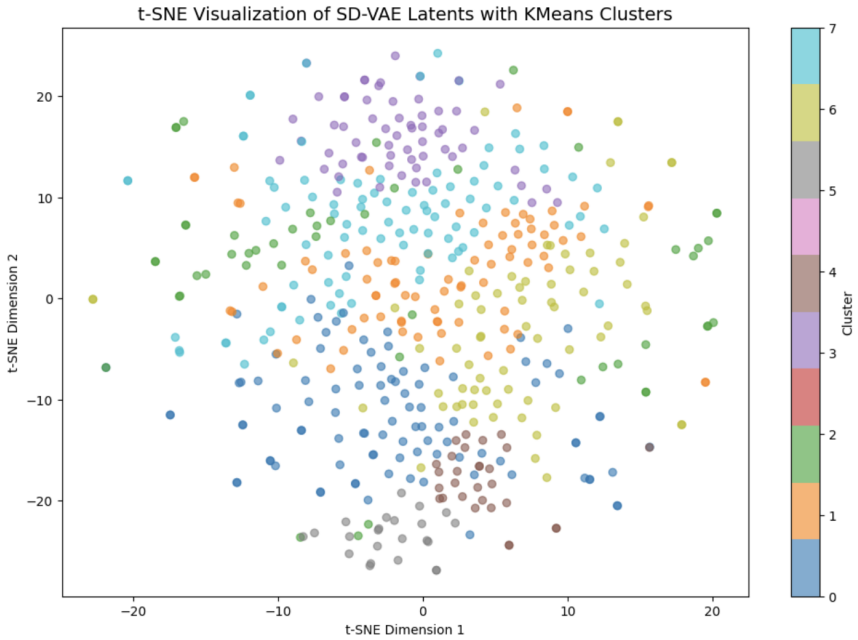


Figure 3: t-SNE visualization of SD-VAE latent representations

The plot shows a continuous cloud of points with substantial overlap between the KMeans-assigned colors, rather than clearly separated groups. This indicates that while nearby points in latent space are locally similar, the space does not naturally break into distinct clusters. Instead, the latents appear smoothly distributed, supporting the earlier PCA and clustering results.

To make sure one more time, I visualize the same latent representations using UMAP, which is designed to better preserve local neighborhood relationships while still capturing global structure.

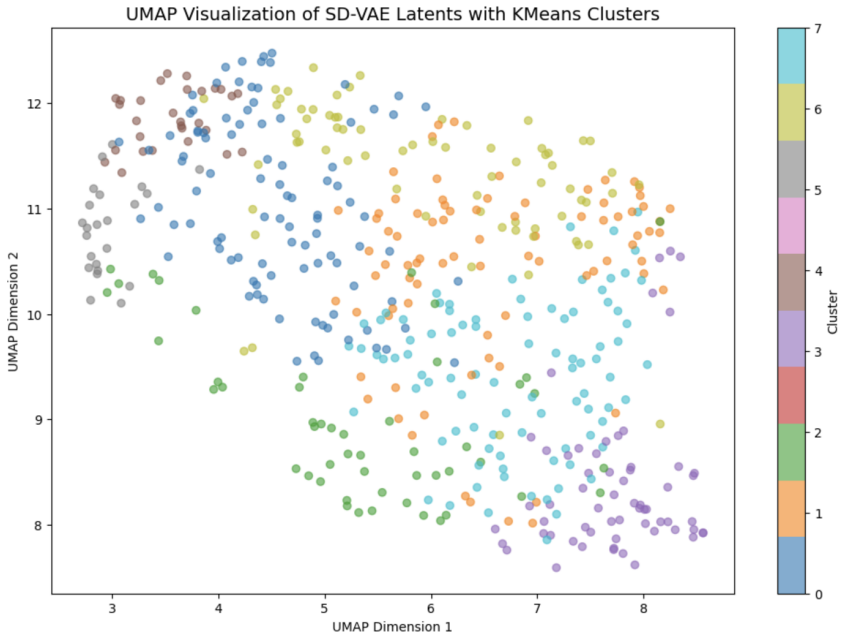


Figure 4: UMAP visualization of SD-VAE latent representations

Compared to t-SNE, the UMAP projection produces more compact and continuous regions, but the KMeans cluster labels still overlap heavily rather than forming clearly separated groups. This suggests that while nearby latents are locally coherent, there are no strong global boundaries between clusters.

Then, I perform a nearest-neighbor search using PCA-reduced latent representations.

Figure 5: Nearest Neighbors in SD-VAE Latent Space (PCA)

Starting from a randomly selected query image, I retrieve the closest images based on Euclidean distance in the PCA-reduced SD-VAE latent space. The closest neighbor is almost identical to the query image and appears to be essentially an augmented version of it, likely differing only in minor factors such as orientation or lighting, which indicates that very small latent distances correspond to near-duplicates. As the distance increases, the retrieved images become more visually distinct, but they still share high-level semantic properties such as being food images, having similar layouts, or comparable color distributions. This gradual change suggests that the latent space is organized smoothly, where nearby points represent perceptually similar images rather than exact copies. So, this provides further evidence that the SD-VAE latent space preserves semantic similarity in a continuous way, making it well-suited for tasks such as similarity search, interpolation, and diffusion-based generation.

Data Augmentation

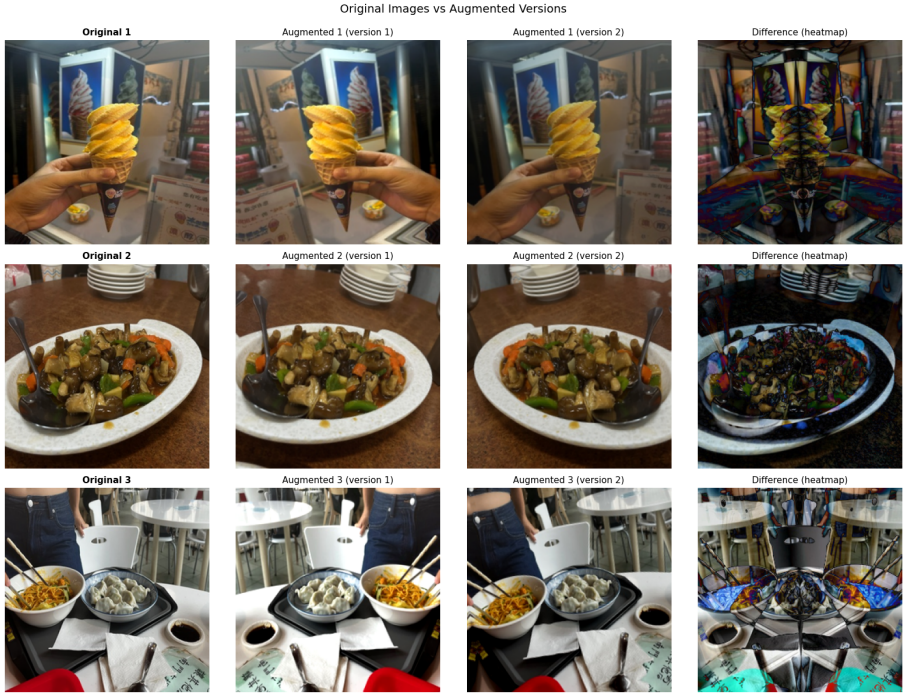


Figure 6: Original images, augmented versions, and pixel-wise difference heatmaps.

I apply data augmentation to increase the size and diversity of the dataset and reduce overfitting, since the original collection is relatively small and visually complex. The dataset initially contains 569 images, which is limited for training deep models on high-dimensional image data. After augmentation, the dataset expands to 1,707 images in total, combining the original photos with 1,138 augmented versions. This gives the model exposure to a wider range of visual variations without changing the underlying content of the images.

The augmentations are intentionally simple and realistic. Each image may be randomly flipped horizontally, rotated by up to ± 15 degrees, and adjusted in brightness and contrast within a $\pm 20\%$ range. The main scene stays the same, but the images change a bit in angle, lighting, and how they

look overall. The figure shows an original image next to two augmented versions, along with a difference heatmap that visualizes how the pixels change after augmentation. Looking at the heatmaps, most of the differences appear around edges, textures, and lighting, rather than changing the overall structure of the scene. This is a good sign, since the food and general layout stay recognizable while the details shift slightly. At the same time, it also shows why this data is hard to model. Even small visual changes end up affecting a lot of pixels, so it is easier for simpler models to smooth everything out and lose detail during reconstruction.

Stable Diffusion VAE Setup

Figure 7 provides a high-level overview of how Stable Diffusion operates in latent space, showing the interaction between the VAE and the diffusion UNet during the generation process. Including this diagram helps understand the implementation details that follow in a clear conceptual picture of the model's overall workflow.

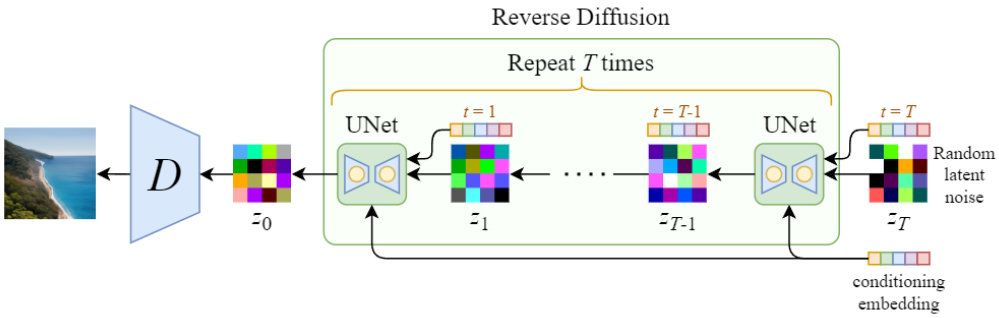


Figure 7: Reverse diffusion process in latent space, where a UNet iteratively denoises Gaussian noise and the final latent is decoded into an image using a VAE decoder (Steins, 2023).

The diagram illustrates the reverse diffusion process performed in the latent space learned by the Stable Diffusion VAE. Generation begins from a random Gaussian latent

$$z_T \sim \mathcal{N}(0, I),$$

which has the same shape as the VAE latents, and proceeds backward through time for T steps. At each timestep t, a UNet predicts the noise component present in the current latent z_t and removes a small amount of noise, producing a slightly cleaner latent z_{t-1}. This iterative denoising process continues until a final clean latent z_0 is obtained. Once denoising is complete, the VAE decoder D maps this latent back into image space, producing the final generated image. Although the diagram includes a conditioning embedding, this project uses unconditional generation, so the conditioning input is replaced with zero-valued embeddings while preserving the same overall architectural structure.

I start the project by establishing a latent preprocessing and validation pipeline based on the pre-trained Stable Diffusion VAE. The goal at this stage is not image generation, but to define and verify the latent space in which a diffusion model will later be trained. By encoding the dataset into this fixed, learned representation and decoding it back to image space, I can evaluate whether the latent variables preserve the essential semantic structure of the images while reducing dimensionality. Keeping the VAE frozen allows me to isolate the properties of the latent space itself and avoid introducing additional sources of instability. To make this setup concrete, I first load the pretrained Stable Diffusion v1.5 VAE and put it in evaluation mode, so that all subsequent experiments reflect the fixed latent geometry learned during pretraining.

```
device = cuda  if torch.cuda.is_available() else cpu
print(fUsing device: {device})

vae = AutoencoderKL.from_pretrained(
    runwayml/stable-diffusion-v1-5,
    subfolder=vae
)
```

```
vae = vae.to(device)
vae.eval() # Set to evaluation mode
print(f Latent channels: {vae.config.latent_channels})
```

Formally, the encoder learns an approximate posterior distribution

$$q_\phi(\mathbf{z} \mid \mathbf{x}),$$

while the decoder defines a conditional likelihood

$$p_\theta(\mathbf{x} \mid \mathbf{z}),$$

where $\mathbf{x} \in \mathbb{R}^{256\times256\times3}$ denotes an RGB image and $\mathbf{z} \in \mathbb{R}^{4\times32\times32}$ represents the corresponding latent variable. Unlike a deterministic autoencoder, this model explicitly captures uncertainty by modeling the encoder output as a Gaussian distribution,

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}^2_\phi(\mathbf{x})\big)\,,$$

which encourages continuity and smoothness in the learned latent space.

Latent samples are obtained using the reparameterization trick, given by

$$\mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{x}) + \boldsymbol{\sigma}_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

which allows gradients to propagate through the stochastic sampling process. Although the VAE parameters are fixed in this stage, this formulation is important for the overall pipeline because it defines the statistical structure of the latent space in which diffusion will later take place. The KL regularization inherent to the VAE objective encourages the latent distribution to remain close to a standard normal prior, a property that aligns naturally with the assumptions of denoising diffusion models.

The Stable Diffusion VAE was originally trained on a very large and diverse image dataset, and because of that it tends to preserve high-level semantic structure while compressing away fine textures and pixel-level detail. In practice, this means that images which look similar at a semantic level end up close to each other in latent space, even if they differ in small visual details. This is exactly the kind of representation that diffusion models work well with, since diffusion assumes a smooth, roughly Gaussian space where noise can be added and removed gradually. By checking that this encoder–decoder pair produces stable and visually reasonable reconstructions on my own dataset, I can be more confident that the diffusion model trained later will focus on learning global structure rather than getting distracted by low-level pixel noise.

To further validate that the latent space defined by the Stable Diffusion VAE is suitable here, I analyze how individual images are encoded and reconstructed. Each image is normalized to the range $[-1, 1]$, consistent with the scale used during the VAE's original training, and then passed through the encoder to obtain a latent sample $\mathbf{z}$ drawn from the approximate posterior distribution

$$\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x}).$$

This latent representation is decoded back into image space using the decoder

$$\hat{\mathbf{x}} = \mathbb{E}_{p_\theta(\mathbf{x}|\mathbf{z})},$$

producing a reconstruction $\hat{\mathbf{x}}$ of the original image. Before processing the full dataset, I verify the behavior of the encoder–decoder pair on individual images.

```
# Test encoding/decoding
rand_ind = np.random.randint(0, len(X_train))
test_image = X_train[rand_ind:rand_ind+1] # Shape: (1, 256, 256, 3) in [0, 1]

# Convert to tensor and normalize to [-1, 1] for encoding
test_tensor = torch.from_numpy(test_image).permute(0, 3, 1, 2).float()
test_tensor = test_tensor * 2.0 - 1.0
test_tensor = test_tensor.to(device)

# Encode
with torch.no_grad():
```

```
    encoded = vae.encode(test_tensor)
    latent = encoded.latent_dist.sample()
    print(f   Encoded. Latent shape: {latent.shape})
    print(f   Latent stats: mean={latent.mean():.4f}, std={latent.std():.4f})

# Decode (no scaling needed)
with torch.no_grad():
    decoded = vae.decode(latent).sample
    decoded_np = decoded.cpu().numpy()
    decoded_np = (decoded_np + 1) / 2
    decoded_np = np.clip(decoded_np, 0, 1)
    decoded_np = np.transpose(decoded_np, (0, 2, 3, 1))
    mse = np.mean((test_image - decoded_np) ** 2)
    print(f   Decoded. Output shape: {decoded_np.shape})
    print(f   MSE: {mse:.6f})
```

Reconstruction quality is measured using mean squared error (MSE), defined as

$$\mathrm{MSE}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{HWC} \sum_{i=1}^{HWC} (x_i - \hat{x}_i)^2,$$

where H, W, and C denote the image height, width, and number of channels, respectively. Across multiple examples, the MSE remains consistently low (on the order of 10^{-3}), with most reconstruction errors concentrated around edges and fine textures rather than large-scale structure. We would expect that, since VAEs prioritize a smooth and regularized latent space over perfect pixel-level fidelity due to the KL divergence term in the objective. Because diffusion models focus on modeling global semantic structure and operate under Gaussian noise assumptions, these results support the use of this latent representation as a foundation for the diffusion process that follows. Even though this test uses only one image, it confirms that the latent representation preserves global structure while smoothing fine details.

Figure 8: Original image, Stable Diffusion VAE reconstruction, and difference map.

Figure 8 shows an example of an original food image, its reconstruction produced by the Stable Diffusion VAE, and the corresponding pixel-wise difference map. Visually, the reconstructed image closely matches the original: the plate shape, food arrangement, colors of the vegetables, and overall lighting are all well preserved. Fine details such as the sliced radishes and texture on the food are slightly smoother in the reconstruction, but their positions and shapes remain clearly recognizable. This behavior is consistent with the very low reconstruction error (MSE = 0.0009). The difference map shows that most of the errors are around edges and fine textures rather than large parts of the image, which is expected since VAEs tend to smooth out high-frequency details.

To better understand where reconstruction errors occur, the pixel-wise absolute difference between the original image $\mathbf{x}$ and its reconstruction $\hat{\mathbf{x}}$ is computed as

$$\mathbf{D} = |\mathbf{x} - \hat{\mathbf{x}}|.$$

Visualizing the reconstruction error as a heatmap highlights where the VAE introduces discrepancies beyond what is captured by the MSE. Errors are small in smooth regions and mainly appear around edges and fine textures, which is expected given the KL-regularized nature of the model. The absence of large-scale structural errors indicates that scene layout and object identity are preserved, even though

high-frequency details are partially smoothed. This behavior is well suited for latent diffusion, which relies on stable global structure rather than exact pixel-level reconstruction.

Then, the images are encoded into latent space in a streamed, batch-wise manner rather than all at once. For a dataset of images $\{x_i\}_{i=1}^{N}$, each image is independently mapped to a latent variable by sampling from the VAE's approximate posterior,

$$z_i \sim q_\phi(z \mid x_i),$$

and the resulting latent tensors are written to disk.

```
def encode_images_to_latent_streaming(images, vae, device, batch_size=8, save_dir=latents):
    os.makedirs(save_dir, exist_ok=True)
    n = len(images)
    latent_paths = []

    for i in range(0, n, batch_size):
        batch = images[i:i+batch_size]

        batch_tensor = torch.from_numpy(batch).permute(0,3,1,2).float()
        batch_tensor = batch_tensor * 2 - 1
        batch_tensor = batch_tensor.to(device)

        with torch.inference_mode():
            latents = vae.encode(batch_tensor).latent_dist.sample()

        lat_np = latents.cpu().numpy().astype(float16)

        path = f{save_dir}/batch_{i}.npy
        np.save(path, lat_np)
        latent_paths.append(path)

        del batch, batch_tensor, latents, lat_np
        gc.collect()

        print(fEncoded {min(i+batch_size, n)}/{n})

    return latent_paths
```

This replaces the original high-dimensional image data with a compact latent representation while preserving the semantic content needed for downstream modeling.

From a practical standpoint, this approach significantly reduces memory pressure during training. Whereas an RGB image of size $256 \times 256 \times 3$ requires storing nearly two hundred thousand pixel values, its corresponding latent representation has shape $4 \times 32 \times 32$, reducing the dimensionality by roughly a factor of 48. As a result, latent tensors are faster to load, cheaper to store, and more stable to process in large batches.

This streaming strategy decouples the expensive image encoding step from the diffusion training phase. Once the latent dataset has been generated, the diffusion model can be trained entirely in latent space without repeatedly passing images through the VAE. This not only improves computational efficiency, but also ensures that diffusion training remains feasible under limited GPU memory constraints, while operating on a representation that has already been validated to preserve global structure and semantic consistency.

After encoding and storing the dataset in latent space, the latent representations are decoded back into image space as a consistency check. Given a latent variable z, the decoder defines a conditional distribution over images,

$$\hat{x} = p_\theta(x \mid z),$$

which maps each stored latent tensor back to the pixel domain. This decoding step is not used for training, but serves purely as a sanity check on the encoding, saving, and loading pipeline. In particular, this confirms that no numerical corruption is introduced during disk I/O, that normalization is applied consistently, and that tensor shapes remain compatible with the decoder.

```
def decode_latents_streaming(latent_paths, vae, device, save_dir=recon):
```

```
    os.makedirs(save_dir, exist_ok=True)
    out_paths = []
    idx = 0

    for path in latent_paths:
        lat = np.load(path)
        lat_tensor = torch.from_numpy(lat).float().to(device)

        with torch.inference_mode():
            dec = vae.decode(lat_tensor).sample

        dec = (dec + 1) / 2
        dec = dec.clamp(0,1)
        dec_np = dec.permute(0,2,3,1).cpu().numpy()

        for j in range(len(dec_np)):
            out_path = f{save_dir}/img_{idx}.npy
            np.save(out_path, dec_np[j])
            out_paths.append(out_path)
            idx += 1

        del lat, lat_tensor, dec, dec_np
        gc.collect()

        print(fDecoded {idx} images)

    return out_paths
```

Beyond individual examples, reconstruction quality is evaluated at the batch level by comparing original images to their decoded counterparts across multiple samples. For each image, reconstruction error is measured using mean squared error (MSE), and these values are aggregated across a batch using the root mean squared error (RMSE),

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathrm{MSE}_i},$$

where N denotes the number of images in the batch. This provides a more robust view of reconstruction behavior than single-image inspection alone.

Consistently low batch-level RMSE indicates that the latent space is stable across the dataset and that reconstruction quality does not degrade for particular subsets of images. The absence of large deviations or outliers suggests that the preprocessing pipeline does not introduce numerical instabilities or mode collapse, and that the latent representations maintain uniform quality. Together, these results validate the full latent preprocessing workflow and confirm that the dataset is well-prepared for training a diffusion model that operates entirely in latent space.

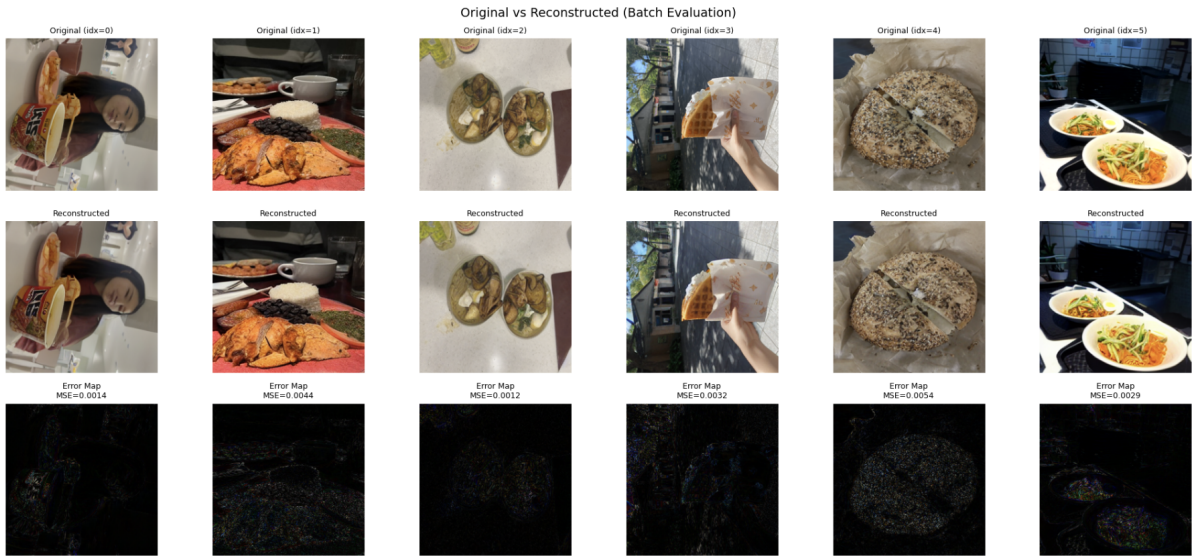


Figure 9: Batch-level comparison of original images & Stable Diffusion VAE reconstructions

Figure 9 builds on the single-image results by showing how the reconstruction behaves across a batch of different images. Overall, the reconstructions stay very close to the originals, with the main scene layout, object identity, and overall colors clearly preserved. When looking at the error maps, most of the differences are small and tend to appear around edges and fine textures, while smoother regions show very little error. This behavior is consistent across the batch and matches what we would expect from a KL-regularized VAE, which trades off some high-frequency detail in favor of a smooth and well-behaved latent space. The consistently low batch-level RMSE suggests that this latent representation is stable across the dataset and does not break down for particular images, which is important for training a diffusion model on top of it.

To further contextualize the quality of the Stable Diffusion VAE used in this project, I include a reconstruction comparison from a previous assignment, where a separately trained VAE was evaluated on the same data.

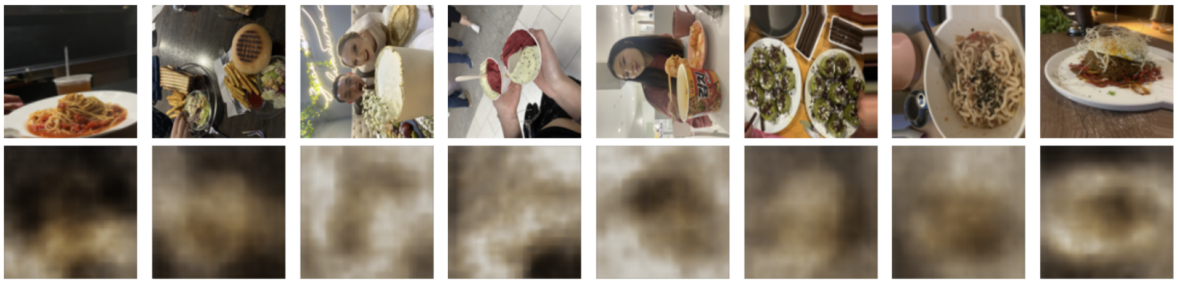


Figure 10: Enter Caption

Comparing these reconstructions to the Stable Diffusion VAE results shows a clear difference in representation quality and stability. In the previous assignment, the VAE reconstructions appear much blurrier, with fine details really washed out and object boundaries poorly defined, indicating a more aggressive information bottleneck in the learned latent space. While the overall color distribution is somewhat preserved, much of the semantic structure is lost, meaning that the latent variables struggle to retain meaningful visual content. In contrast, the Stable Diffusion VAE reconstructions maintain recognizable objects, sharper edges, and more coherent spatial layouts across the batch, as also reflected in the lower and more consistent reconstruction error. This comparison reinforces that the pretrained Stable Diffusion VAE provides a much richer and more stable latent representation, making it far better suited as a foundation for downstream diffusion-based generation.

Overall, these results show that the Stable Diffusion VAE provides a stable and reliable latent representation, keeping the main structure of the images while smoothing out small details. This gives

confidence that the data is well prepared for training a diffusion model in latent space, where the focus can be on learning meaningful structure instead of pixel-level noise.

Latent Diffusion Model: Generative Training and Sampling

The next stage of the project focuses on training a generative model directly in latent space. Instead of operating on high-dimensional RGB images, the diffusion process is learned over the compressed VAE latents, which reduces computational cost while preserving global semantic structure.

Latent DDPM with a Minimal Time-Conditioned CNN (Baseline Model)

```
class TinyUNet_v2(nn.Module):
    def __init__(self, channels=4, hidden=64, tdim=64):
        super().__init__()

        # Timestep embedding
        self.time_mlp = nn.Sequential(
            nn.Linear(1, tdim),
            nn.ReLU(),
            nn.Linear(tdim, tdim),
            nn.ReLU()
        )

        self.conv1 = nn.Conv2d(channels + tdim, hidden, 3, padding=1)
        self.conv2 = nn.Conv2d(hidden, hidden, 3, padding=1)
        self.conv3 = nn.Conv2d(hidden, channels, 3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x, t):
        t = t.view(-1, 1)
        t_embed = self.time_mlp(t).unsqueeze(-1).unsqueeze(-1)
        t_embed = t_embed.expand(-1, -1, x.shape[2], x.shape[3])

        x = torch.cat([x, t_embed], dim=1)
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.conv3(x)
        return x

diffusion_model = TinyUNet_v2().to(device)
print(Loaded TinyUNet v2)
```

This code cell defines the neural network used to model the denoising step in the latent diffusion process. The goal of this model is to learn a function that predicts the Gaussian noise added to a latent representation at a given diffusion timestep, which is the core learning task in denoising diffusion probabilistic models.

Let $x_t \in \mathbb{R}^{4\times32\times32}$ denote a noisy latent variable at diffusion timestep t. The network is trained to approximate the noise component ϵ used to construct x_t from a clean latent x_0. Formally, the model learns a function

$$\hat{\epsilon}_\theta(x_t, t) \approx \epsilon,$$

where θ denotes the trainable parameters of the network.

To condition the model on the timestep, the scalar diffusion index t is first passed through a small multilayer perceptron to produce a learned timestep embedding,

$$\mathbf{e}_t = f_{\text{MLP}}(t) \in \mathbb{R}^d.$$

This embedding is then broadcast spatially and concatenated with the noisy latent along the channel dimension, allowing the network to adapt its denoising behavior depending on how much noise is present.

The combined input is processed by a sequence of convolutional layers that preserve spatial resolution while transforming channel-wise features. The final output has the same shape as the input latent and represents the predicted noise $\hat{\epsilon}_\theta(x_t, t)$.

This code cell implements the training procedure for a Denoising Diffusion Probabilistic Model (DDPM) operating directly in the Stable Diffusion VAE latent space.

```
class TinyUNet_v2(nn.Module):
T = 1000

beta = torch.linspace(1e-4, 0.02, T).to(device)
alpha = 1 - beta
alpha_bar = torch.cumprod(alpha, dim=0)

optimizer = torch.optim.Adam(diffusion_model.parameters(), lr=1e-4)

latents_tensor = torch.from_numpy(X_train_latent[:500]).float().to(device)

steps = 1500
batch_size = 16

print(Training DDPM...)

for step in range(steps):
    idx = torch.randint(0, len(latents_tensor), (batch_size,))
    x0 = latents_tensor[idx]

    t = torch.randint(1, T, (batch_size,), device=device)
    noise = torch.randn_like(x0)

    a_bar = alpha_bar[t].view(-1, 1, 1, 1)
    x_t = torch.sqrt(a_bar) * x0 + torch.sqrt(1 - a_bar) * noise

    pred_noise = diffusion_model(x_t, t.float())

    loss = nn.functional.mse_loss(pred_noise, noise)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if step % 200 == 0:
        print(step, loss.item())
```

The diffusion process is defined over a fixed number of timesteps $T = 1000$, where Gaussian noise is gradually added to a clean latent variable x_0. At each timestep t, the forward noising process produces a corrupted latent x_t according to

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I),$$

where

$$\alpha_t = 1 - \beta_t, \quad \bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s.$$

The noise schedule β_t is chosen as a linear interpolation between 10^{-4} and 0.02, which controls the rate at which information is destroyed over time.

The model is trained to predict the noise term ε that was added during the forward process. Given a noisy latent x_t and timestep t, the neural network outputs a noise estimate

$$\hat{\varepsilon}_\theta(x_t, t).$$

Training is performed using a mean squared error (MSE) loss,

$$\mathcal{L}_{\text{DDPM}} = \mathbb{E}_{x_0, t, \varepsilon}\left[\left\|\varepsilon - \hat{\varepsilon}_\theta(x_t, t)\right\|^2\right],$$

which encourages the model to accurately estimate the injected noise across different timesteps.

At each training iteration, a batch of latent vectors x_0 is sampled from the precomputed VAE latent dataset. A random timestep $t \sim \text{Uniform}\{1, \ldots, T\}$ is selected, Gaussian noise ε is added to form x_t, and the model predicts the corresponding noise. Model parameters are updated using the Adam optimizer to minimize the MSE loss.

The loss going down shows that the model is gradually learning how to remove noise from the latent inputs. This is the main skill the diffusion model needs in order to generate new samples.

```
diffusion_model.eval()

with torch.no_grad():
    x = torch.randn(1, 4, 32, 32).to(device)

    for t in reversed(range(1, T)):
        beta_t = beta[t]
        alpha_t = alpha[t]
        alpha_bar_t = alpha_bar[t]

        t_tensor = torch.tensor([t], device=device).float()
        pred_noise = diffusion_model(x, t_tensor)

        x = (1 / torch.sqrt(alpha_t)) * (
            x - ((1 - alpha_t) / torch.sqrt(1 - alpha_bar_t)) * pred_noise
        )

        if t > 1:
            x = x + torch.sqrt(beta_t) * torch.randn_like(x)

# Decode
with torch.no_grad():
    decoded = vae.decode(x).sample.cpu().detach().numpy()[0]

decoded = (decoded + 1) / 2
decoded = decoded.clip(0, 1)
decoded = decoded.transpose(1, 2, 0)

plt.figure(figsize=(6,6))
plt.imshow(decoded)
plt.title(Generated Image (Fixed DDPM))
plt.axis(off )
plt.show()
```

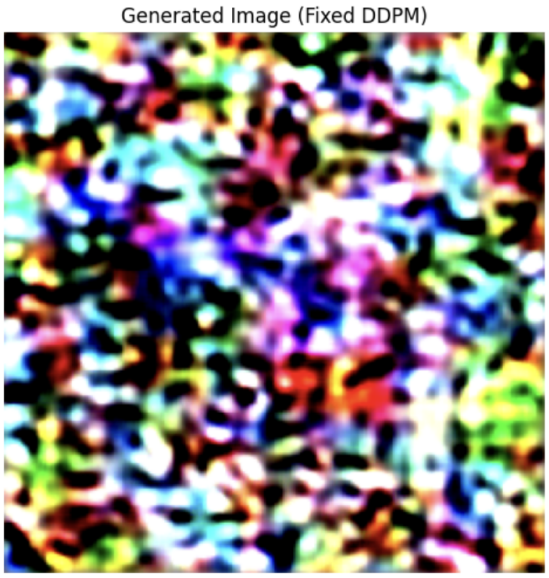


Figure 11: Image generated by the latent-space DDPM after reverse diffusion sampling.

Now, it performs the sampling step of the diffusion model. It starts from pure Gaussian noise in latent space and then iteratively applies the learned reverse diffusion process, using the model's noise predictions at each timestep to gradually denoise the latent. After the final step, the clean latent is decoded back into image space using the frozen VAE decoder, producing a generated image. This confirms that the trained model can generate samples by reversing the forward noising process step by step.

The generated image is still very noisy and does not show clear or recognizable structure, which suggests that this version of the model is not performing well. Even though the reverse diffusion process technically works and produces a sample that can be decoded by the VAE, the result looks more like colored noise than a meaningful image. This likely comes from a combination of factors, such as the very small and simple UNet architecture, the limited number of training steps, and the relatively small training subset used here. In practice, diffusion models usually need deeper networks, more capacity, and longer training to learn strong denoising behavior. So, the model learns how to reduce noise in a general way and does not to reconstruct coherent global structure.

Because the simple latent DDPM struggles to produce coherent structure, the next step is to switch to a stronger pretrained diffusion backbone and adapt it using a more efficient fine-tuning strategy.

LoRA Fine-Tuning of a Pretrained Stable Diffusion Model

Since the simple latent DDPM struggles to generate coherent structure, the next step is to move away from training a diffusion model from scratch and instead leverage a strong pretrained backbone. In this section, I introduce LoRA fine-tuning on a pretrained Stable Diffusion model, which allows the model to adapt to the dataset while preserving the powerful denoising and generative capabilities learned from large-scale training.

The goal of this code is to fine-tune a large pretrained Stable Diffusion UNet efficiently by modifying only a small number of parameters, instead of updating the full model. This is done using LoRA (Low-Rank Adaptation), which injects trainable low-rank matrices into existing linear layers while keeping the original pretrained weights frozen.

```
# Increase recursion limit for deep models like UNet
sys.setrecursionlimit(4000)

# Custom LoRALinear module to replace nn.Linear
class LoRALinear(nn.Module):
    def __init__(self, linear_layer, rank=8, lora_alpha=1.0):
        super().__init__()
        self.linear_layer = linear_layer # The original nn.Linear
        self.in_features = linear_layer.in_features
        self.out_features = linear_layer.out_features
```

```
        self.rank = rank
        self.lora_alpha = lora_alpha
        self.scaling = self.lora_alpha / self.rank

        self.lora_down = nn.Linear(self.in_features, rank, bias=False)
        self.lora_up = nn.Linear(rank, self.out_features, bias=False)

        # Initialize lora_up weights to zero and lora_down with Kaiming uniform
        nn.init.zeros_(self.lora_up.weight)
        nn.init.kaiming_uniform_(self.lora_down.weight, a=math.sqrt(5))

        # Freeze the original linear layer's parameters
        for param in self.linear_layer.parameters():
            param.requires_grad = False

    def forward(self, x):
        # Original forward pass
        original_output = self.linear_layer(x)

        # LoRA additive component
        lora_output = self.lora_up(self.lora_down(x)) * self.scaling
        return original_output + lora_output

# Collect (parent_module, child_name, original_child_module) to replace
modules_to_replace = []
for name, module in unet.named_modules():
    for child_name, child_module in module.named_children():
        if isinstance(child_module, nn.Linear):
            modules_to_replace.append((module, child_name, child_module))

# Perform replacement
for parent_module, child_name, original_child_module in modules_to_replace:
    new_lora_layer = LoRALinear(original_child_module, rank=8)
    setattr(parent_module, child_name, new_lora_layer)

# Move the entire UNet to the device again to ensure new LoRA layers are on GPU
unet.to(device)

# Freeze everything except LoRA weights
for name, param in unet.named_parameters():
    # Only parameters within LoRALinear (lora_down and lora_up) should be trainable
    if  lora_down in name or lora_up in name:
        param.requires_grad = True
    else:
        param.requires_grad = False # Ensures original_linear_layer params remain frozen
```

Stable Diffusion's UNet contains many `nn.Linear` layers, particularly inside attention blocks. Instead of retraining these full weight matrices, each linear layer

$$W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$$

is replaced by a LoRA-augmented version that keeps W frozen and learns only a low-rank update. Concretely, the original linear transformation

$$y = Wx$$

is modified to

$$y = Wx + \alpha \cdot BAx,$$

where:

- $A \in \mathbb{R}^{r \times d_{\text{in}}}$ corresponds to the `lora_down` projection,
- $B \in \mathbb{R}^{d_{\text{out}} \times r}$ corresponds to the `lora_up` projection,

- $r \ll \min(d_{\text{in}}, d_{\text{out}})$ is the LoRA rank,
- α/r is a scaling factor used to keep the updates numerically stable.

This low-rank decomposition dramatically reduces the number of trainable parameters while still allowing the model to adapt effectively to new data.

Inside the custom `LoRALinear` module, the original pretrained linear layer is kept fully intact and its parameters are frozen, so that

$$\frac{\partial W}{\partial \theta} = 0,$$

which preserves the general knowledge learned during large-scale pretraining. Instead of updating the full weight matrix, two new trainable matrices are introduced to form a low-rank adaptation. The first projection, `lora_down`, maps the input into a lower-dimensional subspace according to

$$x \mapsto Ax,$$

while the second projection, `lora_up`, maps it back to the original output space via

$$Ax \mapsto BAx.$$

To ensure stable and controlled training, the matrix B is initialized to zero so that the LoRA path initially has no effect on the model's output, while the matrix A is initialized using Kaiming uniform initialization. As a result, training begins exactly from the pretrained model and gradually adapts as the low-rank parameters are learned. During the forward pass, the output of the layer is computed as

$$\text{output} = Wx + \frac{\alpha}{r} BAx,$$

where the scaling factor α/r keeps the LoRA update numerically stable. This additive structure allows the model to gently adjust pretrained representations rather than relearning them from scratch.

After defining the `LoRALinear` module, LoRA is automatically injected into the UNet by iterating over all submodules and identifying every `nn.Linear` layer. Each such layer is replaced *in place* with a `LoRALinear` wrapper, ensuring that low-rank adaptation is applied consistently across the entire UNet architecture, including attention blocks where linear projections are most critical. Following this replacement, all original UNet parameters are frozen, and only the LoRA parameters are left trainable. Formally, optimization is restricted to the set

$$\theta = \{A_i, B_i\}_{i=1}^{L},$$

where A_i and B_i denote the low-rank LoRA matrices associated with each of the L adapted linear layers, rather than the full parameter set of the UNet. This design drastically reduces the number of trainable parameters, lowers memory and compute requirements, and helps prevent catastrophic forgetting by preserving pretrained weights. As a result, fine-tuning remains stable and effective even when training on relatively small datasets.

Thus, this LoRA setup lets the model adapt efficiently by making small, controlled updates to a pretrained UNet instead of retraining everything from scratch, which is especially practical when data and compute are limited.

After injecting LoRA layers into the pretrained Stable Diffusion UNet, the next step is to fine-tune only these low-rank parameters using the standard diffusion training objective. This training loop follows the same noise-prediction framework used in DDPMs, but restricts learning to the LoRA weights while keeping the VAE and original UNet parameters fixed.

```
optimizer = optim.Adam(filter(lambda p: p.requires_grad, unet.parameters()), lr=1e-4)

vae.eval()
unet.train()

num_steps = 1500 # can increase to 5000 for better quality
step = 0

for epoch in range(10):
    for batch in dataloader:
        step += 1
        if step > num_steps:
```

```
            break

        batch = batch.to(device)

        # Encode images to SD latent space
        with torch.no_grad():
            latents = vae.encode(batch).latent_dist.sample() * 0.18215

        # Sample random timestep
        noise = torch.randn_like(latents)
        timesteps = torch.randint(0, scheduler.num_train_timesteps, (latents.size(0),),
            device=device)

        noisy_latents = scheduler.add_noise(latents, noise, timesteps)

        batch_size_current = noisy_latents.shape[0]
        dummy_encoder_hidden_states = torch.zeros(
            (batch_size_current, 77, 768), # 77 tokens, 768 features (standard for SD1.5 CLIP)
            dtype=noisy_latents.dtype,
            device=device
        )
        noise_pred = unet(noisy_latents, timesteps,
            encoder_hidden_states=dummy_encoder_hidden_states).sample

        loss = torch.nn.functional.mse_loss(noise_pred, noise)

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

        if step % 100 == 0:
            print(fStep {step} / {num_steps}, Loss = {loss.item():.6f})

    if step > num_steps:
        break

print(LoRA fine-tuning complete!)
```

During training, each batch of images is first encoded into the Stable Diffusion latent space using the frozen VAE encoder. Given an input image x, the VAE produces a latent representation

$$z = \mathrm{VAE}(x),$$

which is scaled by a constant factor 0.18215 to match the latent distribution used during Stable Diffusion pretraining. A random diffusion timestep t is then sampled, and Gaussian noise $\epsilon \sim \mathcal{N}(0, I)$ is added according to the forward diffusion process,

$$z_t = \sqrt{\bar{\alpha}_t}\, z + \sqrt{1 - \bar{\alpha}_t}\, \epsilon,$$

where $\bar{\alpha}_t$ is determined by the predefined noise schedule. The UNet, augmented with LoRA layers, is trained to predict the noise component ϵ from the noisy latent z_t and timestep t. Because this setup does not use text conditioning, dummy encoder hidden states are provided to match the expected input format of the Stable Diffusion UNet.

The training objective is the standard mean squared error loss between the true noise and the predicted noise,

$$\mathcal{L} = \mathbb{E}_{z,t,\epsilon}\left[\|\epsilon - \epsilon_\theta(z_t, t)\|^2\right],$$

which encourages the model to accurately reverse the forward noising process. Importantly, optimization is performed only over parameters that require gradients, meaning that updates are restricted to the LoRA matrices $\{A_i, B_i\}$, while the original UNet weights remain frozen. This allows the model to adapt its denoising behavior through low-rank updates without overwriting pretrained knowledge. As training progresses, the loss gradually decreases, indicating that the LoRA-augmented UNet is learning to better

predict noise in latent space, enabling improved reverse diffusion while remaining stable even with a limited number of training steps.

After fine-tuning the UNet with LoRA, the final step is to test the model by actually generating an image.

```
unet.eval()

with torch.no_grad():
    latents = torch.randn(1, 4, 64, 64).to(device)

    for t in reversed(range(scheduler.num_train_timesteps)):
        latent_model_input = latents

        # Explicitly create dummy encoder_hidden_states for unconditional generation
        # The UNet2DConditionModel expects (batch_size, sequence_length, cross_attention_dim),
            typically (B, 77, 768)
        batch_size_current = latent_model_input.shape[0]
        dummy_encoder_hidden_states = torch.zeros(
            (batch_size_current, 77, 768), # 77 tokens, 768 features (standard for SD1.5 CLIP)
            dtype=latent_model_input.dtype,
            device=device
        )
        noise_pred = unet(latent_model_input, torch.tensor([t], device=device),
            encoder_hidden_states=dummy_encoder_hidden_states).sample

        latents = scheduler.step(noise_pred, t, latents).prev_sample

    # Decode via VAE
    image = vae.decode(latents / 0.18215).sample
    image = (image.clamp(-1,1) + 1) / 2
    image = image.cpu().permute(0,2,3,1).numpy()[0]

plt.figure(figsize=(6,6))
plt.imshow(image)
plt.axis(off )
plt.title(LoRA Fine-tuned Stable Diffusion Output)
```

Generation starts from pure Gaussian noise in latent space,

$$z_T \sim \mathcal{N}(0, I),$$

where z_T has the same shape as Stable Diffusion latents. The model then iteratively applies the learned reverse diffusion process by looping backward over timesteps $t = T, \ldots, 1$. At each step, the LoRA-augmented UNet predicts the noise component $\hat{\epsilon}_\theta(z_t, t)$, and the scheduler uses this prediction to compute a slightly denoised latent,

$$z_{t-1} = \text{SchedulerStep}(z_t, \hat{\epsilon}_\theta, t).$$

Because this is unconditional generation, dummy encoder hidden states filled with zeros are passed to the UNet to satisfy its expected input format. After the final timestep, the clean latent z_0 is decoded using the frozen VAE decoder,

$$x = \text{VAE}^{-1}\left(\frac{z_0}{0.18215}\right),$$

and rescaled to valid pixel values. This process tests whether the LoRA fine-tuning successfully improved the UNet's ability to reverse the diffusion process and produce coherent images, without modifying the original pretrained model weights.

Now, let's look at the generated image.

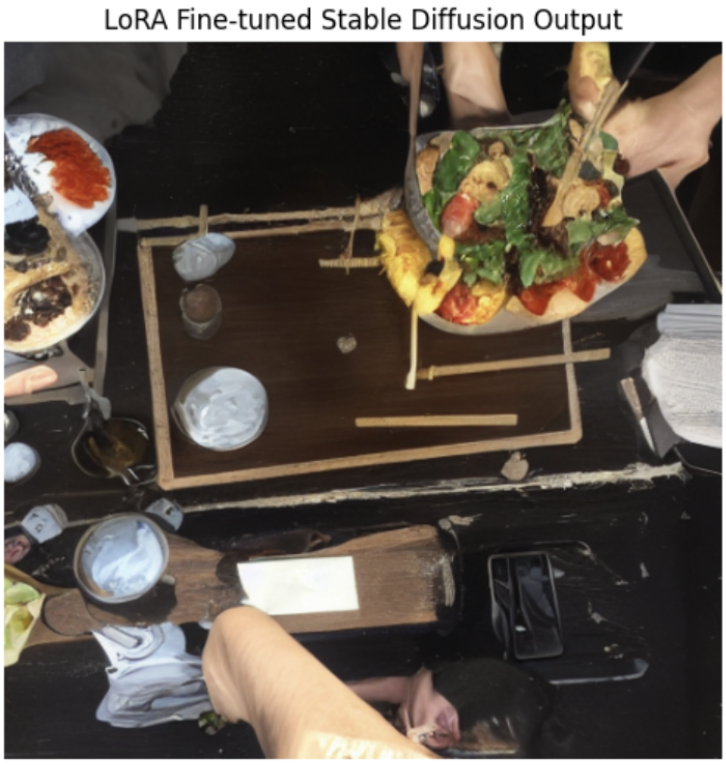


Figure 12: Sample image generated by the LoRA fine-tuned Stable Diffusion model.

Compared to the earlier results, this output is clearly much stronger and shows that the LoRA fine-tuning is actually working. The generated image now clearly resembles a real dining scene rather than random visual noise. From a top-down perspective, we can make out a table surface with plates and bowls spread across it, food piled in the center, and human hands reaching in as if someone is about to grab a bite. The colors and textures loosely match what we would expect from a meal, such as greens that look like vegetables, warmer tones that resemble cooked food, and a dark background that reads as a tabletop. While some objects are still warped in a very "AI-like" way, the scene feels recognizably human, almost like a slightly surreal overhead photo taken mid-dinner.

This suggests that using LoRA lets the pretrained Stable Diffusion UNet reuse what it already knows about images and focus mainly on learning how to denoise better, rather than relearning visual structure from scratch. Overall, the result shows that LoRA fine-tuning clearly improves image quality and makes latent-space diffusion work much better even with limited data and training time.

Conclusion

The main goal of this assignment was to understand how diffusion models work in latent space and to test whether meaningful image generation can be achieved by operating on Stable Diffusion's VAE latents instead of raw pixels. Over the course of the project, I gradually built up the pipeline step by step: first verifying that the VAE could faithfully reconstruct images, then analyzing the structure of the latent space with PCA, clustering, and nearest-neighbor search, and finally training diffusion models to reverse the noising process. An important part of this goal was not just producing an image, but understanding why certain approaches failed and how architectural choices affected the quality of the results. The early experiments with a small, custom DDPM showed that while the model technically learned to denoise, it struggled to produce coherent structure, which motivated switching to a stronger pretrained backbone and using LoRA fine-tuning.

A major challenge throughout the project was computational constraints. Many of the initial models ran extremely slowly or failed altogether when executed on CPU, forcing me to move the pipeline to GPU and carefully adjust batch sizes, training steps, and memory usage. In several cases, increasing batch size or model complexity caused out-of-memory errors, so I had to trade off training stability and speed. The final LoRA-fine-tuned Stable Diffusion model took nearly two hours to train, which is why

I only generated a single sample at the end. Despite this limitation, the improvement in image quality compared to earlier models clearly demonstrates the effectiveness of leveraging pretrained models and low-rank adaptation.

AI Statement

I used ChatGPT as a learning and support tool throughout this assignment to help me understand the models I was implementing at a conceptual and mathematical level. Before relying on AI, I first searched for existing online resources and papers, all of which are listed in the references section. However, I found that much of the available content on diffusion models, latent diffusion, and LoRA fine-tuning was fragmented, highly technical, or focused on narrow implementation details and not the full process. ChatGPT acted as a tutor that helped me connect these pieces together by explaining the overall workflow, the intuition behind each model component, and the underlying math in a step-by-step way. I frequently asked follow-up questions to clarify how the VAE, diffusion process, UNet, and LoRA updates interact, which significantly improved my understanding of the full pipeline rather than just copying code.

In addition to conceptual help, I used ChatGPT extensively for debugging and implementation support. Because this project relied on multiple libraries and frameworks, some packages were not installed by default, and I encountered version conflicts and runtime errors. ChatGPT helped me identify missing dependencies, install the correct packages, and debug errors related to tensor shapes, scheduler usage, and model input. It was also especially helpful when my models were running for extremely long periods of time. AI helped me to reason about computational bottlenecks, adjust batch sizes, move training from CPU to GPU, and reduce training steps when necessary. This was critical since my final LoRA fine-tuned model took nearly two hours to run, which is why I ultimately generated only a single sample image.

Finally, I used ChatGPT as a writing and presentation tool. Since Grammarly does not work reliably in Overleaf, so ChatGPT helped fix spelling, sentence structure, and clarity in my explanations, while keeping the content technically accurate and in my own words. It also helped me correct LaTeX syntax for equations, formatting, and references when I encountered compilation issues. Overall, I found this to be a very effective and productive way of using AI: not as a shortcut, but as an interactive tutor, debugger, and editor that supported both my technical learning and my ability to clearly communicate what I built. The process felt like a genuine learning experience, and AI played a key role in helping me complete a complex assignment that would have been very difficult to manage independently.

References

GeeksforGeeks. (2024, March 28). *Generate images from text in Python Stable Diffusion.* GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning/generate-images-from-text-in-python-stable-diffusion/

GeeksforGeeks. (2025, February 13). *What is Low Rank Adaptation (LoRA)?* GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning/what-is-low-rank-adaptation-lora/

Ho, J., Jain, A., & Abbeel, P. (2020). *Denoising diffusion probabilistic models.* In *Advances in Neural Information Processing Systems (NeurIPS 2020)* (Vol. 33, pp. 6840–6851). https://proceedings.neurips.cc/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf

Latent diffusion model (LDM). (2025, August 9). *Avahi.* https://avahi.ai/glossary/latent-diffusion-model-ldm/

Noble, J. (2025, January 28). *LoRA.* IBM. https://www.ibm.com/think/topics/lora

Steins. (2023, January 2). *Stable Diffusion clearly explained!* Medium. https://medium.com/@steinsfu/stable-diffusion-clearly-explained-ed008044e07e

Tam, A. (2024, June 5). *Running Stable Diffusion with Python.* MachineLearningMastery.com. https://machinelearningmastery.com/running-stable-diffusion-with-python/

Void. (2025, March 20). *Stable Diffusion II — Implementing it from scratch in Python.* Medium. https://medium.com/@atulit23/stable-diffusion-ii-implementing-it-from-scratch-in-python-a646156414f8

Copy of Final_Draft_Pipeline

December 19, 2025

1 Importing the libraries & Loading the data

[ ]:
```
# Standard library
import os
import sys
import glob
import math
import gc
import random

# Arrays and image processing
import numpy as np
from PIL import Image
from scipy.ndimage import rotate

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import rgb_to_hsv

# TensorFlow / Keras (keep only if needed)
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras import backend as K

# scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, DBSCAN
from sklearn.mixture import GaussianMixture
from sklearn.manifold import TSNE
from sklearn.metrics import silhouette_score, adjusted_rand_score,␣
↪mean_squared_error
```

```
# UMAP
try:
    import umap
except ImportError:
    !pip install umap-learn
    import umap

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

# Diffusers (Stable Diffusion components)
from diffusers import AutoencoderKL, UNet2DConditionModel, DDPMScheduler

# Reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
```

[ ]:
```
# Install and import required libraries for Stable Diffusion VAE
try:
    import diffusers
    import torch
    print(" diffusers and torch already installed")
except ImportError:
    print("Installing diffusers, transformers, and accelerate...")
    import subprocess
    import sys
    subprocess.check_call([sys.executable, "-m", "pip", "install", "diffusers",␣
 ↪"transformers", "accelerate", "torch"])
    import diffusers
    import torch
    print(" Libraries installed successfully")

from diffusers import AutoencoderKL

print(f"\n All required libraries loaded")
print(f"  PyTorch version: {torch.__version__}")
print(f"  Device available: {'cuda' if torch.cuda.is_available() else 'cpu'}")
```

```
 diffusers and torch already installed
Flax classes are deprecated and will be removed in Diffusers v1.0.0. We
recommend migrating to PyTorch classes or pinning your version of Diffusers.
Flax classes are deprecated and will be removed in Diffusers v1.0.0. We
```

```
recommend migrating to PyTorch classes or pinning your version of Diffusers.


 All required libraries loaded
 PyTorch version: 2.9.0+cu126
 Device available: cuda
```

[ ]:
```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```python
# Path to the images
DATA_GLOB = "/content/drive/MyDrive/food_images/*"

# Target image size
IMG_HEIGHT = 128
IMG_WIDTH = 128
IMG_CHANNELS = 3
TARGET_SIZE = (IMG_WIDTH, IMG_HEIGHT)   # PIL uses (width, height)
```

[ ]:
```python
# Collect all image paths
image_paths = sorted(glob.glob(DATA_GLOB))

print(f"Found {len(image_paths)} image files.")
image_paths[:10]
```

```
Found 569 image files.
```

[ ]:
```
['/content/drive/MyDrive/food_images/01ABB14B-7FA2-43C6-83EA-
5D7F1EE0B1A7_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/04D815BE-3905-4DDF-A9E2-
5259211A2A1F_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/04E32FFC-567F-450F-B20C-
C0D26E043A37_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/057248DA-87C3-4AAC-B36D-
9B79E743A258_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/05D4B527-9EEC-484B-886B-
ABD97B1E9365_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/0719C9D7-1F9E-4AED-A6AB-
B81BF8B437DA_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/0813A2B5-E3B5-44CC-8A0D-
2D9BEB0A3007_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/08EC11C4-EA23-45B1-82F2-
381FB4FCD903_4_5005_c.jpeg',
 '/content/drive/MyDrive/food_images/0BCB7385-B0A5-40EF-933F-
05A1AFB5959F_4_5005_c.jpeg',
```

```
 '/content/drive/MyDrive/food_images/0D343749-4E82-4C5D-9577-
0F5D7C70E097_4_5005_c.jpeg']
```

[ ]:
```python
# Load all images and preprocess them
def preprocess_images(image_paths, target_size=(128, 128), verbose=True):
    """
    Load and preprocess all images.

    Parameters:
    - image_paths: List of image file paths
    - target_size: Tuple (height, width) for resizing
    - verbose: Whether to print progress

    Returns:
    - X_raw: NumPy array of shape (N, H, W, C) with uint8 values [0, 255]
    - X_proc: NumPy array of shape (N, H, W, C) with float32 values [0, 1]
    """
    images = []
    failed = []

    if verbose:
        print(f"Loading and preprocessing {len(image_paths)} images...")
        print(f"Target size: {target_size}")

    for i, img_path in enumerate(image_paths):
        try:
            # Load image
            img = Image.open(img_path)

            if img.mode != 'RGB':
                img = img.convert('RGB')

            # Resize to target size
            img_resized = img.resize(target_size, Image.Resampling.LANCZOS)

            # Convert PIL Image to NumPy array
            img_array = np.array(img_resized, dtype=np.uint8)

            images.append(img_array)

            if verbose and (i + 1) % 50 == 0:
                print(f"  Processed {i + 1}/{len(image_paths)} images...")

        except Exception as e:
            failed.append((img_path, str(e)))
            if verbose:
```

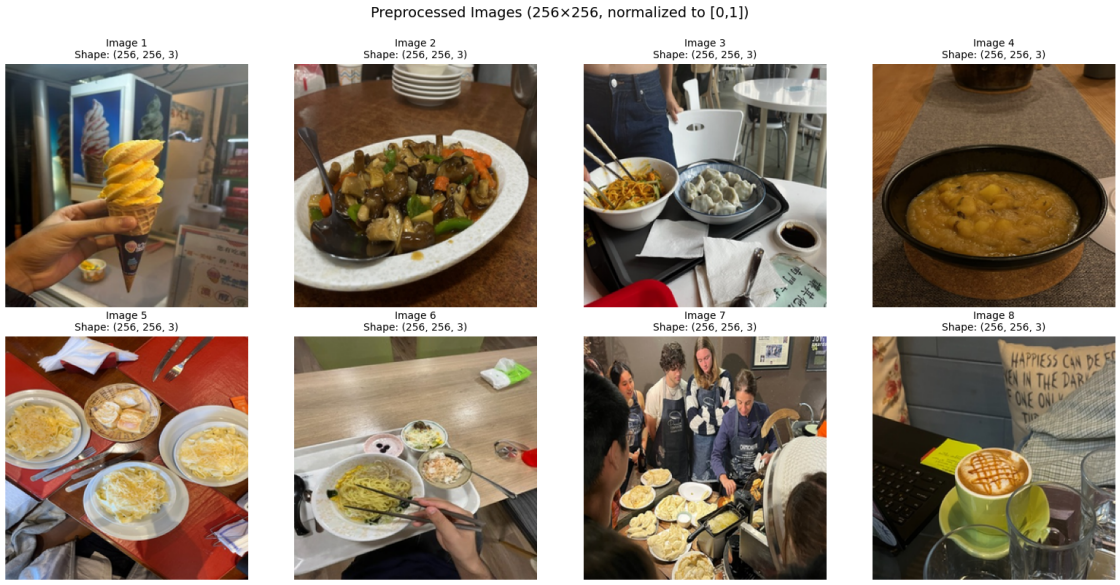
```
                print(f"  Warning: Failed to load {os.path.basename(img_path)}:␣
↪{e}")

    if failed:
        print(f"\n Failed to load {len(failed)} images")

    # Stack all images into a single NumPy array
    # Shape: (N, H, W, C) where N=number of images, H=height, W=width,␣
↪C=channels (3 for RGB)
    X_raw = np.stack(images, axis=0)

    print(f"  X_raw shape: {X_raw.shape}, dtype: {X_raw.dtype}")

    return X_raw, failed

# Preprocess all images
# The Stable Diffusion VAE was trained on 256×256 images and expects that␣
↪resolution
X_raw, failed_images = preprocess_images(image_paths, target_size=(256, 256))
```

```
Loading and preprocessing 569 images…
Target size: (256, 256)
  Processed 50/569 images…
  Processed 100/569 images…
  Processed 150/569 images…
  Processed 200/569 images…
  Processed 250/569 images…
  Processed 300/569 images…
  Processed 350/569 images…
  Processed 400/569 images…
  Processed 450/569 images…
  Processed 500/569 images…
  Processed 550/569 images…
  X_raw shape: (569, 256, 256, 3), dtype: uint8
```

[ ]:
```
# Normalize raw images to float32 in [0,1] and record number of originals
# This is the standard preprocessing step for neural networks

X_proc = X_raw.astype("float32") / 255.0

print("X_proc shape:", X_proc.shape, "dtype:", X_proc.dtype)
print(f"Pixel value range: [{X_proc.min():.3f}, {X_proc.max():.3f}]")

# Store the number of original images (before augmentation)
n_original = X_proc.shape[0]
print(f"\nNumber of original images: {n_original}")
```

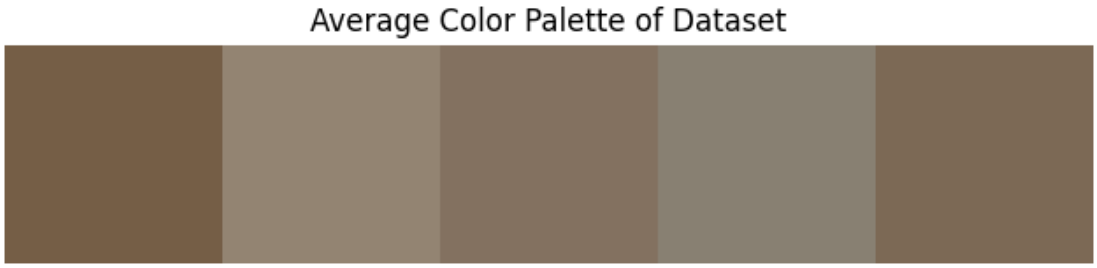
```
X_proc shape: (569, 256, 256, 3) dtype: float32
Pixel value range: [0.000, 1.000]

Number of original images: 569
```

[ ]:
```python
# Visualize a few preprocessed images to verify they look correct
fig, axes = plt.subplots(2, 4, figsize=(16, 8))

for i in range(min(8, len(X_proc))):
    row = i // 4
    col = i % 4

    axes[row, col].imshow(X_proc[i])
    axes[row, col].set_title(f"Image {i+1}\nShape: {X_proc[i].shape}",␣
 ↪fontsize=10)
    axes[row, col].axis('off')

plt.suptitle('Preprocessed Images (256×256, normalized to [0,1])', fontsize=14,␣
 ↪y=1.0)
plt.tight_layout()
plt.show()
```



2 Exploratory Data Analysis

[ ]:
```
def extract_palette(img, k=5):
    pixels = img.reshape(-1,3)
    kmeans = KMeans(n_clusters=k, random_state=0).fit(pixels)
    return kmeans.cluster_centers_

palettes = []
for idx in np.random.choice(len(X_proc), 50, replace=False):
    palettes.append(extract_palette(X_proc[idx]))

palettes = np.array(palettes)

avg_palette = palettes.mean(axis=0)
plt.figure(figsize=(8,2))
plt.imshow([avg_palette])
plt.title("Average Color Palette of Dataset")
plt.axis("off")
plt.show()
```



[ ]:
```
# PCA Latent Embedding

# Load all latent files and concatenate them into a single array
X_train_latent = np.concatenate([np.load(p) for p in train_latent_paths],␣
 ↪axis=0)
X_val_latent = np.concatenate([np.load(p) for p in val_latent_paths], axis=0)

# Flatten latents for PCA input
X_train_latent_flat = X_train_latent.reshape(len(X_train_latent), -1)
X_val_latent_flat = X_val_latent.reshape(len(X_val_latent), -1)

print(f"Flattened latent shape (train): {X_train_latent_flat.shape}")
print(f"Flattened latent shape (val):   {X_val_latent_flat.shape}")

# Compute PCA (keep 50 principal components)
pca = PCA(n_components=50)
```

```
X_train_latent_pca = pca.fit_transform(X_train_latent_flat)
X_val_latent_pca = pca.transform(X_val_latent_flat)

print(f"PCA latent shape (train): {X_train_latent_pca.shape}")
print(f"PCA latent shape (val):   {X_val_latent_pca.shape}")
print(f"Total explained variance (50 components): {pca.
  ↪explained_variance_ratio_.sum():.4f}")
```

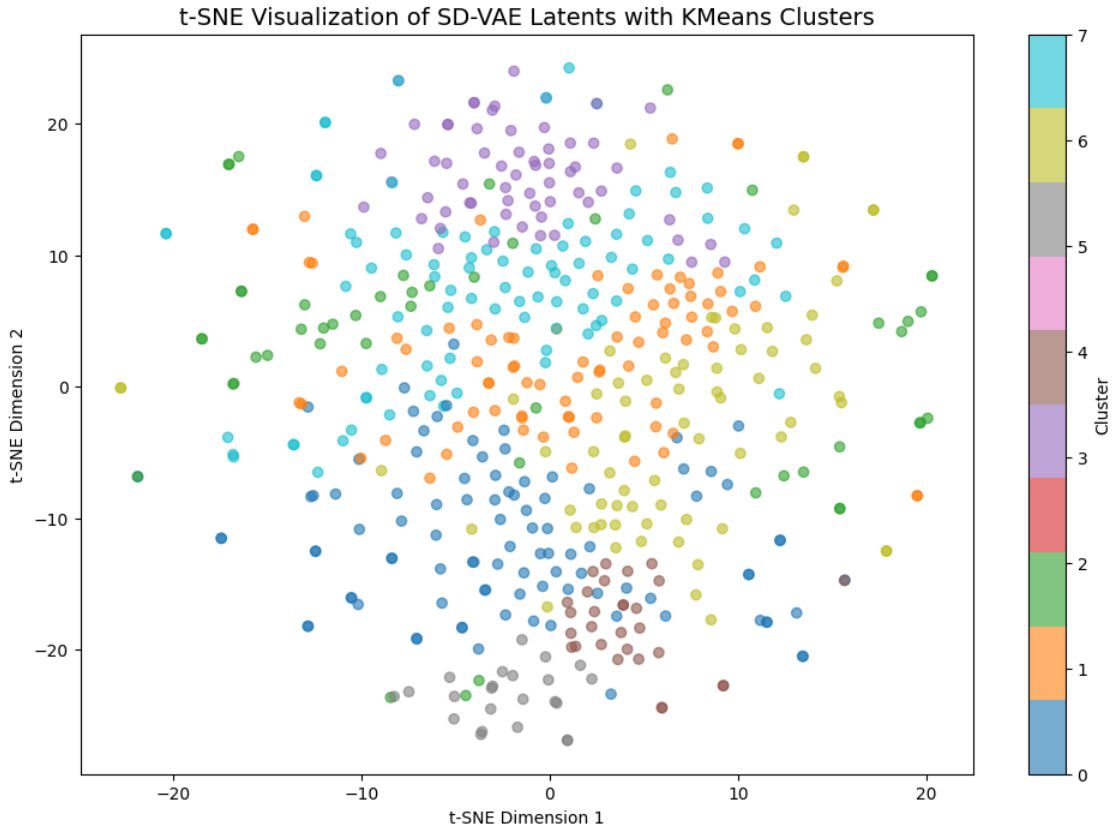
```
Flattened latent shape (train): (1450, 4096)
Flattened latent shape (val):   (257, 4096)
PCA latent shape (train): (1450, 50)
PCA latent shape (val):   (257, 50)
Total explained variance (50 components): 0.3755
```

[ ]:
```
# Clustering on SD-VAE Latent Space
# Using flattened latents for clustering
features = X_train_latent_flat

# KMeans
n_clusters = 8
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
kmeans_labels = kmeans.fit_predict(features)
print(f"KMeans clusters: {len(np.unique(kmeans_labels))}")

# DBSCAN
dbscan = DBSCAN(eps=50, min_samples=5)
dbscan_labels = dbscan.fit_predict(features)
print(f"DBSCAN clusters: {len(np.unique(dbscan_labels[dbscan_labels != -1]))}␣
  ↪(noise: {np.sum(dbscan_labels == -1)})")

# Silhouette scores
kmeans_sil = silhouette_score(features, kmeans_labels)

print(f"\nSilhouette scores:")
print(f"  KMeans: {kmeans_sil:.4f}")
```

```
/usr/local/lib/python3.12/dist-packages/diffusers/configuration_utils.py:141:
FutureWarning: Accessing config attribute `__iter__` directly via
'DDPMScheduler' object attribute is deprecated. Please access '__iter__' over
'DDPMScheduler's config object instead, e.g. 'scheduler.config.__iter__'.
  deprecate("direct config name access", "1.0.0", deprecation_message,
standard_warn=False)
/usr/local/lib/python3.12/dist-
packages/debugpy/_vendored/pydevd/_pydevd_bundle/pydevd_safe_repr.py:128:
FutureWarning: Accessing config attribute `__iter__` directly via
'UNet2DConditionModel' object attribute is deprecated. Please access '__iter__'
over 'UNet2DConditionModel's config object instead, e.g. 'unet.config.__iter__'.
```

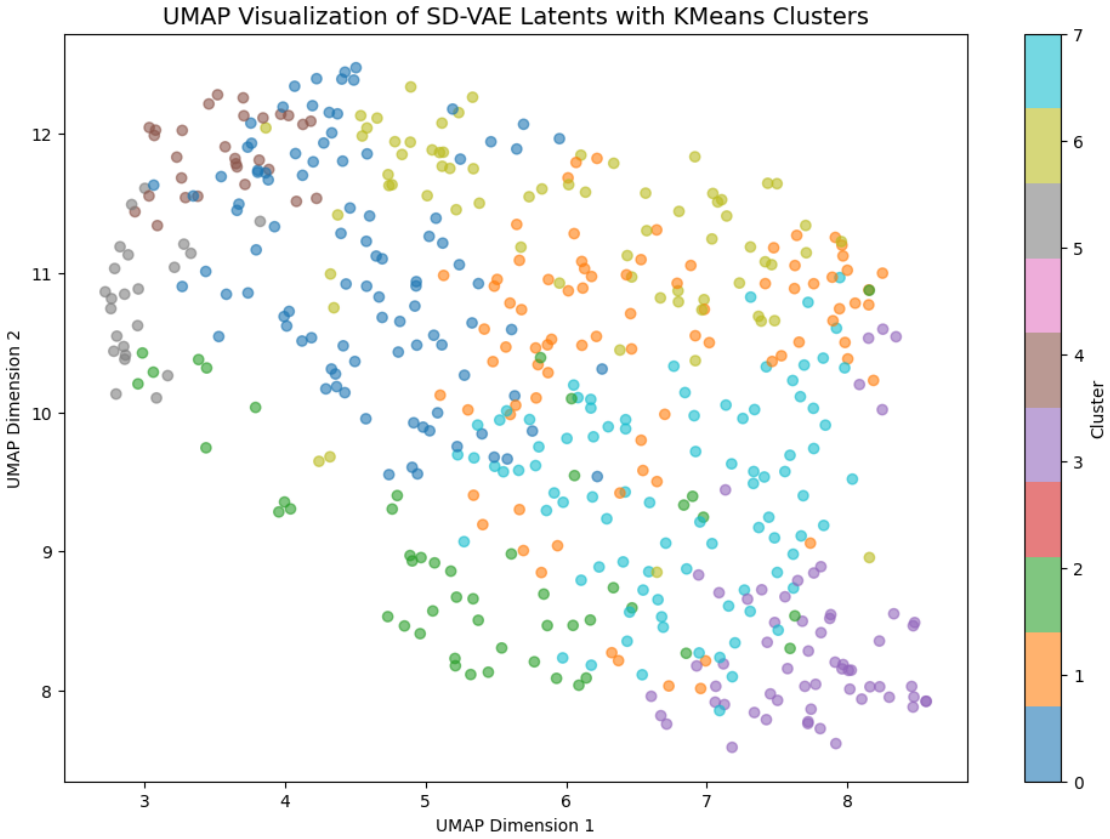
```
  if not hasattr(obj, "__iter__"):
/usr/local/lib/python3.12/dist-
packages/debugpy/_vendored/pydevd/_pydevd_bundle/pydevd_safe_repr.py:128:
FutureWarning: Accessing config attribute `__iter__` directly via
'AutoencoderKL' object attribute is deprecated. Please access '__iter__' over
'AutoencoderKL's config object instead, e.g. 'unet.config.__iter__'.
  if not hasattr(obj, "__iter__"):

KMeans clusters: 8
DBSCAN clusters: 0 (noise: 1450)

Silhouette scores:
  KMeans: 0.0143
```

[ ]:
```
# t-SNE visualization
tsne = TSNE(n_components=2, random_state=42, perplexity=30)
features_2d = features[:500]
labels_2d = kmeans_labels[:500]
tsne_embedding = tsne.fit_transform(features_2d)

plt.figure(figsize=(12, 8))
scatter = plt.scatter(tsne_embedding[:, 0], tsne_embedding[:, 1], c=labels_2d,␣
 ↪cmap='tab10', alpha=0.6)
plt.colorbar(scatter, label='Cluster')
plt.title('t-SNE Visualization of SD-VAE Latents with KMeans Clusters',␣
 ↪fontsize=14)
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.show()
```



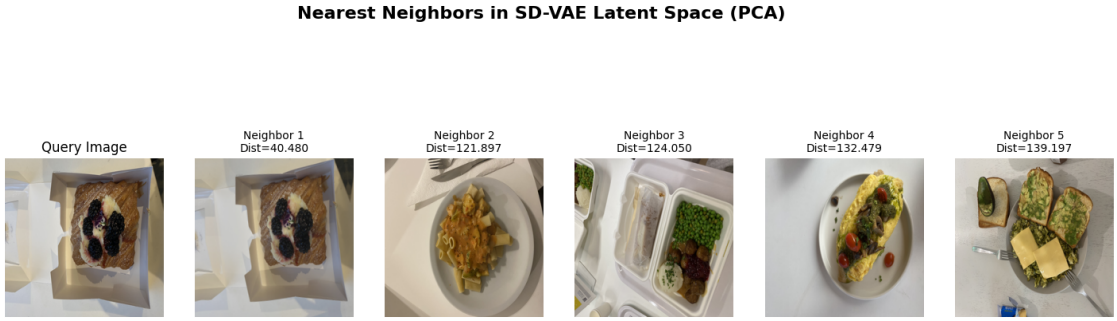
[ ]:
```
# UMAP visualization
reducer = umap.UMAP(n_components=2, random_state=42, n_neighbors=15, min_dist=0.
↪1)
umap_embedding = reducer.fit_transform(features[:500])

plt.figure(figsize=(12, 8))
scatter = plt.scatter(umap_embedding[:, 0], umap_embedding[:, 1], c=labels_2d,␣
↪cmap='tab10', alpha=0.6)
plt.colorbar(scatter, label='Cluster')
plt.title('UMAP Visualization of SD-VAE Latents with KMeans Clusters',␣
↪fontsize=14)
plt.xlabel('UMAP Dimension 1')
plt.ylabel('UMAP Dimension 2')
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/umap/umap_.py:1952: UserWarning: n_jobs
value 1 overridden to 1 by setting random_state. Use no seed for parallelism.
  warn(
```



[ ]:
```python
# Nearest-Neighbor Search in Latent Space (PCA)
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt

# Use PCA latents
features = X_train_latent_pca    # shape: (N, 50)
images = X_train                 # shape: (N, 256, 256, 3)

# Fit nearest-neighbor model
nn_model = NearestNeighbors(n_neighbors=6, metric='euclidean')
nn_model.fit(features)

query_index = np.random.randint(0, len(features))
query_latent = features[query_index].reshape(1, -1)

# Find nearest neighbors
distances, indices = nn_model.kneighbors(query_latent)

plt.figure(figsize=(18, 6))
```

```
plt.suptitle("Nearest Neighbors in SD-VAE Latent Space (PCA)", fontsize=16,␣
 ↪fontweight='bold')

# Show query image
plt.subplot(1, 6, 1)
plt.imshow(images[query_index])
plt.title("Query Image", fontsize=12)
plt.axis("off")

# Show 5 nearest neighbors
for i, idx in enumerate(indices[0][1:]):
    plt.subplot(1, 6, i + 2)
    plt.imshow(images[idx])
    plt.title(f"Neighbor {i+1}\nDist={distances[0][i+1]:.3f}", fontsize=10)
    plt.axis("off")

plt.show()
```



3 Data Augmentation

[ ]:
```
def augment_image(image, seed=None):
    """
    Apply random augmentations to a single image.

    Parameters:
    - image: NumPy array of shape (H, W, C) with values in [0, 1]
    - seed: Random seed for reproducibility

    Returns:
    - Augmented image array
    """
    if seed is not None:
        np.random.seed(seed)
        random.seed(seed)
```

```
    aug_image = image.copy()

    # 1. Random horizontal flip (50% probability)
    if np.random.random() > 0.5:
        aug_image = np.fliplr(aug_image)

    # 2. Random rotation (±15 degrees)
    angle = np.random.uniform(-15, 15)
    aug_image = rotate(aug_image, angle, axes=(0, 1), reshape=False,␣
 ↪mode='reflect', order=1)

    # 3. Random brightness adjustment (±20%)
    brightness_factor = np.random.uniform(0.8, 1.2)
    aug_image = np.clip(aug_image * brightness_factor, 0, 1)

    # 4. Random contrast adjustment (±20%)
    contrast_factor = np.random.uniform(0.8, 1.2)
    mean = aug_image.mean()
    aug_image = np.clip((aug_image - mean) * contrast_factor + mean, 0, 1)

    return aug_image

# Test augmentation on a sample image
sample_idx = 0
original = X_proc[sample_idx]
augmented = augment_image(original)

fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(original)
axes[0].set_title('Original Image', fontsize=12)
axes[0].axis('off')

axes[1].imshow(augmented)
axes[1].set_title('Augmented Image', fontsize=12)
axes[1].axis('off')

plt.tight_layout()
plt.show()
```

[ ]:
```
# Create augmented dataset
# 2 augmented versions per original image (3x total dataset size)

n_augmentations_per_image = 2
augmented_images = []

print(f"Creating {n_augmentations_per_image} augmented versions per image...")
print(f"This will create {n_original * n_augmentations_per_image} additional␣
 ↪images")

for i in range(n_original):
    for aug_idx in range(n_augmentations_per_image):
        # Use a unique seed for each augmentation: image_index *␣
 ↪n_augmentations + aug_index
        seed = i * n_augmentations_per_image + aug_idx
        aug_img = augment_image(X_proc[i], seed=seed)
        augmented_images.append(aug_img)

    if (i + 1) % 50 == 0:
        print(f"  Augmented {i + 1}/{n_original} images...")

# Stack augmented images
X_aug = np.stack(augmented_images, axis=0)

print(f"\n Created {len(augmented_images)} augmented images")
print(f"  X_aug shape: {X_aug.shape}, dtype: {X_aug.dtype}")
```

```
Creating 2 augmented versions per image…
```

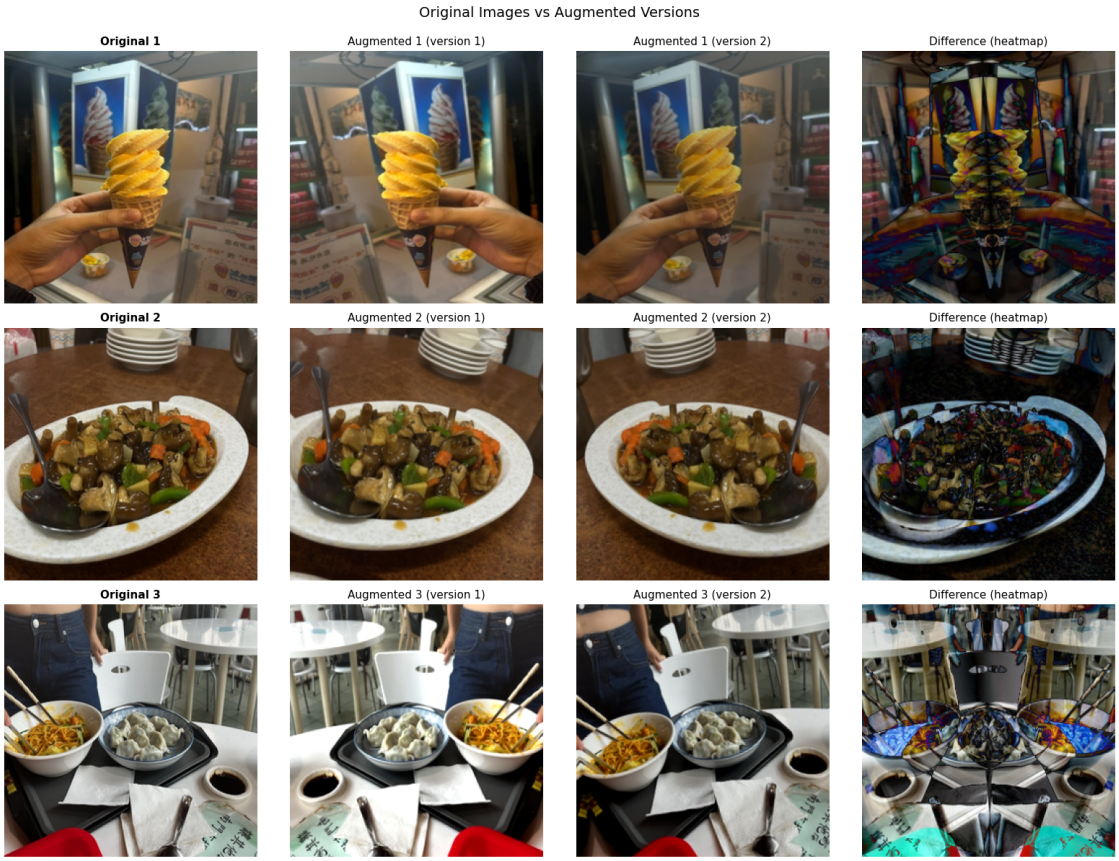
```
This will create 1138 additional images
  Augmented 50/569 images…
  Augmented 100/569 images…
  Augmented 150/569 images…
  Augmented 200/569 images…
  Augmented 250/569 images…
  Augmented 300/569 images…
  Augmented 350/569 images…
  Augmented 400/569 images…
  Augmented 450/569 images…
  Augmented 500/569 images…
  Augmented 550/569 images…

 Created 1138 augmented images
 X_aug shape: (1138, 256, 256, 3), dtype: float32
```

[ ]:
```python
# Combine original and augmented images
X_combined = np.concatenate([X_proc, X_aug], axis=0)

print(f"Final dataset shape: {X_combined.shape}")
print(f"  - Original images: {n_original}")
print(f"  - Augmented images: {X_aug.shape[0]}")
print(f"  - Total images: {X_combined.shape[0]}")
print(f"  - Image dimensions: {X_combined.shape[1]}×{X_combined.shape[2]}")
print(f"  - Channels: {X_combined.shape[3]}")
print(f"  - Data type: {X_combined.dtype}")
print(f"  - Value range: [{X_combined.min():.3f}, {X_combined.max():.3f}]")
```

```
Final dataset shape: (1707, 256, 256, 3)
  - Original images: 569
  - Augmented images: 1138
  - Total images: 1707
  - Image dimensions: 256×256
  - Channels: 3
  - Data type: float32
  - Value range: [0.000, 1.000]
```

[ ]:
```python
# Visualize original vs augmented images side by side
fig, axes = plt.subplots(3, 4, figsize=(16, 12))

for i in range(3):
    # Original image
    axes[i, 0].imshow(X_proc[i])
    axes[i, 0].set_title(f'Original {i+1}', fontsize=11, fontweight='bold')
    axes[i, 0].axis('off')

    # First augmentation
```

```
    axes[i, 1].imshow(X_aug[i * n_augmentations_per_image])
    axes[i, 1].set_title(f'Augmented {i+1} (version 1)', fontsize=11)
    axes[i, 1].axis('off')

    # Second augmentation
    axes[i, 2].imshow(X_aug[i * n_augmentations_per_image + 1])
    axes[i, 2].set_title(f'Augmented {i+1} (version 2)', fontsize=11)
    axes[i, 2].axis('off')

    # Show difference (for visualization)
    diff = np.abs(X_proc[i] - X_aug[i * n_augmentations_per_image])
    axes[i, 3].imshow(diff, cmap='hot')
    axes[i, 3].set_title(f'Difference (heatmap)', fontsize=11)
    axes[i, 3].axis('off')

plt.suptitle('Original Images vs Augmented Versions', fontsize=14, y=0.995)
plt.tight_layout()
plt.show()
```



4 Splitting the data

[ ]:
```
# Split the data into training and validation sets
# We'll use 85% train, 15% validation

train_size = 0.85
random_state = 42

X_train, X_val = train_test_split(
    X_combined,
    train_size=train_size,
    random_state=random_state,
    shuffle=True
)

print(f"Dataset Split:")
print(f"  Training set:   {X_train.shape[0]} images ({X_train.shape[0]/
 ↪len(X_combined)*100:.1f}%)")
print(f"  Validation set: {X_val.shape[0]} images ({X_val.shape[0]/
 ↪len(X_combined)*100:.1f}%)")
print(f"  Total:          {len(X_combined)} images")
print(f"\nShapes:")
print(f"  X_train: {X_train.shape}")
print(f"  X_val:   {X_val.shape}")
```

```
Dataset Split:
  Training set:   1450 images (84.9%)
  Validation set: 257 images (15.1%)
  Total:          1707 images

Shapes:
  X_train: (1450, 256, 256, 3)
  X_val:   (257, 256, 256, 3)
```

5 Setting up the model

5.1 Stable Diffusion VAE Setup

[ ]:
```
# Load Stable Diffusion VAE encoder/decoder

device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

vae = AutoencoderKL.from_pretrained(
    "runwayml/stable-diffusion-v1-5",
    subfolder="vae"
)
```

```
vae = vae.to(device)
vae.eval()  # Set to evaluation mode
print(f"  Latent channels: {vae.config.latent_channels}")
```

```
Using device: cuda
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
```

```
config.json:   0%|          | 0.00/547 [00:00<?, ?B/s]
```

```
vae/diffusion_pytorch_model.safetensors:   0%|          | 0.00/335M [00:00<?, ?B/
↪s]
```

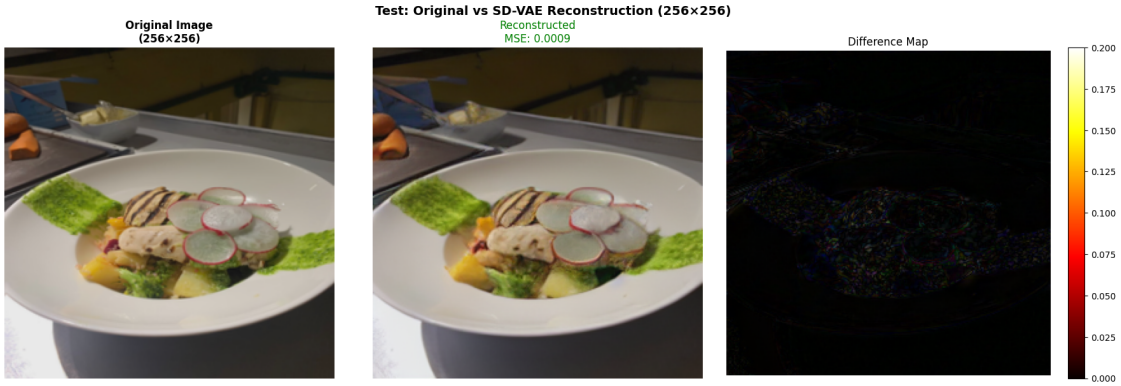
```
  Latent channels: 4
```

[ ]:
```
# Test encoding/decoding with correct 256×256 resolution
rand_ind = np.random.randint(0, len(X_train))
test_image = X_train[rand_ind:rand_ind+1]  # Shape: (1, 256, 256, 3) in [0, 1]

# Convert to tensor and normalize to [-1, 1] for encoding
test_tensor = torch.from_numpy(test_image).permute(0, 3, 1, 2).float()
test_tensor = test_tensor * 2.0 - 1.0  # [0, 1] → [-1, 1]
test_tensor = test_tensor.to(device)

# Encode
with torch.no_grad():
    encoded = vae.encode(test_tensor)
    latent = encoded.latent_dist.sample()
    print(f"   Encoded. Latent shape: {latent.shape}")
    print(f"   Expected: (1, 4, 32, 32) for 256×256 input")
    print(f"   Latent stats: mean={latent.mean():.4f}, std={latent.std():.4f}")

# Decode (no scaling needed)
with torch.no_grad():
    decoded = vae.decode(latent).sample
    decoded_np = decoded.cpu().numpy()
    decoded_np = (decoded_np + 1) / 2
    decoded_np = np.clip(decoded_np, 0, 1)
    decoded_np = np.transpose(decoded_np, (0, 2, 3, 1))
    mse = np.mean((test_image - decoded_np) ** 2)
```

```
    print(f"   Decoded. Output shape: {decoded_np.shape}")
    print(f"   MSE: {mse:.6f}")
```

```
Encoded. Latent shape: torch.Size([1, 4, 32, 32])
Expected: (1, 4, 32, 32) for 256×256 input
Latent stats: mean=0.9247, std=4.6344
Decoded. Output shape: (1, 256, 256, 3)
MSE: 0.000909
```

[ ]:
```
# Original vs Reconstructed

if 'decoded_np' in locals():
    fig, axes = plt.subplots(1, 3, figsize=(18, 6))

    # Original
    axes[0].imshow(test_image[0])
    axes[0].set_title('Original Image\n(256×256)', fontsize=12,␣
 ↪fontweight='bold')
    axes[0].axis('off')

    # Reconstructed
    axes[1].imshow(decoded_np[0])
    axes[1].set_title(f'Reconstructed\nMSE: {mse:.4f}', fontsize=12,
                      color='green' if mse < 0.01 else 'orange' if mse < 0.05␣
 ↪else 'red')
    axes[1].axis('off')

    # Difference
    diff = np.abs(test_image[0] - decoded_np[0])
    im = axes[2].imshow(diff, cmap='hot', vmin=0, vmax=0.2)
    axes[2].set_title('Difference Map', fontsize=12)
    axes[2].axis('off')
    plt.colorbar(im, ax=axes[2], fraction=0.046)

    plt.suptitle('Test: Original vs SD-VAE Reconstruction (256×256)',
                 fontsize=14, y=1.0, fontweight='bold')
    plt.tight_layout()
    plt.show()
```



6 Running the model

[ ]:
```python
def encode_images_to_latent_streaming(images, vae, device, batch_size=8,␣
↪save_dir="latents"):
    os.makedirs(save_dir, exist_ok=True)
    n = len(images)
    latent_paths = []

    for i in range(0, n, batch_size):
        batch = images[i:i+batch_size]

        batch_tensor = torch.from_numpy(batch).permute(0,3,1,2).float()
        batch_tensor = batch_tensor * 2 - 1
        batch_tensor = batch_tensor.to(device)

        with torch.inference_mode():
            latents = vae.encode(batch_tensor).latent_dist.sample()

        lat_np = latents.cpu().numpy().astype("float16")

        path = f"{save_dir}/batch_{i}.npy"
        np.save(path, lat_np)
        latent_paths.append(path)

        del batch, batch_tensor, latents, lat_np
        gc.collect()

        print(f"Encoded {min(i+batch_size, n)}/{n}")

    return latent_paths
```

[ ]:
```python
def decode_latents_streaming(latent_paths, vae, device, save_dir="recon"):
    os.makedirs(save_dir, exist_ok=True)
    out_paths = []
    idx = 0

    for path in latent_paths:
        lat = np.load(path)
        lat_tensor = torch.from_numpy(lat).float().to(device)

        with torch.inference_mode():
            dec = vae.decode(lat_tensor).sample

        dec = (dec + 1) / 2
        dec = dec.clamp(0,1)
        dec_np = dec.permute(0,2,3,1).cpu().numpy()

        for j in range(len(dec_np)):
            out_path = f"{save_dir}/img_{idx}.npy"
            np.save(out_path, dec_np[j])
            out_paths.append(out_path)
            idx += 1

        del lat, lat_tensor, dec, dec_np
        gc.collect()

        print(f"Decoded {idx} images")

    return out_paths
```

[ ]:
```python
train_latent_paths = encode_images_to_latent_streaming(
    X_train, vae, device, batch_size=8, save_dir="latents_train"
)
```

```
Encoded 8/1450
Encoded 16/1450
Encoded 24/1450
Encoded 32/1450
Encoded 40/1450
Encoded 48/1450
Encoded 56/1450
Encoded 64/1450
Encoded 72/1450
Encoded 80/1450
Encoded 88/1450
Encoded 96/1450
Encoded 104/1450
Encoded 112/1450
```

```
Encoded 120/1450
Encoded 128/1450
Encoded 136/1450
Encoded 144/1450
Encoded 152/1450
Encoded 160/1450
Encoded 168/1450
Encoded 176/1450
Encoded 184/1450
Encoded 192/1450
Encoded 200/1450
Encoded 208/1450
Encoded 216/1450
Encoded 224/1450
Encoded 232/1450
Encoded 240/1450
Encoded 248/1450
Encoded 256/1450
Encoded 264/1450
Encoded 272/1450
Encoded 280/1450
Encoded 288/1450
Encoded 296/1450
Encoded 304/1450
Encoded 312/1450
Encoded 320/1450
Encoded 328/1450
Encoded 336/1450
Encoded 344/1450
Encoded 352/1450
Encoded 360/1450
Encoded 368/1450
Encoded 376/1450
Encoded 384/1450
Encoded 392/1450
Encoded 400/1450
Encoded 408/1450
Encoded 416/1450
Encoded 424/1450
Encoded 432/1450
Encoded 440/1450
Encoded 448/1450
Encoded 456/1450
Encoded 464/1450
Encoded 472/1450
Encoded 480/1450
Encoded 488/1450
Encoded 496/1450
```

```
Encoded 504/1450
Encoded 512/1450
Encoded 520/1450
Encoded 528/1450
Encoded 536/1450
Encoded 544/1450
Encoded 552/1450
Encoded 560/1450
Encoded 568/1450
Encoded 576/1450
Encoded 584/1450
Encoded 592/1450
Encoded 600/1450
Encoded 608/1450
Encoded 616/1450
Encoded 624/1450
Encoded 632/1450
Encoded 640/1450
Encoded 648/1450
Encoded 656/1450
Encoded 664/1450
Encoded 672/1450
Encoded 680/1450
Encoded 688/1450
Encoded 696/1450
Encoded 704/1450
Encoded 712/1450
Encoded 720/1450
Encoded 728/1450
Encoded 736/1450
Encoded 744/1450
Encoded 752/1450
Encoded 760/1450
Encoded 768/1450
Encoded 776/1450
Encoded 784/1450
Encoded 792/1450
Encoded 800/1450
Encoded 808/1450
Encoded 816/1450
Encoded 824/1450
Encoded 832/1450
Encoded 840/1450
Encoded 848/1450
Encoded 856/1450
Encoded 864/1450
Encoded 872/1450
Encoded 880/1450
```

```
Encoded 888/1450
Encoded 896/1450
Encoded 904/1450
Encoded 912/1450
Encoded 920/1450
Encoded 928/1450
Encoded 936/1450
Encoded 944/1450
Encoded 952/1450
Encoded 960/1450
Encoded 968/1450
Encoded 976/1450
Encoded 984/1450
Encoded 992/1450
Encoded 1000/1450
Encoded 1008/1450
Encoded 1016/1450
Encoded 1024/1450
Encoded 1032/1450
Encoded 1040/1450
Encoded 1048/1450
Encoded 1056/1450
Encoded 1064/1450
Encoded 1072/1450
Encoded 1080/1450
Encoded 1088/1450
Encoded 1096/1450
Encoded 1104/1450
Encoded 1112/1450
Encoded 1120/1450
Encoded 1128/1450
Encoded 1136/1450
Encoded 1144/1450
Encoded 1152/1450
Encoded 1160/1450
Encoded 1168/1450
Encoded 1176/1450
Encoded 1184/1450
Encoded 1192/1450
Encoded 1200/1450
Encoded 1208/1450
Encoded 1216/1450
Encoded 1224/1450
Encoded 1232/1450
Encoded 1240/1450
Encoded 1248/1450
Encoded 1256/1450
Encoded 1264/1450
```

```
Encoded 1272/1450
Encoded 1280/1450
Encoded 1288/1450
Encoded 1296/1450
Encoded 1304/1450
Encoded 1312/1450
Encoded 1320/1450
Encoded 1328/1450
Encoded 1336/1450
Encoded 1344/1450
Encoded 1352/1450
Encoded 1360/1450
Encoded 1368/1450
Encoded 1376/1450
Encoded 1384/1450
Encoded 1392/1450
Encoded 1400/1450
Encoded 1408/1450
Encoded 1416/1450
Encoded 1424/1450
Encoded 1432/1450
Encoded 1440/1450
Encoded 1448/1450
Encoded 1450/1450
```

[ ]:
```
val_latent_paths = encode_images_to_latent_streaming(
    X_val, vae, device, batch_size=8, save_dir="latents_val"
)
```

```
Encoded 8/257
Encoded 16/257
Encoded 24/257
Encoded 32/257
Encoded 40/257
Encoded 48/257
Encoded 56/257
Encoded 64/257
Encoded 72/257
Encoded 80/257
Encoded 88/257
Encoded 96/257
Encoded 104/257
Encoded 112/257
Encoded 120/257
Encoded 128/257
Encoded 136/257
Encoded 144/257
Encoded 152/257
```

```
Encoded 160/257
Encoded 168/257
Encoded 176/257
Encoded 184/257
Encoded 192/257
Encoded 200/257
Encoded 208/257
Encoded 216/257
Encoded 224/257
Encoded 232/257
Encoded 240/257
Encoded 248/257
Encoded 256/257
Encoded 257/257
```

[ ]:
```
subset_train = train_latent_paths[:10]   # choose first 10 latent batches
decoded_train = decode_latents_streaming(subset_train, vae, device,␣
 ↪save_dir="recon_train")
```

```
Decoded 8 images
Decoded 16 images
Decoded 24 images
Decoded 32 images
Decoded 40 images
Decoded 48 images
Decoded 56 images
Decoded 64 images
Decoded 72 images
Decoded 80 images
```

[ ]:
```
subset_val = val_latent_paths[:10]      # choose first 10 latent batches
decoded_val = decode_latents_streaming(subset_val, vae, device,␣
 ↪save_dir="recon_val")
```

```
Decoded 8 images
Decoded 16 images
Decoded 24 images
Decoded 32 images
Decoded 40 images
Decoded 48 images
Decoded 56 images
Decoded 64 images
Decoded 72 images
Decoded 80 images
```

[ ]:
```
batch_size = 8

# Choose any batch to evaluate (first batch is simplest)
```

```
batch_path = train_latent_paths[0]
print("Selected latent file:", batch_path)

# Extract batch ID from filename, e.g. "batch_000.npy" → 0
batch_id = int(batch_path.split("_")[-1].split(".")[0])
print("Batch ID:", batch_id)

# Compute original image indices corresponding to this batch
start = batch_id * batch_size
end = start + batch_size
orig_indices = list(range(start, end))
print("Original image indices:", orig_indices)

# Load the latent batch
latents = np.load(batch_path)
print("Latent batch shape:", latents.shape)

# Decode this batch USING STREAMING. This returns a list of paths to individual␣
 ↪decoded images.
decoded_paths_for_batch = decode_latents_streaming([batch_path], vae, device,␣
 ↪save_dir="eval_temp")

# Load the individual decoded images from their paths and stack them into a␣
 ↪single NumPy array
decoded_batch = []
for p in decoded_paths_for_batch:
    decoded_batch.append(np.load(p))
decoded_batch = np.stack(decoded_batch, axis=0)

print("Decoded batch shape:", decoded_batch.shape)
```

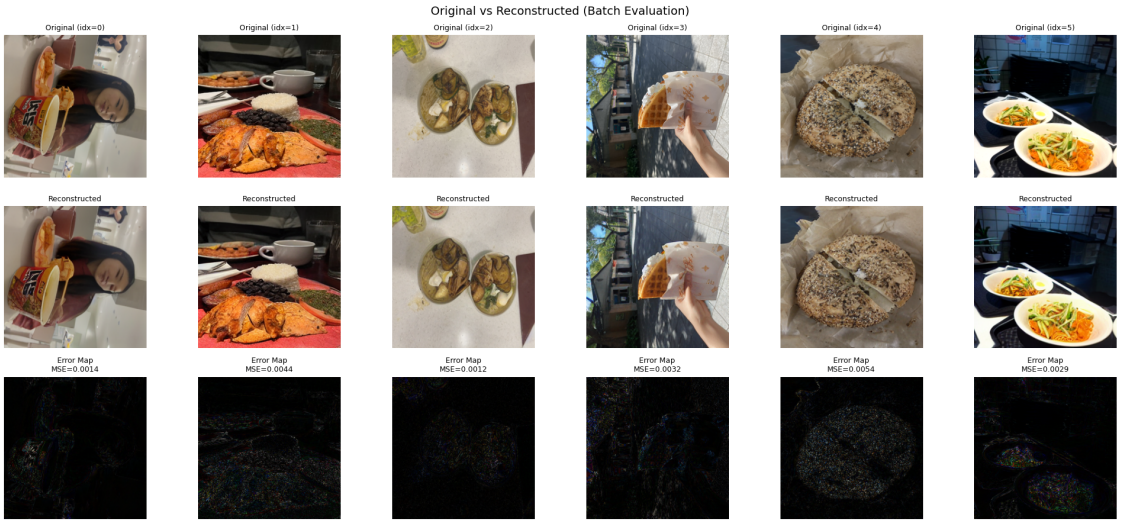
```
Selected latent file: latents_train/batch_0.npy
Batch ID: 0
Original image indices: [0, 1, 2, 3, 4, 5, 6, 7]
Latent batch shape: (8, 4, 32, 32)
Decoded 8 images
Decoded batch shape: (8, 256, 256, 3)
```

[ ]:
```
mse_list = []

for i, orig_idx in enumerate(orig_indices):
    orig_img = X_train[orig_idx]
    recon_img = decoded_batch[i]

    mse = np.mean((orig_img - recon_img) ** 2)
    mse_list.append(mse)
```

```
mse_batch = np.mean(mse_list)

print("\nReconstruction Quality for Batch:")
print(f"  Mean MSE:  {mse_batch:.6f}")
print(f"  RMSE:      {np.sqrt(mse_batch):.6f}")
```

```
Reconstruction Quality for Batch:
  Mean MSE:  0.003308
  RMSE:      0.057511
```

[ ]:
```
n_compare = min(6, len(orig_indices))

fig, axes = plt.subplots(3, n_compare, figsize=(20, 9))

for i in range(n_compare):
    orig_idx = orig_indices[i]
    orig_img = X_train[orig_idx]
    recon_img = decoded_batch[i]

    # Row 1 - Original image
    axes[0, i].imshow(orig_img)
    axes[0, i].set_title(f"Original (idx={orig_idx})", fontsize=9)
    axes[0, i].axis("off")

    # Row 2 - Reconstructed
    axes[1, i].imshow(recon_img)
    axes[1, i].set_title("Reconstructed", fontsize=9)
    axes[1, i].axis("off")

    # Row 3 - Difference heatmap
    diff = np.abs(orig_img - recon_img)
    mse = np.mean(diff**2)

    im = axes[2, i].imshow(diff, cmap="hot", vmin=0, vmax=0.2)
    axes[2, i].set_title(f"Error Map\nMSE={mse:.4f}", fontsize=9)
    axes[2, i].axis("off")

plt.suptitle("Original vs Reconstructed (Batch Evaluation)", fontsize=14)
plt.tight_layout()
plt.show()
```



7 Latent Diffusion Model: Generative Training and Sampling

7.1 Latent DDPM with a Minimal Time-Conditioned CNN (Baseline Model)

[ ]:
```python
class TinyUNet_v2(nn.Module):
    def __init__(self, channels=4, hidden=64, tdim=64):
        super().__init__()

        # Timestep embedding
        self.time_mlp = nn.Sequential(
            nn.Linear(1, tdim),
            nn.ReLU(),
            nn.Linear(tdim, tdim),
            nn.ReLU()
        )

        self.conv1 = nn.Conv2d(channels + tdim, hidden, 3, padding=1)
        self.conv2 = nn.Conv2d(hidden, hidden, 3, padding=1)
        self.conv3 = nn.Conv2d(hidden, channels, 3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x, t):
        t = t.view(-1, 1)
        t_embed = self.time_mlp(t).unsqueeze(-1).unsqueeze(-1)
        t_embed = t_embed.expand(-1, -1, x.shape[2], x.shape[3])

        x = torch.cat([x, t_embed], dim=1)
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
```

```
        x = self.conv3(x)
        return x

diffusion_model = TinyUNet_v2().to(device)
print("Loaded TinyUNet v2")
```

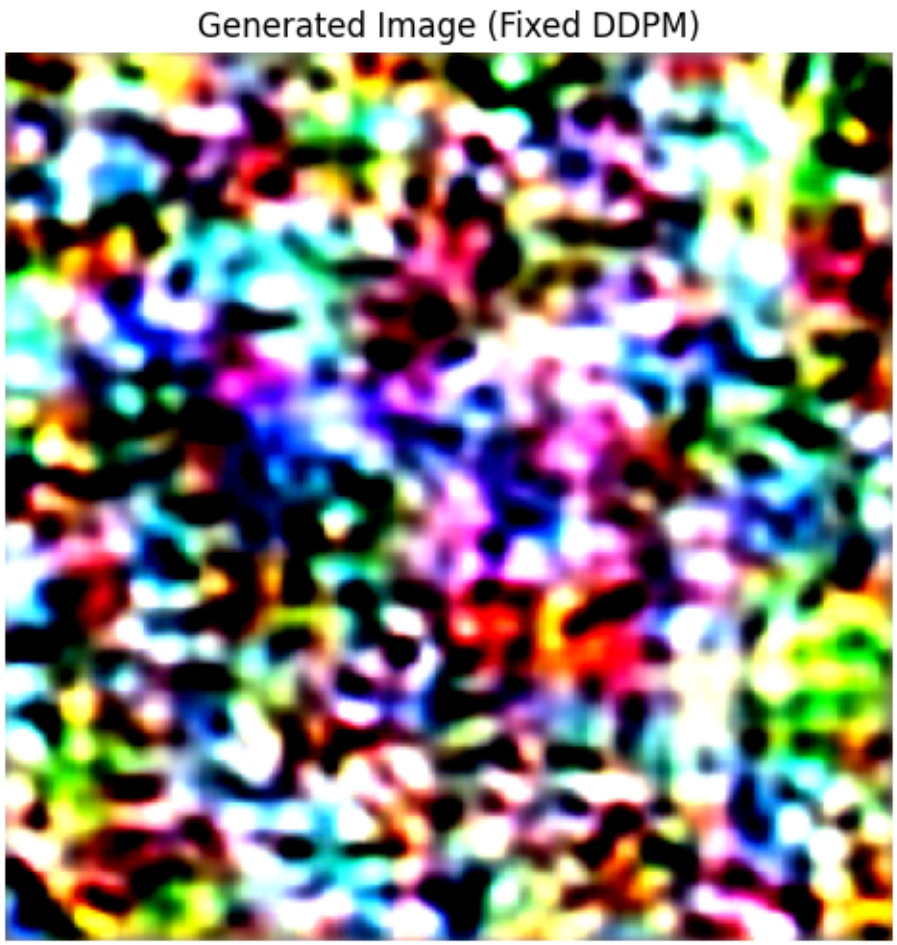
```
Loaded TinyUNet v2
```

[ ]:
```
T = 1000

beta = torch.linspace(1e-4, 0.02, T).to(device)
alpha = 1 - beta
alpha_bar = torch.cumprod(alpha, dim=0)

optimizer = torch.optim.Adam(diffusion_model.parameters(), lr=1e-4)

latents_tensor = torch.from_numpy(X_train_latent[:500]).float().to(device)

steps = 1500
batch_size = 16

print("Training DDPM...")

for step in range(steps):
    idx = torch.randint(0, len(latents_tensor), (batch_size,))
    x0 = latents_tensor[idx]

    t = torch.randint(1, T, (batch_size,), device=device)
    noise = torch.randn_like(x0)

    a_bar = alpha_bar[t].view(-1, 1, 1, 1)
    x_t = torch.sqrt(a_bar) * x0 + torch.sqrt(1 - a_bar) * noise

    pred_noise = diffusion_model(x_t, t.float())

    loss = nn.functional.mse_loss(pred_noise, noise)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if step % 200 == 0:
        print(step, loss.item())
```

```
Training DDPM…
0 93.811279296875
200 0.9896387457847595
400 0.9691115617752075
600 0.9378674030303955
```

```
800 0.9131677150726318
1000 0.8981828689575195
1200 0.8686449527740479
1400 0.8666119575500488
```

[ ]:
```python
diffusion_model.eval()

with torch.no_grad():
    x = torch.randn(1, 4, 32, 32).to(device)

    for t in reversed(range(1, T)):
        beta_t = beta[t]
        alpha_t = alpha[t]
        alpha_bar_t = alpha_bar[t]

        t_tensor = torch.tensor([t], device=device).float()
        pred_noise = diffusion_model(x, t_tensor)

        x = (1 / torch.sqrt(alpha_t)) * (
            x - ((1 - alpha_t) / torch.sqrt(1 - alpha_bar_t)) * pred_noise
        )

        if t > 1:
            x = x + torch.sqrt(beta_t) * torch.randn_like(x)

# Decode
with torch.no_grad():
    decoded = vae.decode(x).sample.cpu().detach().numpy()[0]

decoded = (decoded + 1) / 2
decoded = decoded.clip(0, 1)
decoded = decoded.transpose(1, 2, 0)

plt.figure(figsize=(6,6))
plt.imshow(decoded)
plt.title("Generated Image (Fixed DDPM)")
plt.axis("off")
plt.show()
```



8 LoRA Fine-Tuning of a Pretrained Stable Diffusion Model

[ ]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"

# Load SD components
vae = AutoencoderKL.from_pretrained("runwayml/stable-diffusion-v1-5",␣
 ↪subfolder="vae").to(device)
unet = UNet2DConditionModel.from_pretrained("runwayml/stable-diffusion-v1-5",␣
 ↪subfolder="unet").to(device)

scheduler = DDPMScheduler.from_pretrained("runwayml/stable-diffusion-v1-5",␣
 ↪subfolder="scheduler")

# --- Dataset loader ---
class FoodDataset(Dataset):
    def __init__(self, folder):
```

```
        # Use glob.glob to get files matching the pattern
        self.files = glob.glob(folder)
        self.transform = transforms.Compose([
            transforms.Resize((512,512)),
            transforms.ToTensor(),
            transforms.Normalize([0.5], [0.5])
        ])

    def __len__(self):
        return len(self.files)

    def __getitem__(self, idx):
        img = Image.open(self.files[idx]).convert("RGB")
        img = self.transform(img)
        return img

dataset = FoodDataset("/content/drive/MyDrive/food_images/*")
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)
```

```
config.json:   0%|          | 0.00/743 [00:00<?, ?B/s]
```

```
unet/diffusion_pytorch_model.safetensors:   0%|          | 0.00/3.44G [00:00<?, ?
↪B/s]
```

```
scheduler_config.json:   0%|          | 0.00/308 [00:00<?, ?B/s]
```

[ ]:
```
# Increase recursion limit for deep models like UNet
sys.setrecursionlimit(4000)

# Custom LoRALinear module to replace nn.Linear
class LoRALinear(nn.Module):
    def __init__(self, linear_layer, rank=8, lora_alpha=1.0):
        super().__init__()
        self.linear_layer = linear_layer # The original nn.Linear
        self.in_features = linear_layer.in_features
        self.out_features = linear_layer.out_features
        self.rank = rank
        self.lora_alpha = lora_alpha
        self.scaling = self.lora_alpha / self.rank

        self.lora_down = nn.Linear(self.in_features, rank, bias=False)
        self.lora_up = nn.Linear(rank, self.out_features, bias=False)

        # Initialize lora_up weights to zero and lora_down with Kaiming uniform
        nn.init.zeros_(self.lora_up.weight)
        nn.init.kaiming_uniform_(self.lora_down.weight, a=math.sqrt(5))

        # Freeze the original linear layer's parameters
```

```
        for param in self.linear_layer.parameters():
            param.requires_grad = False

    def forward(self, x):
        # Original forward pass
        original_output = self.linear_layer(x)

        # LoRA additive component
        lora_output = self.lora_up(self.lora_down(x)) * self.scaling
        return original_output + lora_output

# Collect (parent_module, child_name, original_child_module) to replace
modules_to_replace = []
for name, module in unet.named_modules():
    for child_name, child_module in module.named_children():
        if isinstance(child_module, nn.Linear):
            modules_to_replace.append((module, child_name, child_module))

# Perform replacement
for parent_module, child_name, original_child_module in modules_to_replace:
    new_lora_layer = LoRALinear(original_child_module, rank=8)
    setattr(parent_module, child_name, new_lora_layer)

# IMPORTANT: Move the entire UNet to the device again to ensure new LoRA layers␣
 ↪are on GPU
unet.to(device)

# Freeze everything except LoRA weights
for name, param in unet.named_parameters():
    # Only parameters within LoRALinear (lora_down and lora_up) should be␣
 ↪trainable
    if "lora_down" in name or "lora_up" in name:
        param.requires_grad = True
    else:
        param.requires_grad = False # Ensures original_linear_layer params␣
 ↪remain frozen
```

[ ]:
```
optimizer = optim.Adam(filter(lambda p: p.requires_grad, unet.parameters()),␣
 ↪lr=1e-4)

vae.eval()
unet.train()

num_steps = 1500  # can increase to 5000 for better quality
step = 0

for epoch in range(10):
```

```
    for batch in dataloader:
        step += 1
        if step > num_steps:
            break

        batch = batch.to(device)

        # Encode images to SD latent space
        with torch.no_grad():
            latents = vae.encode(batch).latent_dist.sample() * 0.18215

        # Sample random timestep
        noise = torch.randn_like(latents)
        timesteps = torch.randint(0, scheduler.num_train_timesteps, (latents.
↪size(0),), device=device)

        noisy_latents = scheduler.add_noise(latents, noise, timesteps)

        batch_size_current = noisy_latents.shape[0]
        dummy_encoder_hidden_states = torch.zeros(
            (batch_size_current, 77, 768), # 77 tokens, 768 features (standard␣
↪for SD1.5 CLIP)
            dtype=noisy_latents.dtype,
            device=device
        )
        noise_pred = unet(noisy_latents, timesteps,␣
↪encoder_hidden_states=dummy_encoder_hidden_states).sample

        loss = torch.nn.functional.mse_loss(noise_pred, noise)

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

        if step % 100 == 0:
            print(f"Step {step} / {num_steps}, Loss = {loss.item():.6f}")

    if step > num_steps:
        break

print("LoRA fine-tuning complete!")
```

```
/usr/local/lib/python3.12/dist-packages/diffusers/configuration_utils.py:141:
FutureWarning: Accessing config attribute `num_train_timesteps` directly via
'DDPMScheduler' object attribute is deprecated. Please access
'num_train_timesteps' over 'DDPMScheduler's config object instead, e.g.
'scheduler.config.num_train_timesteps'.
```

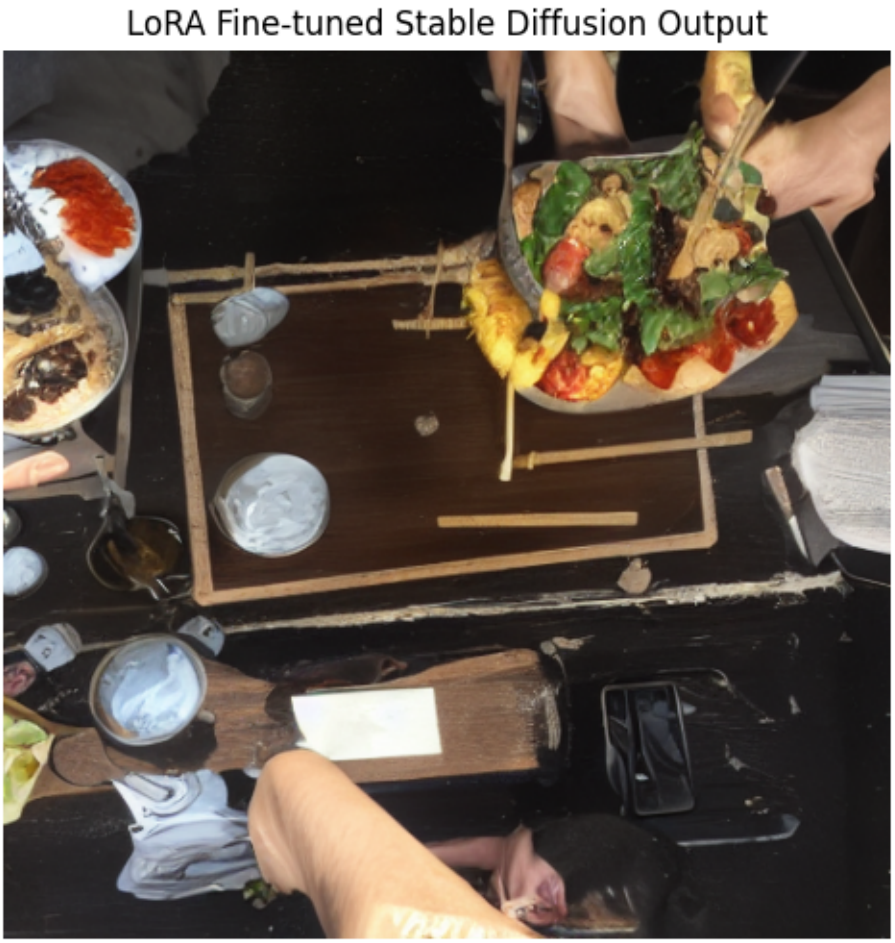
```
  deprecate("direct config name access", "1.0.0", deprecation_message,
standard_warn=False)
```

```
Step 100 / 1500, Loss = 0.116833
Step 200 / 1500, Loss = 0.223462
Step 300 / 1500, Loss = 0.229546
Step 400 / 1500, Loss = 0.124641
Step 500 / 1500, Loss = 0.147510
Step 600 / 1500, Loss = 0.053833
Step 700 / 1500, Loss = 0.179403
Step 800 / 1500, Loss = 0.075595
Step 900 / 1500, Loss = 0.042246
Step 1000 / 1500, Loss = 0.196417
Step 1100 / 1500, Loss = 0.099877
Step 1200 / 1500, Loss = 0.217164
Step 1300 / 1500, Loss = 0.250977
Step 1400 / 1500, Loss = 0.323580
LoRA fine-tuning complete!
```

[ ]:
```
unet.eval()

with torch.no_grad():
    latents = torch.randn(1, 4, 64, 64).to(device)

    for t in reversed(range(scheduler.num_train_timesteps)):
        latent_model_input = latents

        # Explicitly create dummy encoder_hidden_states for unconditional␣
↪generation
        # The UNet2DConditionModel expects (batch_size, sequence_length,␣
↪cross_attention_dim), typically (B, 77, 768)
        batch_size_current = latent_model_input.shape[0]
        dummy_encoder_hidden_states = torch.zeros(
            (batch_size_current, 77, 768), # 77 tokens, 768 features (standard␣
↪for SD1.5 CLIP)
            dtype=latent_model_input.dtype,
            device=device
        )
        noise_pred = unet(latent_model_input, torch.tensor([t], device=device),␣
↪encoder_hidden_states=dummy_encoder_hidden_states).sample

        latents = scheduler.step(noise_pred, t, latents).prev_sample

    # Decode via VAE
    image = vae.decode(latents / 0.18215).sample
    image = (image.clamp(-1,1) + 1) / 2
    image = image.cpu().permute(0,2,3,1).numpy()[0]
```

```
plt.figure(figsize=(6,6))
plt.imshow(image)
plt.axis("off")
plt.title("LoRA Fine-tuned Stable Diffusion Output")
```

[ ]: Text(0.5, 1.0, 'LoRA Fine-tuned Stable Diffusion Output')



[3]:
```
!apt-get update
!apt-get install -y pandoc texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

```
 0% [Working]              Hit:1 https://cli.github.com/packages stable
InRelease
 0% [Connecting to archive.ubuntu.com (185.125.190.83)] [Connecting to security.
Hit:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64
InRelease
```

```
Hit:3 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ InRelease
Hit:4 http://archive.ubuntu.com/ubuntu jammy InRelease
Hit:5 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:6 https://r2u.stat.illinois.edu/ubuntu jammy InRelease
Hit:7 http://archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:8 http://archive.ubuntu.com/ubuntu jammy-backports InRelease
Hit:9 https://ppa.launchpadcontent.net/deadsnakes/ppa/ubuntu jammy InRelease
Hit:10 https://ppa.launchpadcontent.net/graphics-drivers/ppa/ubuntu jammy
InRelease
Hit:11 https://ppa.launchpadcontent.net/ubuntugis/ppa/ubuntu jammy InRelease
Reading package lists… Done
W: Skipping acquire of configured file 'main/source/Sources' as repository
'https://r2u.stat.illinois.edu/ubuntu jammy InRelease' does not seem to provide
it (sources.list entry misspelt?)
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
pandoc is already the newest version (2.9.2.1-3ubuntu2).
The following additional packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 libapache-pom-java libcommons-logging-java
  libcommons-parent-java libfontbox-java libgs9 libgs9-common libidn12
  libijs-0.35 libjbig2dec0 libkpathsea6 libpdfbox-java libptexenc1 libruby3.0
  libsynctex2 libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13
  lmodern poppler-data preview-latex-style rake ruby ruby-net-telnet
  ruby-rubygems ruby-webrick ruby-xmlrpc ruby3.0 rubygems-integration t1utils
  teckit tex-common tex-gyre texlive-base texlive-binaries texlive-latex-base
  texlive-latex-extra texlive-latex-recommended texlive-pictures tipa
  xfonts-encodings xfonts-utils
Suggested packages:
  fonts-noto fonts-freefont-otf | fonts-freefont-ttf libavalon-framework-java
  libcommons-logging-java-doc libexcalibur-logkit-java liblog4j1.2-java
  poppler-utils ghostscript fonts-japanese-mincho | fonts-ipafont-mincho
  fonts-japanese-gothic | fonts-ipafont-gothic fonts-arphic-ukai
  fonts-arphic-uming fonts-nanum ri ruby-dev bundler debhelper gv
  | postscript-viewer perl-tk xpdf | pdf-viewer xzdec
  texlive-fonts-recommended-doc texlive-latex-base-doc python3-pygments
  icc-profiles libfile-which-perl libspreadsheet-parseexcel-perl
  texlive-latex-extra-doc texlive-latex-recommended-doc texlive-luatex
  texlive-pstricks dot2tex prerex texlive-pictures-doc vprerex
  default-jre-headless tipa-doc
The following NEW packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 libapache-pom-java libcommons-logging-java
  libcommons-parent-java libfontbox-java libgs9 libgs9-common libidn12
  libijs-0.35 libjbig2dec0 libkpathsea6 libpdfbox-java libptexenc1 libruby3.0
  libsynctex2 libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13
  lmodern poppler-data preview-latex-style rake ruby ruby-net-telnet
```

```
  ruby-rubygems ruby-webrick ruby-xmlrpc ruby3.0 rubygems-integration t1utils
  teckit tex-common tex-gyre texlive-base texlive-binaries
  texlive-fonts-recommended texlive-latex-base texlive-latex-extra
  texlive-latex-recommended texlive-pictures texlive-plain-generic
  texlive-xetex tipa xfonts-encodings xfonts-utils
0 upgraded, 53 newly installed, 0 to remove and 50 not upgraded.
Need to get 182 MB of archives.
After this operation, 571 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-droid-fallback all
1:6.0.1r16-1.1build1 [1,805 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-lato all 2.0-2.1
[2,696 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy/main amd64 poppler-data all
0.4.11-1 [2,171 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tex-common all 6.17
[33.7 kB]
Get:5 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-urw-base35 all
20200910-1 [6,367 kB]
Get:6 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libgs9-common
all 9.55.0~dfsg1-0ubuntu5.13 [753 kB]
Get:7 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libidn12 amd64
1.38-4ubuntu1 [60.0 kB]
Get:8 http://archive.ubuntu.com/ubuntu jammy/main amd64 libijs-0.35 amd64
0.35-15build2 [16.5 kB]
Get:9 http://archive.ubuntu.com/ubuntu jammy/main amd64 libjbig2dec0 amd64
0.19-3build2 [64.7 kB]
Get:10 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libgs9 amd64
9.55.0~dfsg1-0ubuntu5.13 [5,032 kB]
Get:11 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libkpathsea6
amd64 2021.20210626.59705-1ubuntu0.2 [60.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu jammy/main amd64 libwoff1 amd64
1.0.2-1build4 [45.2 kB]
Get:13 http://archive.ubuntu.com/ubuntu jammy/universe amd64 dvisvgm amd64
2.13.1-1 [1,221 kB]
Get:14 http://archive.ubuntu.com/ubuntu jammy/universe amd64 fonts-lmodern all
2.004.5-6.1 [4,532 kB]
Get:15 http://archive.ubuntu.com/ubuntu jammy/main amd64 fonts-noto-mono all
20201225-1build1 [397 kB]
Get:16 http://archive.ubuntu.com/ubuntu jammy/universe amd64 fonts-texgyre all
20180621-3.1 [10.2 MB]
Get:17 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libapache-pom-java
all 18-1 [4,720 B]
Get:18 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcommons-parent-
java all 43-1 [10.8 kB]
Get:19 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libcommons-logging-
java all 1.2-2 [60.3 kB]
Get:20 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libptexenc1
amd64 2021.20210626.59705-1ubuntu0.2 [39.1 kB]
```

```
Get:21 http://archive.ubuntu.com/ubuntu jammy/main amd64 rubygems-integration
all 1.18 [5,336 B]
Get:22 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby3.0 amd64
3.0.2-7ubuntu2.11 [50.1 kB]
Get:23 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby-rubygems
all 3.3.5-2ubuntu1.2 [228 kB]
Get:24 http://archive.ubuntu.com/ubuntu jammy/main amd64 ruby amd64 1:3.0~exp1
[5,100 B]
Get:25 http://archive.ubuntu.com/ubuntu jammy/main amd64 rake all 13.0.6-2 [61.7
kB]
Get:26 http://archive.ubuntu.com/ubuntu jammy/main amd64 ruby-net-telnet all
0.1.1-2 [12.6 kB]
Get:27 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby-webrick
all 1.7.0-3ubuntu0.2 [52.5 kB]
Get:28 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 ruby-xmlrpc all
0.3.2-1ubuntu0.1 [24.9 kB]
Get:29 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libruby3.0
amd64 3.0.2-7ubuntu2.11 [5,114 kB]
Get:30 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libsynctex2
amd64 2021.20210626.59705-1ubuntu0.2 [55.6 kB]
Get:31 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libteckit0 amd64
2.5.11+ds1-1 [421 kB]
Get:32 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libtexlua53
amd64 2021.20210626.59705-1ubuntu0.2 [120 kB]
Get:33 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 libtexluajit2
amd64 2021.20210626.59705-1ubuntu0.2 [267 kB]
Get:34 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libzzip-0-13 amd64
0.13.72+dfsg.1-1.1 [27.0 kB]
Get:35 http://archive.ubuntu.com/ubuntu jammy/main amd64 xfonts-encodings all
1:1.0.5-0ubuntu2 [578 kB]
Get:36 http://archive.ubuntu.com/ubuntu jammy/main amd64 xfonts-utils amd64
1:7.7+6build2 [94.6 kB]
Get:37 http://archive.ubuntu.com/ubuntu jammy/universe amd64 lmodern all
2.004.5-6.1 [9,471 kB]
Get:38 http://archive.ubuntu.com/ubuntu jammy/universe amd64 preview-latex-style
all 12.2-1ubuntu1 [185 kB]
Get:39 http://archive.ubuntu.com/ubuntu jammy/main amd64 t1utils amd64
1.41-4build2 [61.3 kB]
Get:40 http://archive.ubuntu.com/ubuntu jammy/universe amd64 teckit amd64
2.5.11+ds1-1 [699 kB]
Get:41 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tex-gyre all
20180621-3.1 [6,209 kB]
Get:42 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 texlive-
binaries amd64 2021.20210626.59705-1ubuntu0.2 [9,860 kB]
Get:43 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-base all
2021.20220204-1 [21.0 MB]
Get:44 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-fonts-
recommended all 2021.20220204-1 [4,972 kB]
```

```
Get:45 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-base
all 2021.20220204-1 [1,128 kB]
Get:46 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libfontbox-java all
1:1.8.16-2 [207 kB]
Get:47 http://archive.ubuntu.com/ubuntu jammy/universe amd64 libpdfbox-java all
1:1.8.16-2 [5,199 kB]
Get:48 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-
recommended all 2021.20220204-1 [14.4 MB]
Get:49 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-pictures
all 2021.20220204-1 [8,720 kB]
Get:50 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-latex-extra
all 2021.20220204-1 [13.9 MB]
Get:51 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-plain-
generic all 2021.20220204-1 [27.5 MB]
Get:52 http://archive.ubuntu.com/ubuntu jammy/universe amd64 tipa all 2:1.3-21
[2,967 kB]
Get:53 http://archive.ubuntu.com/ubuntu jammy/universe amd64 texlive-xetex all
2021.20220204-1 [12.4 MB]
Fetched 182 MB in 9s (21.3 MB/s)
Extracting templates from packages: 100%
Preconfiguring packages …
Selecting previously unselected package fonts-droid-fallback.
(Reading database … 121689 files and directories currently installed.)
Preparing to unpack …/00-fonts-droid-fallback_1%3a6.0.1r16-1.1build1_all.deb
…
Unpacking fonts-droid-fallback (1:6.0.1r16-1.1build1) …
Selecting previously unselected package fonts-lato.
Preparing to unpack …/01-fonts-lato_2.0-2.1_all.deb …
Unpacking fonts-lato (2.0-2.1) …
Selecting previously unselected package poppler-data.
Preparing to unpack …/02-poppler-data_0.4.11-1_all.deb …
Unpacking poppler-data (0.4.11-1) …
Selecting previously unselected package tex-common.
Preparing to unpack …/03-tex-common_6.17_all.deb …
Unpacking tex-common (6.17) …
Selecting previously unselected package fonts-urw-base35.
Preparing to unpack …/04-fonts-urw-base35_20200910-1_all.deb …
Unpacking fonts-urw-base35 (20200910-1) …
Selecting previously unselected package libgs9-common.
Preparing to unpack …/05-libgs9-common_9.55.0~dfsg1-0ubuntu5.13_all.deb …
Unpacking libgs9-common (9.55.0~dfsg1-0ubuntu5.13) …
Selecting previously unselected package libidn12:amd64.
Preparing to unpack …/06-libidn12_1.38-4ubuntu1_amd64.deb …
Unpacking libidn12:amd64 (1.38-4ubuntu1) …
Selecting previously unselected package libijs-0.35:amd64.
Preparing to unpack …/07-libijs-0.35_0.35-15build2_amd64.deb …
Unpacking libijs-0.35:amd64 (0.35-15build2) …
Selecting previously unselected package libjbig2dec0:amd64.
```

```
Preparing to unpack …/08-libjbig2dec0_0.19-3build2_amd64.deb …
Unpacking libjbig2dec0:amd64 (0.19-3build2) …
Selecting previously unselected package libgs9:amd64.
Preparing to unpack …/09-libgs9_9.55.0~dfsg1-0ubuntu5.13_amd64.deb …
Unpacking libgs9:amd64 (9.55.0~dfsg1-0ubuntu5.13) …
Selecting previously unselected package libkpathsea6:amd64.
Preparing to unpack …/10-libkpathsea6_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libkpathsea6:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libwoff1:amd64.
Preparing to unpack …/11-libwoff1_1.0.2-1build4_amd64.deb …
Unpacking libwoff1:amd64 (1.0.2-1build4) …
Selecting previously unselected package dvisvgm.
Preparing to unpack …/12-dvisvgm_2.13.1-1_amd64.deb …
Unpacking dvisvgm (2.13.1-1) …
Selecting previously unselected package fonts-lmodern.
Preparing to unpack …/13-fonts-lmodern_2.004.5-6.1_all.deb …
Unpacking fonts-lmodern (2.004.5-6.1) …
Selecting previously unselected package fonts-noto-mono.
Preparing to unpack …/14-fonts-noto-mono_20201225-1build1_all.deb …
Unpacking fonts-noto-mono (20201225-1build1) …
Selecting previously unselected package fonts-texgyre.
Preparing to unpack …/15-fonts-texgyre_20180621-3.1_all.deb …
Unpacking fonts-texgyre (20180621-3.1) …
Selecting previously unselected package libapache-pom-java.
Preparing to unpack …/16-libapache-pom-java_18-1_all.deb …
Unpacking libapache-pom-java (18-1) …
Selecting previously unselected package libcommons-parent-java.
Preparing to unpack …/17-libcommons-parent-java_43-1_all.deb …
Unpacking libcommons-parent-java (43-1) …
Selecting previously unselected package libcommons-logging-java.
Preparing to unpack …/18-libcommons-logging-java_1.2-2_all.deb …
Unpacking libcommons-logging-java (1.2-2) …
Selecting previously unselected package libptexenc1:amd64.
Preparing to unpack …/19-libptexenc1_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libptexenc1:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package rubygems-integration.
Preparing to unpack …/20-rubygems-integration_1.18_all.deb …
Unpacking rubygems-integration (1.18) …
Selecting previously unselected package ruby3.0.
Preparing to unpack …/21-ruby3.0_3.0.2-7ubuntu2.11_amd64.deb …
Unpacking ruby3.0 (3.0.2-7ubuntu2.11) …
Selecting previously unselected package ruby-rubygems.
Preparing to unpack …/22-ruby-rubygems_3.3.5-2ubuntu1.2_all.deb …
Unpacking ruby-rubygems (3.3.5-2ubuntu1.2) …
Selecting previously unselected package ruby.
Preparing to unpack …/23-ruby_1%3a3.0~exp1_amd64.deb …
```

```
Unpacking ruby (1:3.0~exp1) …
Selecting previously unselected package rake.
Preparing to unpack …/24-rake_13.0.6-2_all.deb …
Unpacking rake (13.0.6-2) …
Selecting previously unselected package ruby-net-telnet.
Preparing to unpack …/25-ruby-net-telnet_0.1.1-2_all.deb …
Unpacking ruby-net-telnet (0.1.1-2) …
Selecting previously unselected package ruby-webrick.
Preparing to unpack …/26-ruby-webrick_1.7.0-3ubuntu0.2_all.deb …
Unpacking ruby-webrick (1.7.0-3ubuntu0.2) …
Selecting previously unselected package ruby-xmlrpc.
Preparing to unpack …/27-ruby-xmlrpc_0.3.2-1ubuntu0.1_all.deb …
Unpacking ruby-xmlrpc (0.3.2-1ubuntu0.1) …
Selecting previously unselected package libruby3.0:amd64.
Preparing to unpack …/28-libruby3.0_3.0.2-7ubuntu2.11_amd64.deb …
Unpacking libruby3.0:amd64 (3.0.2-7ubuntu2.11) …
Selecting previously unselected package libsynctex2:amd64.
Preparing to unpack …/29-libsynctex2_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libsynctex2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libteckit0:amd64.
Preparing to unpack …/30-libteckit0_2.5.11+ds1-1_amd64.deb …
Unpacking libteckit0:amd64 (2.5.11+ds1-1) …
Selecting previously unselected package libtexlua53:amd64.
Preparing to unpack …/31-libtexlua53_2021.20210626.59705-1ubuntu0.2_amd64.deb
…
Unpacking libtexlua53:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libtexluajit2:amd64.
Preparing to unpack
…/32-libtexluajit2_2021.20210626.59705-1ubuntu0.2_amd64.deb …
Unpacking libtexluajit2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package libzzip-0-13:amd64.
Preparing to unpack …/33-libzzip-0-13_0.13.72+dfsg.1-1.1_amd64.deb …
Unpacking libzzip-0-13:amd64 (0.13.72+dfsg.1-1.1) …
Selecting previously unselected package xfonts-encodings.
Preparing to unpack …/34-xfonts-encodings_1%3a1.0.5-0ubuntu2_all.deb …
Unpacking xfonts-encodings (1:1.0.5-0ubuntu2) …
Selecting previously unselected package xfonts-utils.
Preparing to unpack …/35-xfonts-utils_1%3a7.7+6build2_amd64.deb …
Unpacking xfonts-utils (1:7.7+6build2) …
Selecting previously unselected package lmodern.
Preparing to unpack …/36-lmodern_2.004.5-6.1_all.deb …
Unpacking lmodern (2.004.5-6.1) …
Selecting previously unselected package preview-latex-style.
Preparing to unpack …/37-preview-latex-style_12.2-1ubuntu1_all.deb …
Unpacking preview-latex-style (12.2-1ubuntu1) …
Selecting previously unselected package t1utils.
Preparing to unpack …/38-t1utils_1.41-4build2_amd64.deb …
```

```
Unpacking t1utils (1.41-4build2) …
Selecting previously unselected package teckit.
Preparing to unpack …/39-teckit_2.5.11+ds1-1_amd64.deb …
Unpacking teckit (2.5.11+ds1-1) …
Selecting previously unselected package tex-gyre.
Preparing to unpack …/40-tex-gyre_20180621-3.1_all.deb …
Unpacking tex-gyre (20180621-3.1) …
Selecting previously unselected package texlive-binaries.
Preparing to unpack …/41-texlive-
binaries_2021.20210626.59705-1ubuntu0.2_amd64.deb …
Unpacking texlive-binaries (2021.20210626.59705-1ubuntu0.2) …
Selecting previously unselected package texlive-base.
Preparing to unpack …/42-texlive-base_2021.20220204-1_all.deb …
Unpacking texlive-base (2021.20220204-1) …
Selecting previously unselected package texlive-fonts-recommended.
Preparing to unpack …/43-texlive-fonts-recommended_2021.20220204-1_all.deb …
Unpacking texlive-fonts-recommended (2021.20220204-1) …
Selecting previously unselected package texlive-latex-base.
Preparing to unpack …/44-texlive-latex-base_2021.20220204-1_all.deb …
Unpacking texlive-latex-base (2021.20220204-1) …
Selecting previously unselected package libfontbox-java.
Preparing to unpack …/45-libfontbox-java_1%3a1.8.16-2_all.deb …
Unpacking libfontbox-java (1:1.8.16-2) …
Selecting previously unselected package libpdfbox-java.
Preparing to unpack …/46-libpdfbox-java_1%3a1.8.16-2_all.deb …
Unpacking libpdfbox-java (1:1.8.16-2) …
Selecting previously unselected package texlive-latex-recommended.
Preparing to unpack …/47-texlive-latex-recommended_2021.20220204-1_all.deb …
Unpacking texlive-latex-recommended (2021.20220204-1) …
Selecting previously unselected package texlive-pictures.
Preparing to unpack …/48-texlive-pictures_2021.20220204-1_all.deb …
Unpacking texlive-pictures (2021.20220204-1) …
Selecting previously unselected package texlive-latex-extra.
Preparing to unpack …/49-texlive-latex-extra_2021.20220204-1_all.deb …
Unpacking texlive-latex-extra (2021.20220204-1) …
Selecting previously unselected package texlive-plain-generic.
Preparing to unpack …/50-texlive-plain-generic_2021.20220204-1_all.deb …
Unpacking texlive-plain-generic (2021.20220204-1) …
Selecting previously unselected package tipa.
Preparing to unpack …/51-tipa_2%3a1.3-21_all.deb …
Unpacking tipa (2:1.3-21) …
Selecting previously unselected package texlive-xetex.
Preparing to unpack …/52-texlive-xetex_2021.20220204-1_all.deb …
Unpacking texlive-xetex (2021.20220204-1) …
Setting up fonts-lato (2.0-2.1) …
Setting up fonts-noto-mono (20201225-1build1) …
Setting up libwoff1:amd64 (1.0.2-1build4) …
Setting up libtexlua53:amd64 (2021.20210626.59705-1ubuntu0.2) …
```

```
Setting up libijs-0.35:amd64 (0.35-15build2) …
Setting up libtexluajit2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up libfontbox-java (1:1.8.16-2) …
Setting up rubygems-integration (1.18) …
Setting up libzzip-0-13:amd64 (0.13.72+dfsg.1-1.1) …
Setting up fonts-urw-base35 (20200910-1) …
Setting up poppler-data (0.4.11-1) …
Setting up tex-common (6.17) …
update-language: texlive-base not installed and configured, doing nothing!
Setting up libjbig2dec0:amd64 (0.19-3build2) …
Setting up libteckit0:amd64 (2.5.11+ds1-1) …
Setting up libapache-pom-java (18-1) …
Setting up ruby-net-telnet (0.1.1-2) …
Setting up xfonts-encodings (1:1.0.5-0ubuntu2) …
Setting up t1utils (1.41-4build2) …
Setting up libidn12:amd64 (1.38-4ubuntu1) …
Setting up fonts-texgyre (20180621-3.1) …
Setting up libkpathsea6:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up ruby-webrick (1.7.0-3ubuntu0.2) …
Setting up fonts-lmodern (2.004.5-6.1) …
Setting up fonts-droid-fallback (1:6.0.1r16-1.1build1) …
Setting up ruby-xmlrpc (0.3.2-1ubuntu0.1) …
Setting up libsynctex2:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up libgs9-common (9.55.0~dfsg1-0ubuntu5.13) …
Setting up teckit (2.5.11+ds1-1) …
Setting up libpdfbox-java (1:1.8.16-2) …
Setting up libgs9:amd64 (9.55.0~dfsg1-0ubuntu5.13) …
Setting up preview-latex-style (12.2-1ubuntu1) …
Setting up libcommons-parent-java (43-1) …
Setting up dvisvgm (2.13.1-1) …
Setting up libcommons-logging-java (1.2-2) …
Setting up xfonts-utils (1:7.7+6build2) …
Setting up libptexenc1:amd64 (2021.20210626.59705-1ubuntu0.2) …
Setting up texlive-binaries (2021.20210626.59705-1ubuntu0.2) …
update-alternatives: using /usr/bin/xdvi-xaw to provide /usr/bin/xdvi.bin
(xdvi.bin) in auto mode
update-alternatives: using /usr/bin/bibtex.original to provide /usr/bin/bibtex
(bibtex) in auto mode
Setting up lmodern (2.004.5-6.1) …
Setting up texlive-base (2021.20220204-1) …
/usr/bin/ucfr
/usr/bin/ucfr
/usr/bin/ucfr
/usr/bin/ucfr
mktexlsr: Updating /var/lib/texmf/ls-R-TEXLIVEDIST…
mktexlsr: Updating /var/lib/texmf/ls-R-TEXMFMAIN…
mktexlsr: Updating /var/lib/texmf/ls-R…
mktexlsr: Done.
```

```
tl-paper: setting paper size for dvips to a4:
/var/lib/texmf/dvips/config/config-paper.ps
tl-paper: setting paper size for dvipdfmx to a4:
/var/lib/texmf/dvipdfmx/dvipdfmx-paper.cfg
tl-paper: setting paper size for xdvi to a4: /var/lib/texmf/xdvi/XDvi-paper
tl-paper: setting paper size for pdftex to a4: /var/lib/texmf/tex/generic/tex-
ini-files/pdftexconfig.tex
Setting up tex-gyre (20180621-3.1) …
Setting up texlive-plain-generic (2021.20220204-1) …
Setting up texlive-latex-base (2021.20220204-1) …
Setting up texlive-latex-recommended (2021.20220204-1) …
Setting up texlive-pictures (2021.20220204-1) …
Setting up texlive-fonts-recommended (2021.20220204-1) …
Setting up tipa (2:1.3-21) …
Setting up texlive-latex-extra (2021.20220204-1) …
Setting up texlive-xetex (2021.20220204-1) …
Setting up rake (13.0.6-2) …
Setting up libruby3.0:amd64 (3.0.2-7ubuntu2.11) …
Setting up ruby3.0 (3.0.2-7ubuntu2.11) …
Setting up ruby (1:3.0~exp1) …
Setting up ruby-rubygems (3.3.5-2ubuntu1.2) …
Processing triggers for man-db (2.10.2-1) …
Processing triggers for mailcap (3.70+nmu1ubuntu1) …
Processing triggers for fontconfig (2.13.1-4.2ubuntu5) …
Processing triggers for libc-bin (2.35-0ubuntu3.8) …
/sbin/ldconfig.real: /usr/local/lib/libtcm.so.1 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc.so.2 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libur_adapter_opencl.so.0 is not a symbolic
link

/sbin/ldconfig.real: /usr/local/lib/libur_adapter_level_zero.so.0 is not a
symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbb.so.12 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbmalloc_proxy.so.2 is not a symbolic
link

/sbin/ldconfig.real: /usr/local/lib/libumf.so.1 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtcm_debug.so.1 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libhwloc.so.15 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind.so.3 is not a symbolic link
```

```
/sbin/ldconfig.real: /usr/local/lib/libur_loader.so.0 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libur_adapter_level_zero_v2.so.0 is not a
symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_5.so.3 is not a symbolic link

/sbin/ldconfig.real: /usr/local/lib/libtbbbind_2_0.so.3 is not a symbolic link

Processing triggers for tex-common (6.17) …
Running updmap-sys. This may take some time… done.
Running mktexlsr /var/lib/texmf … done.
Building format(s) --all.
        This may take some time… done.
```

[4]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
!jupyter nbconvert --to pdf "/content/drive/MyDrive/Colab Notebooks/Copy of␣
↪Final_Draft_Pipeline.ipynb"
```

[ ]:
```
from google.colab import files

files.download("/content/drive/MyDrive/Copy of Final_Draft_Pipeline.pdf")
```