AVR Project Report

Exploration of VAEs and Diffusion Models

Variational AutoEncoders (VAEs)

In this notebook we will be using a variational autoencoder to augment MNIST digit images

Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimize the reconstruction error between the encoded-decoded data and the initial data.

However, in order to introduce some regularization of the latent space, we proceed to a slight modification of the encoding-decoding process: instead of encoding an input as a single point, we encode it as a distribution over the latent space. The model is then trained as follows:

- The input is encoded as distribution over the latent space
- A point from the latent space is sampled from that distribution
- The sampled point is decoded and the reconstruction error can be computed
- The reconstruction error is backpropagated through the network

We utilize three different components in order to build the model:

- **Sampling Layer**, to draw a point from the distribution
- **Encoder**, typically conssisting of Convolutional Layers and Max Pooling Layers
- **Decoder**, typically consisting of Transposed Convolutional Layers and Upsampling Layers

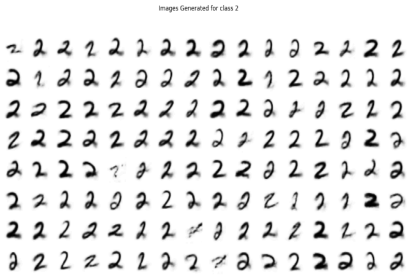
We have experimented with various hyperparameters such as Batch Size, Embedded Codings Size and number of epochs in order to generate augmented images for each class of the MNIST dataset.

The experiments performed have been documented below:

Experiment #	BATCH SIZE	CODINGS SIZE	N_EPOCHS
1	128	32	150
2	128	64	150
3	256	128	200
4	128	128	200
5	64	64	200

Qualitative Analysis:

- It has been observed from experiments 1 and 2 that generally there is **lower intra class variability** upon increasing the coding size (the dimension of the latent space). The following image depicts the augmented images generated in **experiments 1 and 2** for class '2' in the MNIST dataset:



It is clear that the intra class variability has decreased by increasing the coding size from 32 to 64. This could be due to the following reasons:

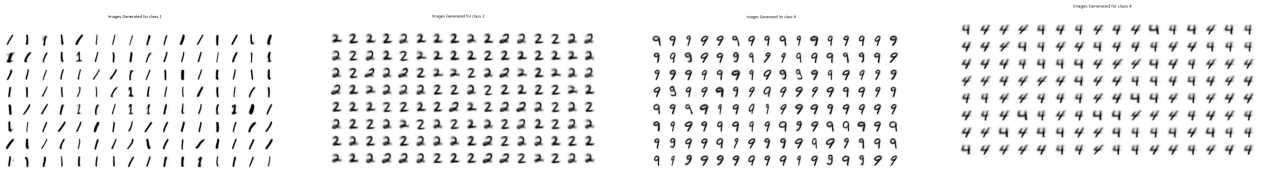
★ **Over-regularization**: As the coding size increases, the regularization term in the VAE loss might become relatively less influential. However, if other regularization

techniques are not adjusted accordingly, the model might still enforce strong constraints on the latent space, limiting its ability to represent diverse instances of the same class.

★ **Model Convergence Issues:** Increasing the coding size introduces additional parameters to be learned, and the training process might become more challenging. If the model does not converge properly or gets stuck in a suboptimal solution, it may not effectively utilize the increased capacity to capture intra-class variability**.**

- It has been observed from experiments 3 and 4 that increasing the batch size from 128 to 256 while keeping the other parameters epochs and codings size constant leads to **lower intra class variability.**
  - ★ For classes ‘1’, ‘2’, ‘4’ and ‘9’, all the augmented images generated by the VAE using batch size 256 are exactly the same.

  - ★ The corresponding classes generated by the VAE using batch size 128 are shown below:



  - ★ It is evident that the diversity of images generated by the VAE trained using a lower batch size is much higher. The reasons for this could be:

    - ➢ **Averaging Effects:** Larger batch sizes lead to more stable and less noisy gradient estimates because the gradients are computed based on a larger set of samples. While this can result in smoother convergence, it might also lead to a situation where the model averages out subtle variations within the same class, reducing intra-class variability.

- ➢ **Limited Exploration in Latent Space:**In a VAE, the training process involves sampling from the latent space. Larger batch sizes might lead to a more conservative exploration of the latent space, where the model converges to a more restricted region, resulting in less diverse samples within each class.

- The resultant images for all the classes 1-9 from the various experiments performed can be obtained from the version history of our kaggle notebook linked above.

Conditional VAE:

In this notebook we build the conditional variation of the autoencoder, train it and analyze the latent space with the t-SNE technique.

A Conditional Variational Autoencoder (CVAE) is a type of generative model that extends the traditional VAE by incorporating conditional information, allowing for the generation of data conditioned on specific attributes or labels.

In a CVAE, the encoder not only maps the input data to a probability distribution in the latent space but also takes conditional information into account. This is achieved by conditioning the latent space distribution on the provided conditional information.

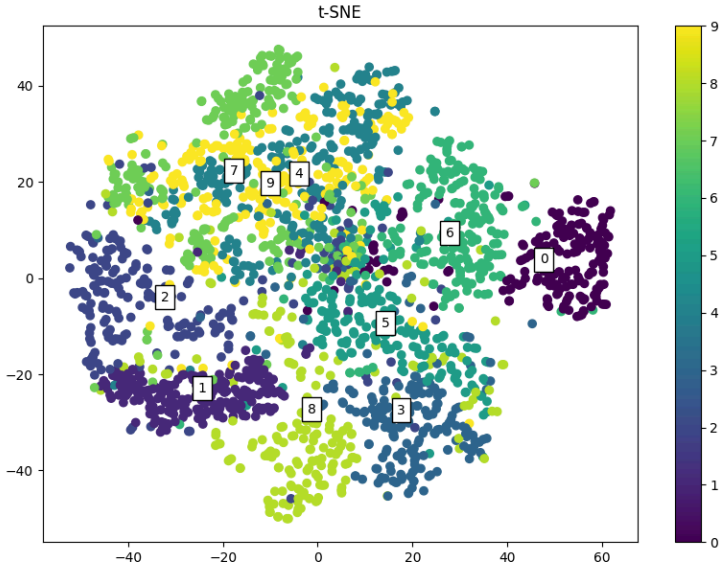
During the decoding process, the decoder takes samples from the conditioned latent space and, along with the conditional information, generates reconstructed data. During inference, given a set of conditional information, a CVAE can generate new samples by sampling from the conditioned latent space and decoding them.

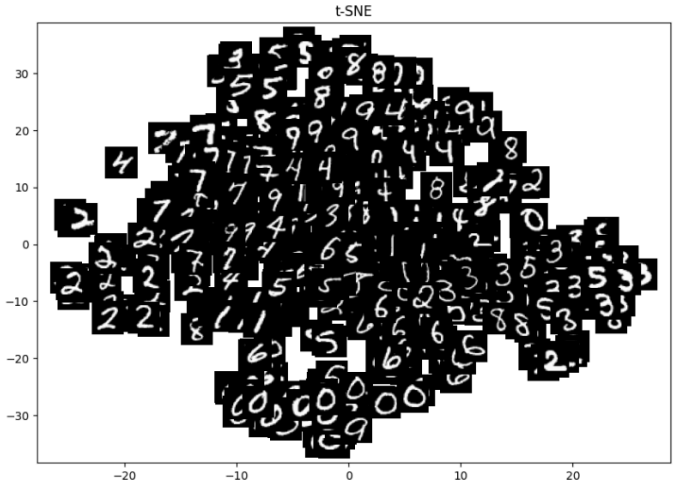
We have experimented with various values of epochs as well as batch size in order to see the effect on the t-SNE visualization of the latent space on the MNIST dataset.

Experiment #	EPOCHS	BATCH SIZE	Val_loss
1	200	128	0.8499
2	200	64	0.8501
3	150	256	0.8508
4	250	64	0.8505

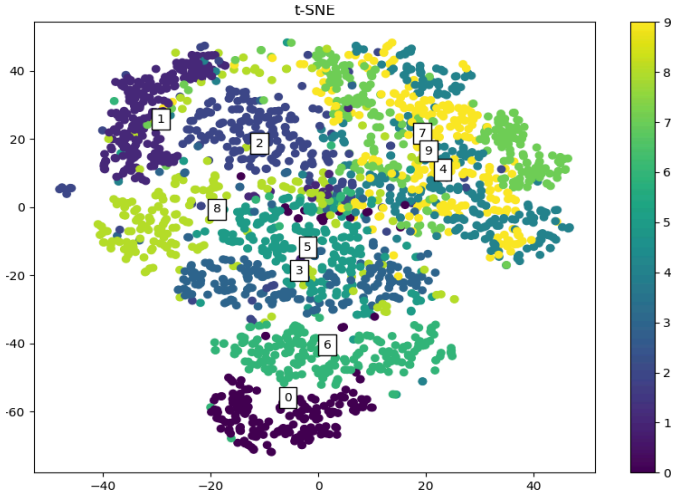
The t-SNE visualizations for the above experiments are given below:

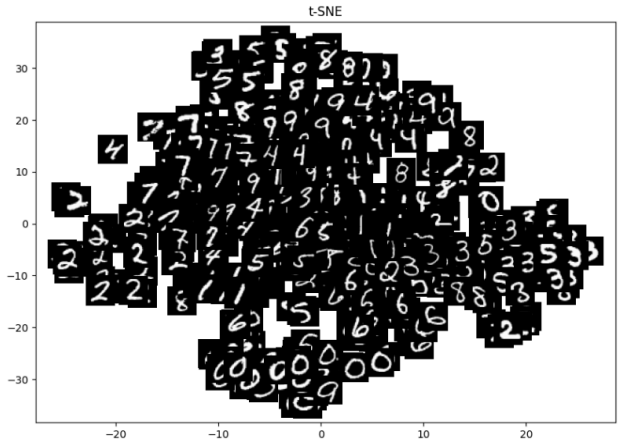
Experiment 1:



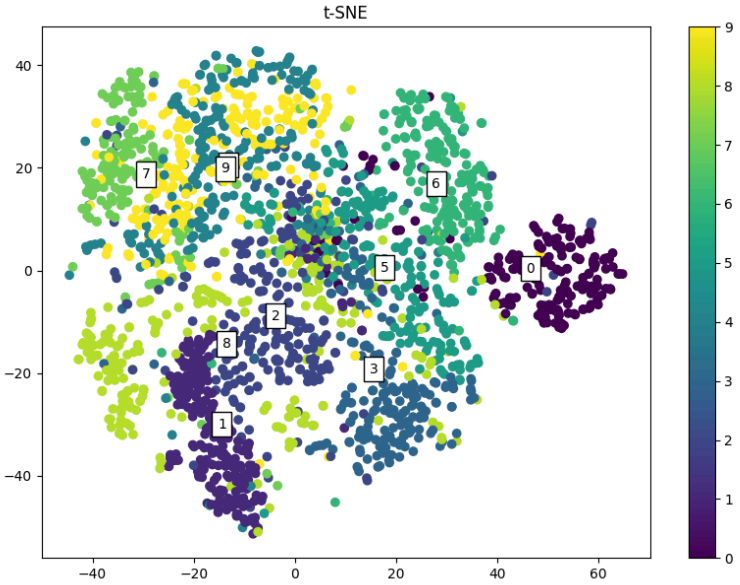


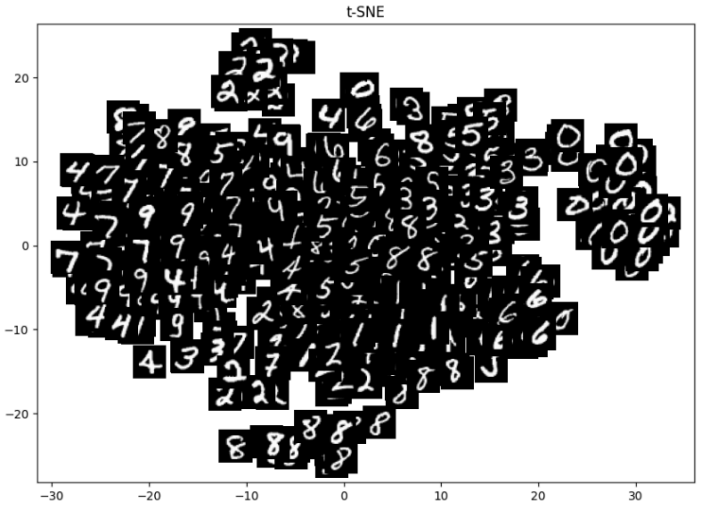
Experiment 2:



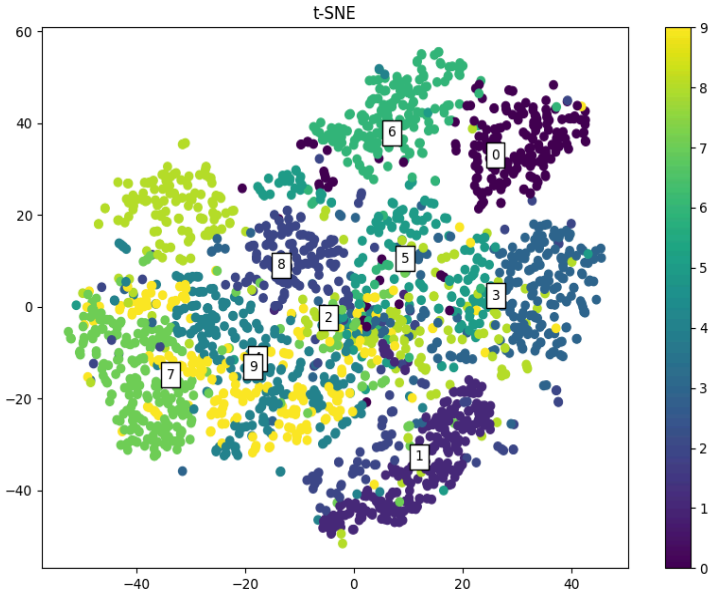


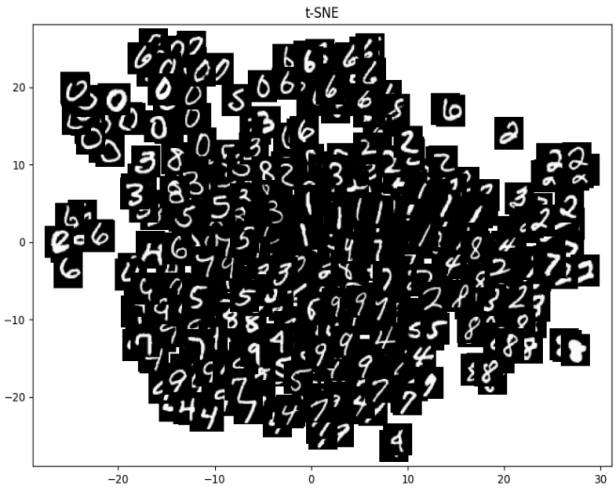
Experiment 3:





Experiment 4:





Some observations:

- Classes 4, 9 and 7 are very close to each other in all the plots. This means that these classes might share certain **structural features** that the CVAE struggles to disentangle. For example, "4" and "9" can have similar shapes, and "7" might have shared characteristics, leading to **proximity in the latent space**.

- Class 0 is well separated from all the other digits in all the plots. This implies that the model has successfully learned to create a **distinct and isolated cluster** for the digit "0" in the latent space. The CVAE has established a **clear decision boundary** between the latent representations of "0" and other digits. This indicates that the learned features for "0" are **not easily confused** with those of other digits.

- By concurrently optimizing both the reconstruction loss and KL divergence loss, we have achieved the creation of a latent space that, on a local scale, retains the similarity of neighboring encodings through clustering.

Vector Quantised VAE:

In this notebook, we construct a Vector Quantized Variational Autoencoder (VQ-VAE). Unlike traditional VAEs, where the latent space is continuous and sampled from a Gaussian distribution, VQ-VAEs adopt a discrete latent space, simplifying the optimization process.

This is achieved by managing a discrete set known as a **codebook**. The creation of this codebook involves **discretizing the space between continuous embeddings and the encoded outputs**. These discrete code words are subsequently input into the decoder, which is trained to produce reconstructed samples.

To establish a codebook, we initialize an embedding table. The L2-normalized distance is computed between the flattened encoder outputs and the code words in this codebook. The code that minimizes the distance is chosen, and one-hot encoding is applied for quantization.

Given that the quantization process lacks differentiability, we introduce a straight-through estimator between the decoder and encoder. This ensures that the decoder gradients directly influence the encoder. Since both the encoder and decoder share the same channel space, the gradients from the decoder remain meaningful to the encoder.

PixelCNN has been trained and used to generate new images by sampling from the codebook. A PixelCNN generates an image on a pixel-by-pixel basis. For the purpose in this example, however, its task is to generate code book indices instead of pixels directly. The trained VQ-VAE decoder is used to map the indices generated by the PixelCNN back into the pixel space.

The training process for the PixelCNN involves learning a categorical distribution for the discrete codes. Initially, we generate code indices using the recently trained encoder and vector quantizer. The objective during training is to minimize the cross-entropy loss between these indices and the PixelCNN outputs.

Once our PixelCNN is trained, we sample distinct codes from its outputs and pass them to our decoder to generate novel images.

Results:

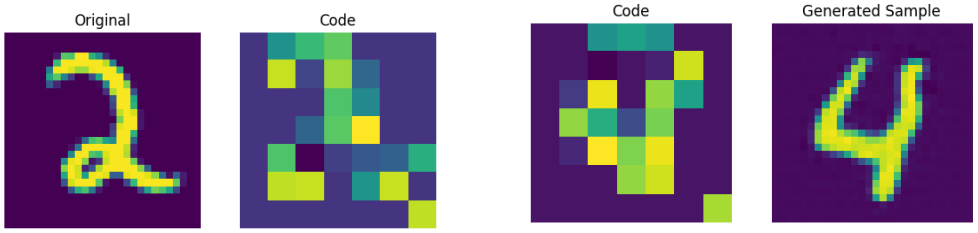
We have experimented with various **hyperparameters** of both the encoder-decoder model as well as the PixelCNN model as shown below:

Exp #	**vq-vae embeddings**	**epochs**	**Batch size**	**Batch Size(pixelCNN)**	**Epochs (PixelCNN)**	**Vqvae Loss**	**PixelCNN Accuracy**
1	256	50	256	256	50	0.1787	0.7044
2	128	30	128	128	30	0.1561	0.7107
3	64	50	64	64	50	0.1480	0.6724
4	128	50	64	128	50	0.1318	0.6975
5	128	30	128	64	30	0.1616	0.7084

Some observations which can made from the results are:

- The quality of the generated images is directly proportional to the PixelCNN accuracy, Hence by tweaking the PixelCNN model, the quality of images can be improved.
- PixelCNN **Batch Size of 128** has been found to be optimal for this particular task and it gives the highest accuracy.
- Apon decreasing the number of VQ-VAE embeddings, the VQ-VAE **loss decreases and accuracy increases.**

Examples of code generation from original image and generation of a new image from a code:



Diffusion Models:

Inner working of diffusion models:

In this section we provide an introduction to Diffusion Models as well as **score matching** based on the swiss roll dataset.

Score matching:

Score matching aims to learn the *gradients* (termed *score*) of log *p(x)* with respect to x. Hence, we seek a model to approximate $\nabla_x \log p(x)$. An efficient formulation of sliced score matching, which relies on random projections to approximate the computation of the Jacobian is given below:

$$E_{\mathbf{v}\sim\mathcal{N}(0,1)}E_{\mathbf{x}\sim p(\mathbf{x})}\left[\mathbf{v}^T\nabla_{\mathbf{x}}\mathcal{F}_\theta(\mathbf{x})\mathbf{v}+\frac{1}{2}\left\|\mathbf{v}^T\mathcal{F}_\theta(\mathbf{x})\right\|_2^2\right],$$

where v~N(0,1) are a set of Normal-distributed vectors. They show that this can be computed by using forward mode auto-differentiation, which is computationally efficient.

Denoising Score Matching:

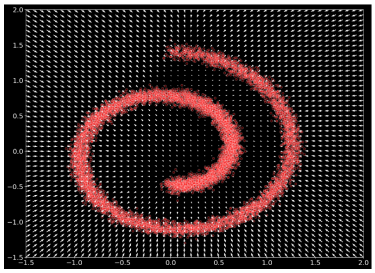
The denoising score matching loss is as follows:

$$l(\theta;\sigma)=E_{q_\sigma(\tilde{\mathbf{x}}|\mathbf{x})}E_{\mathbf{x}\sim p(\mathbf{x})}\left[\left\|\mathcal{F}_\theta(\tilde{\mathbf{x}})+\frac{\tilde{\mathbf{x}}-\mathbf{x}}{\sigma^2}\right\|_2^2\right].$$

We implement the denoising score matching loss as follows:

```
def denoising_score_matching(scorenet, samples, sigma=0.01):
    perturbed_samples = samples + torch.randn_like(samples) * sigma
    target = - 1 / (sigma ** 2) * (perturbed_samples - samples)
    scores = scorenet(perturbed_samples)
    target = target.view(target.shape[0], -1)
    scores = scores.view(scores.shape[0], -1)
    loss = 1 / 2. * ((scores - target) ** 2).sum(dim=-1).mean(dim=0)
    return loss
```

Plotting the output value across the input space using the swiss roll dataset:



Diffusion Models

Diffusion probabilistic models are based on two reciprocal processes that represent two Markov chains of random variables. One process gradually adds noise to the input data (called the **diffusion or forward process**), destroying the signal up to full noise. In the opposite direction, the **reverse process** tries to learn how to invert this diffusion process (transform random noise into a high-quality waveform).

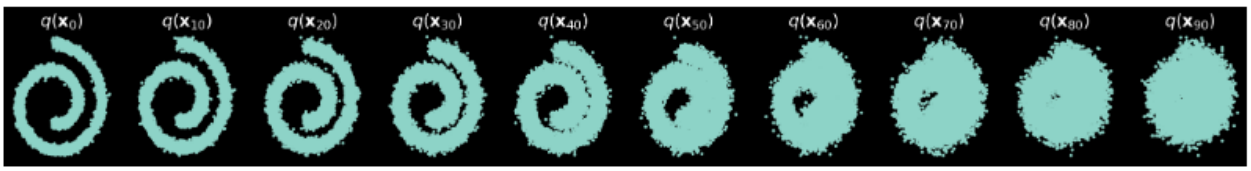
- **Forward Process**:

  The forward process in a diffusion model generates data through sequential transformations, starting from a simple distribution and iteratively refining the sample in a series of diffusion steps. These steps, driven by learned transformations, enhance the sample's quality, capturing intricate patterns from the training data.

  Implementation:

```
def forward_process(x_start, n_steps, noise=None):
    """ Diffuse the data (t == 0 means diffused for 1 step) """
    x_seq = [x_start]
    for n in range(n_steps):
        x_seq.append((torch.sqrt(1 - betas[n]) * x_seq[-1]) + (betas[n] * torch.rand_like(x_s
tart)))
    return x_seq
n_steps = 100
betas = torch.tensor([0.035] * n_steps)
dataset = torch.Tensor(data.T).float()
x_seq = forward_process(dataset, n_steps, betas)
fig, axs = plt.subplots(1, 10, figsize=(28, 3))
for i in range(10):
    axs[i].scatter(x_seq[int((i / 10.0) * n_steps)][:, 0], x_seq[int((i / 10.0) * n_steps)]
[:, 1], s=10);
    axs[i].set_axis_off(); axs[i].set_title('$q(\mathbf{x}_{'+str(int((i / 10.0) * n_steps))
+'})$')
```

  Result:



- **Reverse Process:**

  The reverse process in a diffusion model focuses on reconstructing data, aiming to invert the forward process. By applying learned inverse transformations to generated samples, this process accurately recovers fine-grained details, undoing diffusion steps and removing added noise.

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

  The two functions defining the mean μθ(xt,t) and covariance Σθ(xt,t) can be parametrized by deep neural networks. Note also that these functions are parametrized by t, which means that a single model can be used for all time steps.

- **Training Loss:**

  The training loss is given by

$$K = -\mathbb{E}_q[D_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \| p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) + H_q(\mathbf{X}_T|\mathbf{X}_0) - H_q(\mathbf{X}_1|\mathbf{X}_0) - H_p(\mathbf{X}_T)]$$

  Implementation:

```
def compute_loss(true_mean, true_var, model_mean, model_var):
    # the KL divergence between model transition and posterior from data
    KL = normal_kl(true_mean, true_var, model_mean, model_var).float()
    # conditional entropies H_q(x^T|x^0) and H_q(x^1|x^0)
    H_start = entropy(betas[0].float()).float()
    beta_full_trajectory = 1. - torch.exp(torch.sum(torch.log(alphas))).float()
    H_end = entropy(beta_full_trajectory.float()).float()
    H_prior = entropy(torch.tensor([1.])).float()
    negL_bound = KL * n_steps + H_start - H_end + H_prior
    # the negL_bound if this was an isotropic Gaussian model of the data
    negL_gauss = entropy(torch.tensor([1.])).float()
    negL_diff = negL_bound - negL_gauss
    L_diff_bits = negL_diff / np.log(2.)
    L_diff_bits_avg = L_diff_bits.mean()
    return L_diff_bits_avg
```

The terms discussed in this notebook are to understand the inner workings and components of diffusion models in detail. The rest of the notebooks deal with particular tasks and experiments performed using various diffusion models.

Various experiments performed using diffusion models:

❖ Diffusion model training from scratch:

1. Predicting the noise:

In this notebook, we have trained a diffusion model to generate images of butterflies. We have also explored the core components of the Hugginface Diffusers library.

What Did we do:

- Experimented with a powerful custom diffusion model pipeline in action
- Create our own mini pipeline by:
    - Recapping the core ideas behind diffusion models
    - Loading in data from the Hub for training
    - Exploring how we add noise to this data with a scheduler
    - Creating and training the UNet model
    - Putting the pieces together into a working pipeline

Introduction:

- Stable Diffusion is a powerful **text-conditioned latent diffusion model.**
- However, Stable Diffusion has a limitation: it lacks knowledge of our appearance unless we're widely recognized and our images are all over the internet.
- To mitigate this limitation, we looked at the **Stable Diffusion Dreambooth Library.**

What is Dreambooth?:

- Dreambooth allows us to create a customized model variant by injecting extra knowledge about a specific face, object, or style.
- It's like having our own photo booth where we capture a subject and synthesize it in various contexts, wherever our dreams take us.
- Dreambooth works seamlessly even with just a small set of images (typically 10 to 20).

Example Application:

To demonstrate its effectiveness, we used a powerful custom model which is trained on 5 photos of a popular children's toy called "Mr Potato Head" to generate some images given a text prompt.

model url =*`https://huggingface.co/sd-dreambooth-library/mr-potato-head`

The steps we followed are:

- First, we load the pipeline. This will download model weights etc. from the Hub.
- Once the pipeline has finished loading, we can generate images with it.

Experimentation:

- For a given prompt, we tried to change the parameters to see how the results change correspondingly. We Experiment with changing the number of **inference steps** and the **guidance scale** (determines how closely the model matches the prompt).
    - number of **inference steps = 25** and the **guidance scale=7.5**

```
prompt = "an abstract oil painting of sks mr potato head by picasso"
image = pipe(prompt, num_inference_steps=25, guidance_scale=7.5).images[0]
image
```

```
100% 25/25 [00:06<00:00, 3.77it/s]
```

    - number of **inference steps = 50** and the **guidance scale=7.5**

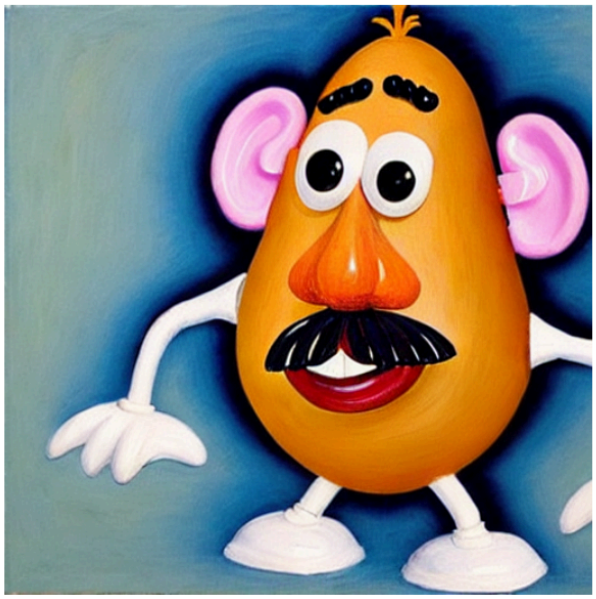
```
[33]: prompt = "an abstract oil painting of sks mr potato head by picasso"
image = pipe(prompt, num_inference_steps=50, guidance_scale=7.5).images[0]
image
```

```
100% 50/50 [00:13<00:00, 3.76it/s]
[33]:
```

- number of **inference steps = 75** and the **guidance scale=7.5**

```
prompt = "an abstract oil painting of sks mr potato head by picasso"
image = pipe(prompt, num_inference_steps=75, guidance_scale=7.5).images[0]
image
```

```
100% 75/75 [00:20<00:00, 3.76it/s]
```



- number of **inference steps = 50** and the **guidance scale=5**

```
prompt = "an abstract oil painting of sks mr potato head by picasso"
image = pipe(prompt, num_inference_steps=50, guidance_scale=5).images[0]
image
```

```
100% 50/50 [00:13<00:00, 3.76it/s]
```

Qualitative Analysis:

Inference Steps:

- In a diffusion model, the process involves gradually revealing an image by adding noise to it over a series of steps.
- Each inference step corresponds to a stage where the model refines the image by reducing noise.
- We observe that when the number of inference steps is increased in a diffusion model, it generally leads to smoother and more detailed generated images

Guidance Scale:

- The guidance scale determines the **strength of influence** that the prompt has on the generated output.
- A higher guidance scale means the model will **more closely follow** the prompt.
- We observe that when the **guidance scale** is increased in a diffusion model, the model becomes more focused on matching the prompt and hence the generated output will closely resemble the prompt, potentially sacrificing creativity or divergence.

We also looked at how we can train a diffusion model (to generate images of butterflies) from scratch.

MVP (Minimum Viable Pipeline)

The core API of Huggingface Diffusers is divided into three main components:

1. **Pipelines**: high-level classes designed to rapidly generate samples from popular trained diffusion models in a user-friendly fashion.
2. **Models**: popular architectures for training new diffusion models, *e.g.* UNet
3. **Schedulers**: various techniques for generating images from noise during *inference* as well as to generate noisy images for *training*.

Training a diffusion model looks something like this:

1. Load in some images from the training data
2. Add noise, in different amounts.
3. Feed the noisy versions of the inputs into the model
4. Evaluate how well the model does at denoising these inputs
5. Use this information to update the model weights, and repeat

Step 1: Download a training dataset

For this project, we'll use a dataset of images from the Hugging Face Hub.

Specifically, this collection of 1000 butterfly pictures: [huggan/smithsonian_butterflies_subset · Datasets at Hugging Face]

Step 2: Define the Scheduler

Our plan for training is to take these input images and add noise to them, then feed the noisy images to the model. And during inference, we will use the model predictions to iteratively remove noise. In `diffusers`, these processes are both handled by the **scheduler**.

The noise schedule determines how much noise is added at different timesteps. Here's how we might create a scheduler using the default settings for 'DDPM' training and sampling (based on the paper "Denoising Diffusion Probabilistic Models"):

Noise Scheduling:

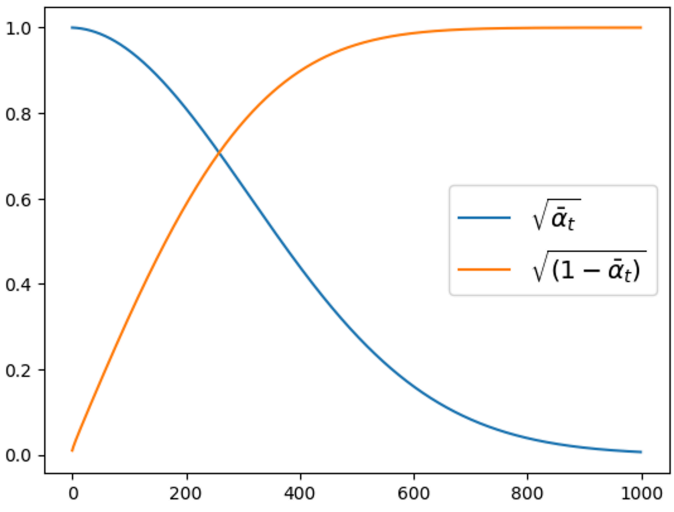
- The `DDPMScheduler` class manages both adding noise during training and removing it during inference.
- It defines a noise schedule that determines how much noise is added for each "timestep" of the diffusion process.
- The default schedule for "DDPM" training is used here, with 1000 timesteps.

Noisy Images:

- The corruption process iteratively adds noise to an image until it becomes significantly distorted.
- Each timestep, it takes the previous noisy version (`x_t-1`) and:
    - **Scales it down:** Multiplies by `sqrt(1 - beta_t)`, where `beta_t` comes from the schedule.
    - **Adds scaled noise:** Adds noise (e.g., random pixels) multiplied by `sqrt(beta_t)`.
- This formula is applied repeatedly from `t=1` to `t=T` to get the final noisy image.

Visualizing Noise Schedule:

- The code plots two curves to show how the scaling factors for image and noise change over time:
    - `sqrt(alpha_prod_t)` represents the cumulative scaling of the image across timesteps.
    - `sqrt(1 - alpha_prod_t)` represents the cumulative scaling of the noise added.



- This visualization helps understand how the model gradually transitions from the noisy input to the denoised final image.

Adding Noise:

- The `noise_scheduler.add_noise` function adds noise to a given image (`xb`) at specified timesteps (`timesteps`).

- This creates a sequence of noisy images, representing the intermediate stages of the denoising process.

Overall, this section defines and uses the scheduler to manage noise addition and removal for diffusion models, allowing for gradual denoising and better understanding of the model's behavior at different steps.

Step 3: Define the Model

Model Architecture:

- U-Net: The core model is a U-Net architecture, commonly used in diffusion models.
- Encoder-Decoder: It encodes the noisy image through several downsampling blocks with ResNet layers, then decodes it back to the original size through upsampling blocks.
- Skip Connections: Connections between corresponding encoder and decoder layers preserve relevant features for accurate reconstruction.
- Unet2DModel from Diffusers: This convenient class handles creating the desired U-Net architecture in PyTorch.

Step 4: Create a Training Loop

Training Loop:

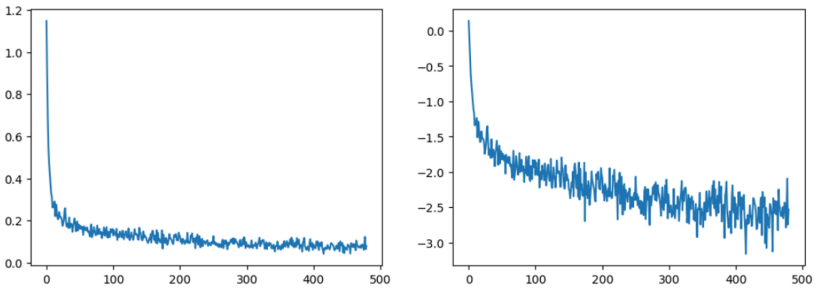
- Standard PyTorch optimization loop with AdamW optimizer and MSE loss.

- Each iteration:

    - Sample random timesteps for noise scheduling.
    - Add noise to clean images according to timesteps.
    - Feed noisy images to the model.
    - Compare model prediction (noise) with target noise using MSE loss.
    - Backpropagate and update model parameters with optimizer.

- Finally we Log and plot losses over time.

```
Epoch:5, loss: 0.15121025778353214
Epoch:10, loss: 0.10957387881353498
Epoch:15, loss: 0.09144688723608851
Epoch:20, loss: 0.08069301722571254
Epoch:25, loss: 0.06832898547872901
Epoch:30, loss: 0.07114456943236291
```



Alternative:

- Load a pre-trained model from the provided pipeline to bypass training.

Overall, this section demonstrates setting up and training a U-Net diffusion model using Diffusers and its UNet2DModel class.

Step 5: Generate Images

We explore two options for generating images using a trained diffusion denoising model:

Option 1: Creating a Pipeline:

1. **DDPMPipeline:** Uses the `DDPMPipeline` class from Diffusers to create a pipeline object that encapsulates the trained model (`unet`) and noise scheduler (`noise_scheduler`).
2. **Image Generation:** Calling the pipeline (`image_pipe()`) automatically generates denoised images from random noise, and returns the results.
3. **Saving and Sharing:** You can save the pipeline to a local folder (`"my_pipeline"`) for later use or upload it to the Hugging Face Hub for sharing with others.
4. **Pipeline Contents:** The saved folder contains subfolders for the scheduler and model, including their weights and configuration files. These files can be used to recreate the pipeline and generate images.

Option 2: Writing a Sampling Loop:

1. **Manual Sampling Loop:** This option demonstrates the underlying process of denoising through the model and scheduler.
2. **Random Start:** Starts with random noise as the initial image.
3. **Iterative Denoising:** Loops through the noise scheduler timesteps (most to least noisy).
4. **Model Prediction:** At each step, the model predicts the residual noise remaining in the current image.
5. **Noise Removal:** Uses the `noise_scheduler.step()` function to update the image by removing the predicted noise based on the current time step.
6. **Final Image:** The result after iterating through all timesteps is the denoised image.

The below images are some samples of the results which we get:

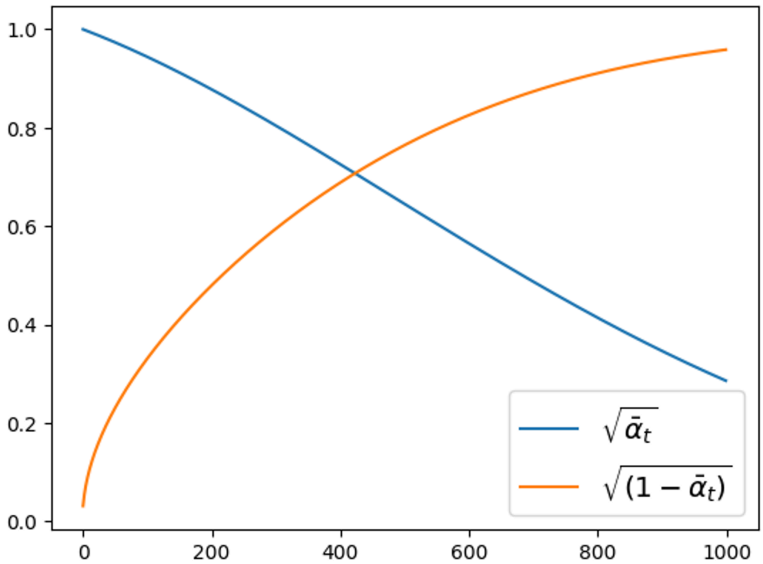
The above are the results which we get when we use a default schedule for "DDPM" training, with 1000 timesteps.

Different Schedulers

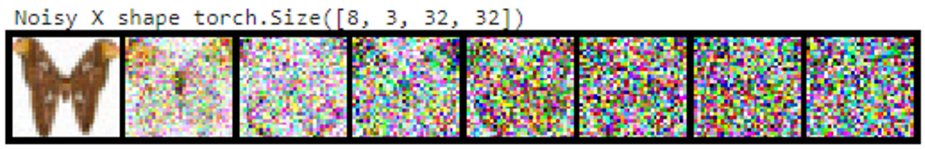
Different Schedulers by adjusting the scheduler settings by using `beta_start` and `beta_end` control the minimum and maximum noise level added at each time step.

- `beta_start`=0.001 `beta_end`=0.004
    - Results of the graph which how the scaling factors for image and noise change over time:

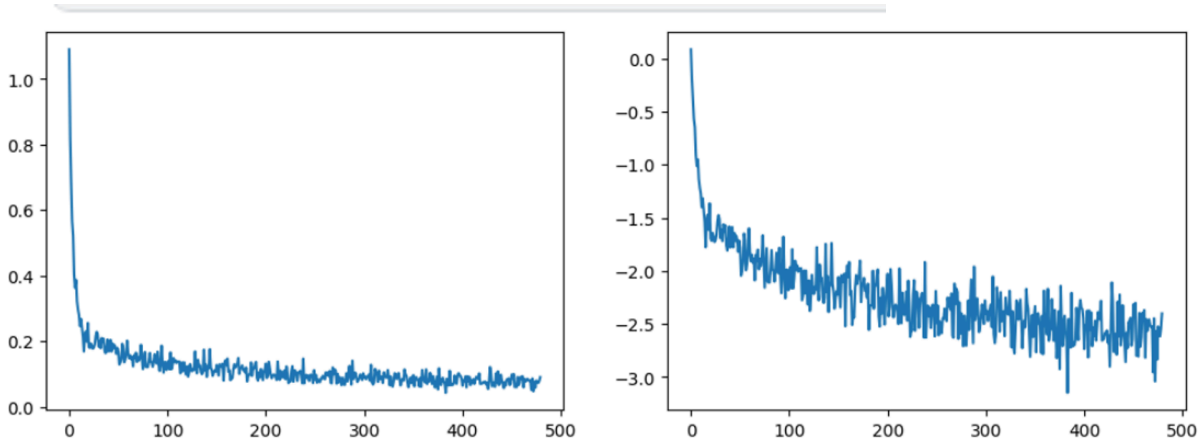
      This visualization helps understand how the model gradually transitions from the noisy input to the denoised final image.



- results of sequence of noisy images, representing the intermediate stages of the denoising process.



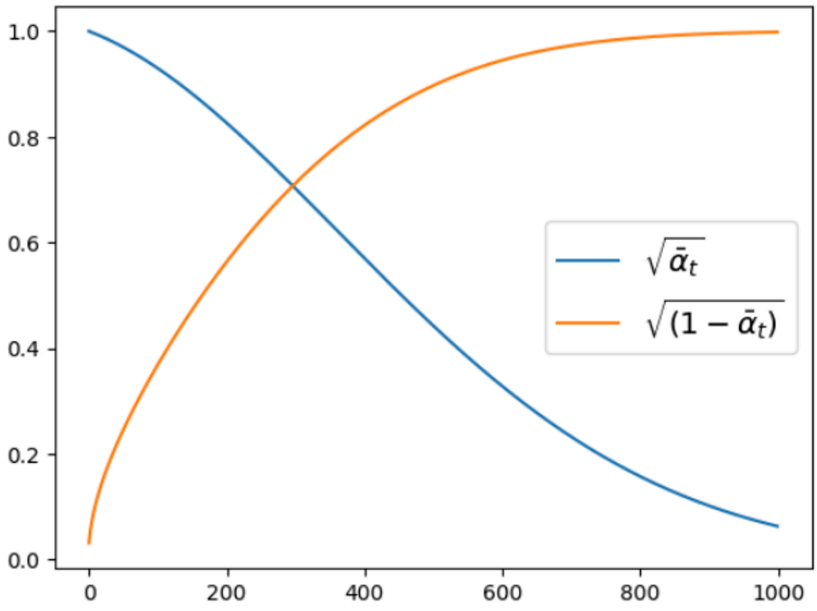
- Results of the losses in the training loop over time



- The below images are some samples of the final results which we get:

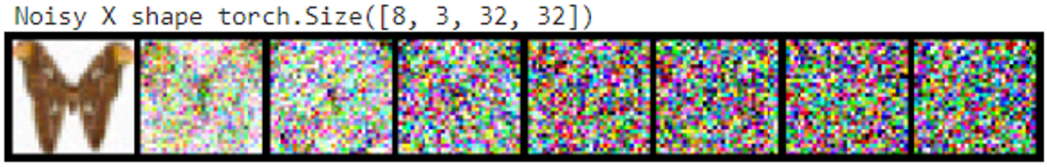
- `beta_start`=0.001 `beta_end`=0.01
  - Results of the graph which how the scaling factors for image and noise change over time:

    This visualization helps understand how the model gradually transitions from the noisy



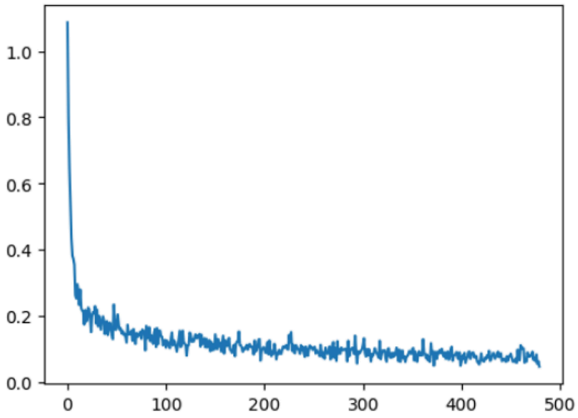
input to the denoised final image.

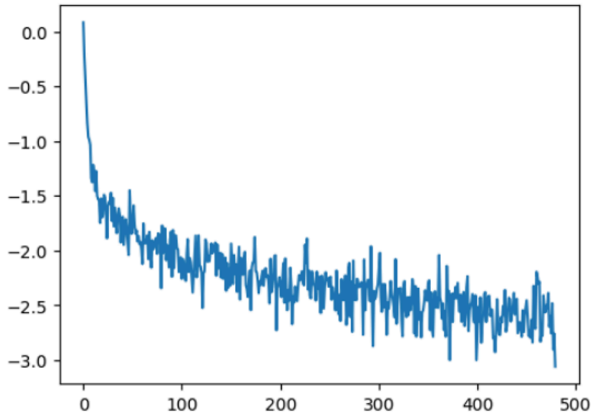
- results of sequence of noisy images, representing the intermediate stages of the denoising process.



- Results of the losses in the training loop over time

```
Epoch:5, loss: 0.14045090414583683
Epoch:10, loss: 0.11292926874011755
Epoch:15, loss: 0.10698695108294487
Epoch:20, loss: 0.08738408843055367
Epoch:25, loss: 0.08001358178444207
Epoch:30, loss: 0.07121107610873878
```

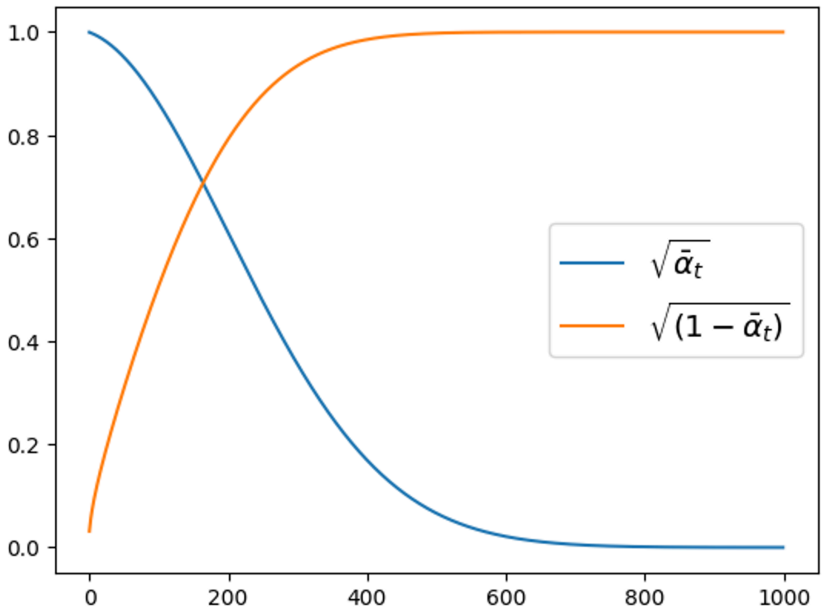




- The below images are some samples of the final results which we get:

- `beta_start`=0.001 `beta_end`=0.04
  - Results of the graph which how the scaling factors for image and noise change over time:

    This visualization helps understand how the model gradually transitions from the noisy input to the denoised final image.

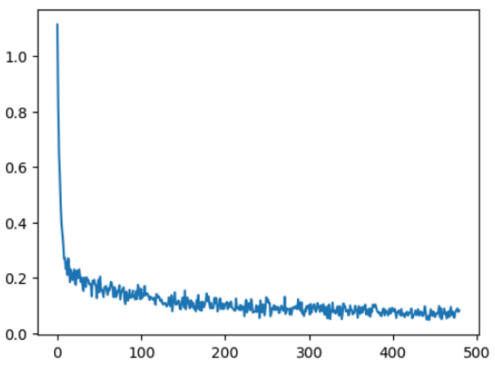


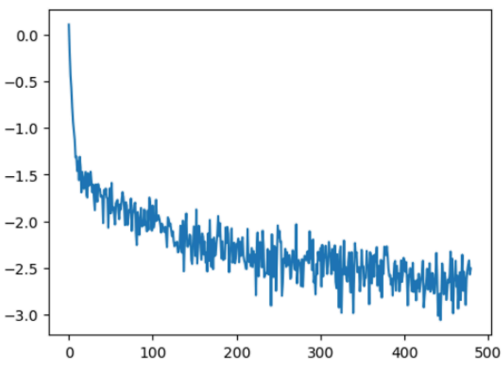
- results of sequence of noisy images, representing the intermediate stages of the

denoising process.

- Results of the losses in the training loop over time

```
Epoch:5, loss: 0.1538833174854517
Epoch:10, loss: 0.11005082353949547
Epoch:15, loss: 0.09224994527176023
Epoch:20, loss: 0.0885396241210401
Epoch:25, loss: 0.07565561309456825
Epoch:30, loss: 0.07354211015626788
```





- The below images are some samples of the final results which we get:

Qualitative Analysis:

When we increase the beta_end from 0.001 to 0.04 while keeping beta_start constant at 0.001 in a diffusion model, you observe the following changes:

Noise Graphs:

- **Higher peak noise:** The graph showing the amount of noise added at each timestep (sqrt_alpha_prod over timesteps) has a higher peak, meaning more noise will be added at some point during the diffusion process. This is because `beta_end` defines the maximum noise level added, and increasing it directly introduces more noise.
- **Faster noise decay:** The graph showing the remaining noise (sqrt(1 - alpha_prod) over timesteps) will decay faster initially. This is because the higher `beta_end` leads to larger steps in noise removal during early timesteps.

Training Losses:

- **Higher initial losses:** During training, the loss will likely be lower initially but will be higher after a certain point compared to lower `beta_end` values. This is because the model has to learn to denoise images with significantly more noise at some point.
- **Faster convergence:** The training has converged faster with higher `beta_end`. This is because the larger initial noise variations can provide more information for the model to learn from. However, it can also lead to instability or difficulties in fine-tuning the denoising process.

Conditioning

Guidance is a great way to get some additional mileage from an unconditional diffusion model, but if we have additional information (such as a class label or an image caption) available during training then we can also feed this to the model for it to use as it makes its predictions. In doing so, we create a conditional model, which we can control at inference time by controlling what is fed in as conditioning. The Conditional Sampling notebook shows an example of a class-conditioned model which learns to generate images according to a class label.

There are a number of ways to pass in this conditioning information, such as

- Feeding it in as additional channels in the input to the UNet. This is often used when the conditioning information is the same shape as the image, such as a segmentation mask, a depth map or a blurry version of the image (in the case of a restoration/superresolution model). It does work for other types of conditioning too. For example, in the notebook, the class label is mapped to an embedding and then expanded to be the same width and height as the input image so that it can be fed in as additional channels.
- Creating an embedding and then projecting it down to a size that matches the number of channels at the output of one or more internal layers of the UNet, and then adding it to those outputs. This is how the timestep conditioning is handled, for example. The output of each Resnet block has a projected timestep embedding added to it. This is useful when you have a vector such as a CLIP image embedding as your conditioning information.
- Adding cross-attention layers that can 'attend' to a sequence passed in as conditioning. This is most useful when the conditioning is in the form of some text - the text is mapped to a sequence of embeddings using a transformer model, and then cross-attention layers in the UNet are used to incorporate this information into the denoising path. We'll see this in action in Unit 3 as we examine how Stable Diffusion handles text conditioning.

❖ DDPM Vs DDIM (sampling speed):

Link to code:

DDIM = Denoising Diffusion Implicit Models

DDPM = Denoising Diffusion Probabilistic Models

Here our focus is on how to use a pre-trained DDPM pipeline for image generation and then the need for sampling optimization.

Loading the Pre-Trained Pipeline:

1. `DDPMPipeline.from_pretrained`: This line loads a pre-trained DDPM pipeline for generating images of faces from the "google/ddpm-celebahq-256" model.
2. `.to(device)`: This moves the pipeline to the desired device (CPU or GPU) for faster processing.

Generating Images:

1. `__call__ method`: The pipeline can be used like a function by calling its `__call__` method.
2. `images = image_pipe().images`: This generates a batch of images and stores them in the `images` variable.
3. `images[0]`: You can access individual images from the batch using indexing (e.g., `images[0]` for the first image).

DDIM (Denoising Diffusion Implicit Models) is a valuable addition to the field of generative models and image denoising.

Here are some reasons why DDIM is required:

1. **Efficiency**:
    - DDIMs prioritize computational efficiency by achieving good-quality samples with significantly fewer steps compared to traditional methods like DDPMs.
    - In scenarios where computational resources are limited, DDIMs provide a practical solution.
2. **Faster Sampling**:
    - DDIMs allow faster sampling by using an iterative process that goes “backwards” in time from high noise to low noise.

    - The scheduler manages the noise schedule during sampling, resulting in quicker convergence.
3. **Reduced Complexity**:
    - Traditional Markov chain-based methods (such as DDPMs) require a large number of steps (e.g., 1000 steps) to explore the joint distribution thoroughly.
    - DDIMs achieve comparable results with significantly fewer steps (e.g., 40 steps), simplifying the sampling process.

In summary, DDIMs strike a balance between quality and efficiency, making them a valuable tool for denoising and generating high-quality samples in a resource-efficient manner.

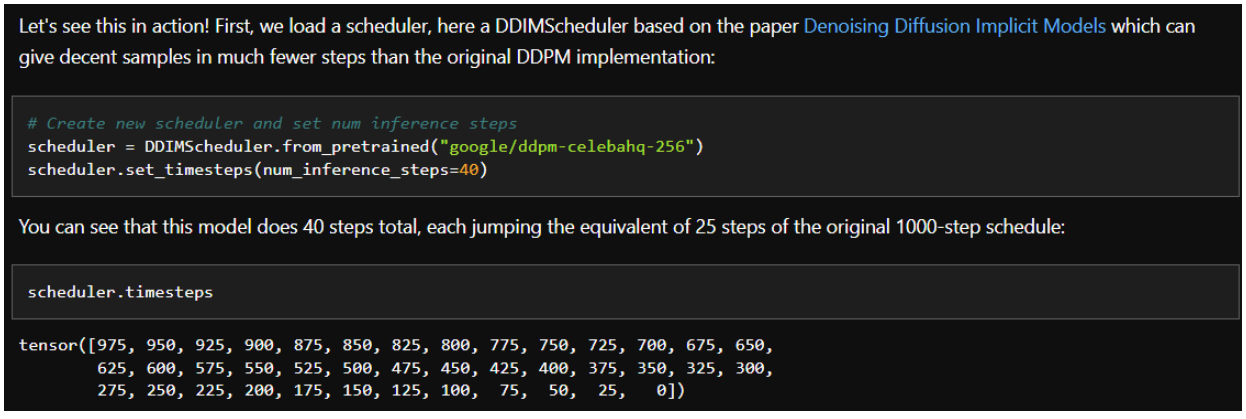
Let's explore the differences between **DDIM** and **DDPM**:

1. **Model Type**:
    - **DDIM**:
        - They use an iterative process to predict noise and estimate fully denoised images.
        - The goal is to achieve high-quality samples with computational efficiency.
    - **DDPM**:
        - They follow a probabilistic approach based on Markov chain sampling.
        - DDPMs aim to model the joint distribution of noisy and clean images.
2. **Sampling Process**:
    - **DDIM**:
        - DDIMs perform a reverse process, going from high noise to low noise.
        - They use fewer steps (e.g., 40 steps) compared to DDPMs.
        - The scheduler manages the noise schedule during sampling.
    - **DDPM**:
        - DDPMs follow a forward diffusion process, starting from a clean image.
        - They typically require more steps (e.g., 1000 steps) for accurate sampling.
        - Markov chain-based methods guide the sampling.
3. **Efficiency vs. Accuracy**:
    - **DDIM**:
        - Prioritizes efficiency by achieving good samples with fewer steps.
        - Suitable for scenarios where computational resources are limited.
    - **DDPM**:
        - Emphasizes accuracy by thoroughly exploring the joint distribution.
        - May be computationally expensive but provides high-quality samples.

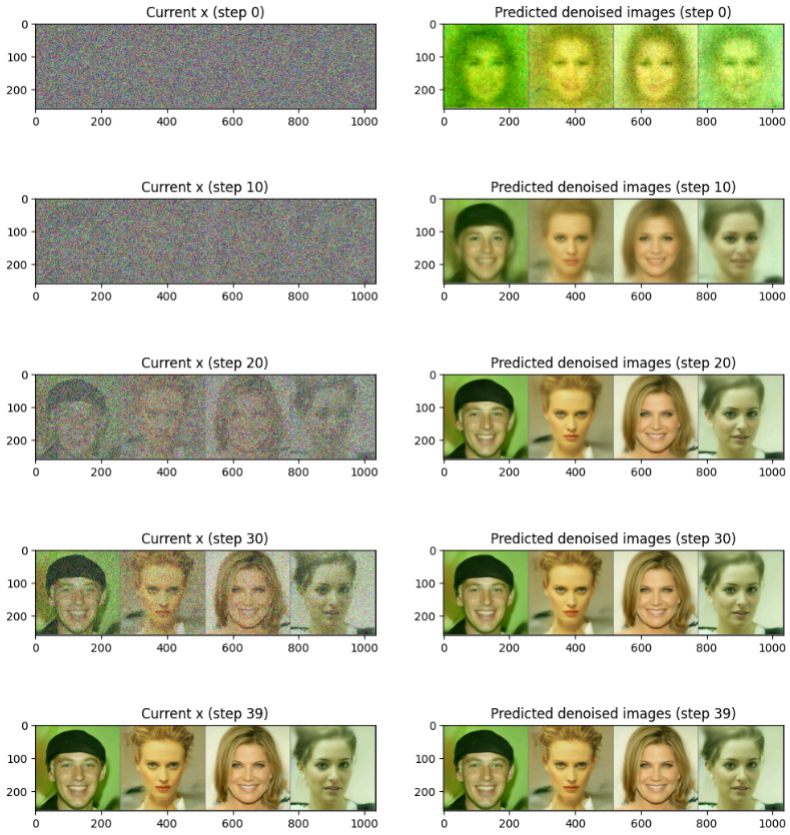
In summary, DDIMs offer faster sampling with fewer steps, while DDPMs provide a more comprehensive exploration of the joint distribution. Both approaches contribute to the field of generative models and denoising techniques.

Process overview of Faster Sampling with DDIM:

1. **Objective**: The goal is to generate high-quality samples efficiently using the Denoising Diffusion Implicit Model (DDIM).
2. **Input Noise Prediction**: At each step, the model receives a **noisy input** and predicts the noise. Essentially, it estimates what the fully denoised image might look like.
3. **Iterative Process**: Initially, these predictions may not be accurate. To improve them, we break down the process into multiple steps. However, recent research has shown that using over 1000 steps is unnecessary.
4. **Scheduler and Step Function**: In the Huggingface Diffusers library, sampling methods are managed by a **scheduler**. This scheduler updates the model via the `step()` function.
5. **Image Generation**: To create an image:
    - Start with **random noise**.
    - For each timestep in the scheduler's noise schedule:
        - Feed the noisy input to the model.
        - Pass the resulting prediction to the `step()` function.
        - The output includes a `prev_sample` attribute, representing the denoised image at that step.
    - This process goes "backwards" in time from high noise to low noise (opposite of forward diffusion).



We generate 4 random images and run through the sampling loop, and visualize both the current and the predicted denoised version as the process progresses as below:



As you can see, the initial predictions are not great but as the process goes on the predicted outputs get more and more refined.

❖ Class conditioned diffusion model:

Link to code:

- **Objective:**
  - ➢ The goal of this notebook was to create a class-conditioned diffusion model using the MNIST dataset.
  - ➢ The model will generate images of handwritten digits (0-9) while being conditioned on a specific digit class.

Technique for incorporating class information into a UNet model, called "class-conditioned UNet:

- **Class-Conditioning Approach:** We achieve class conditioning in the following manner:
  - ➢ Create a Standard UNet2DModel:
    - Start with a standard UNet architecture, which is commonly used for image segmentation tasks.
    - UNet consists of an encoder (downsampling path) and a decoder (upsampling path).
  - ➢ Map Class Label to a Learned Vector:
    - Use an embedding layer to map the class label (digit) to a learned vector.
    - This vector has a shape of `(class_emb_size)`.
  - ➢ Concatenate Class Information:
    - Concatenate the class embedding vector with the input data.
    - The resulting input has `(class_emb_size + 1)` channels.
  - ➢ Feed Input into UNet:
    - Pass this modified input (with class information) through the UNet.
    - The UNet processes it to generate the final prediction (image).
- **Class-Conditioned UNet Architecture:**
  - ➢ The `ClassConditionedUnet` class is defined, inheriting from `nn.Module`.
  - ➢ It has the following components:

        - `self.class_emb`: An embedding layer that maps the class label (digit) to a vector of size `class_emb_size`.
        - `self.model`: An unconditional UNet with additional input channels for class conditioning.
            - The UNet architecture consists of an encoder (downsampling path) and a decoder (upsampling path).
            - The class embedding is concatenated with the input before feeding it into the UNet.

- **Forward Method:**
    - The `forward` method now takes the class labels as an additional argument.
    - The input data `x` has shape `(bs, 1, 28, 28)` (batch size, channels, width, height).
    - Class conditioning information is prepared:
        - The class embedding is obtained by mapping the class labels using `self.class_emb`.
        - The class embedding is reshaped to match the input dimensions.
        - The class embedding is expanded to have the same spatial dimensions as the input.
    - The net input is formed by concatenating the input data `x` and the class conditioning (`class_cond`) along dimension 1.
        - The resulting net input has shape `(bs, 5, 28, 28)`.
- **Usage:**
    - During training or inference, we would pass the input data, class labels, and other relevant information to this class to obtain predictions.
- This approach helps the UNet specialize in segmenting images based on their class.

Implementation:

This is what the implementation looks like:

In [ ]:

```python
class ClassConditionedUnet(nn.Module):
  def __init__(self, num_classes=10, class_emb_size=4):
    super().__init__()

    # The embedding layer will map the class label to a vector of size class_emb_size
    self.class_emb = nn.Embedding(num_classes, class_emb_size)

    # Self.model is an unconditional UNet with extra input channels to accept the conditioning information (the class embed
    self.model = UNet2DModel(
        sample_size=28,           # the target image resolution
        in_channels=1 + class_emb_size, # Additional input channels for class cond.
        out_channels=1,           # the number of output channels
        layers_per_block=2,       # how many ResNet layers to use per UNet block
        block_out_channels=(32, 64, 64),
        down_block_types=(
            "DownBlock2D",        # a regular ResNet downsampling block
            "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
            "AttnDownBlock2D",
        ),
        up_block_types=(
            "AttnUpBlock2D",
            "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
            "UpBlock2D",          # a regular ResNet upsampling block
          ),
    )

  # Our forward method now takes the class labels as an additional argument
  def forward(self, x, t, class_labels):
    # Shape of x:
    bs, ch, w, h = x.shape

    # class conditioning in right shape to add as additional input channels
    class_cond = self.class_emb(class_labels) # Map to embedding dimension
    class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
    # x is shape (bs, 1, 28, 28) and class_cond is now (bs, 4, 28, 28)

    # Net input is now x and class cond concatenated together along dimension 1
    net_input = torch.cat((x, class_cond), 1) # (bs, 5, 28, 28)

    # Feed this to the UNet alongside the timestep and return the prediction
    return self.model(net_input, t).sample # (bs, 1, 28, 28)
```

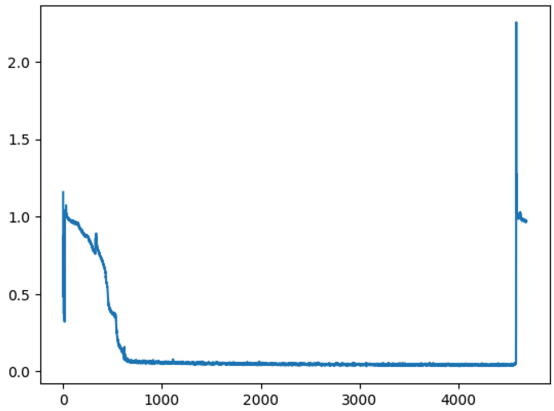
Details the training process and subsequent sampling of the trained model:

1. **Training and Sampling**:
    - Previously, we used to predict denoised images using a diffusion model with the command `prediction = unet(x, t)`.
    - Now, we're adding an additional argument to the prediction process during training:
        - `y`: These are the correct labels (class information) corresponding to the MNIST digits (values from 0 to 9).

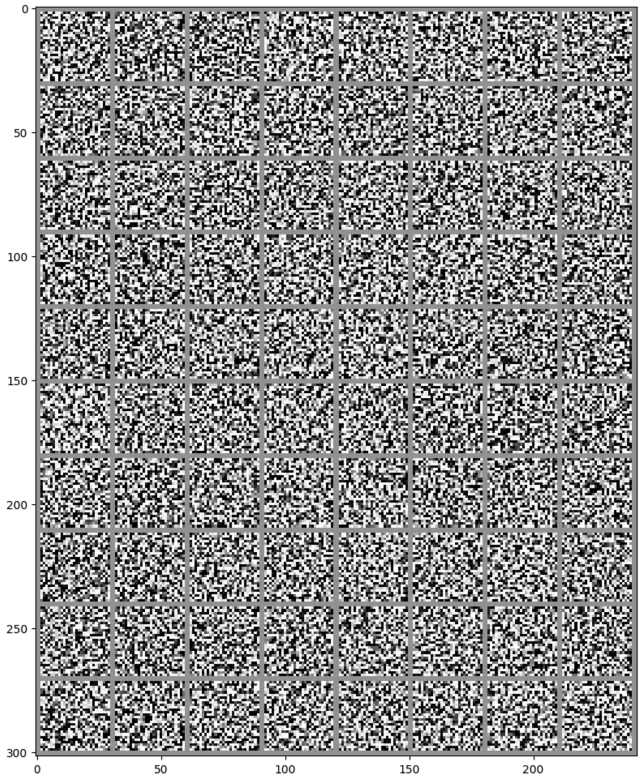
        - So, during training, the prediction becomes `prediction = unet(x, t, y)`.
    - At inference time, we can pass any class labels we want, and if everything works correctly, the model should generate images that match the specified class.
    - Essentially, we're conditioning the model on specific class labels to guide its image generation.
2. **Training Loop**:
    - However, there's a key difference to the normal training loop:
        - Instead of predicting the denoised image, we're now predicting the **noise**.
        - This aligns with the objective expected by the default `DDPMScheduler`.
        - The scheduler adds noise during training and generates samples at inference time.
    - Note that training can take some time, but you can skim the code without running it since we're primarily illustrating the concept.
3. **Scheduler**:
    - The `noise_scheduler` is created with the following parameters:
        - `num_train_timesteps`: The number of training steps (in this case, 1000).
        - `beta_schedule`: The schedule for the beta parameter (used in the diffusion process).

The results obtained are as below:

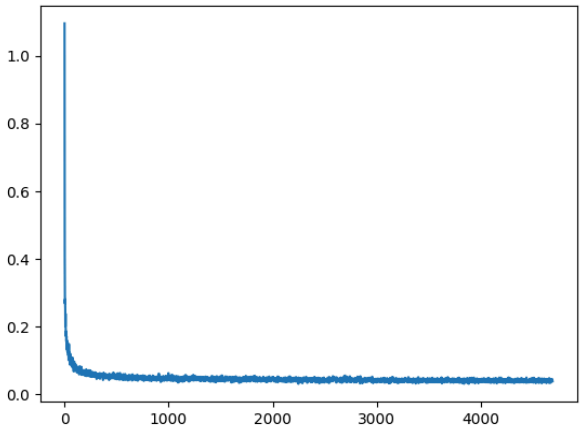
- learning_rate= 0.002, batch_size= 128, no_of_epocs=10
    - Finished epoch 0. Average of the last 100 loss values: 0.628539
    - Finished epoch 5. Average of the last 100 loss values: 0.044570
    - Finished epoch 9. Average of the last 100 loss values: 0.990907
    - Training Loss curve:



Results of the image generated by sampling for different digits conditionally:

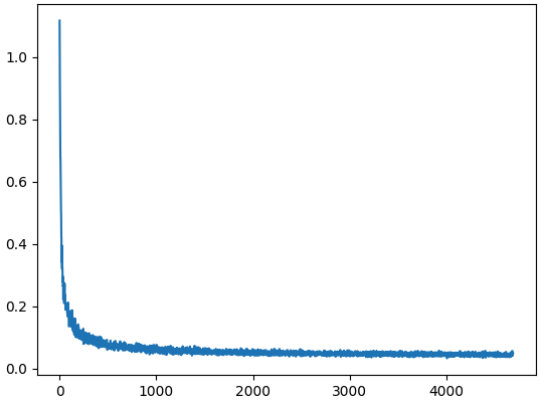


- learning_rate= 0.003, batch_size= 128, no_of_epocs=10
    - Finished epoch 0. Average of the last 100 loss values: 0.051552
    - Finished epoch 5. Average of the last 100 loss values: 0.040603
    - Finished epoch 9. Average of the last 100 loss values: 0.039175
    - Training Loss curve:

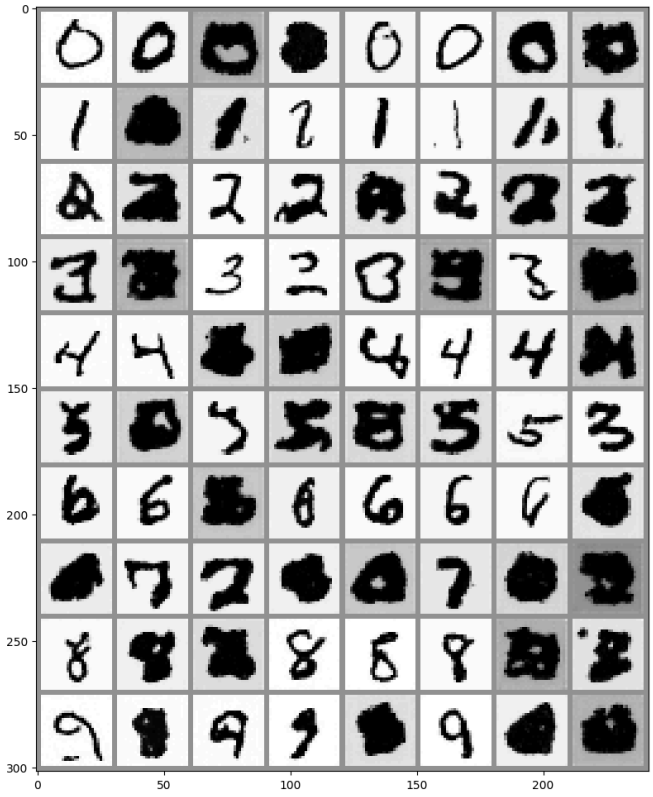


Results of the image generated by sampling for different digits conditionally:

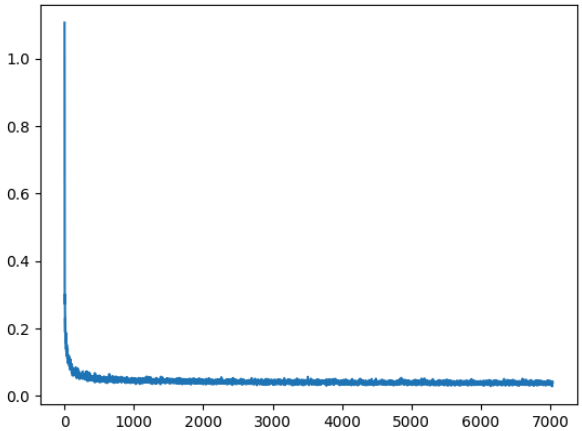
- learning_rate= 0.004, batch_size= 128, no_of_epocs=10
    - Finished epoch 0. Average of the last 100 loss values: 0.081995
    - Finished epoch 5. Average of the last 100 loss values: 0.047996
    - Finished epoch 9. Average of the last 100 loss values: 0.043310
    - Training Loss curve:



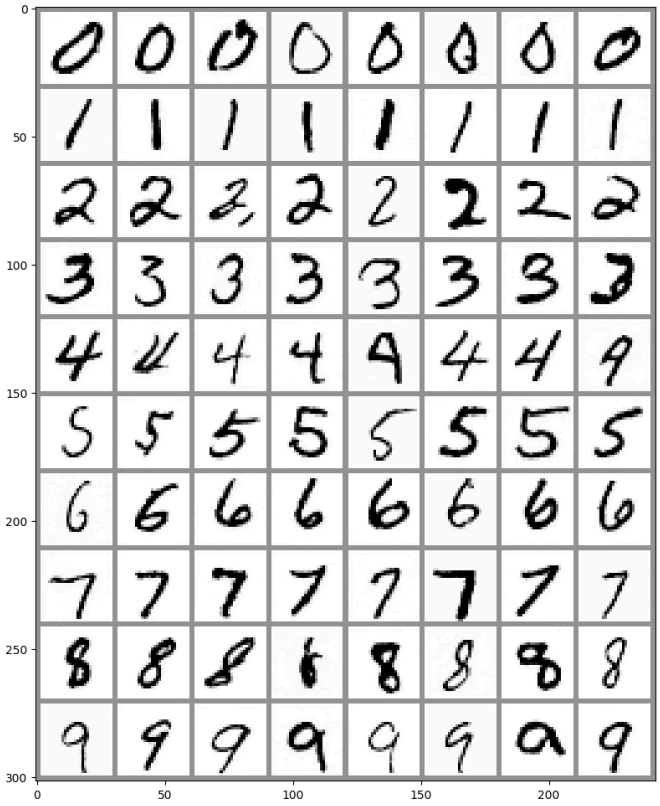
Results of the image generated by sampling for different digits conditionally:



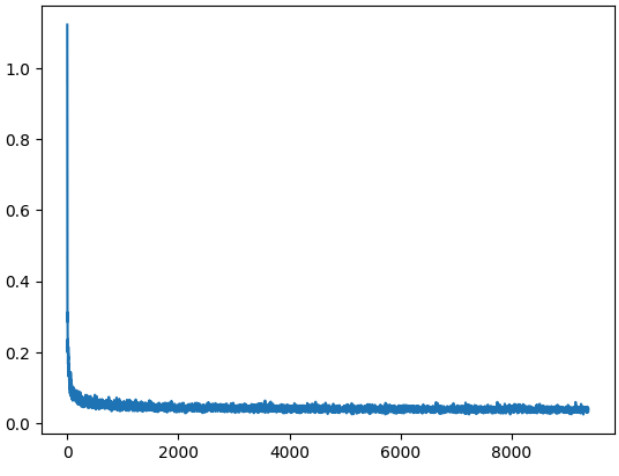
- learning_rate= 0.003, batch_size= 128, no_of_epocs=15
    - Finished epoch 0. Average of the last 100 loss values: 0.052219
    - Finished epoch 10. Average of the last 100 loss values: 0.038881
    - Finished epoch 14. Average of the last 100 loss values: 0.038116
    - Training Loss curve:



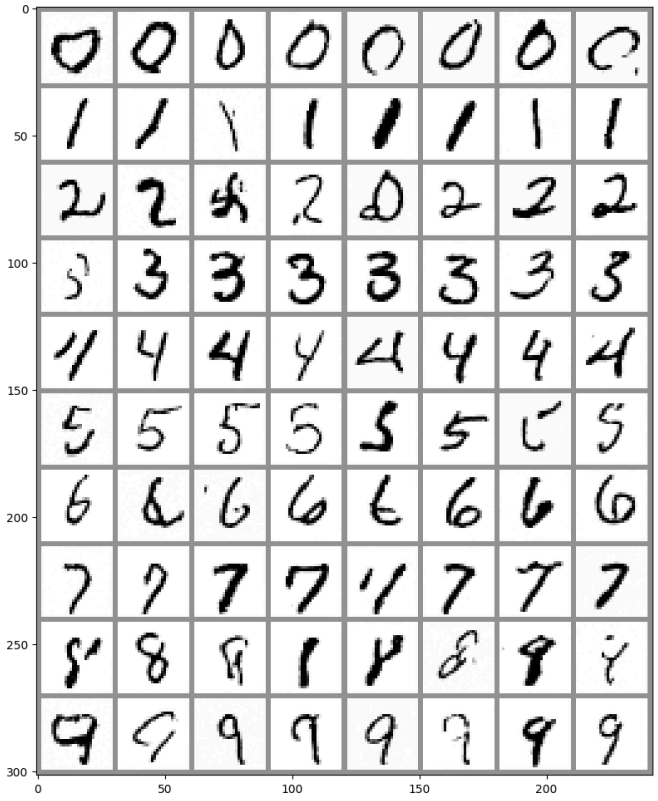
Results of the image generated by sampling for different digits conditionally:



- learning_rate= 0.003, batch_size= 64, no_of_epocs=10
    - Finished epoch 0. Average of the last 100 loss values: 0.049489
    - Finished epoch 5. Average of the last 100 loss values: 0.039875
    - Finished epoch 9. Average of the last 100 loss values: 0.038359
    - Training Loss curve:



Results of the image generated by sampling for different digits conditionally:



Quantitative Analysis:

- Increasing the batch size from 64 to 128 while keeping other parameters constant results in the following observations:
    - Loss curve: The shape of the loss curve becomes smoother and converges faster, which suggests reduced variance and better training stability.
    - Training time: The training time decreases significantly, which suggests the positive effect of faster convergence is dominant.
- Increasing the learning rate from 0.02 to 0.03 and 0.04 while keeping other parameters constant results in the following observations:
    - Convergence: A higher learning rate allows the model to take larger steps in the parameter space, leading to faster convergence and quicker training times, especially in the initial stages. But on increasing the learning rate further, the model updates can become too large, causing the loss to start fluctuating dramatically or even diverging altogether. This indicates instability and difficulty in finding the optimal parameters.

    - Loss reduction: Initially, a higher learning rate might result in more significant reductions in the loss function as the model rapidly descends the error landscape but on increasing the learning rate further increases the loss as the model becomes insatiable.
- Increasing the number of epochs from 10 to 15 while keeping other parameters constant results in the following observations:
    - Loss curve: The loss curve becomes smoother as the model has more opportunities to adjust its parameters and refine its predictions.
    - Improved Loss and Accuracy: If the model hasn't fully converged by epoch 10, increasing the epochs allows it to further optimize its parameters, leading to lower training loss and potentially higher validation accuracy.

❖ Stable diffusion:

This notebook covers the usage of Stable Diffusion to create and modify images using existing pipelines. Some of the key ideas explored within this notebook are as follows:

- Generating images from text using the StableDiffusionPipeline and experimenting with the available arguments
- Exploring some of the key pipeline components in action
  - The VAE that makes this a 'latent diffusion model'
  - The tokenizer and text encoder that process the text prompt
  - The UNet itself
  - The scheduler, and exploring different schedulers
- Replicating the **sampling loop** with the pipeline components
- Editing existing images with the **Img2Img** pipeline
- Using **inpainting** and **Depth2Img** pipelines

Generating Images from Text

- The stable diffusion pipeline which has been used for this purpose has been taken from https://huggingface.co/stabilityai/stable-diffusion-2-1-base.

```
pipe_output = pipe(
    prompt="Palette knife painting of an autumn cityscape", # What to generate
    negative_prompt="Oversaturated, blurry, low quality", # What NOT to generate
    height=480, width=640,      # Specify the image size
    guidance_scale=8,           # How strongly to follow the prompt
    num_inference_steps=35,     # How many steps to take
    generator=generator         # Fixed random seed
)
```

- The settings and prompt above can be tweaked to see how it affects the output of the pipeline
- Some of the Key arguments to tweak are:
  - ❖ **Width and height** specify the size of the generated image. They must be divisible by 8 for the VAE to work
  - ❖ The **number of steps** influences the generation quality. The default (50) works well.
  - ❖ The **negative prompt** is used during the classifier-free guidance process, and can be a useful way to add additional control.
  - ❖ The **guidance_scale** argument determines how strong the classifier-free guidance (CFG) is. Higher scales push the generated images to better match the prompt, but if the scale is too high the results can become **over-saturated** and unpleasant.

:

- The various parameters shown above have been tinkered with and the resultant changes in images generated have been captured as follows:

❖ **Removing Negative Prompt:**

Original Image: **New Image:**


The new image has more artifacts, and is a little blurrier compared to the original image due to removing the negative prompt argument.

❖ **Lowering Guidance Scale from 8 to 2:**

Original Image: **New Image:**


It is evident that by lowering the guidance scale, the prompt is not strictly followed and the new image is distorted.

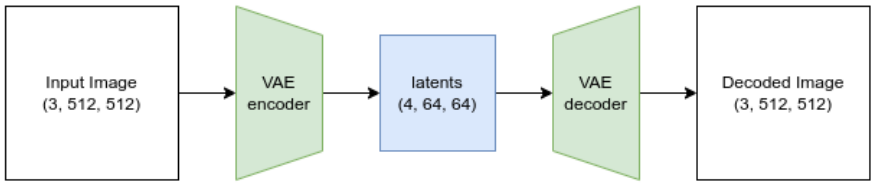
❖ **Reducing the number of inference steps from 35 to 15:**

Original Image: **New Image:**


With fewer inference steps, the model performs fewer refinement iterations, leading to coarser sampling. The generated image exhibits less fine-grained detail and does not capture intricate features as effectively.

Stable Diffusion Pipeline Components:

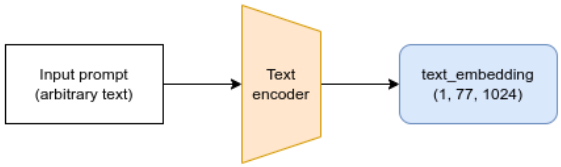
➔ VAE



The 512x512 image is compressed to a 64x64 latent representation (with four channels). This 8x reduction in each spatial dimension is the reason the specified width and height need to be multiples of 8.

Working with these information-rich 4x64x64 latents is more efficient than working with massive 512px images, allowing for faster diffusion models that take less resources to train and use.

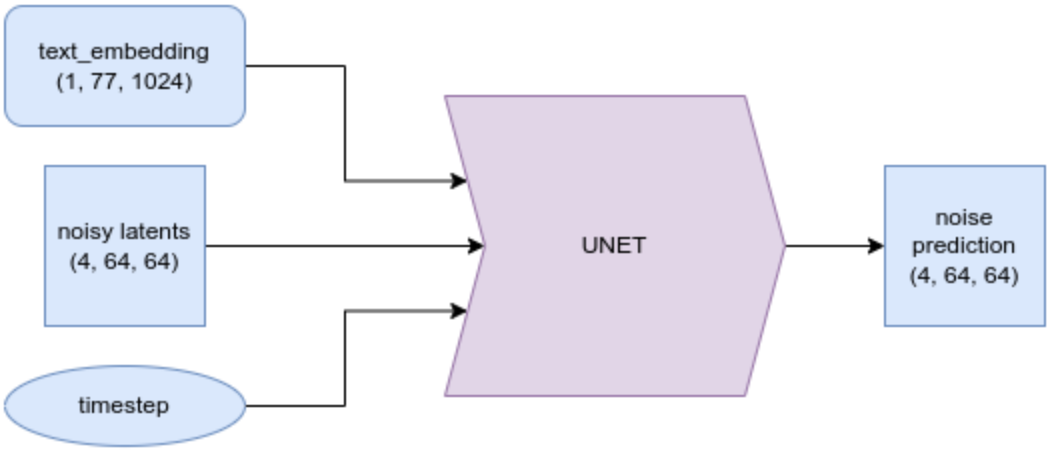
➔ The Tokenizer and Text Encoder



The objective of the text encoder is to convert an input string (the prompt) into a numerical representation suitable for conditioning the UNet. Initially, the text undergoes tokenization using the pipeline's tokenizer, resulting in a series of tokens.

Subsequently, the tokens are processed through the text encoder model itself, which is a transformer model initially trained as the text encoder for CLIP. The expectation is that this pre-trained transformer model has acquired valuable representations of text, which can prove beneficial for the diffusion task as well.

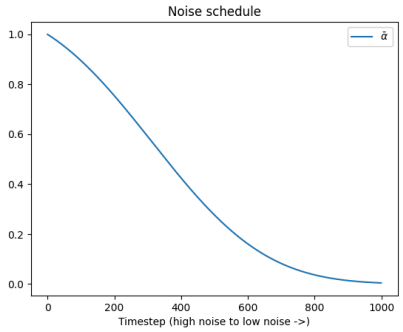
➔ The UNet



The UNet takes a noisy input and predicts the noise. The input is not an image but is instead a latent representation of an image. And in addition to the timestep conditioning, this UNet also takes in the text embeddings of the prompt as an additional input

➔ The Scheduler

We plot the noise schedule to see the noise level over time:




```
Scheduler config: LMSDiscreteScheduler {
  "_class_name": "LMSDiscreteScheduler",
  "_diffusers_version": "0.24.0",
  "beta_end": 0.012,
  "beta_schedule": "scaled_linear",
  "beta_start": 0.00085,
  "clip_sample": false,
  "num_train_timesteps": 1000,
  "prediction_type": "epsilon",
  "set_alpha_to_one": false,
  "skip_prk_steps": true,
  "steps_offset": 1,
  "timestep_spacing": "linspace",
  "trained_betas": null,
  "use_karras_sigmas": false
}
```

We also experiment with using another scheduler (LMSDiscreteScheduler) in order to generate the same image:

The image generated by the above scheduler is given below:

Additional Pipelines:

- **Img2Img:**

  In the previous instances, we created images entirely from random latents through the complete diffusion sampling loop. However, starting from scratch is not the only option.

  In the Img2Img pipeline, an existing image is initially encoded into a set of latents. Subsequently, some noise is introduced to these latents, serving as the starting point. The extent of noise added and the number of denoising steps applied determine the 'strength' of the img2img process

```
result_image = img2img_pipe(
    prompt="An oil painting of a man on a bench",
    image=init_image, # The starting image
    strength=0.6, # 0 for no change, 1.0 for max strength
).images[0]
```

  The strength parameter represents how closely the final image follows the given prompt, it can be varied from 0 (for no change) to 1 (for max strength).

  In this example, we start off with an image of dog sitting on a bench and generate an oil painting of a man sitting on a bench.

Original Image:

New Image:

Now, we **decrease** the strength hyperparameter from **0.6 to 0.2** and observe the change in the generated image:

Original Image:

New Image:

Clearly, we see that the new image **closely resembles** the starting image and does not follow the prompt strictly since the strength parameter was decreased.

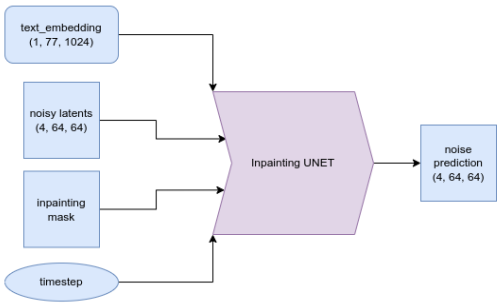
Now, we **increase** the strength hyperparameter from **0.6 to 0.9** and observe the change in the generated image:

Original Image:

New Image:

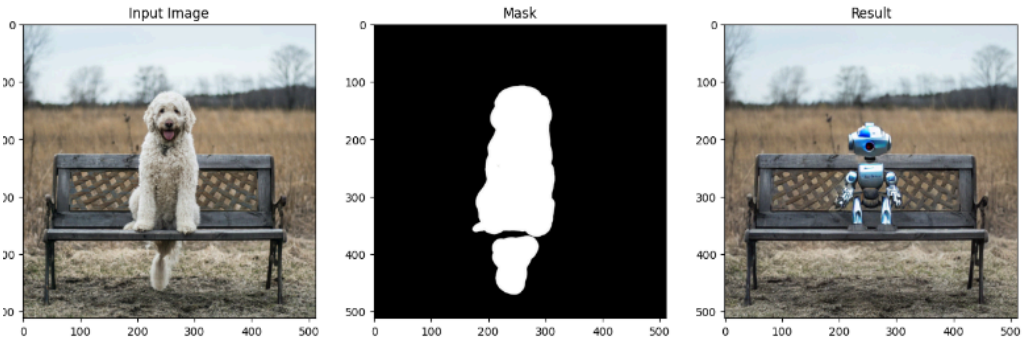
We can see that since the strength parameter has been increased to a large value, the generated image is **completely different** from the input image.

- **In - Painting:**



Consider the scenario where we wish to retain certain portions of the input image unchanged while generating new content in other regions. This process is known as 'inpainting.'

Superior results have been obtained by utilizing a custom fine-tuned version of Stable Diffusion that incorporates a mask as additional conditioning. The mask image should share the same shape as the input image, with white representing the areas to be replaced and black indicating the regions to remain unchanged.

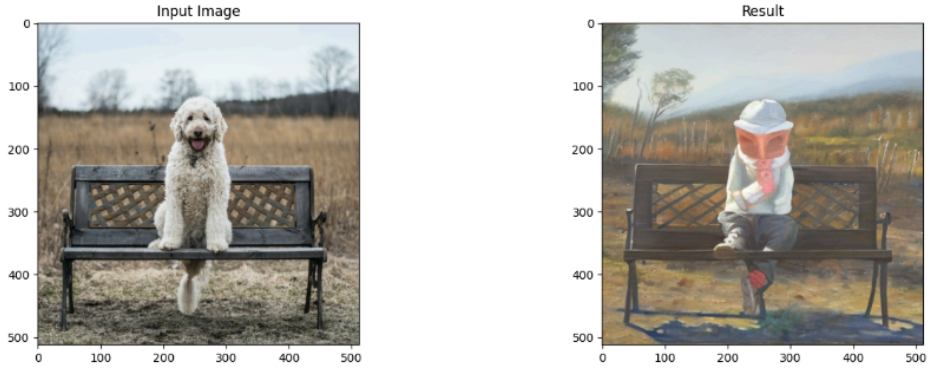


- **Depth2Image:**

  Depth2Image is used to create a new image with the composition of the original but completely different colours or textures.

  It takes in depth information as additional conditioning when generating. The pipeline uses a depth estimation model to create a depth map, which is then fed to the fine-tuned UNet when generating images to (hopefully) preserve the depth and structure of the initial image while filling in completely new content.

  An example of **Depth2Image** using the prompt “An oil painting of a man on a bench”:



❖ Diffusion model for audio generation:

In this notebook, we take a brief look at generating audio with diffusion models.

For the purpose of the audio generation task, we first load a pre-existing audio diffusion model from https://huggingface.co/teticio/audio-diffusion-instrumental-hiphop-256

We can then create audio samples by calling the pipeline as follows:

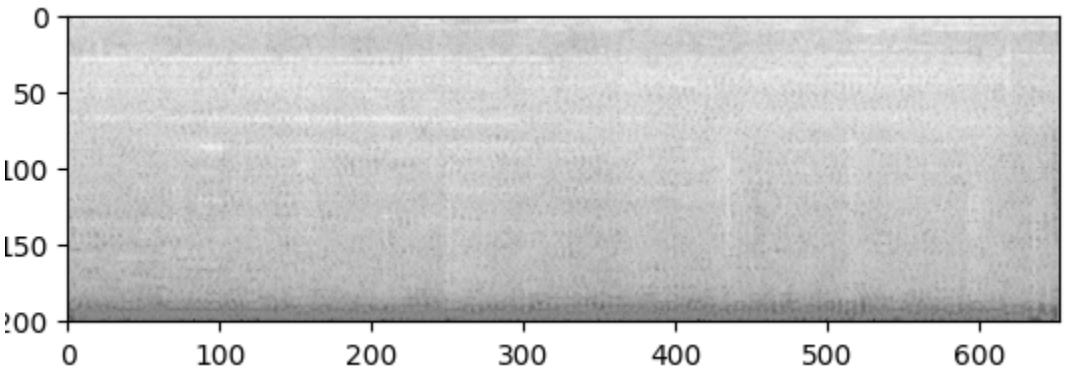
```
# Sample from the pipeline and display the outputs
output = pipe()
display(output.images[0])
display(Audio(output.audios[0], rate=pipe.mel.get_sample_rate()))
```

The audio is not directly generated with diffusion - instead, the pipeline has the same kind of 2D UNet as the unconditional image generation pipelines we saw in the previous sections that is used to generate the **spectrogram**, which is then **post-processed into the final audio.**

The pipe has an extra component that handles these conversions, which we can access via pipe.mel:


```
Mel {
  "_class_name": "Mel",
  "_diffusers_version": "0.24.0",
  "hop_length": 512,
  "n_fft": 2048,
  "n_iter": 32,
  "sample_rate": 22050,
  "top_db": 80,
  "x_res": 256,
  "y_res": 256
}
```


An audio 'waveform' encodes the raw audio samples over time - this could be the electrical signal received from a microphone, for example. Working with this 'Time Domain' representation can be tricky, so it is a common practice to convert it into some other form, commonly something called a spectrogram. A spectrogram shows the intensity of different frequencies (y axis) vs time (x axis):



We convert an array of audio data into spectrogram images by first loading the raw audio data and then calling the **pipe.mel.audio_slice_to_image()** function. Longer clips are automatically sliced into chunks of the correct length to produce a 256x256 spectrogram image.

The audio is represented as a long array of numbers. To play this out loud we need one more key piece of information: The sample rate which describes how many samples (individual values) we use to represent a single second of audio. Increasing the sample rate results in the audio getting sped up.

The dataset which we have used has a collection of audio clips in multiple genres. The genre used for training as well as the batch size are taken as a hyperparameters.

The resultant audio clips have been uploaded in the folder below:

Some observations:

- The audio generated is more crisp and sharp upon increasing the batch size while training.

- The audio generated sounds vastly different depending upon the genre on which the pipeline has been trained on.

- Different audio clips in the training data have different sample rates, Hence resampling needed to be done before training in order to adjust the sampling rate.

References:

1. **https://en.wikipedia.org/wiki/Diffusion_model**

2. **https://pytorch.org/vision/stable/generated/torchvision.datasets.StanfordCars.html**
3. **https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers/stable_diffusion.ipynb#scrollTo=yEErJFjlrSWS**
4. **https://huggingface.co/teticio/audio-diffusion-instrumental-hiphop-256**
5. **https://huggingface.co/stabilityai/stable-diffusion-2-1-base**
6. **https://huggingface.co/docs/diffusers/api/pipelines/audio_diffusion**
7. **https://huggingface.co/keras-io/vq-vae**
8. **https://towardsdatascience.com/understanding-conditional-variational-autoencoders-cd62b4f57bf8**
9. Denoising Diffusion Implicit Models - Introduced the DDIM sampling method (used by DDIMScheduler)
10. GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models - Introduced methods for conditioning diffusion models on text
11. eDiffi: Text-to-Image Diffusion Models with an Ensemble of Expert Denoisers - Shows how many different kinds of conditioning can be used together to give even more control over the kinds of samples generated

12.
