JANUARY 25, 2025

ANN FINAL POJECT

HESAM KOUCHEKI | 403 72 39 87

HESSAM KOUCHEKI
IRAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Table of Contents

Problem 1

VAE

In the first part of the project, we completed and trained the VAE model for generating images from texts using PyTorch. A deeper description is available step by step below.

Parameter Settings

In this project, we set several important parameters for training the model. The latent space has 2 dimensions, which makes it easy to visualize the results. Training runs for 30 epochs, meaning the model goes through the dataset 30 times to learn. A batch size of 128 was chosen but I increased that to 512 for a faster training, increasing speed and a little memory usage.

The model's capacity is set to 64, which controls its complexity without making it too heavy to train. The learning rate is 0.001, ensuring steady and stable learning. We also set the variational beta to 1, which balances how well the model reconstructs data with how well it organizes the latent space. Finally, GPU support is enabled to make the training faster, thus, I run the code on the Kaggle platform to make use of free 16GB GPU and faster internet speed for donwloading the datasets and other resources. Moreover, the Kaggle provides us with pre-installed packages and libraries.

Loading MNIST

In this part of the project, we prepared the MNIST dataset for training and testing. The `torchvision.transforms` module was used to define a transformation for preprocessing the images. Here, the images are converted to tensors using `transforms.ToTensor()`, which normalizes pixel values to a range between 0 and 1. This transformation is applied to both the training and testing datasets.

The *MNIST dataset* was loaded using the `torchvision.datasets.MNIST` class. For the training set, `train=True` was specified, and `download=True` ensured that the dataset is downloaded if it wasn't already available locally. A similar process was followed for the testing dataset with `train=False`. The datasets were wrapped in `DataLoader` objects, which handle batching and shuffling during training. For training, a batch size of 64 was used, and shuffling was enabled to improve model generalization. For testing, the same

batch size was used, but shuffling was disabled to maintain a consistent order for evaluation.

Variational Autoencoder (VAE) Implementation

This part implements a VAE using PyTorch. A VAE is a generative model designed to encode input data into a compressed latent space and then reconstruct the original data from this latent representation. The model learns both the encoding (mapping data to the latent space) and decoding (reconstructing data) processes while ensuring the latent space follows a smooth, continuous distribution. This approach allows the VAE to generate new data samples by sampling from the learned latent space.

Encoder Class

The Encoder class defines the process of mapping input images into a latent space representation in the VAE. It consists of convolutional layers and fully connected layers, each designed for specific tasks in the encoding process.

The **convolutional layers**, `conv1` and `conv2`, extract features from the input images. They progressively reduce the spatial dimensions while increasing the depth (number of channels) of the data. This allows the encoder to capture meaningful patterns and features necessary for reconstruction.

After the feature extraction, the output of the convolutional layers is flattened into a vector and passed through two **fully connected** layers: `fc_mu` and `fc_logvar`. The `fc_mu` layer predicts the mean (`mu`) of the latent space distribution, while the `fc_logvar` layer predicts the log variance (`logvar`). These two outputs define the parameters of the latent space distribution for the input image.

The forward pass starts with applying ReLU activation to the outputs of the convolutional layers, which introduces non-linearity to the system. The flattened features are then passed to the fully connected layers to compute `mu` and `logvar`. These values are crucial for sampling in the latent space, as they enable the VAE to generate variations of the input data while maintaining similarity to the original image. The encoder essentially learns a compact, probabilistic representation of the input data.

Decoder Class

The Decoder class is responsible for reconstructing the original input data from the latent space in the VAE. It essentially performs the reverse operation of the Encoder class: taking a latent vector as input and generating an image as output.

The reconstruction process begins with a fully connected layer (fc), which expands the latent vector into a larger feature map. This feature map serves as the starting point for reconstructing the original image. The output of the fully connected layer is reshaped into a 4D tensor to be processed by the transpose convolutional layers.

The transpose convolutional layers, `conv2` and `conv1`, upsample the feature maps back to the spatial dimensions of the original image. These layers perform the opposite operation of the convolutional layers in the encoder, increasing the spatial size of the feature maps while reducing their depth.

In the forward pass, the output of the fully connected layer is reshaped and passed through `conv2` and `conv1`. ReLU activation is applied after `conv2` to introduce non-linearity and help the model learn complex patterns. Finally, a *sigmoid* activation function is applied to the output of `conv1`. This ensures the reconstructed image has pixel values in the range of 0 to 1, making it suitable for use with binary cross-entropy loss during training.

The Decoder takes the compressed representation from the latent space (a vector of two numbers, in this case) and reconstructs an image that resembles the original input. By working with the encoder, the decoder ensures the VAE can effectively compress and reconstruct data while allowing for meaningful sampling from the latent space.

VariationalAutoencoder Class

The VariationalAutoencoder class integrates the encoder and decoder into a single model, defining the full pipeline of the VAE. It handles the encoding of input data, the sampling from the latent space, and the decoding back to reconstruct the input.

In the forward pass, the input is first passed through the encoder, which outputs the mean (`mu`) and log variance (`logvar`) of the latent space distribution. These parameters define the Gaussian distribution for the latent representation. The latent vector is then sampled using the `latent_sample` method, which implements the reparameterization trick. This trick is essential as it allows the model to backpropagate through the stochastic sampling step by expressing the sampled latent vector as a deterministic function of `mu`, `logvar`, and random noise.

Once the latent vector is obtained, it is passed to the decoder, which reconstructs the input image from the latent space. The decoder produces an output that matches the original input's dimensions, completing the VAE pipeline.

The latent_sample method is a key component of the VAE. During training, it adds randomness to the latent representation by sampling from the learned distribution using the standard deviation (`std`) and random noise (`eps`). This ensures the model learns to map input data to a continuous and smooth latent space. However, during evaluation, it simply returns mu as the latent vector, avoiding stochasticity.

This class ties together the encoder and decoder, ensuring they work together to compress input data into a latent space and reconstruct it with minimal loss. It also enables the model to generate new samples by sampling from the learned latent distribution.

Loss Function

This part defines the core loss function, which is crucial for training the model to meet its objectives. The loss function combines two key components: reconstruction loss and KL divergence, both of which play distinct roles in guiding the model.

The **reconstruction loss** is calculated using binary cross-entropy (BCE) between the original input image and the reconstructed image. This loss ensures that the decoder learns to produce outputs that closely match the original input, effectively minimizing the pixel-level differences. By using the sum reduction in BCE, the reconstruction loss accounts for all pixels in the image.

The **KL divergence** term acts as a regularizer for the latent space. It encourages the learned latent distribution (defined by `mu` and `logvar`) to be close to a standard normal distribution. This ensures that the latent space is smooth and well-structured, making it possible to generate diverse and meaningful samples by simply sampling from a normal distribution during inference. The KL divergence formula is derived mathematically from the difference between the learned and standard normal distributions.

The total VAE loss is a combination of these two terms, weighted by the variational_beta parameter, which balances the importance of reconstruction accuracy and latent space regularization.

Once the loss function is defined, the VariationalAutoencoder instance is initialized and moved to the specified GPU device (or CPU, if GPU is not available) for training. The number of trainable parameters in the model is also calculated and printed

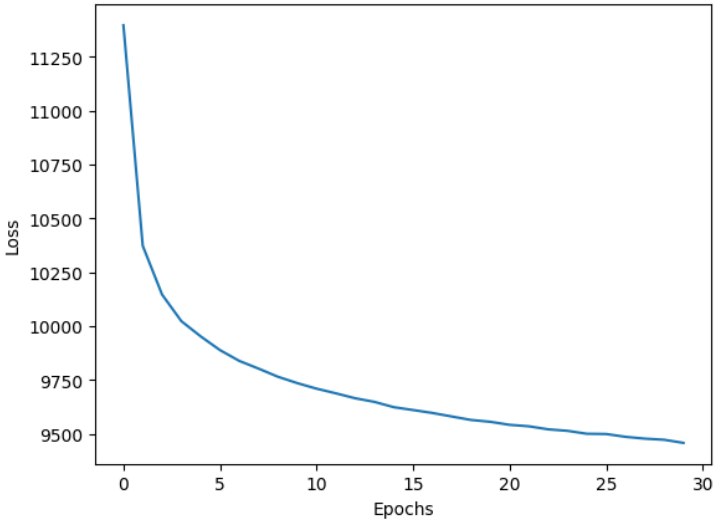
Train VAE

This section shows the training process for the VAE. The *Adam* optimizer is used here, with a learning rate of `learning_rate` and a `weight decay` of 1e-5 to prevent overfitting. The model is set to training mode using `vae.train()` for gradient computation and updates.

The training loop runs for a specified number of epochs (30), iterating through mini-batches of images from the training dataset. For each batch, the images are first moved to the appropriate device (GPU or CPU) for faster computation. The optimizer gradients are zeroed at the beginning of each iteration to ensure proper backpropagation.

The VAE processes the input batch through its encoder and decoder, producing the reconstructed batch, the mean (`mu`), and the log variance (`logvar`). These outputs are used to compute the total loss using the `vae_loss` function, which combines the reconstruction loss and KL divergence.

Once the loss is calculated, the `backward()` function is called to compute the gradients of the loss with respect to the model's parameters. The optimizer then updates the model's parameters using these gradients in the `step()` function.



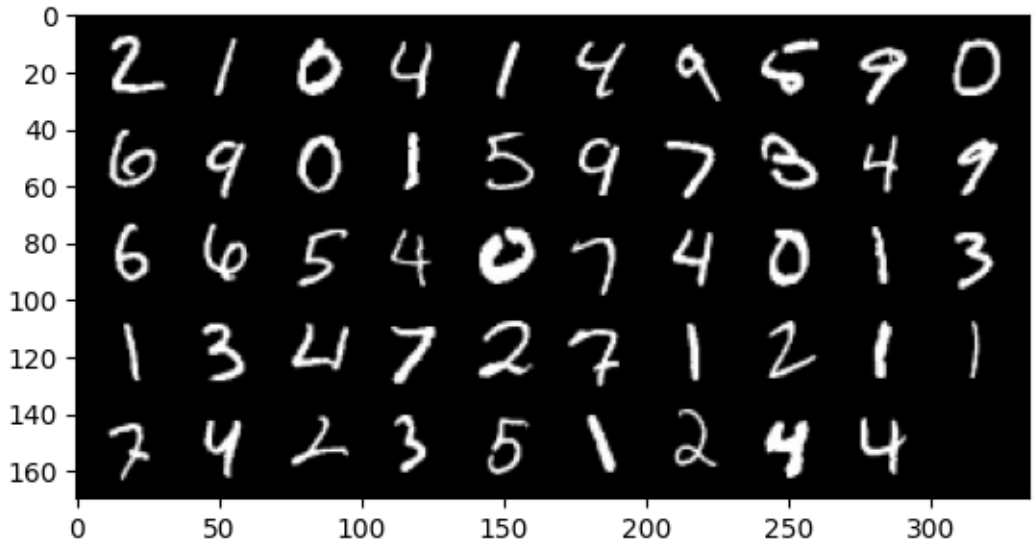
The average loss for each epoch is calculated by summing the losses for all batches and dividing by the number of batches. This provides a clear indication of the model's performance and convergence over time. The average reconstruction error for each epoch is printed, offering insight into how well the model is learning to encode and decode the data. After 30 epochs, the average reconstruction error is about 9457.781046.

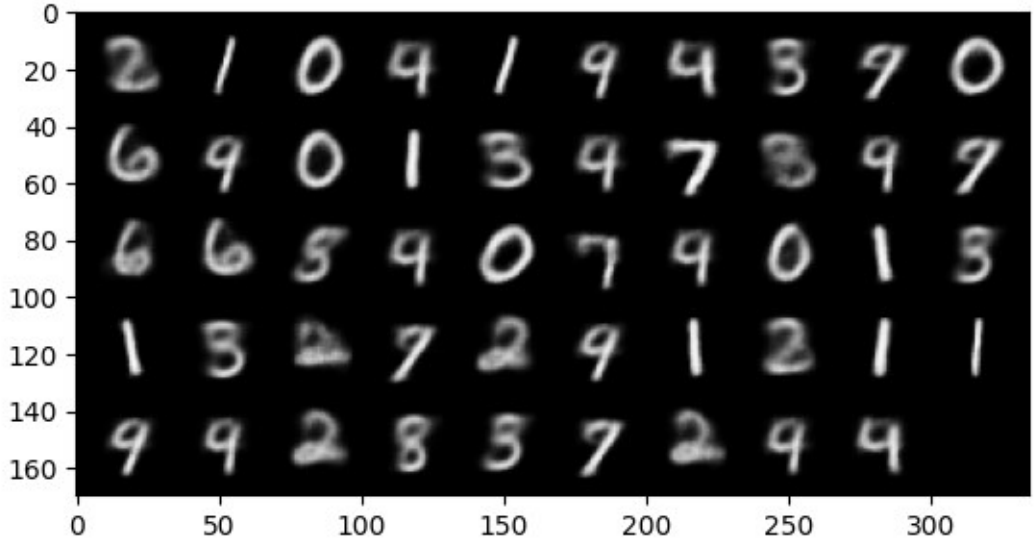
Evaluate on the Test Set

This section evaluates the VAE on the test dataset. The model is set to evaluation mode with `vae.eval()`, and gradients are disabled using `torch.no_grad()` since no updates are required. For each batch in the test set, images are moved to the device, and the VAE generates reconstructions along with the latent `mu` and `logvar`. The total loss, combining reconstruction error and KL divergence, is computed using `vae_loss`. The average reconstruction error is then calculated by summing the losses and dividing by the number of batches. The average reconstruction error is 9560.857860 for test which is pretty near to the train which means a good generalization and training.

Visualize Reconstructions

This section visualizes the VAE's performance by comparing original test images to their reconstructed counterparts. The goal is to observe how closely the model can reproduce input images. First, the function `to_img` clamps image pixel values to the range [0, 1], ensuring they are suitable for visualization. The `show_image` function displays a batch of images in a grid format, converting them from tensors to NumPy arrays for plotting. The main function, visualise_output, takes input images and the VAE model. It moves the images to the device, generates their reconstructions, and converts the outputs back to CPU for visualization. The reconstructed images are displayed in a grid using `torchvision.utils.make_grid`. To demonstrate, a batch of images is taken from the test dataset. The original images are visualized first, followed by their reconstructed versions generated by the VAE. This comparison highlights the VAE's ability to encode and decode data, showing how well it captures the features of the input images.

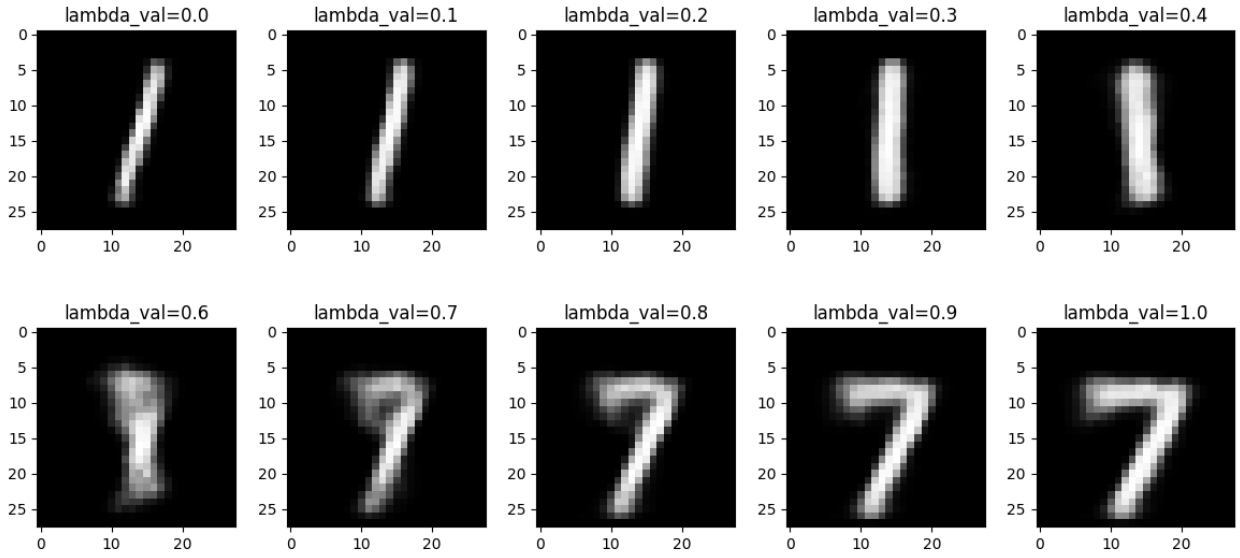




Here, we can see the produced images by VAE and observe that they are pretty similar to the original ones. However, there are some problems when creating the close numbers, i.e. close in shape e.g. 4 and 9.

Interpolate in Latent Space

This section demonstrates the VAE's ability to interpolate between two images in the latent space, highlighting the smoothness and structure of the learned latent representation. Interpolation allows us to generate intermediate representations that gradually transition from one image to another. The interpolation process begins by encoding two images (e.g., of digits 7 and 1) using the encoder. From the encoder, we extract only the mu values for both images, as the mean of the latent space distribution provides a deterministic representation. Since we are interested in interpolation, the variance (`logvar`) is not used. The interpolation function computes a weighted average of the two latent vectors using a parameter `lambda1` (ranging from 0 to 1). This weighted average creates a point in the latent space that is interpolated between the two input images. The decoder then reconstructs an image from this interpolated latent vector. After defining the interpolation function, we sort part of the test set by digits to select specific images for interpolation. Using a range of lambda values from 0 to 1, we generate and display a series of reconstructed images. These images transition smoothly between the two input images, demonstrating the VAE's ability to interpolate in the latent space.



Show 2D Latent Space

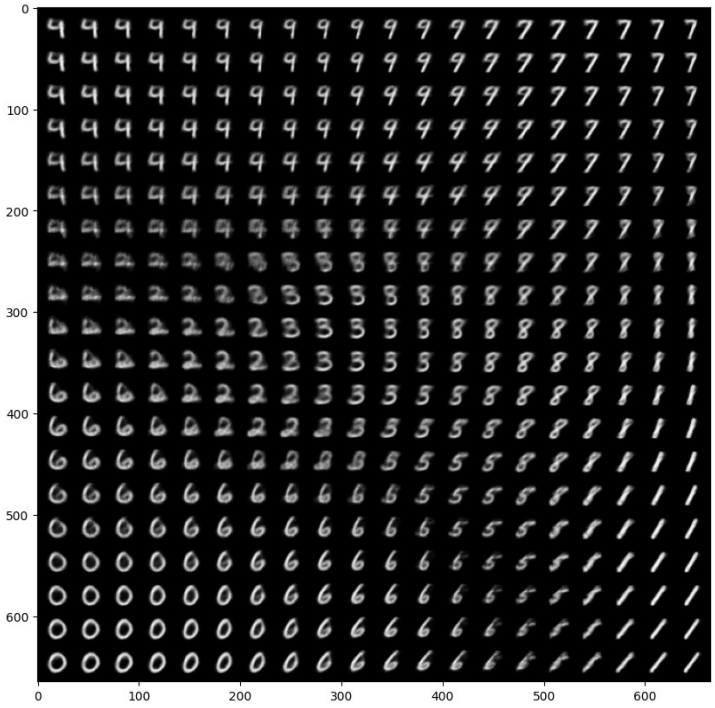
In this experiment, the VAE's decoder is tested on a grid of points sampled directly from the 2D latent space. This allows us to visualize how the decoder interprets various parts of

the latent space and reconstructs corresponding images. The latent space grid spans values between -1.5 and 1.5 along both dimensions, representing a wide range of latent vectors. These vectors are fed into the decoder to generate reconstructions.

The resulting visualization shows how the decoder produces images for different points in the latent space. While the overall results are promising, with distinct patterns resembling the training data, the outputs tend to appear blurry.

Why are most of the outputs blurry?

This blurriness arises because the VAE optimizes for a trade-off between reconstruction quality and regularization of the latent space. The reconstruction loss (binary cross-entropy) encourages the decoder to create images that closely match the input, but the KL divergence term ensures the latent space approximates a smooth Gaussian distribution. This regularization forces the decoder to generalize across the latent space, which can result in less sharp outputs, especially for points sampled far from the training data's distribution.



Additionally, the probabilistic nature of the VAE means the model doesn't focus on reconstructing fine details but rather captures the general structure of the data. While this ensures smooth latent space transitions, it can lead to the observed blurriness in individual reconstructions. This is a known limitation of VAEs and is why alternative generative models, like GANs, are often used when sharper outputs are required.

Problem 2

Stable Diffusion

Stable Diffusion is a generative technique for creating high-quality images from text descriptions. It uses a process of iterative denoising, where the model starts with random noise and gradually refines it into a coherent image, guided by the input text. By leveraging a powerful combination of text-to-image alignment and noise modeling, Stable Diffusion is widely used for creative tasks, such as art generation and visual storytelling. In this setup, the image generation parameters are defined. The default image dimensions are set to 512x512 pixels, providing a standard resolution for outputs. The process involves 50 inference steps, which control the number of iterations used to refine the image from noise. A guidance scale of 8 is used for classifier-free guidance, striking a balance between creativity and faithfulness to the text prompt. The random seed for generating initial noise is set using `torch.manual_seed(64)`, ensuring reproducibility of the outputs. The batch size is set to 1 for generating one image at a time. Finally, the device is set to "cuda" if a GPU is available, otherwise defaulting to the CPU. This configuration ensures an efficient and reproducible setup for generating high-quality images with Stable Diffusion.

Importing Models

This section describes the process of importing and initializing the models used in Stable Diffusion. The generation pipeline involves multiple components working together to transform text prompts into high-quality images. The pipeline begins with the VAE, which is responsible for decoding latent representations into image space. The pretrained VAE model is loaded from the Stable Diffusion checkpoint and moved to the appropriate device (`torch_device`). Next, the CLIP Tokenizer and Text Encoder are imported. The tokenizer converts user-provided text prompts into tokenized inputs, while the text encoder generates corresponding embeddings for these tokens. Both are based on the `openai/clip-vit-large-patch14` model, which excels at aligning text and image embeddings. The core of the generation process is the U-Net model, which iteratively denoises the latent embeddings, gradually refining the image representation. A scheduler, specifically the PNDM scheduler, guides the noise reduction process by defining the denoising steps and algorithm. A utility function, `latents_to_pil`, is defined to convert latent representations into final image outputs. This function scales the latent values appropriately and decodes them into image tensors using the VAE. The decoded tensors are then normalized, converted to NumPy arrays, and finally transformed into a list of PIL images for easy visualization or saving. By combining these components, the Stable Diffusion pipeline enables the transformation of random latent noise and text prompts

into detailed and coherent images. This modular setup ensures flexibility and efficiency for text-to-image generation tasks.

Prompt to Embedding

This section converts text prompts into embeddings to provide guidance for the U-Net model in the Stable Diffusion pipeline. The embeddings consist of two parts: positive embeddings for the text prompt and negative embeddings for unconditional guidance. These embeddings are essential for aligning the denoising process with the desired output.

The `prompt_to_emb` function begins by tokenizing the text prompts using the tokenizer, ensuring a fixed maximum length of 77 tokens. The tokenized inputs are passed through the `text_encoder` to generate the positive embeddings. These embeddings are processed to ensure they have the correct data type and are moved to the appropriate device (`device`), allowing the U-Net model to utilize them effectively. The function also generates negative embeddings by following a similar process. Since no explicit negative prompts are provided, empty strings are used as placeholders. These negative embeddings act as an unconditional reference, guiding the U-Net model during the denoising steps to maintain coherence. Finally, the positive and negative embeddings are concatenated along the batch dimension, creating a combined tensor with a shape of [2, 77, 768] for each text prompt. This concatenated embedding provides the U-Net with both conditional and unconditional guidance, enabling it to align the output with the text while maintaining flexibility in generating diverse results. The design of this function ensures compatibility with the Stable Diffusion framework and emphasizes the importance of both positive and negative embeddings in guiding the generative process.

Embeding to Latents

The emb_to_latents function refines random latent noise into meaningful latent representations using the U-Net and scheduler. Starting with random noise of shape [1, 4, 64, 64], the scheduler iterates over `num_inference_steps`, progressively denoising the latents. At each timestep, the noise is duplicated, scaled, and passed to the U-Net along with the text embeddings. The U-Net predicts the noise to remove, combining unconditional and text-guided components through classifier-free guidance, controlled by guidance_scale. The scheduler then updates the latents based on the predicted noise. This process gradually aligns the latent representation with the text prompt, producing refined latents ready for decoding into images with the VAE. This iterative refinement ensures coherence between the text input and the generated image.

Evaluate

Here are some examples made by the model.

“A photograph of an astronaut riding a horse.”

“A campfire (oil on canvas)”