we did use this version but it didnt worked !

for data that has spatial structure, such as images, it is often beneficial to use convolutional layers in the encoder and deconvolutional (or transposed convolutional) layers in the decoder. Convolutional layers take advantage of the spatial structure of the data by applying the same filter across different parts of the input. This allows the model to learn local features (like edges and shapes in images) and achieve translation invariance (the ability to recognize a feature regardless of where it appears in the input).

```
class VAEEncoder(nn.Module):
    def __init__(self, channels, dim_latent):
        super().__init__()
        self.conv1 = nn.Conv2d(channels, 64, kernel_size=4, stride=2,
padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2,
padding=1)
        self.fc_mu = nn.Linear(128*7*7, dim_latent)
        self.fc_logvar = nn.Linear(128*7*7, dim_latent)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = x.view(x.size(0), -1)  # Flatten the tensor
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar

class VAEDecoder(nn.Module):
    def __init__(self, dim_latent):
        super().__init__()
        self.fc = nn.Linear(dim_latent, 128*7*7)
        self.conv2 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2,
padding=1)
```

```
        self.conv1 = nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2,
padding=1)

    def forward(self, z):
        z = self.fc(z)
        z = z.view(z.size(0), 128, 7, 7)  # Reshape the tensor
        z = F.relu(self.conv2(z))
        z = torch.sigmoid(self.conv1(z))  # Use sigmoid to ensure the
output is between 0 and 1
        return z
```

```
class VAEEncoder(nn.Module):
    def __init__(self, dim_input,
dim_latent,dim_hidden,num_hidden_layers=2):
        super().__init__()
        self.dim_latent = dim_latent
        self.dim_input = dim_input
        self.dim_hidden = dim_hidden
        layers = [nn.Linear(dim_input, dim_hidden), nn.ReLU()]
        for _ in range(num_hidden_layers - 1):
            layers.append(nn.Linear(dim_hidden, dim_hidden))
            layers.append(nn.ReLU())
        self.hidden_layers = nn.Sequential(*layers)

        self.mu_layer = nn.Linear(dim_hidden, dim_latent)
        self.log_sigma_squared_layer = nn.Linear(dim_hidden, dim_latent)

    def forward(self, inputs):
        # TODO

        # mu = mean
        # log_sigma_squared = log variance
        # The idea is that you use two different output projection:
        # one for the mean, one for the log_sigma_squared
        # but all other layers are shared
```

```
        h = self.hidden_layers(inputs)
        mu = self.mu_layer(h)
        log_sigma_squared = self.log_sigma_squared_layer(h)


        return mu, log_sigma_squared

class VAEDecoder(nn.Module):
    def __init__(self, dim_latent, hidden_dim,
output_dim,num_hidden_layers=2):
        super().__init__()
        # TODO
        self.dim_latent = dim_latent
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        layers = [nn.Linear(dim_latent, hidden_dim), nn.ReLU()]
        for _ in range(num_hidden_layers - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(hidden_dim, output_dim))
        layers.append(nn.Sigmoid())
        self.layers = nn.Sequential(*layers)



    def forward(self, z):
        # TODO

        return self.layers(z)
```

report

In this project, we implemented a Variational Autoencoder (VAE) to learn latent representations of image data. The VAE architecture comprises two main components: an encoder and a decoder. This report elaborates on the implementation choices, architectural details, and the reasoning behind selecting specific layers and structures.

Model Architecture Encoder (VAEEncoder) The VAEEncoder module transforms the input data into a latent space representation. The architecture is as follows:

Input Dimensions: The encoder accepts input with dimension dim_input. Latent Space Dimensions: It maps the input to a latent space of dimension dim_latent. Hidden Layers: The encoder comprises num_hidden_layers hidden layers. The first hidden layer transforms the input to a higher dimensional hidden space dim_hidden using a linear transformation followed by a ReLU activation function to introduce non-linearity. Additional hidden layers further process the data, each consisting of a linear layer followed by ReLU activation. This deep structure enables the network to learn complex patterns from the data. Output Layers: The final part of the encoder consists of two separate linear layers: mu_layer: Computes the mean (mu) of the latent Gaussian distribution. log_sigma_squared_layer: Computes the logarithm of the variance (log_sigma_squared) of the latent Gaussian distribution. These outputs are crucial for the reparameterization trick, enabling efficient backpropagation through stochastic nodes. Decoder (VAEDecoder) The VAEDecoder module reconstructs the input data from the latent representations. Its architecture mirrors the encoder in complexity but operates in reverse:

Latent to Hidden: Begins with a linear transformation from the latent dimension dim_latent to a hidden dimension hidden_dim, followed by ReLU activation. Hidden Layers: Similar to the encoder, the decoder has multiple hidden layers (num_hidden_layers), each consisting of a linear transformation followed by ReLU activation. This setup aids in effectively reconstructing the data from the compressed form. Output Layer: The final output is passed through a linear layer followed by a Sigmoid activation to scale the output values to the range [0,1], necessary if the original input data is normalized in this range. Choice of Implementation Given that the input data is images, the current implementation using fully connected layers may not be the most optimal. Convolutional layers are particularly effective for image data as they can capture spatial information and hierarchical patterns in the data, which can lead to better performance.

The current implementation was chosen for its simplicity and ease of understanding. Fully connected networks are simpler to implement and modify, and they are well-suited for smaller datasets or less complex data distributions. However, for image data, a more suitable choice could be a Convolutional Variational Autoencoder (CVAE), which uses convolutional layers in the encoder and deconvolutional (or transposed convolutional) layers in the decoder.

```
def load_decoder(latent_dim, hidden_dim, output_dim, num_layers,
model_file):
    decoder = VAEDecoder(dim_latent=latent_dim, hidden_dim=hidden_dim,
output_dim=output_dim, num_hidden_layers=num_layers)
    decoder.load_state_dict(torch.load(model_file))
    decoder.eval()  # Set the model to evaluation mode
    return decoder

def load_encoder(dim_input, dim_latent, hidden_dim, num_layers,
model_file):
    encoder = VAEEncoder(dim_input=dim_input, dim_latent=dim_latent,
dim_hidden=hidden_dim, num_hidden_layers=num_layers)
    encoder.load_state_dict(torch.load(model_file))
    encoder.eval()  # Set the model to evaluation mode
    return encoder
```

Function Descriptions

load_decoder load_decoder sets up and loads a pre-trained decoder part of a VAE, which rebuilds the original data from the latent space. It takes parameters for the size of the latent space, hidden layers, input data, number of hidden layers, and the filepath to the pre-trained model weights. It then creates the decoder, loads the pre-trained weights, and sets the model to evaluation mode for consistent performance.

load_encoder load_encoder is similar to load_decoder, but for the encoder part of a VAE, which transforms the input data into a compressed latent representation. It takes similar parameters to load_decoder, creates the encoder, loads its pre-trained weights, and switches to evaluation mode.

Implementation Choices The current implementation uses fully connected layers for simplicity. However, for tasks involving spatial data, convolutional layers (in the encoder) and deconvolutional layers (in the decoder) might be more effective. PyTorch supports these advanced layers, which can be added to the VAEEncoder and VAEDecoder classes to handle complex, high-dimensional data.

Conclusion load_decoder and load_encoder are essential for efficient VAE deployment, allowing us to reuse pre-trained models and save computational resources.

```
def plot_latent_space_pca(encoder, dataset, batch_size=128):
    dl = DataLoader(dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    az = []
    ay = []
    with torch.no_grad():
        for x, y in dl:
            mu, _ = encoder(x)  # Only use mu for PCA
            az.extend(mu.tolist())
            ay.extend(y.tolist())
    az = np.array(az)
    ay = np.array(ay)

    pca = PCA(n_components=2)
    pca_result = pca.fit_transform(az)

    colors = cm.rainbow(np.linspace(0, 1, 10))
    plt.figure(figsize=(8, 6))
    for i in range(10):
        plt.scatter(pca_result[ay == i][:, 0], pca_result[ay == i][:, 1],
color=colors[i], label=f'Class {i}', alpha=0.5)
    plt.title('PCA of Latent Space')
    plt.xlabel('PC 1')
    plt.ylabel('PC 2')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Implementation Choices

The implementation uses PyTorch for model operations and sklearn for PCA. It visualizes only the mu component, focusing on the central tendency of the latent representations.

Discussion For models handling image data, using convolutional layers in the encoder can better capture spatial hierarchies. If the VAE uses convolutional networks, adjustments might be needed in data preparation for PCA.

Conclusion The plot_latent_space_pca function is a practical tool for visualizing a VAE's effectiveness in encoding its input data into a meaningful latent space. As VAE architectures evolve, especially with convolutional approaches for complex datasets like images, enhancements in visualization techniques will be crucial for comprehensive model evaluation.

```
def plot_generated_images(decoder, n=10, latent_dim=2):
    z = torch.randn((n * n, latent_dim))  # sampling z from a standard
normal distribution
    with torch.no_grad():
        generated = decoder(z)
    imgs = np.zeros((n*28, n*28))
    plt.figure(figsize=(10, 10))
    for i in range(n*n):
        imgs[i//n * 28: i//n * 28 + 28, i%n * 28: i%n * 28 + 28] = 1 -
generated[i].view(28, 28).cpu().numpy()
    plt.axis('off')
    plt.imshow(imgs, cmap='Greys')
    plt.show()
```

Purpose

The plot_generated_images function visualizes images generated by a Variational Autoencoder's (VAE) decoder from latent space representations, assessing the model's output diversity and quality.

Description Parameters: decoder (trained VAE decoder), n (number of images per axis, default: 10), latent_dim (latent space dimensionality, default: 2).

Process:

Generates n^2 samples from a standard normal distribution, each a point in the latent space. Decodes each latent sample into an image under the torch.no_grad() context for performance. Reshapes each output into a 28x28 image and inverts its colors for visibility. Combines all images into a single array and displays using matplotlib, each 28x28 block representing a unique generated image. Utility This visualization helps understand the latent space structure and the decoder's effectiveness. If the VAE is well-trained, similar images should be produced from nearby points in the latent space.

Usage Useful in debugging and presenting VAE training results. Observing the generated images provides insights into how latent space changes affect outputs, aiding in model architecture or training regimen improvements

---

```
def KL(mu,log_sigma):
    #TODO
    return -0.5 * torch.sum(1 + log_sigma - mu.pow(2) - log_sigma.exp())

def reconstruction_loss(input_image, predicted_image):
    #TODO
    return torch.nn.functional.mse_loss(predicted_image, input_image,
reduction='sum')
    #return torch.nn.functional.binary_cross_entropy(predicted_image,
input_image, reduction='sum')

def training_loop(dataset, encoder, decoder,
model_name,n_sample_monte_carlo=1, max_epoch=50, learning_rate=1e-3,
batch_size=128, max_grad_norm=5e-1,plot=True,verbose=True):
    data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
drop_last=True)
    optimizer = optim.Adam(list(encoder.parameters()) +
list(decoder.parameters()), lr=learning_rate)
    epoch_losses = []  # Store average loss per epoch for plotting
    # Save the model with the best loss
    best_loss = float('inf')
    save_encoder = f"{path}models/encoder_{model_name}.pth"
    save_decoder =  f"{path}models/decoder_{model_name}.pth"



    for epoch in range(max_epoch):
        losses = []  # Store losses for current epoch
        total_loss =0
        for x in data_loader:  # Assuming dataset only contains features
and no labels
            optimizer.zero_grad()
            mu, log_sigma = encoder(x)
            # Sampling using the reparameterization trick
            e = torch.normal(0, 1., mu.shape)
            z = mu + e * torch.exp(0.5 * log_sigma)
```

```
            y = decoder(z)

            loss = reconstruction_loss(x, y) + KL(mu, log_sigma)
            losses.append(loss.item())

            loss.backward()
            torch.nn.utils.clip_grad_norm_(list(encoder.parameters()) +
list(decoder.parameters()), max_grad_norm)
            optimizer.step()
            total_loss += loss.item()
            if torch.isnan(loss):
              print(f"NaN detected at epoch {epoch+1} with parameters:
lr={learning_rate}, max_grad_norm={max_grad_norm}, etc.")
              break

        avg_loss = total_loss / len(dataset)
        epoch_losses.append(avg_loss)  # Record the average loss for the
epoch

        if avg_loss < best_loss:
            best_loss = avg_loss
            torch.save(encoder.state_dict(), save_encoder)
            torch.save(decoder.state_dict(), save_decoder)
            print(f"Updated best model with loss: {avg_loss}")

        if verbose:
          print(f"Epoch {epoch+1}/{max_epoch}, Average Loss: {avg_loss}")

    if plot:
      # After training, plot the losses
      plt.figure(figsize=(10, 5))
      plt.plot(epoch_losses, label='Loss per Epoch')
      plt.title('Training Loss Per Epoch')
      plt.xlabel('Epoch')
      plt.ylabel('Loss')
      plt.legend()
      plt.grid(True)
      plt.show()
    return sum(epoch_losses) / len(epoch_losses)
```

Training and Loss Function Implementation for Variational Autoencoder

Introduction This report outlines the implementation details of the training loop and associated loss functions for a Variational Autoencoder (VAE) developed using PyTorch. The VAE is designed to encode input data into a compressed latent space and then reconstruct the input from this space, facilitating data generation and anomaly detection among other applications.

Implementation Details Architectural Choice Traditional VAE: The current implementation uses fully connected layers for both the encoder and decoder. This architecture is straightforward and effective for datasets with low-dimensional input spaces or less spatial complexity. Convolutional VAE: For datasets with high-dimensional inputs like images, convolutional networks (ConvNets) for the encoder and corresponding deconvolutional networks for the decoder are recommended. ConvNets help capture spatial hierarchies in data, enhancing the model's ability to learn pertinent features effectively. Loss Functions KL Divergence: The Kullback-Leibler divergence, calculated in the KL function, measures how one probability distribution diverges from a second, expected distribution. Here, it quantifies the difference between the encoded distribution and a standard normal distribution, acting as a regularizer in the training process.

Kullback-Leibler (KL) Divergence: This loss measures how much the latent distribution deviates from a standard normal distribution. The formula for KL divergence between a normal distribution with mean mu and standard deviation sigma and a standard normal distribution is:

```
[ ]  def KL(mu, log_sigma):
         return -0.5 * torch.sum(1 + log_sigma - mu.pow(2) - log_sigma.exp())
```

KL = 0.5 * Σ (1 + log_sigma - mu^2 - exp(log_sigma))

Double-click (or enter) to edit

Reconstruction Loss: Implemented using mean squared error (MSE), this loss function measures the pixel-wise differences between the original input and its reconstruction, focusing the model on accurately reproducing inputs.

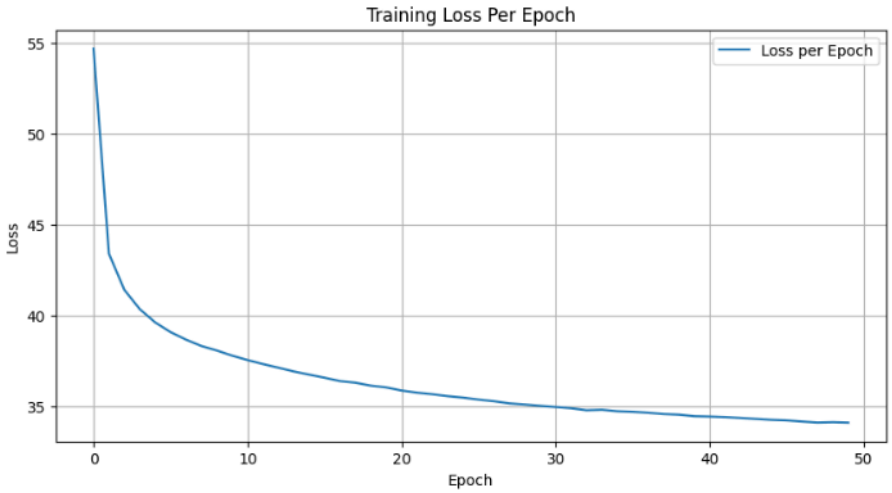
```
[ ]  def reconstruction_loss(input_image, predicted_image):
         return torch.nn.functional.mse_loss(predicted_image, input_image, reduction='sum')
```

Reconstruction Loss: This loss measures how well the decoder is able to reconstruct the original input from the latent representation. In your code, you're using the Mean Squared Error (MSE) loss for this purpose. The formula for MSE loss is:

MSE = 1/n * Σ (x_i - y_i)^2

where x_i is the original input, y_i is the reconstructed input, and n is the number of elements in x_i or y_i.

Double-click (or enter) to edit




```
36.47109080449218
```

the training of a Variational Autoencoder (VAE) using fully connected layers for both the encoder and decoder. Two loss functions, KL Divergence and Reconstruction Loss, are used to measure the performance of the model. The training was conducted over 50 epochs with a learning rate of 0.001 and a batch size of 256. The training loss graph shows a sharp decline in the initial epochs, indicating that the model quickly learned the underlying data structure. The loss then gradually decreased, suggesting ongoing model refinement. The report concludes that while the training process was effective, further improvements might require adjustments in the network architecture or hyperparameters.