Understanding the latent space of Variational Autoencoders and Oversampling Imbalanced Classes

George Luther (George.Luther1@hotmail.com)

Introduction

The aim of this project is to explore the use of **Variational Autoencoders (VAEs)** in tackling the issue of imbalanced datasets using the **MNIST dataset**. The project focuses on generating synthetic data via a VAE to oversample underrepresented classes, balancing the dataset and evaluating both datasets on a **Multilayer Perceptron**. Additionally, this project delves deep into the VAE model by examining its latent space, which can reveal valuable insights about how the model learns representations of the data. Although the MNIST dataset does not have widely imbalanced classes, it was primarily used to examine the latent space of the model more intuitively. This project is therefore a combination of delving deep into a specific model and programming an application of its use-case (both options).

The inspiration behind this project came from an article (Jordan, 2018) that visualizes the latent space and explains VAEs intuitively. As an extension to this article, further visualizations of the latent space with different parameters were done along with an application in oversampling imbalanced datasets.

Variational Autoencoders (VAE)

VAE Fundamentals

The Variational Autoencoder (VAE) works by inserting a stochastic element into its latent space. Instead of encoding the input directly into a low dimensional representation, the VAE encodes the latent space as a probability distribution, calculating the mean and variance of the latent vectors. These vectors are sampled using the reparameterization trick and are compared to a prior distribution (in this project, we set this as the normal distribution). The stochastic elements are called the latent attributes which describe different characteristic cs of the input data. This probabilistic nature of the VAE's latent space is what differentiates them from deterministic models like transformers.

Loss Function

The VAE combines principles from Bayesian statistics and neural networks, making it particularly useful for generative tasks. The model's architecture includes key components such as **Kullback-Leibler Divergence (KLD)** and **Binary Cross Entropy (BCE)**, which combine to form the loss function that is used to train the model. These will be further explained in the application part of the guide, where a simpler version of the KLD which uses the normal distribution as a prior probability distribution was used.

Application of the model and outputs

The encoder and decoder phases of the model are described in **Code Snippet 1 (CS 1)**, where the input data is first flattened into a vector of 28 x 28 = 784 elements (equivalent to pixel size of MNIST images). Next, it is passed through a fully connected layer with 200 neurons compressing the input data, it is then passed into the latent dimension with 200 elements and dimensionality equivalent to the code_size. This dimension parameter controls how many latent attributes we have, where latent attributes explain the variability in the data. In the VAE, the mean and variance are computed, defined as *loc* and *scale* in the code snippet:

- **loc:** The mean of the learned latent distribution.
- **scale:** The standard deviation (or variance) of the learned latent distribution, where the soft plus activation function ensures positive values.

The decoder phase takes the input of the has the same dimensions of the encoder but in reverse order, starting with the input being code_size, through a fully connected layer and expanding the latent representation back to 28 x 28, reconstructing the image.

```python
# Define the encoder network
class Encoder(nn.Module):
    def __init__(self, code_size):
        super(Encoder, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc_loc = nn.Linear(200, code_size)
        self.fc_scale = nn.Linear(200, code_size)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        # loc is the mean of the latent space
        loc = self.fc_loc(x)
        # scale is the log of the standard deviation
        scale = torch.nn.functional.softplus(self.fc_scale(x))
        return loc, scale
```

✓ 0.0s

```python
# Define the decoder network
class Decoder(nn.Module):
    def __init__(self, code_size):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(code_size, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 28 * 28)

    def forward(self, z):
        x = torch.relu(self.fc1(z))
        x = torch.relu(self.fc2(x))
        # logit is the output of the decoder
        logit = self.fc3(x)
        logit = logit.view(-1, 1, 28, 28)
        return logit
```

✓ 0.0s

Code Snippet 1. The encoder and decoder definitions of our VAE model.

In **CS 2**, we can see the VAE model defined as containing the previously mentioned encoder and decoder architecture from **CS 1**. In the forward pass function, the **loc** and **scale** are obtained as an output from the encoder and the reparameterization trick is used. The reparameterization trick is sampling a point from a standard normal distribution (mean 0, variance 1) with the same shape as std. This random value represents the noise component needed to introduce stochasticity in the latent space, allowing us to generate new unseen samples. This value is then passed through the decoder, representing the input.

```python
# Define the VAE model
class VAE(nn.Module):
    def __init__(self, code_size):
        super(VAE, self).__init__()
        self.encoder = Encoder(code_size)
        self.decoder = Decoder(code_size)

    def reparameterize(self, loc, scale):
        """
        reparameterization trick to sample from the latent space.
        """
        std = torch.exp(0.5 * scale)
        eps = torch.randn_like(std)
        return loc + eps * std

    def forward(self, x):
        """
        forward pass of the VAE.
        """
        loc, scale = self.encoder(x)
        z = self.reparameterize(loc, scale)
        logit = self.decoder(z)
        return logit, loc, scale
```

✓ 0.0s

Code Snippet 2. VAE model combining both the encoder and decoder, including the reparameterization trick.

The total loss function is defined in **CS 3** as a combination of BCE and KLD. The BCE loss measures the difference between the reconstructed data, recon_x and the input, x. The KLD on the other hand measures the difference between two probability distributions. It measures how close the distribution between the two latent variables, loc and scale (or mean and variance) are to the normal distribution (mean=0 and variance = 1). KLD acts as a regularization term by encouraging the latent space to be like a standard normal distribution and prevents overfitting by ensuring the latent variables follow a smooth, continuous distribution. The original paper on VAEs contains more information and derivations of the KLD and other loss functions that could possibly be used such as Monte-Carlo sampling (Kingma, 2013)

```python
# Loss function
def loss_function(recon_x, x, loc, scale):
    BCE = nn.BCEWithLogitsLoss(reduction='sum')(recon_x, x)
    # When prior distribution is normal, KLD is -0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD = -0.5 * torch.sum(1 + scale - loc.pow(2) - scale.exp()) # scale is the log of the standa
    return BCE + KLD
```

✓ 0.0s Python

Code Snippet 3. The loss function for our VAE defined as BCE+KLD.

Preprocessing and training the models

The dataset is loaded and a **batch size of 100** is set throughout this report, where the batch size controls how often we update the gradient during backpropagation. Three models are instantiated with different dimensions in their latent space: **2, 10 and 100**. These models are then trained for **25 epochs** where the training loop can be seen in **CS 4** where the gradient is stored for each input and updated on every 100th input. Adam (Adaptive Moment Estimation), a version of gradient descent that dynamically adjusts the learning rate is used as the optimization function during training, with learning rate set to **0.001**. The functions defined in previous code snippets are used in the training loop and the model is set to evaluation mode, and gradient updates have been turned off to finally test the model.

```python
trained_models = {}

for code_size, model in tqdm(models_n_dim.items()):
    optimizer = optim.Adam(model.parameters(), lr=0.01)
    num_epochs = 25
    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            optimizer.zero_grad()  # Zero out gradients before the backward pass
            recon_batch, loc, scale = model(data)  # Forward pass
            loss = loss_function(recon_batch, data, loc, scale)  # Compute loss
            loss.backward()  # Backward pass to compute gradients
            train_loss += loss.item()  # Accumulate the training loss
            optimizer.step()  # Update the model parameters

        print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset)}')

    # Testing the model
    model.eval()  # Set model to evaluation mode
    test_loss = 0
    with torch.no_grad():  # Disable gradient calculation for efficiency
        for data, _ in test_loader:
            data = data.to(device)
            recon_batch, loc, scale = model(data)  # Forward pass for test data
            test_loss += loss_function(recon_batch, data, loc, scale).item()  # Compute test loss

    test_loss /= len(test_loader.dataset)
    print(f'Test set loss: {test_loss}')
    trained_models[code_size] = model  # Store the trained model
```

49.6s Python

Code Snippet 4. The training loop for multiple VAE models with different dimensions in their latent space, then added to a dictionary called trained_models.

Exploring the latent space using different code_size

The code_size refers to the number of dimensions in the latent space. In a VAE, the number of dimensions refer to two vectors which are described by a mean and variance. In **Figure 1**, we can see a that when the latent space is set to 2-dimensions (i.e., code_size=2) using PCA for dimensionality reduction on the encoded data, the classes are more separable as the model only relies on these two dimensions to capture the patterns within the data. When we include 10 or 100 dimensions to describe the MNIST data, projecting this higher dimensional representation into two dimensions using PCA results in less visible variance. This occurs because the other dimensions encode various other characteristics of the digits, such as different strokes, thickness of lines, sharp edges of digits that cannot all be captured in a 2D projection.

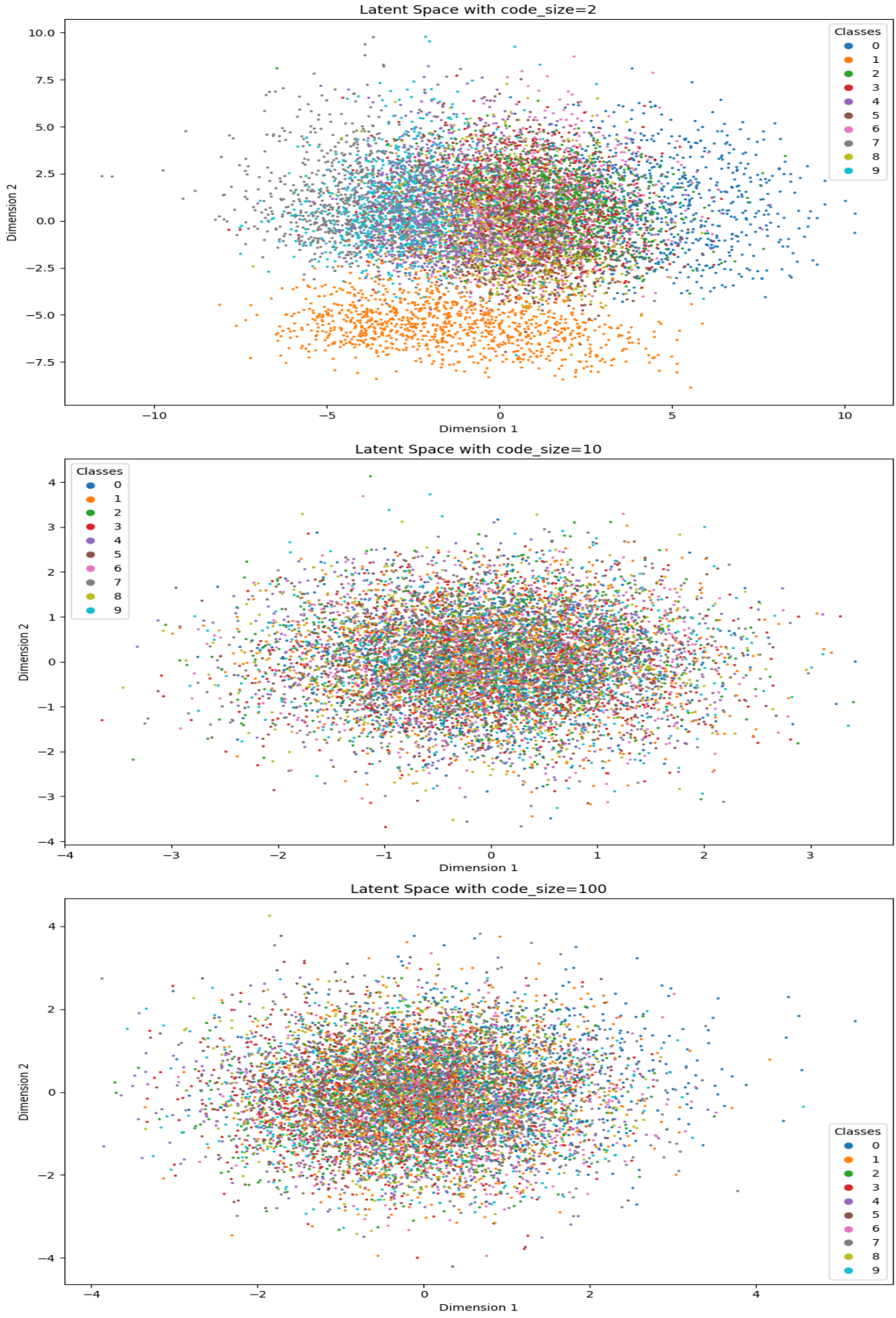

Figure 1. Different dimensions of the latent space for the MNIST data.

Figure 2 shows the latent space of a 2-dimensional VAE. In this example, we traverse the 2-dimensional latent space at each coordinate and pass the coordinate through the decoder, allowing us to visualize the distribution of the latent space. In this image, we can see digits with similar characteristics closer together. For example, digit 1 is in the top-left hand corner and slowly transitions to 9, i.e., the model determines that these digits have similar characteristics and places them near each other in the latent space. Because the number of dimensions is limited, the other digits are less distinct or "fuzzified" but can be distinguished.

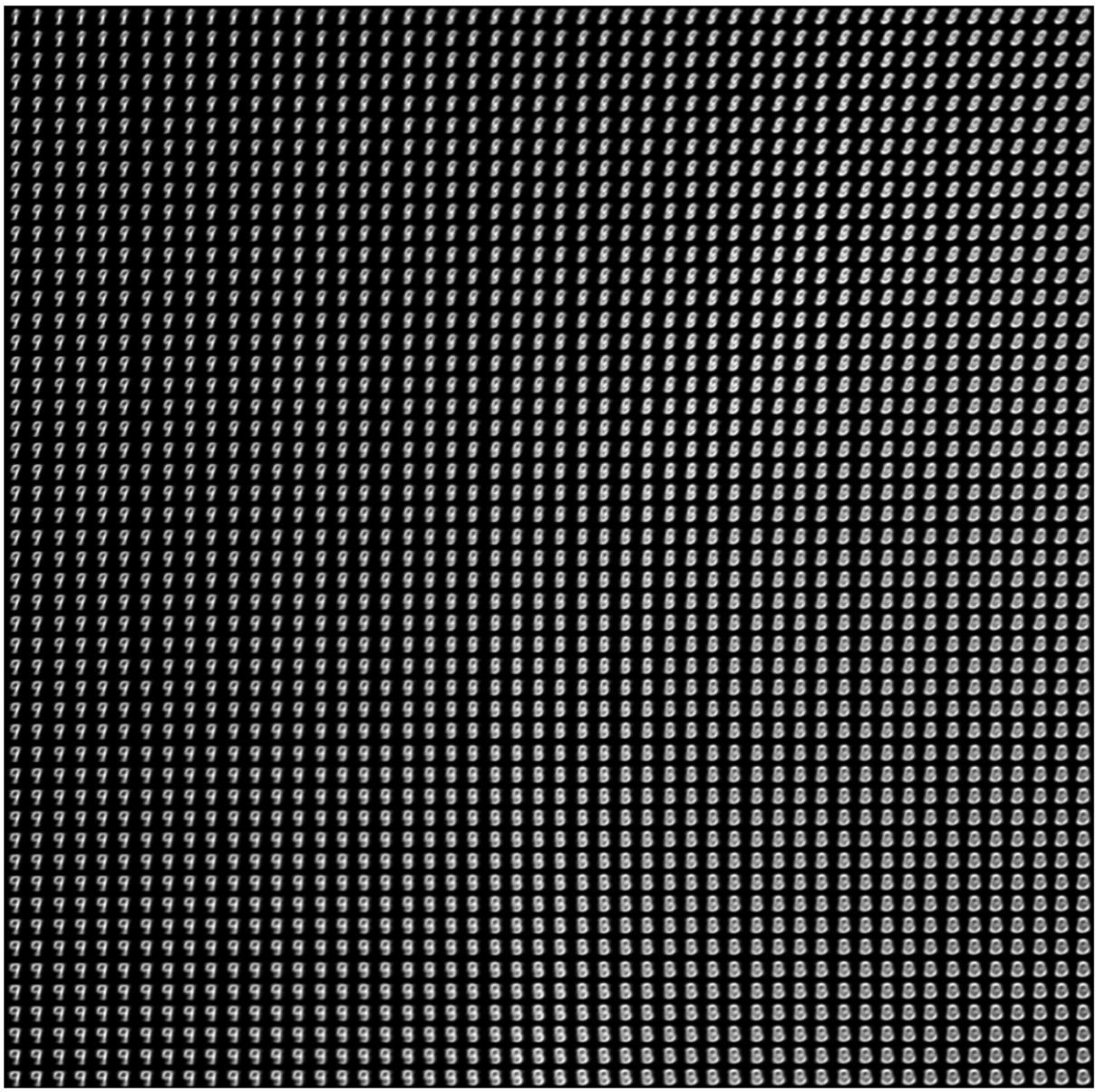

Figure 2. The latent space of a 2-dimensional VAE projected into a grid.

Oversampling with VAEs

Referring to **Figure 3,** we can see the distribution of the MNIST data where the dataset is not balanced. This provides an opportunity to generate synthetic data learned on the training samples and run the model through a Multilayer perceptron and assess model performance. In this section, we use the same model architecture and parameters from **CS 1-3** to provide a solution for imbalanced datasets.

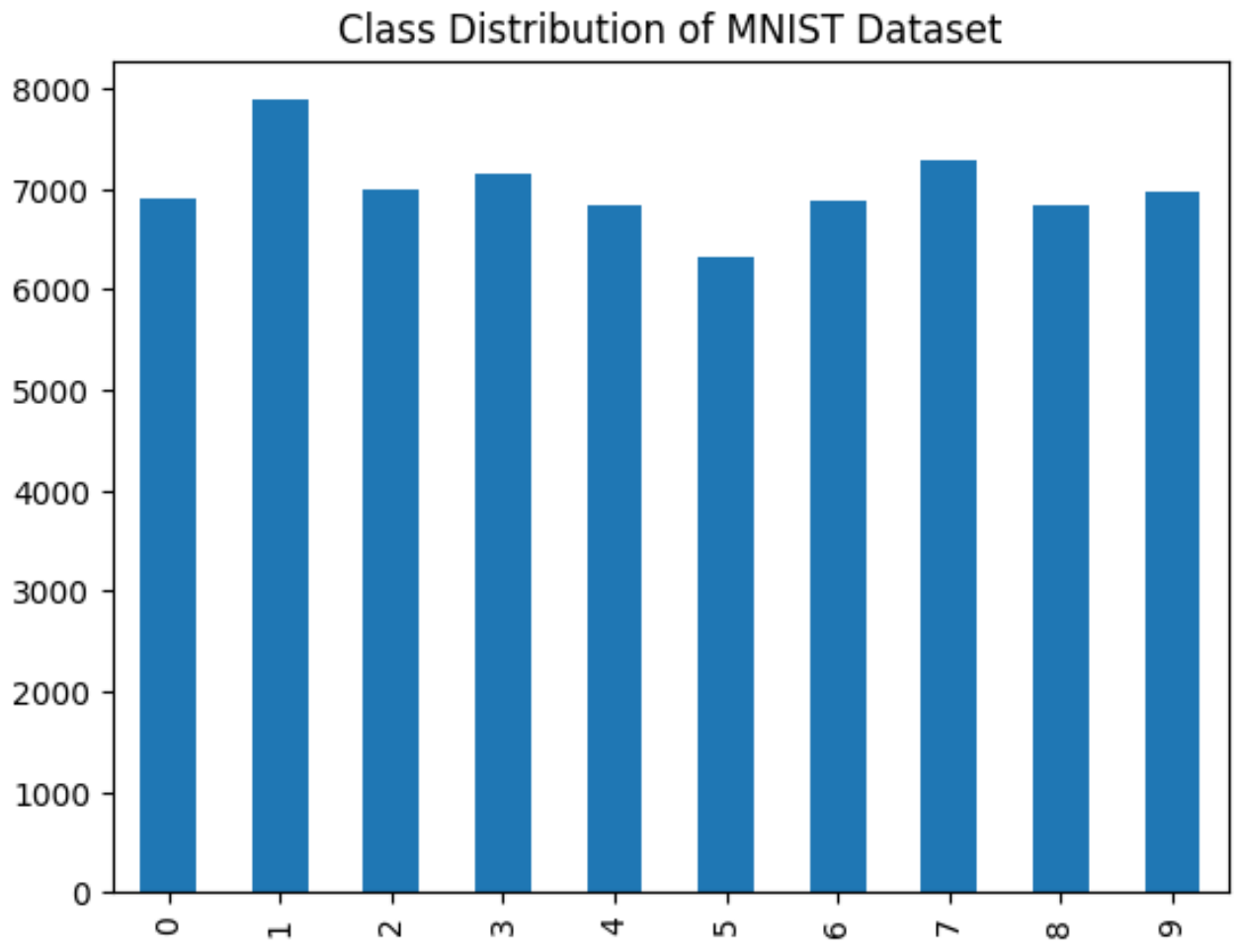

Figure 3. The class distribution of the MNIST dataset.

Initially, we train 10 different VAE models, one for each class of the MNIST digit. These models are stored in a dictionary and accessed at a later stage to sample points from the latent space of each model. **CS 5** shows the training of the model, where 50 epochs were used with a batch size of 100.

```python
vae_models = defaultdict()
for label in train_by_label.keys():
    # Create a DataLoader with tuples of (data, label)
    labeled_data = [(data, label) for data in train_by_label[label]]
    # iter_train_loader is the training data for a specific label, separate from train_loader to
    iter_train_loader = DataLoader(labeled_data, batch_size=100, shuffle=True)

    vae_models[label] = VAE(code_size=10).to(device)
    optimizer = optim.Adam(vae_models[label].parameters(), lr=0.001)
    num_epochs = 50
    for epoch in range(num_epochs):
        vae_models[label].train()
        train_loss = 0
        for batch_idx, (data, _) in enumerate(iter_train_loader):  # Unpack data and ignore label
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, loc, scale = vae_models[label](data)
            loss = loss_function(recon_batch, data, loc, scale)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

    print(f'Label: {label}, Loss: {train_loss / len(train_loader.dataset)}')

    # Latent space visualization for the current label
    vae_models[label].eval()  # Switch to evaluation mode
    latent_points = []

    with torch.no_grad():
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            loc, _ = vae_models[label].encoder(data)  # Get the latent variables
            latent_points.append(loc.cpu().numpy())  # Collect latent space representations

    latent_points = np.concatenate(latent_points, axis=0)  # Stack all latent points
```

Code Snippet 5. The code used to create 10 different VAE models, one for each class label in the MNIST dataset.

Synthetically generated inputs that we will be using as training data are shown in **Figure 4,** where the digits are seen as noisy. This is a typical output from a VAE as it generates points from a distribution. After this stage, 10,000 inputs are generated for each class and are used as training. The test set was the unseen MNIST data that was held-out when training. **CS 6** shows a simple MLP architecture used to train the synthetic and original inputs. Here, the model has a 28 x 28 = 784 input along with a fully connected layer with 200 neurons. The final layer contains 10 neurons where a soft-max function is applied to squeeze the values across all classes between 0 and 1, where the neuron with the highest value is the class that is selected by the model.

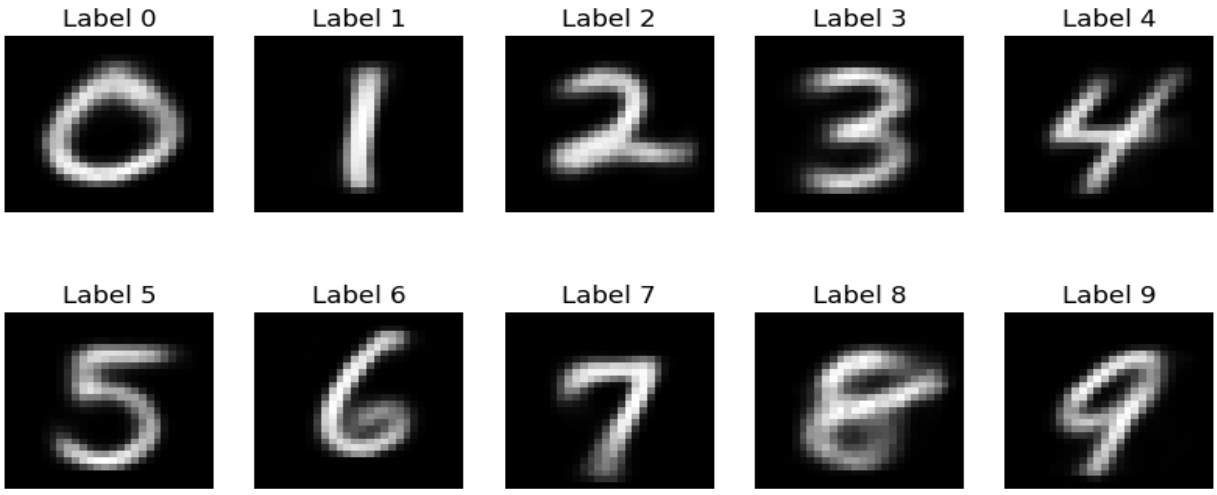

Figure 4. Synthetically generated inputs used as training from each VAE model, one for each class.

```python
# multilayer perceptron
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = nn.Flatten()
        # 28x28 input, 200 neurons in first hidden layer...
        self.fc1 = nn.Linear(28 * 28, 200)
        # 200 input, 200 neurons next layer
        self.fc2 = nn.Linear(200, 200)
        # 200 input, 10 output (i.e. n classes)
        self.fc3 = nn.Linear(200, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

✓ 0.0s

Code Snippet 6. MLP classifier used to train synthetic and non-synthetic data.

6. Handling Imbalanced Datasets

The MNIST dataset is slightly imbalanced, with certain digits being more frequent than others. To address this, we used the **VAE to generate synthetic samples** for the underrepresented classes. These synthetic samples were added to the training set to balance the dataset. Two models were trained, one with synthetic and one with the non-synthetic samples and evaluated. **Figures 5-6** show the confusion matrix for a basic MLP model trained on Imbalanced (Original) MNIST data and synthetic data, respectively. The original dataset

outperformed the synthetically generated one based on the confusion matrix. The original data had a hard time differentiating between 2 and 7, and 4 and 9, where the synthetically trained model had a hard time differentiating between 5 and 8, and 5 and 3. The synthetically trained model relies on the output of a VAE as input where 5, 3, and 8 are mapped around the same space. Thus, a synthetically generated 5 could have some attributes of 3 and 8, and vice versa, explaining this phenomenon. The original and synthetically trained models had an accuracy score across all 10 class labels of **98.05%** and **89.60%** respectively, where the original data outperformed synthetic by **8.45%**. This shows that the synthetically generated dataset is not useful in this context of slightly imbalanced classes. However, with further hyperparameter fine-tuning and construction of the latent space, model performance can potentially exceed the original data by introducing gaussian variations to the input that act as a regularization term.

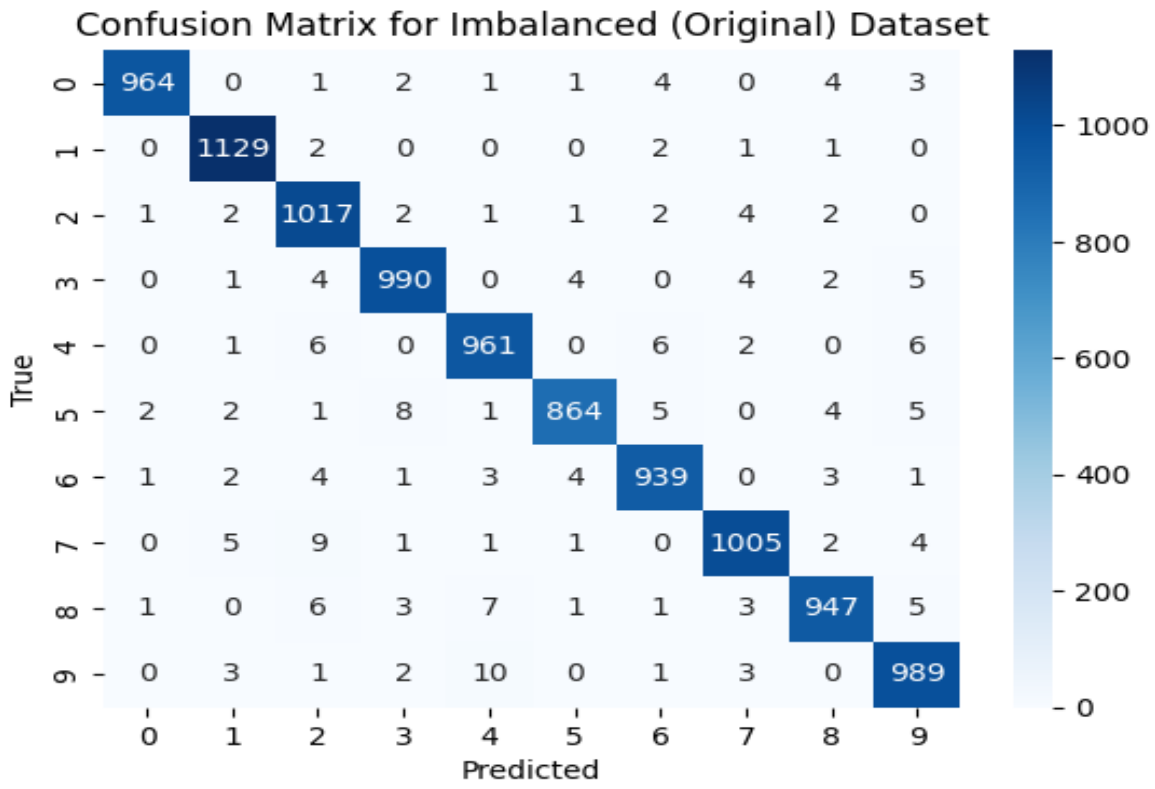

Figure 5. Confusion Matrix for an MLP trained on an Imbalanced Dataset.

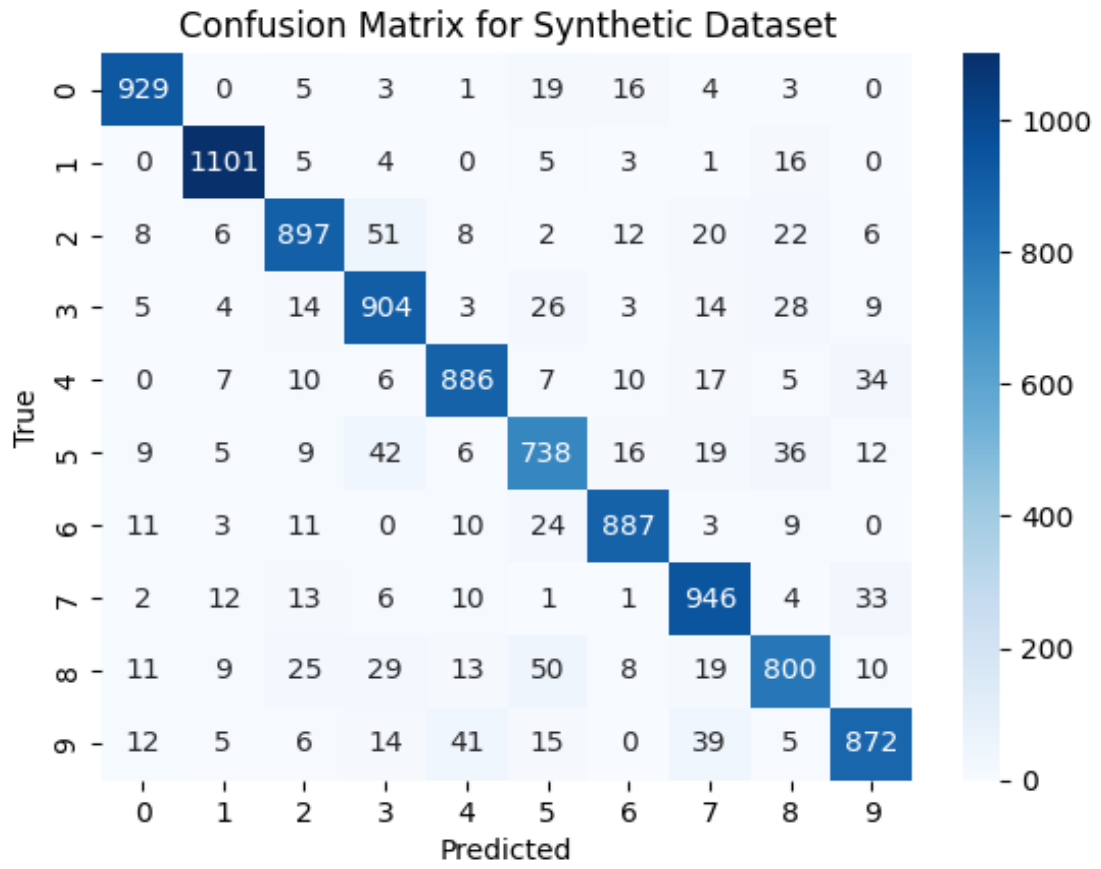

Figure 6. Confusion Matrix for an MLP trained on a Synthetically Generated Dataset via a VAE.

In **Figure 7**, we see the MLP models predictions, trained on the imbalanced MNIST dataset, where it struggles with certain class predictions. Digits such as 2, 7, 4 and 9 appear to be challenging for the model to differentiate. This confusion may be due to the visual similarities between the shapes of these digits, especially when the digits are written in an italics style, where they appear ambiguous even to the human eye. For instance, handwritten 7s may sometimes be mistaken for 2s if the writing style lacks distinct features and vice versa.

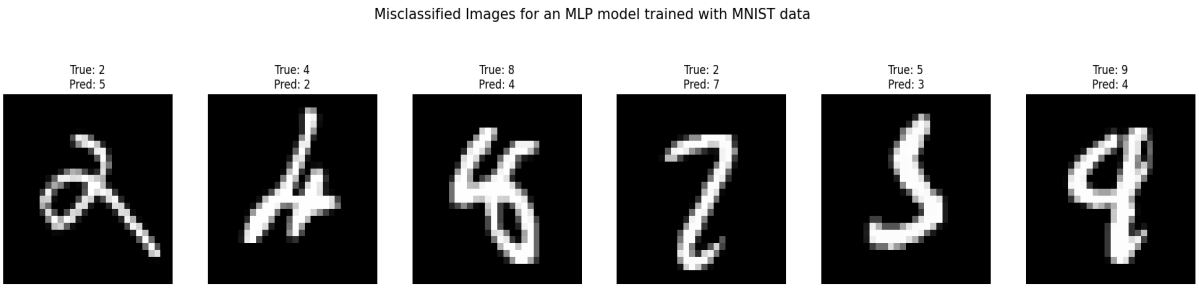

Figure 7. Misclassified Images for an MLP model trained with the original MNIST data.

Figure 8 illustrates incorrectly predicted class labels for an MLP model trained on synthetically generated data. The MLP model has difficulty distinguishing between digits like 5, 8, and 3, where the latent space created by the Variational Autoencoder might have mapped these digits close together, also supported by the confusion matrix in **Figure 6**. This results in overlapping features being assigned to different classes. For example, synthetic digits generated by the VAE for the 5 class could share attributes with the 3 or 8 class, leading to misclassification. We suggest that while VAEs are useful for generating synthetic data, fine-tuning of the latent space is required to ensure that there is no confusion for classes with similar characteristics.

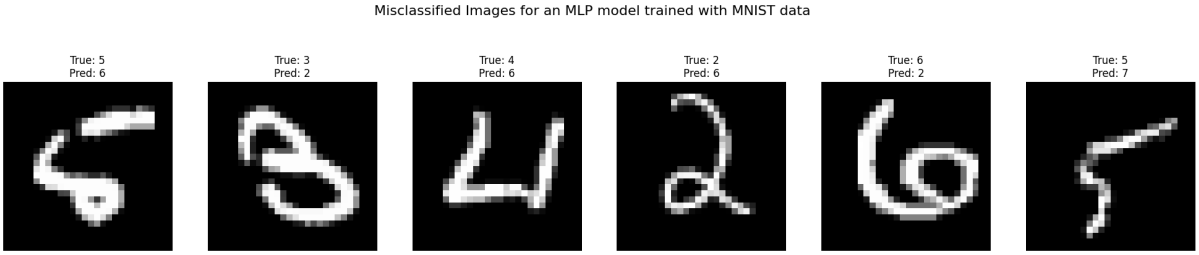

Figure 6. Misclassified Images for an MLP model trained with the original MNIST data.

Concluding Remarks

In conclusion, the latent space was thoroughly examined in a VAE model with different dimensions. It was found that an increase in the dimension decreases the variability between two principal components as there are more latent attributes to explain variability. When synthetic and original MNIST data were compared in an MLP model, the MLP that was trained on the original data outperformed the synthetic version by **8.45%.**

The intuition behind why this oversampling with a potential model performance improvement would prove useful was due to the gaussian noise that a VAE can add from generating synthetic data, acting as a regularization factor and generalizing better to unseen data. In this project, this was not confirmed due to time and computation constraints however, with more hyperparameter fine-tuning and experimentation on different datasets, this approach could potentially prove useful in boosting model performance.

References

Jordan, J. (2018, March 19). *Variational autoencoders.* Jeremy Jordan. https://www.jeremyjordan.me/variational-autoencoders/

Kingma, D. P., & Welling, M. (2013, December 20). *Auto-Encoding Variational Bayes*. ArXiv.org. https://arxiv.org/abs/1312.6114