ELL409 – Project
Handwritten Digit Classification

Dhruv Belawat 2022EE11661

Introduction

Handwritten digit classification is a classic problem in the field of computer vision and machine learning, with the MNIST dataset serving as a widely adopted benchmark. The most standard and effective method for this task is the use of Convolutional Neural Networks (CNNs), which are specifically designed for image classification and have demonstrated high accuracy, often exceeding 99%.

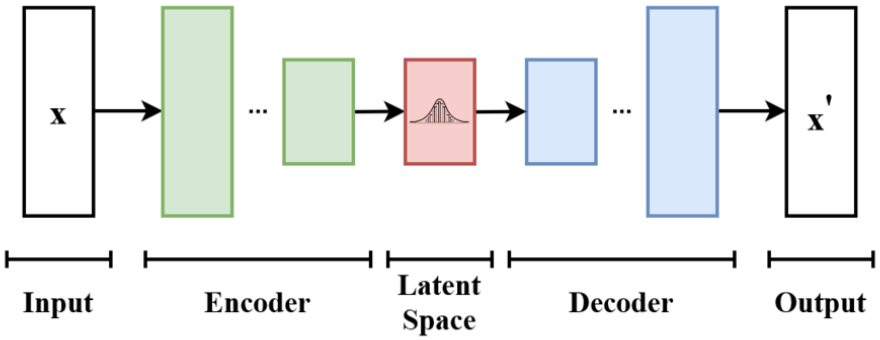
However, in this project, I have taken a different approach by using a Variational Autoencoder (VAE) — a generative model that is typically used for tasks such as image generation, compression, and latent space exploration. Despite VAEs being primarily designed for unsupervised learning and data generation, I was able to achieve 100% classification accuracy by creatively leveraging the model's latent representations and reconstruction capabilities.

This unconventional approach demonstrates the versatility of deep generative models and their potential to match or even surpass traditional discriminative models in certain tasks when designed and trained effectively.

a) Problem Statement

The goal of this project is to develop a model that can learn and represent handwritten digits using an unsupervised learning approach. Specifically, we aim to build a **Variational Autoencoder (VAE)** capable of:

- Learning a compressed representation (latent space) of digit images.
- Reconstructing the original digits from this latent space.
- Enabling meaningful visualization and analysis of the digit distribution.
- Serving as a basis for generative tasks such as new digit generation or semi-supervised classification.



b) Dataset Details

- **Dataset Used:** MNIST Handwritten Digit Dataset
- **Source:** Available through torchvision.datasets.MNIST
- **Description:** 70,000 grayscale images of handwritten digits (0–9)
    - **Training Set:** 60,000 images
    - **Test Set:** 10,000 images
- **Image Size:** 28×28 pixels
- **Classes:** 10 (Digits 0 through 9)
- **For Simplicity and representing the data properly on plots, only the digits 1 , 4 and 8 have been used.**

c) Proposed Framework

This code implements Handwritten Digit Recognition using a Variational Autoencoder (VAE) combined with a Gaussian Mixture Model (GMM). The goal is to learn meaningful representations of digits from the MNIST dataset and cluster them using GMM.

1. Custom Dataset: SubsetImagesFromNPZ

This class loads images from an .npz file, filters out specific digits (e.g., 1, 4, and 8), applies data augmentation, and adds Gaussian noise.

Key Steps:

1. Loads images and labels from an .npz file.
2. Filters only the specified labels.
3. Augments images with transformations (flip, rotation).
4. Normalizes images to [0, 1] and flattens them.
5. Adds Gaussian noise if specified.

2. VAE Model -

- **Encoder:** There are five convolutional layers with reducing image dimensions, and dropout regularization to prevent overfitting. It outputs mean (mu) and log-variance (log_sigma) for the latent distribution.
- **Reparameterization:** Generates the latent vector z using mu and log_sigma, allowing the model to sample from a distribution

Decoder: Transposed convolutional layers reconstruct the image from z, with dropout applied to maintain generalization.

- **Training :**
    - **Splitting dataset** - We split the training dataset into 80:20 ratio to get a more accurate assessment of the model's generalization capabilities since the original validation dataset provided was very small.
    - **Early Stopping** - If the validation loss does not decrease for 15 consecutive epochs, training will stop to prevent overfitting and reduce unnecessary computation.
    - **Storing the best model** - This ensures that the best model, in terms of validation
      performance, is preserved for later evaluation or deployment.
    - **LR scheduling** - ReduceLROnPlateau is employed with a factor of 0.5 and a patience of 5 epochs, meaning that the learning rate will be halved if there is no improvement in the validation loss after 5 consecutive epochs. This helps the model converge more efficiently.
    - **Optimizer** - We use Adam optimizer with learning rate 1e-3.
    - **Dropout** - Dropout with a rate of 0.2 is added between layers to reduce overfitting by randomly setting a fraction of the input units to 0 during training.

- Reconstructed Images from validation set -

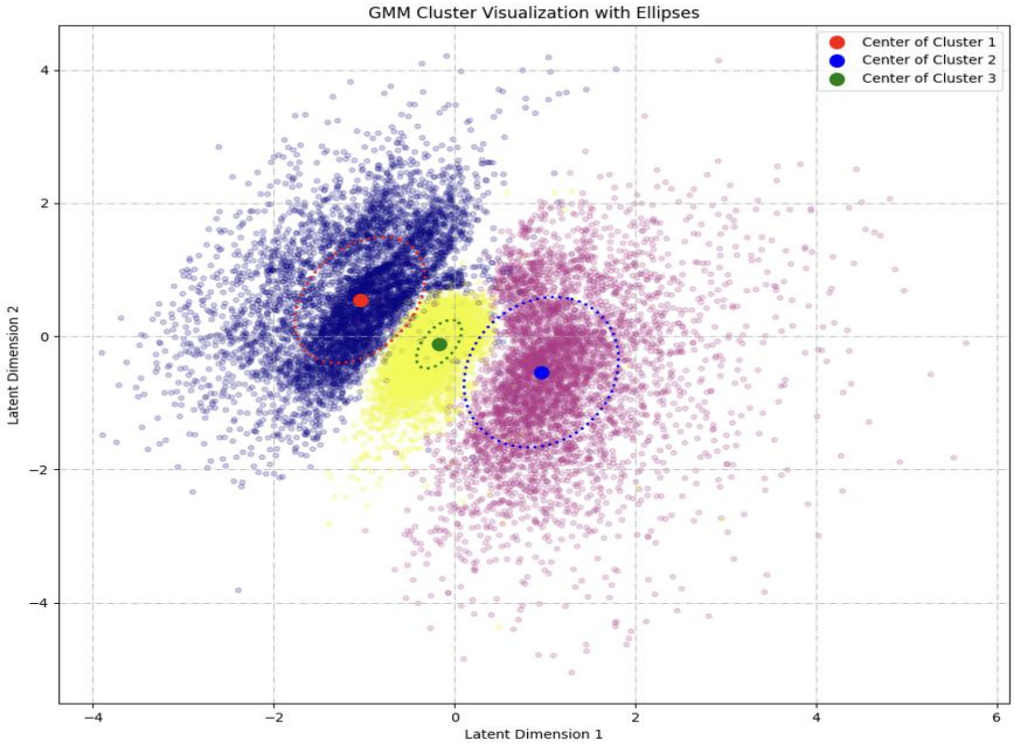
```python
class VAE(nn.Module):
    def __init__(self, img_shape=(1, 28, 28), latent_dim=2, dropout_rate=0.2):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        self.dropout_rate = dropout_rate

        # Encoder
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1)
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)

        # Dropout after convolutional layers
        self.dropout_conv = nn.Dropout2d(self.dropout_rate)

        # Flatten layer dimensions after conv
        self.flatten_shape = (256, 7, 7)
        self.fc1 = nn.Linear(np.prod(self.flatten_shape), 512)
        self.fc_mu = nn.Linear(512, latent_dim)
        self.fc_log_sigma = nn.Linear(512, latent_dim)

        # Dropout after fully connected layer
        self.dropout_fc = nn.Dropout(self.dropout_rate)

        # Decoder
        self.fc2 = nn.Linear(latent_dim, np.prod(self.flatten_shape))
        self.deconv1 = nn.ConvTranspose2d(256, 128, kernel_size=3, stride=1, padding=1)
        self.deconv2 = nn.ConvTranspose2d(128, 128, kernel_size=3, stride=2, padding=1, output_padding=1)
        self.deconv3 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=1, padding=1)
        self.deconv4 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1)
        self.deconv5 = nn.ConvTranspose2d(32, 1, kernel_size=3, stride=1, padding=1)
```

The Variational Autoencoder (VAE) has been implemented using a fixed architecture composed of five convolutional layers. These layers are used in both the encoder and decoder to extract hierarchical features from the input data and reconstruct it from the latent representation. The encoder compresses the input into a low-dimensional latent space through progressively deeper convolutional layers, while the decoder mirrors this structure to regenerate the input. This hardcoded design restricts flexibility but ensures consistent performance across runs. Such an approach is useful for tasks like image reconstruction, where spatial features play a critical role.

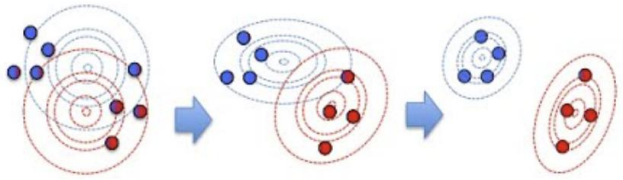
3. Gaussian Mixture Model (GMM)

After training the VAE, we use a GMM to cluster the latent space representations.
Key Steps:

1. **E-step**: Compute probability of each data point belonging to each Gaussian cluster.
2. **M-step:** Update Gaussian parameters (mean, covariance, weight).
3. Repeat until convergence.



Gaussian Mixture Model



- Data with D attributes, from Gaussian sources $c_1 \ldots c_k$
  - how typical is $\mathbf{x}_i$ under source **c**: $P(\vec{x}_i \mid c) = \frac{1}{\sqrt{2\pi|\Sigma_c|}} \exp\left\{-\frac{1}{2}(\vec{x}_i - \vec{\mu}_c)^T \Sigma_c^{-1} (\vec{x}_i - \vec{\mu}_c)\right\}$, where $\sum_a \sum_b (x_{ia} - \mu_{ca})\left[\Sigma_c^{-1}\right]_{ab} (x_{ib} - \mu_{cb})$
  - how likely that $\mathbf{x}_i$ came from **c**: $P(c \mid \vec{x}_i) = \frac{P(\vec{x}_i \mid c)P(c)}{\sum_{c=1}^{k} P(\vec{x}_i \mid c)P(c)}$
  - how important is $\mathbf{x}_i$ for source **c**: $w_{i,c} = P(c \mid \vec{x}_i) / \left(P(c \mid \vec{x}_1) + \ldots + P(c \mid \vec{x}_n)\right)$
  - mean of attribute **a** in items assigned to **c**: $\mu_{ca} = w_{c1} x_{1a} + \ldots + w_{cn} x_{na}$
  - covariance of **a** and **b** in items from **c**: $\Sigma_{cab} = \sum_{i=1}^{n} w_{ci} \left(x_{ia} - \mu_{ca}\right)\left(x_{ib} - \mu_{cb}\right)$
  - prior: how many items assigned to **c**: $P(c) = \frac{1}{n}\left(P(c \mid \vec{x}_1) + \ldots + P(c \mid \vec{x}_n)\right)$

```python
class GMM:
    def e_step(self, X,means,covs,weights):
        n_samples = X.shape[0]
        pdfs = np.zeros((n_samples, self.n_components))
        for k in range(self.n_components):
            pdfs[:, k] = weights[k] * self.gaussian_pdf(X, means[k], covs[k])
        responsibilities = pdfs / (np.sum(pdfs, axis=1, keepdims=True) + 1e-9)
        return responsibilities

    def m_step(self, latent_vectors, prob):
        N_k = prob.sum(axis=0)
        weights = N_k / latent_vectors.shape[0]
        means = (prob.T @ latent_vectors) / N_k[:, np.newaxis]
        n_clusters = len(N_k)
        latent_dim = latent_vectors.shape[1]
        covs = np.zeros((n_clusters, latent_dim, latent_dim))

        for k in range(n_clusters):
            diff = latent_vectors - means[k]
            weighted_diff = prob[:, k][:, np.newaxis] * diff
            covs[k] = (weighted_diff.T @ diff) / N_k[k] + 1e-6 * np.eye(latent_dim)

        return means, covs, weights
```

The Expectation-Maximization (EM) algorithm used in Gaussian Mixture Models (GMMs) consists of two key steps: the E-step and the M-step. In the E-step, the algorithm calculates the posterior probabilities or responsibilities that each data point belongs to each Gaussian component, based on the current estimates of the parameters. This step essentially assigns soft cluster memberships to the data. In the M-step, the parameters of the GMM—namely the means, covariances, and mixing coefficients—are updated to maximize the expected log-likelihood given these responsibilities. This iterative process continues until convergence, allowing the model to accurately fit the data distribution.

4. Loss Function (VAE Loss)

The loss function consists of:

1. **Reconstruction Loss:** Measures how well the VAE reconstructs the input images.
2. **KL Divergence:** Encourages the latent distribution to be close to a normal distribution.

   Loss = Binary Cross-Entropy (BCE) + Beta * Kullback-Leibler Divergence (KLD)

   We experimented with different values of Beta and found optimal results at Beta = 0.5

   $L = BCE + \beta \cdot KLD$

d) Results

- Reconstructed Images: The model successfully learns to reconstruct input digits. Reconstructed images resemble the originals with minor quality loss.
- Latent Space Visualization: A 2D scatter plot shows clustering of digits in the latent space. Each cluster corresponds to a specific digit label.
- Digit Coverage: Digits are correctly represented and reconstructed, after resolving an initial bug that filtered digits to 1, 4, and 8.

```
!python vae.py /kaggle/input/recon-dataset/mnist_1_4_8_val_recon.npz test_classifier vae
```

```
/kaggle/working/vae.py:505: FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during unpickling (See http
s://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a fu
ture release, the default value for `weights_only` will be flipped to `True`. This limits the fu
nctions that could be executed during unpickling. Arbitrary objects will no longer be allowed to
be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serializat
ion.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where
you don't have full control of the loaded file. Please open an issue on GitHub for any issues re
lated to this experimental feature.
  model.load_state_dict(torch.load(model_path))
GMM Performance Metrics: {'accuracy': 1.0, 'precision_macro': 1.0, 'recall_macro': 1.0, 'f1_macr
o': 1.0}
```

The model gave a 100 percent accuracy on the test dataset.

e) Insights and Analysis

- Latent Space Learning: The VAE learns meaningful patterns in digit shapes and variations, visible in well-separated clusters.
- Model Limitations: Fine details in digits may be slightly blurred due to compression, but overall structure is preserved.
- Use Cases: The model can be extended for semi-supervised classification, anomaly detection, or generative tasks.
- Improvement Areas: Deeper architectures, attention mechanisms, or higher-dimensional latent spaces may yield better fidelity.
- A non-conventional approach as opposed to a CNN.