WESTERN SYDNEY UNIVERSITY

SCHOOL of: Computer, Data and Mathematical Sciences

# ASSIGNMENT COVER SHEET

## STUDENT DETAILS

**Name:** Midhun Shyam **Student ID:** 22058122

## SUBJECT AND TUTORIAL DETAILS

**Subject Name:** Probablistic Graphical Models **Subject code:** MATH 7017

**Tutorial Group:** Click or tap here to enter text. **Day:** Monday **Time:** 11:00 - 13:00

**Lecturer or Tutor name:** Prof. Oliver Obst & Stuart Fitzpatrick

## ASSIGNMENT DETAILS

**Title:** Conditional Generation with VAE and GAN **Length:** Click or tap here to enter text.

**Due Date: 17/06/2024** **Date submitted: 17/06/2024**

**Home campus: Parramatta South**

## DECLARATION

**By submitting your work using this link you are certifying that:**

☒ You hold a copy of this submission if the original is lost or damaged.

☒ No part of this submission has been copied from any other student's work or from any other third party (including generative AI) except where due acknowledgment is made in the submission.

☒ No part of this submission has been submitted by you in another (previous or current) assessment, except where appropriately referenced, and with prior permission from the teacher/tutor/supervisor/ Subject Coordinator for this subject.

☒ No part of this submission has been written/produced for you by any other person or technology except where collaboration has been authorised by the teacher/tutor/ supervisor/Subject Coordinator either in the assessment resources section of the Learning Guide for this assessment task, in the instructions for this assessment task, or through vUWS.

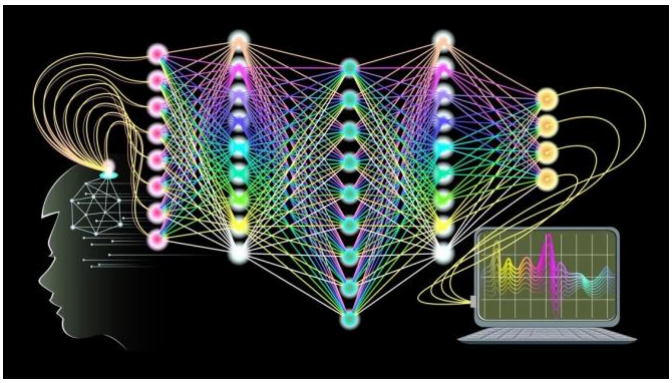
☒ You are aware that this submission will be reproduced and submitted to detection software programs for the purpose of investigating possible breaches of the Student Misconduct Rule, for example, plagiarism, contract cheating, or unauthorised use of generative AI. Turnitin or other tools of investigation may retain a copy of the submission for the purposes of future investigation.

☒ You will not make this submission available to any other person unless required by the University.

***Instructions**: Please complete the requested details in the form, save it and convert to PDF before adding your signature below*

**Student signature:** M. Shyam

Note: An examiner or lecturer/tutor has the right to not mark this assignment if the above declaration has not been completed. Staff may contact you for permission to share a de-identified extract or copy of your submission with students or staff for teaching purposes, following guidelines for requesting and sharing exemplar assessment tasks.



# GENERATIVE MODELS

Conditional Generation with Variational Autoencoders and Generative Adversarial Networks


## ABSTRACT

Image generation models for Kuzushiji-49 Japanese characters



Midhun Shyam

Probabilistic Graphical Models

# Contents

# 1 Introduction

The project focuses on conditional generation using Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). The task involves building and training conditional versions of Variational Autoencoders (CVAE) and Generative Adversarial Networks (CGAN) models to generate Japanese characters from the Kuzushiji-49 dataset. The key objectives are developing efficient generative models that can generate images given a class and style label, the Kuzushiji-49 dataset preparation, implementing a Conditional VAE (C-VAE) and a Conditional GAN (C-GAN), comparing the performance of the two models, and exploring additional experiments with stylistic labels, different architectures, training strategies, and techniques for improving the quality and diversity of the generated images.

# 2 Literature review

## 2.1 Variational Autoencoders

Variational Autoencoders (VAEs) are a class of generative models that combine principles from variational inference and autoencoders. Introduced by Kingma and Welling in 2014, VAEs have become a popular tool in machine learning for generating new data samples and learning efficient data representations (Kingma and Welling, 2014). The model diagram (See Figure 1) explains the architecture of VAEs.

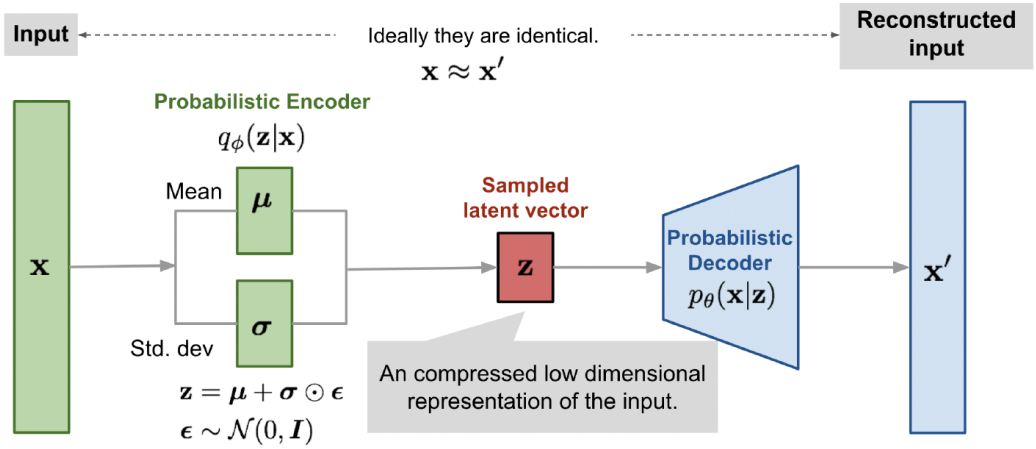



Figure 1: Variational Autoencoder

VAEs are used to:

- Generate new data samples that resemble the training data.
- Learn efficient, low-dimensional representations of high-dimensional data.
- Perform probabilistic inference and data generation.

VAE Architecture

The VAE consists of two main components: the encoder and the decoder.

- **Encoder:** Maps input data $\mathbf{x}$ to a latent space, producing the parameters of the approximate posterior distribution $q_\phi(\mathbf{z}|\mathbf{x})$. Typically, the encoder outputs the mean $\mu$ and the log-variance $\log(\sigma^2)$ of a Gaussian distribution.
- **Decoder:** Maps the latent variables $\mathbf{z}$ back to the data space, generating data samples $\mathbf{x}$ from the latent representation $\mathbf{z}$ using the likelihood $p_\theta(\mathbf{x}|\mathbf{z})$.

Mathematical Notations

- **Prior** $p(\mathbf{z})$**:** Assumed to be a simple distribution, often a standard Gaussian $\mathcal{N}(0, \mathbf{I})$.
- **Posterior** $q_\phi(\mathbf{z}|\mathbf{x})$**:** The encoder's output distribution, approximated as a Gaussian with parameters $\mu$ and $\log(\sigma^2)$.
- **Likelihood** $p_\theta(\mathbf{x}|\mathbf{z})$**:** The decoder's output distribution, generating $\mathbf{x}$ from $\mathbf{z}$.

The Algorithm

**1. Encoder Step**

Compute the mean $\mu$ and log-variance $\log(\sigma^2)$ of the latent variable $\mathbf{z}$ from the input $\mathbf{x}$:

$$\mu, \log(\sigma^2) = \text{Encoder}_\phi(\mathbf{x})$$

Sample $\mathbf{z}$ using the reparameterisation trick:

$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

**2. Decoder Step** Generate the reconstructed input $\hat{\mathbf{x}}$ from $\mathbf{z}$:

$$\hat{\mathbf{x}} = \text{Decoder}_\theta(\mathbf{z})$$

### 3. Loss Function

The VAE loss consists of two parts: the reconstruction loss and the KL divergence.

- **Reconstruction Loss:** Measures how well the VAE reconstructs the input $\mathbf{x}$:
$$\mathcal{L}_{\text{reconstruction}} = -E_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$$
- **KL Divergence:** Ensures the approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$ is close to the prior $p(\mathbf{z})$:
$$\mathcal{L}_{\text{KL}} = \text{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))$$
- **Total Loss:**
$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{reconstruction}} + \mathcal{L}_{\text{KL}}$$

### 4. ELBO (Evidence Lower Bound)

The goal is to maximise the Evidence Lower Bound (ELBO), which serves as a proxy for the log-likelihood of the data. The ELBO is defined as:

$$\text{ELBO} = E_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \text{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))$$

### 5. Reparameterisation Trick

The reparameterisation trick allows backpropagation through the sampling process:

$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

This trick transforms the stochastic sampling into a deterministic process, enabling gradient descent optimisation.

## 2.2 Conditional Variational Autoencoders (CVAEs)

Conditional Variational Autoencoders (CVAE) are an extension of the Variational Autoencoder (VAE) that allow for conditional data generation by incorporating additional information such as class labels into the generative process. This method was developed to address the limitation of VAEs in generating data with specific properties. By conditioning on auxiliary information, CVAEs enable more controlled and directed generation of data samples (Sofeikov, 2023).

Unlike VAEs, which generate samples without specific attributes, CVAEs allow for the generation of samples conditioned on given attributes, such as class labels or style characteristics. This makes CVAEs particularly useful for tasks requiring conditional generation, such as generating images of specific digits or creating specific styles of images such as a thick or thin character of the Kuzushiji 49 Japanese Characters.

CVAEs Algorithm

CVAEs work by conditioning both the encoder and decoder on some additional information $y$, such as class labels. This additional conditioning variable allows the model to generate samples that adhere to specific desired attributes (See Figure 2).

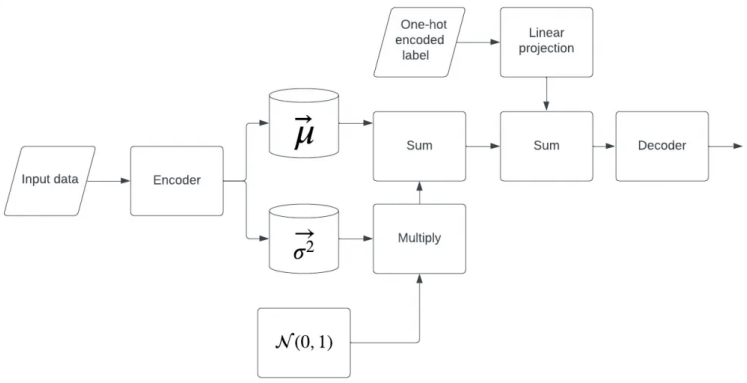



Figure 2: CVAE model diagram
(Sofeikov, 2019)

1. **Encoder:** The encoder maps the input $\mathbf{x}$ and the conditioning variable $y$ to the latent space. It produces the parameters of the approximate posterior distribution $q_\phi(\mathbf{z}|\mathbf{x}, y)$, typically the mean $\mu$ and the log-variance $\log(\sigma^2)$ of a Gaussian distribution:
$$\mu, \log(\sigma^2) = \text{Encoder}_\phi(\mathbf{x}, y)$$

2. **Decoder:** The decoder generates the output $\hat{\mathbf{x}}$ from the latent variables $\mathbf{z}$ conditioned on $y$:
$$\hat{\mathbf{x}} = \text{Decoder}_\theta(\mathbf{z}, y)$$

3. **Reparameterisation Trick:** The latent variable $\mathbf{z}$ is sampled using the reparameterisation trick:
$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

4. **Loss Function:** The loss function for CVAEs is similar to that of VAEs, comprising the reconstruction loss and the KL divergence:
$$\mathcal{L}_{\text{reconstruction}} = -E_{q_\phi(\mathbf{z}|\mathbf{x},y)}[\log p_\theta(\mathbf{x}|\mathbf{z}, y)]$$
$$\mathcal{L}_{\text{KL}} = \text{KL}(q_\phi(\mathbf{z}|\mathbf{x}, y)\|p(\mathbf{z}))$$
$$\mathcal{L}_{\text{CVAE}} = \mathcal{L}_{\text{reconstruction}} + \mathcal{L}_{\text{KL}}$$

### Mathematical Formulation

The key idea behind CVAEs is to model the conditional probability $p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})$ instead of the marginal probability $p_\theta(\mathbf{x}|\mathbf{z})$. The joint distribution over the observed variable $\mathbf{x}$, the latent variable $\mathbf{z}$, and the conditioning variable $\mathbf{y}$ is given by:

$$p_\theta(\mathbf{x}, \mathbf{z}|\mathbf{y}) = p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})p_\theta(\mathbf{z}|\mathbf{y})$$

During the training process, the encoder learns the approximate posterior distribution $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y})$, and the decoder learns the likelihood $p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})$. The objective is to maximise the Evidence Lower Bound (ELBO):

$$\text{ELBO} = E_{q_\phi(\mathbf{z}|\mathbf{x},\mathbf{y})}[\log p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})] - \text{KL}(q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y})\|p_\theta(\mathbf{z}|\mathbf{y}))$$

The training process involves optimising the same ELBO loss function used for VAEs:

```
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum
        ')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

### Implementation Details

In CVAEs, the encoder and decoder are modified to incorporate the conditioning variable $\mathbf{y}$. The conditioning variable is often represented as a one-hot encoded vector.

- **Encoder:** The encoder takes both the input $\mathbf{x}$ and the conditioning variable $\mathbf{y}$ and produces the parameters of the latent distribution $q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y})$.
- **Decoder:** The decoder takes the latent variable $\mathbf{z}$ and the conditioning variable $\mathbf{y}$ to generate the reconstructed input $\hat{\mathbf{x}}$.

### Differences from VAEs

Unlike VAEs, which learn a probabilistic mapping from a latent space to the data space without any conditional inputs, CVAEs introduce a conditioning variable $\mathbf{y}$ that represents additional information such as class labels. This conditioning variable is used both during the encoding and decoding processes, enabling the generation of data that conforms to specific conditions.

The key difference between CVAEs and VAEs is the conditioning variable $y$. In CVAEs, the generation process is controlled by $y$, allowing the model to generate samples with specific properties. For instance, in generating images of digits, $y$ could be a one-hot encoded vector representing the digit class.

Implementation example with Class and Style Labels

CVAEs can be conditioned on both class and style labels. The class label $y_{\text{class}}$ specifies the type of object to generate, while the style label $y_{\text{style}}$ controls the appearance of the object. The encoder and decoder are conditioned on both $y_{\text{class}}$ and $y_{\text{style}}$:

- **Encoder:**
$$\mu, \log(\sigma^2) = \text{Encoder}_\phi(\mathbf{x}, y_{\text{class}}, y_{\text{style}})$$
- **Decoder:**
$$\hat{\mathbf{x}} = \text{Decoder}_\theta(\mathbf{z}, y_{\text{class}}, y_{\text{style}})$$

This conditioning allows the CVAE to generate samples that not only belong to a specific class but also exhibit a particular style, making the model more versatile and powerful for various generative tasks.

CVAEs enhance the capabilities of traditional VAEs by enabling conditional data generation, making them suitable for applications where specific properties or attributes need to be controlled in the generated samples (Sofeikov, 2023). The incorporation of class labels or other auxiliary information provides a more powerful framework for generative modelling.

## 2.3 Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow et al. in 2014, have become one of the most widely used and researched generative models in machine learning. GANs consist of two neural networks, the generator and the discriminator, that compete against each other in a zero-sum game framework (Goodfellow et al., 2014).

Generative Adversarial Networks represent a significant advancement in generative modeling, offering powerful capabilities for generating realistic data. Continued research and development are likely to further enhance their performance and applicability across various fields.

Generative modeling is a type of unsupervised learning that aims to model the distribution of a dataset to generate new samples from the same distribution. Traditional ap-

proaches include density estimation methods, which often struggle with high-dimensional data (Koller and Friedman, 2009).

GANs consist of two main components: the generator $G$ and the discriminator $D$. The generator $G$ takes a random noise vector $z$ and maps it to the data space, creating fake samples. The discriminator $D$ evaluates samples and attempts to distinguish between real samples from the training data and fake samples generated by $G$. The goal of $G$ is to fool $D$ into classifying fake samples as real, while $D$ aims to correctly identify real and fake samples (See Figure 3) (Goodfellow et al., 2014).

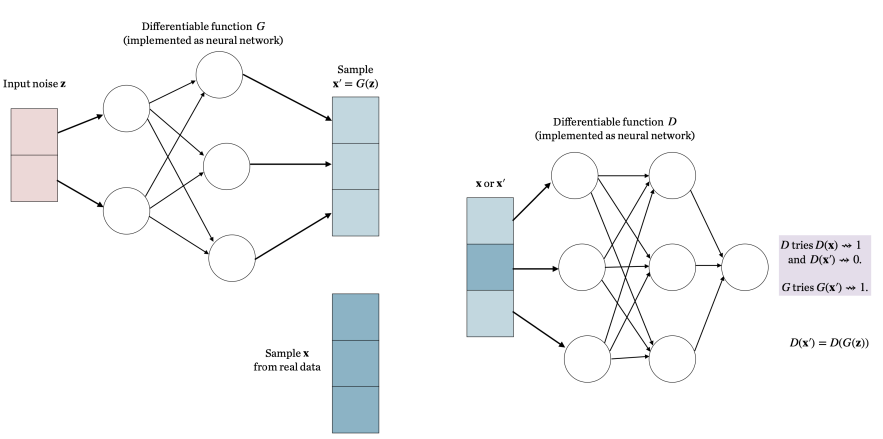


Figure 3: Generative Adversarial Networks

Algorithm

Mathematically, the GAN framework can be described as a minimax game with the following objective function:

$$\min_G \max_D V(D,G) = E_{x\sim p_{\text{data}}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]. \tag{1}$$

The GAN training algorithm involves iteratively updating the generator and discriminator:

1. Sample a batch of real data samples $\{x^{(1)}, \ldots, x^{(m)}\}$ from the training data distribution $p_{\text{data}}(x)$.

2. Sample a batch of random noise vectors $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution $p_z(z)$.

3. Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))\right]. \tag{2}$$

4. Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))). \tag{3}$$

Applications of GANs

GANs have been successfully applied in various domains, including image generation, video prediction, and data augmentation. They are particularly notable for their ability to generate realistic high-resolution images (Goodfellow et al., 2020).

Challenges and scope

Despite their success, GANs face several challenges, such as training instability, mode collapse, and the difficulty of evaluating generated samples. Ongoing research aims to address these issues by developing new architectures, training techniques, and evaluation metrics (Goodfellow et al., 2020; Arjovsky et al., 2017).

## 2.4 Conditional Generative Adversarial Networks (C-GAN)

Generative Adversarial Networks (GANs) have been widely used for generating realistic images. Conditional Generative Adversarial Networks (CGANs) are an extension of GANs where additional information is provided to both the generator and discriminator. CGANs incorporates additional information, such as class labels or data from other modalities, to condition the image generation process. This approach enhances the control over the generated content Mirza and Osindero (2014). CGANs modify the standard GAN framework by conditioning both the generator and discriminator on auxiliary information $y$. This information can be anything from class labels to other modalities of data Goodfellow et al. (2014).

The generator $G$ in a CGAN takes as input both a noise vector $z$ and the conditional information $y$, and generates an image $G(z|y)$. The discriminator $D$ evaluates both the generated image and the real image, along with the conditional information, and tries to distinguish between them (See Figure 4) (Mirza and Osindero, 2014).

The objective function for CGANs can be formulated as:

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x|y)] + E_{z \sim p_z(z)}[\log(1 - D(G(z|y)))]. \tag{4}$$

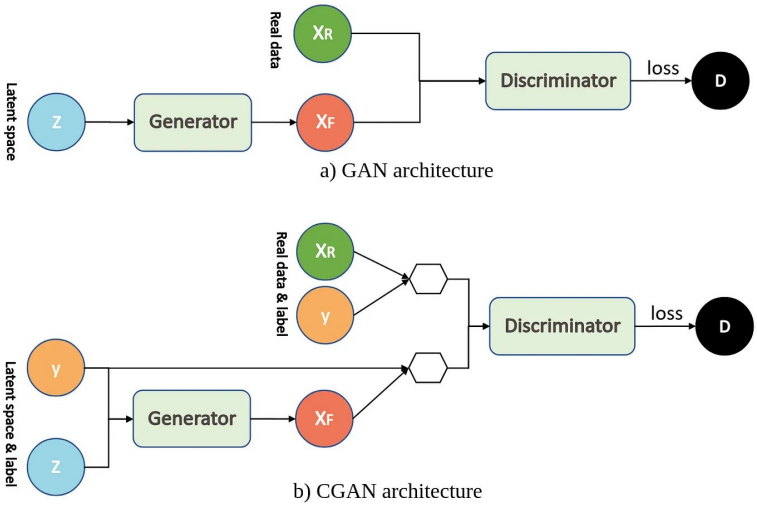



Figure 4: Conditional Generative Adversarial Networks (Bagheri, 2019)

Algorithm

The training algorithm for CGANs involves the following steps:

1. Sample a batch of real data samples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ from the training data distribution $p_{\text{data}}(x)$ and the conditional distribution $p(y)$.

2. Sample a batch of random noise vectors $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution $p_z(z)$.

3. Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D(x^{(i)}|y^{(i)}) + \log(1 - D(G(z^{(i)}|y^{(i)}))) \right]. \tag{5}$$

4. Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}|y^{(i)}))). \tag{6}$$

Applications of CGANs

CGANs have been effectively used in various image generation tasks, including image-to-image translation, where the model learns to map images from one domain to another, and image super-resolution, where low-resolution images are enhanced to high-resolution versions Isola et al. (2017); Ledig et al. (2017).

Challenges and scope

Despite their success, CGANs face challenges such as mode collapse and training instability. Future research aims to address these issues by improving network architectures and training techniques Goodfellow et al. (2020).

Conditional Generative Adversarial Networks provide a powerful framework for controlled image generation. By conditioning on additional information, CGANs offer enhanced flexibility and control over the generated content, making them a valuable tool in various image generation tasks.

# 3 Dataset preparation

The Kuzushiji-49 dataset is a comprehensive collection of 270,912 grayscale images of handwritten Japanese characters, each measuring 28x28 pixels. This dataset, part of the Kuzushiji project, encompasses 49 distinct classes of Kuzushiji, the ancient Japanese cursive script. It is divided into a training set with 232,365 images and a validation set with 38,547 images, facilitating effective training and evaluation of machine learning models. Each class represents a unique Kuzushiji character from historical documents, providing a diverse set of samples for research.

Primarily used for training and testing handwritten character recognition algorithms, the Kuzushiji-49 dataset also plays a significant role in the digitization and preservation of historical documents. Additionally, it serves educational purposes, advancing the study of machine learning and Japanese cultural heritage. The dataset is available for access and download from the Kuzushiji GitHub repository(Clanuwat et al., 2018), which provides detailed documentation and resources to support its use in various projects.

### 3.0.1 Pre-processing using Python

The dataset preparation begins with loading and merging the training and test images and their corresponding labels from `.npz` files. The images are normalized to have pixel values between 0 and 1 by converting the data type to `float32` and dividing by 255. This normalization ensures consistency across the dataset.

A function, `countForegroundPixels`, counts the number of pixels in an image that exceed a given threshold, identifying the presence of foreground elements. Using this function, the median number of foreground pixels is computed for each class. Each image's style is then classified as 0 or 1 based on whether its foreground pixel count is below or above the median for its class.

The class labels and style labels are converted into one-hot encoded vectors using the `to_one_hot` function, creating a binary matrix representation of the labels. The original images are reshaped into flat vectors and concatenated with their one-hot encoded class and style labels, forming a comprehensive feature set for each image.

Finally, the dataset is split into training and test sets using the `train_test_split` function, with 70% of the data used for training and 30% for testing. This split allows for model training on the majority of the data while reserving a portion for performance evaluation.

| Name | Type | Size | Value |
|---|---|---|---|
| class1H | Tensor | (270912, 49) | Tensor object of torch module |
| classLabels | Array of uint8 | (270912,) | [30 19 20 ... 1 27 47] |
| images | Array of float32 | (270912, 28, 28) | [[[0. 0. 0. ... 0. 0. 0. ... |
| img | Array of float32 | (270912, 784) | [[0. 0. 0. ... 0. 0. 0.]<br>[0. 0. 0. ... 0. 0. 0.] |
| medianClasses | dict | 49 | {0:312.0, 1:247.0, 2:141.0, 3:219.0, 4:288.0, 5:241.0, 6:222.0, 7:153.<br>... |
| style | Array of int64 | (270912,) | [0 1 0 ... 1 1 1] |
| style1H | Tensor | (270912, 2) | Tensor object of torch module |
| x | Array of float32 | (270912, 835) | [[0. 0. 0. ... 0. 1. 0.]<br>[0. 0. 0. ... 0. 0. 1.] |
| X_test | Array of float32 | (81274, 835) | [[0. 0. 0. ... 0. 0. 1.]<br>[0. 0. 0. ... 0. 1. 0.] |
| X_train | Array of float32 | (189638, 835) | [[0. 0. 0. ... 0. 1. 0.]<br>[0. 0. 0. ... 0. 0. 1.] |

Figure 5: Overview of variables in the prepared dataset

### 3.0.2 Variable Overview

- `class1H`: Tensor of shape (270,912, 49), representing the one-hot encoded class labels.
- `classLabels`: Array of `uint8` with 270,912 elements, representing the original

class labels.

- `images`: Array of `float32` with shape (270,912, 28, 28), representing the normalized images.
- `img`: Array of `float32` with shape (270,912, 784), representing the flattened image vectors.
- `medianClasses`: Dictionary with 49 elements, representing the median foreground pixel counts for each class.
- `style`: Array of `int64` with 270,912 elements, representing the style classification (0 or 1) for each image.
- `style1H`: Tensor of shape (270,912, 2), representing the one-hot encoded style labels.
- `x`: Array of `float32` with shape (270,912, 835), representing the concatenated feature set for each image.
- `X_train`: Array of `float32` with shape (189,638, 835), representing the training set.
- `X_test`: Array of `float32` with shape (81,274, 835), representing the test set.

### 3.0.3 Visualising the images based on style

The "Thick Images" dataset consists of handwritten Japanese characters from the Kuzushiji-49 dataset, presented in a bold and thickened style (See Figure 6. Each of the 49 classes is visually represented, showcasing the diversity of ancient Japanese script with clear, bold strokes. This visualisation aids in understanding the variability and distinct features of each character, making it easier to train and evaluate models that can recognise and differentiate these stylised characters.

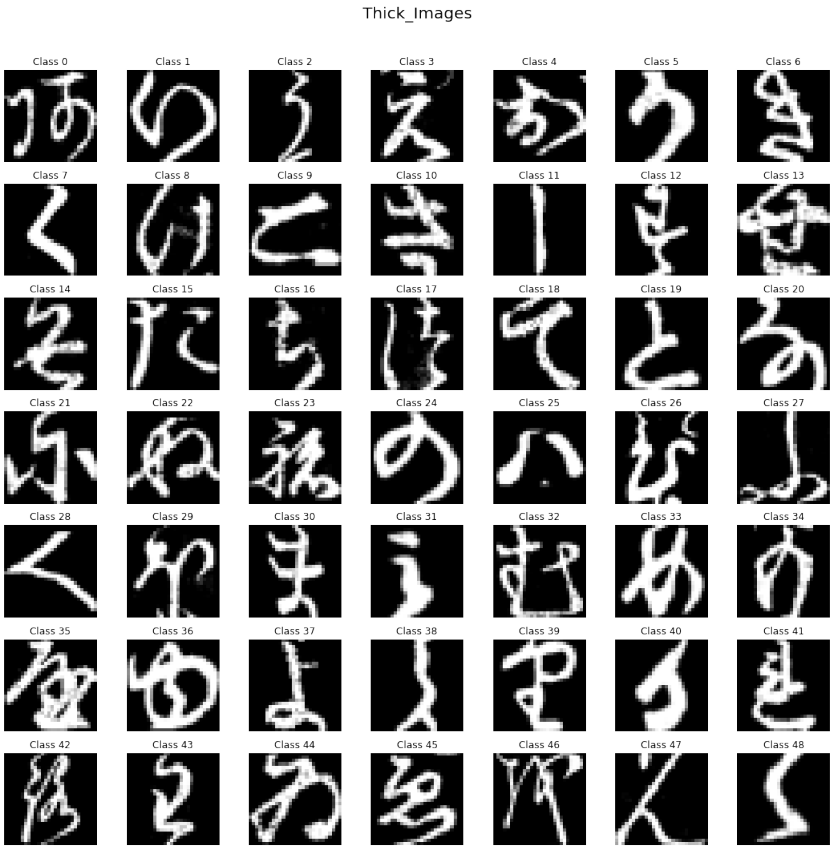


Figure 6: Thick style images from each class

The "Thin Images" dataset features the same set of handwritten Japanese characters but in a thin and less bold style (See Figure7. These images highlight the intricate and delicate aspects of the characters, showing finer details that are essential for accurate recognition. The thinner strokes emphasise the subtle differences and nuances between each character class, which is critical for developing models capable of high precision in distinguishing these ancient scripts.

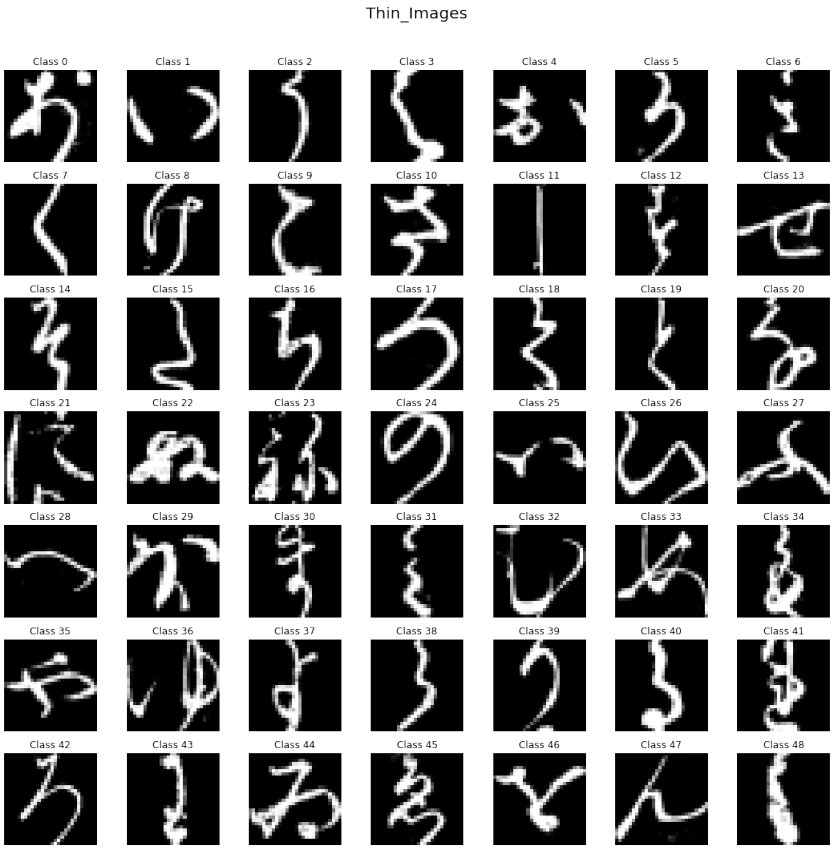


Figure 7: Thin style images from each class

# 4 Model Implementation

## 4.1 Libraries

The implementation of a Conditional Variational Autoencoder (C-VAE) and a Conditional Generative Adversarial Network (CGAN) leverages several essential libraries. PyTorch is used as the primary deep learning framework, with its `nn` module facilitating the construction of neural networks. Data loading and preprocessing are handled by `torch.utils.data` utilities, while `torchvision.utils` provides tools for creating image grids. Numerical operations are performed using NumPy, and visualization is accomplished with Matplotlib. The `torch.nn.functional` module offers a functional interface for various neural network operations, and optimization algorithms are implemented using `torch.optim`. Additionally, the `sklearn.model_selection` module is used for splitting data into training and test sets, and the `torchvision` library provides additional computer vision utilities.

```
import torch                                    # PyTorch library
```

```
from torch import nn                    # Neural network module
from torch.utils.data import DataLoader, TensorDataset  # Data loading
    utility
from torchvision.utils import make_grid # Utility to create image grids
import matplotlib.pyplot as plt         # Plotting library
import numpy as np                      # Numerical operations library
from torch.nn import functional as F  # Functional interface for neural
    network operations
import os                               # Operating system interface
from sklearn.model_selection import train_test_split  # Train-test
    splitting utility
import torch.optim as optim             # Optimization algorithms
import torchvision                      # Computer vision tools
```

## 4.2 Conditional Variational Autoencoder (C-VAE)

A Conditional VAE is implemented to take a Kuzushiji-49 image and its associated labels as input. The model learns to generate new images resembling the input image given a class label and the specified style (thick or thin).

### 4.2.1 A: Design the CVAE Neural Network Architecture

The Conditional Variational Autoencoder (C-VAE) model is implemented using PyTorch. The device for computation is set to use a GPU if available; otherwise, it falls back to using the CPU. The key parameters for the model are defined as follows: the input dimension is 784 (corresponding to 28x28 pixel images), the latent dimension is set to 2, and the hidden dimension is set to 144 (calculated as $12 \times 12$). The dataset comprises 49 classes, with 2 defined styles. The model is trained for 2000 epochs with a batch size of 100 and a learning rate of 0.001.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
inputDim = 784
latentDim = 2
hiddenSqrt = 12
hiddenDim = hiddenSqrt * hiddenSqrt
classDim = 49
styleDim = 2
epochs = 2000
batchSize = 100
learningRate = 0.001
```

### Create DataLoader

The training and test datasets are created by combining the image data with their corresponding class and style labels. These datasets are then loaded into DataLoaders to facilitate batch processing during training.

```
trainDataset = torch.utils.data.TensorDataset(torch.tensor(X_train[:, :
    inputDim]), torch.tensor(X_train[:, inputDim:]))
testDataset = torch.utils.data.TensorDataset(torch.tensor(X_test[:, :
    inputDim]), torch.tensor(X_test[:, inputDim:]))

trainLoader = torch.utils.data.DataLoader(trainDataset, batch_size=
    batchSize, shuffle=True)
testLoader = torch.utils.data.DataLoader(testDataset, batch_size=
    batchSize, shuffle=False)
```

### ConditionalVAE Class

The `ConditionalVAE` class defines the architecture of the C-VAE model. It includes an encoder, a decoder, and a label projector. The encoder consists of fully connected layers that take the concatenated input (image, class, and style) and output the mean and log variance of the latent space. The decoder reconstructs the input from the latent space, conditioned on the class and style labels. The reparameterization trick is used to sample from the latent space.

```
class ConditionalVAE(nn.Module):
    def __init__(self, inputDim, hiddenDim, latentDim, classDim, styleDim
        ):
        super(ConditionalVAE, self).__init__()

        self.inputDim = inputDim
        self.hiddenDim = hiddenDim
        self.latentDim = latentDim
        self.classDim = classDim
        self.styleDim = styleDim

        # Encoder
        self.fc1 = nn.Linear(inputDim + classDim + styleDim, hiddenDim)
        self.fcMu = nn.Linear(hiddenDim, latentDim)
        self.fcLogVar = nn.Linear(hiddenDim, latentDim)

        # Decoder
        self.fc3 = nn.Linear(latentDim + classDim + styleDim, hiddenDim)
        self.fc4 = nn.Linear(hiddenDim, inputDim)

        # Label projector
        self.labelProjector = nn.Sequential(
            nn.Linear(classDim + styleDim, hiddenDim),
            nn.ReLU(),
```

```
        )

    def encode(self, x, c, s):
        # Concatenate input with class and style labels
        x = torch.cat((x, c, s), dim=1)
        h = F.relu(self.fc1(x))
        mu = self.fcMu(h)
        logVar = self.fcLogVar(h)
        return mu, logVar

    def reparameterize(self, mu, logVar):
        std = torch.exp(0.5 * logVar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def condition_on_label(self, z, y):
        projectedLabel = self.labelProjector(y.float())
        return torch.cat((z, projectedLabel), dim=1)

    def decode(self, z, c, s):
        y = torch.cat((c, s), dim=1)
        z = torch.cat((z, c, s), dim=1)
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x, c, s):
        mu, logVar = self.encode(x.view(-1, self.inputDim), c, s)
        z = self.reparameterize(mu, logVar)
        return self.decode(z, c, s), mu, logVar

    def sample(self, num_samples, c, s):
        with torch.no_grad():
            # Generate random noise
            z = torch.randn(num_samples, self.latentDim).to(device)
            # Pass the noise through the decoder to generate samples
            samples = self.decode(z, c, s)
        # Return the generated samples
        return samples
```

### Loss Function

The loss function used in training the C-VAE is a combination of the reconstruction loss (binary cross-entropy) and the KL divergence loss. The reconstruction loss measures how well the decoder is able to reconstruct the input data, while the KL divergence loss ensures that the latent space follows a standard normal distribution.

```
def lossFunction(reconX, x, mu, logVar):
    BCE = F.binary_cross_entropy(reconX, x.view(-1, inputDim), reduction=
        'sum')
    KLD = -0.5 * torch.sum(1 + logVar - mu.pow(2) - logVar.exp())
    return BCE + KLD
```

### Model and Optimizer

The model is instantiated using the defined architecture, and the Adam optimizer is used for training the model with the specified learning rate.

```
model = ConditionalVAE(inputDim, hiddenDim, latentDim, classDim, styleDim
    ).to(device)
optimizer = optim.Adam(model.parameters(), lr=learningRate)
```

### 4.2.2 B: Train the CVAE Model

The training process for the Conditional Variational Autoencoder (C-VAE) involves iterating over the dataset for a specified number of epochs. In each epoch, the model processes batches of images, performs a forward pass to reconstruct the images, and computes the loss using the defined loss function, which combines the reconstruction loss and the KL divergence. The optimizer then updates the model parameters to minimize the loss.

```
# Train the model
for epoch in range(epochs):
    for i, (images, labels) in enumerate(trainLoader):
        images = images.to(device)
        classLabels = labels[:, :classDim].to(device)
        styleLabels = labels[:, classDim:].to(device)

        # Forward pass
        reconImages, mu, logVar = model(images, classLabels, styleLabels)
        loss = lossFunction(reconImages, images, mu, logVar)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print('Epoch [{}/{}], Training Loss: {:.4f}'
                  .format(epoch + 1, epochs, loss.item()))

# Save the trained model
torch.save(model.state_dict(), '/bigdata/users/postgrad/mshyam/PGM Models
    /PGM/cvaeModel.pth')
print('Model saved to /bigdata/users/postgrad/mshyam/PGM Models/PGM/
    cvaeModel.pth')
```

The trained model is saved to a specified path after the training process is completed. This allows for future use of the model without needing to retrain it from scratch.

### 4.2.3 C: Test the Model

The testing process involves evaluating the trained model on the test dataset. The model is set to evaluation mode, and no gradients are computed during this phase to save memory and computation. The test loss is calculated over the entire test dataset. Additionally, original and reconstructed images are displayed to visually assess the performance of the model.

```
# Test the model
model.eval()
total_loss = 0
with torch.no_grad():
    for i, (images, labels) in enumerate(testLoader):
        images = images.to(device)
        classLabels = labels[:, :classDim].to(device)
        styleLabels = labels[:, classDim:].to(device)

        reconImages, mu, logVar = model(images, classLabels, styleLabels)
        loss = lossFunction(reconImages, images, mu, logVar)
        total_loss += loss.item()

        if i == 0:
            print('Original images')
            out_imgs = torchvision.utils.make_grid(images.reshape(-1, 1,
                28, 28)[:8])
            plt.imshow(out_imgs.permute(1, 2, 0).cpu().numpy())
            plt.show()
            print('Reconstructed images')
            out_imgs = torchvision.utils.make_grid(reconImages.reshape
                (-1, 1, 28, 28)[:8])
            plt.imshow(out_imgs.permute(1, 2, 0).cpu().numpy())
            plt.show()

print('Average test loss: {:.4f}'.format(total_loss / len(testLoader)))
```

During testing, the average test loss is computed and printed to provide a quantitative measure of the model's performance on unseen data. This loss can be used to compare different models or configurations and to monitor the model's ability to generalize to new data.

### 4.2.4 D: Generate and Display Images

To visualize the generative capabilities of the C-VAE, the function `generateimages` is defined. This function generates new images by sampling from the latent space, conditioned on specific class and style labels. The generated images are displayed using the `show_images` function.

```
def show_images(images, labels):
```

```
    grid_img = torchvision.utils.make_grid(images.reshape(-1, 1, 28, 28),
        nrow=5)
    plt.imshow(grid_img.permute(1, 2, 0).numpy())
    plt.title(f"Class: {labels[0]}")  # Assuming all labels are the same
    plt.show()

def generateimages(model, num_samples, class_label, style_label):
    # Create one-hot encoded class and style labels
    class_one_hot = to_one_hot(torch.LongTensor([class_label] *
        num_samples), classDim).to(device)
    style_one_hot = to_one_hot(torch.LongTensor([style_label] *
        num_samples), styleDim).to(device)

    # Generate images using the CVAE model
    generated_images = model.sample(num_samples, class_one_hot,
        style_one_hot)

    # Display the generated images
    show_images(generated_images.cpu().detach(), [class_label] *
        num_samples)

generateimages(model, num_samples=10, class_label=0, style_label=0)
```

The function `show_images` arranges the generated images into a grid and displays them using `matplotlib`. The `generateimages` function creates one-hot encoded vectors for the specified class and style labels, samples images from the C-VAE model, and calls `show_images` to display the results.

## 4.3 Conditional Generative Adversarial Network (C-GAN)

A Conditional GAN is implemented that takes a random noise vector and a class and style label as inputs to the generator, and produces a specified Japanese character. The discriminator is also conditioned on the labels.

### 4.3.1 A: Design the CGAN Neural Network Architecture

The data is first renormalised to values between -1 and 1 to match the Tanh activation function used in the generator.

```
x = x * 2 - 1  # Renormalize values in x to be between -1 and 1
```

### 4.3.2 Generator Architecture

The generator class is defined with the ability to generate images from random noise, class labels, and style labels. It includes an input layer that combines the noise, class, and style vectors, followed by a ReLU activation, and finally an output layer that produces the generated image with Tanh activation to ensure the output values are between -1 and 1.

```
class Generator(nn.Module):
    def __init__(self, latent_dim, class_dim, style_dim, img_dim):
        super(Generator, self).__init__()
        self.gen = nn.Sequential(
            nn.Linear(latent_dim + class_dim + style_dim, 256),  # Input
                layer combining noise, class, and style
            nn.ReLU(),  # Activation function
            nn.Linear(256, img_dim),  # Output layer to generate image
            nn.Tanh(),  # Normalize output to be between -1 and 1
        )

    def forward(self, noise, class_labels, style_labels):
        x = torch.cat([noise, class_labels, style_labels], dim=1)  #
            Concatenate noise, class labels, and style labels
        return self.gen(x)  # Generate image
```

### 4.3.3 Discriminator Architecture

The discriminator class is defined to distinguish between real and fake images. It takes an image, class labels, and style labels as input, and passes them through a series of linear layers with ReLU activation, ending with a Sigmoid activation function to output a probability indicating whether the image is real or fake.

```
class Discriminator(nn.Module):
    def __init__(self, img_dim, class_dim, style_dim):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            nn.Linear(img_dim + class_dim + style_dim, 256),  # Input
                layer combining image, class, and style
            nn.ReLU(),  # Activation function
            nn.Linear(256, 1),  # Output layer to predict real/fake
            nn.Sigmoid(),  # Output probability
        )

    def forward(self, img, class_labels, style_labels):
        x = torch.cat([img, class_labels, style_labels], dim=1)  #
            Concatenate image, class labels, and style labels
        return self.disc(x)  # Discriminate real/fake
```

### 4.3.4 B: Training the CGAN Model

The training loop involves alternately training the discriminator and the generator. For the discriminator, the goal is to maximize the log probability of correctly classifying real images and minimize the log probability of classifying fake images as real. For the generator, the goal is to maximize the log probability of the discriminator classifying generated (fake) images as real.

Loss function

The loss function used for both the generator and the discriminator is binary cross-entropy loss (BCELoss), which is suitable for binary classification tasks. Binary Cross-Entropy (BCE) loss is a common loss function used for binary classification tasks, and it measures the performance of a classification model whose output is a probability value between 0 and 1 (Goodfellow et al., 2016).

```
# Training the model

device = 'cuda' if torch.cuda.is_available() else 'cpu'  # Use GPU if
    available, otherwise use CPU

# Hyperparameters etc.
img_x = img_y = 28
img_dim = img_x * img_y  # Flattened K49 28x28=784
latent_dim = 64
class_dim = 49
style_dim = 2
batch_size = 128
lr = 0.0001

# Initialize generator and discriminator
generator = Generator(latent_dim, class_dim, style_dim, img_dim).to(
    device)
discriminator = Discriminator(img_dim, class_dim, style_dim).to(device)

# Optimizers
opt_gen = torch.optim.Adam(generator.parameters(), lr=lr)
opt_disc = torch.optim.Adam(discriminator.parameters(), lr=lr)

# Loss function
criterion = nn.BCELoss()

loader = DataLoader(x, batch_size=batch_size, shuffle=True)

for epoch in range(epochs):
    for batch_idx, real in enumerate(loader):
        real = real.to(device)
        batch_size = real.shape[0]
```

```
# Split real data into image, class labels, and style labels
real_img = real[:, :img_dim]
real_class_labels = real[:, img_dim:img_dim + class_dim]
real_style_labels = real[:, img_dim + class_dim:]

### Train Discriminator: max log(D(x)) + log(1 - D(G(z)))
noise = torch.randn(batch_size, latent_dim).to(device)
fake = generator(noise, real_class_labels, real_style_labels)

disc_real = discriminator(real_img, real_class_labels,
    real_style_labels).view(-1)
lossD_real = criterion(disc_real, torch.ones_like(disc_real))
disc_fake = discriminator(fake.detach(), real_class_labels,
    real_style_labels).view(-1)
lossD_fake = criterion(disc_fake, torch.zeros_like(disc_fake))
lossD = (lossD_real + lossD_fake) / 2
discriminator.zero_grad()
lossD.backward()
opt_disc.step()

### Train Generator: min log(1 - D(G(z))) <-> max log(D(G(z)))
output = discriminator(fake, real_class_labels, real_style_labels
    ).view(-1)
lossG = criterion(output, torch.ones_like(output))
generator.zero_grad()
lossG.backward()
opt_gen.step()

if batch_idx == 0:
    print(
        f"Epoch [{epoch}/{epochs}] Batch {batch_idx}/{len(loader)
            } \
          Loss D: {lossD:.4f}, loss G: {lossG:.4f}"
    )

    if epoch % 100 == 0:
        with torch.no_grad():
            fake = generator(noise, real_class_labels,
                real_style_labels).reshape(-1, 1, img_x, img_y)
            data = real_img.reshape(-1, 1, img_x, img_y)
            img_grid_fake = make_grid(fake, normalize=True)
            img_grid_real = make_grid(data, normalize=True)

            plt.figure(figsize=(10,10))
            plt.subplot(1,2,1)
            plt.title("Real Images")
            plt.imshow(img_grid_real.permute(1, 2, 0).cpu().numpy
                ())
            plt.subplot(1,2,2)
            plt.title("Fake Images")
            plt.imshow(img_grid_fake.permute(1, 2, 0).cpu().numpy
                ())
```

```
# Adding hyperparameter information to the plot

plt.figtext(0.5, 0.1, f'Learning Rate: {lr}, Latent
    Dim: {latent_dim}, '
            f'Batch Size: {batch_size}, Total Epochs:
                {epochs}, '
            f'Epoch: {epoch}',
            wrap=True, horizontalalignment='center',
                fontsize=12)

plt.savefig(f'epoch_{epoch}_batch_{batch_idx}.png')
plt.show()
plt.close()
```

### 4.3.5 C: Saving the model

The models are saved after the training process is completed. This allows for future use of the trained models without needing to retrain them from scratch.

```
torch.save(generator.state_dict(), '/bigdata/users/postgrad/mshyam/PGM
    Models/PGM/generatorModel.pth')
torch.save(discriminator.state_dict(), '/bigdata/users/postgrad/mshyam/
    PGM Models/PGM/discriminatorModel.pth')
print('Generator model saved to /bigdata/users/postgrad/mshyam/PGM Models
    /PGM/generatorModel.pth')
print('Discriminator model saved to /bigdata/users/postgrad/mshyam/PGM
    Models/PGM/discriminatorModel.pth')
```

### 4.3.6 D: Generating images using CGAN

The function `show_images(images, class_label, style_label)` displays a set of images in a grid. It takes three parameters: `images`, a tensor of the images; `class_label`, the class identifier; and `style_label`, the style identifier. The images are reshaped with `images.view(-1, 1, 28, 28)`, and `make_grid` arranges them into a grid with 5 images per row, normalizing their pixel values. The line `plt.imshow(grid_img.permute(1, 2, 0).numpy())` displays the grid of images using matplotlib. The plot title is set with `plt.title`, and the plot is rendered with `plt.show()`.

The function `generatorCGAN(generator, num_samples, class_label, style_label)` generates images using a CGAN generator. It takes four parameters: `generator`, the trained CGAN model; `num_samples`, the number of images to generate; `class_label`, the class label for the images; and `style_label`, the style label. One-hot encoded vectors for class and style labels are created and moved to the appropriate device. Random noise vectors are generated with `torch.randn` and passed to the generator

along with the one-hot vectors. The generated images are detached from the computational graph and moved to the CPU. Finally, `show_images` is called to display the generated images. Example calls to `generatorCGAN` illustrate generating images with specific class and style labels.

```
# Function to show images in a grid
def show_images(images, class_label, style_label):
    # Arrange images into a grid with 5 images per row, normalizing pixel
        values
    grid_img = make_grid(images.view(-1, 1, 28, 28), nrow=5, normalize=
        True)
    # Display the grid of images using matplotlib
    plt.imshow(grid_img.permute(1, 2, 0).numpy())
    # Set the title of the plot to show class and style labels
    plt.title(f"Class: {class_label}, Style: {style_label}")
    # Render the plot
    plt.show()

# Function to generate images using the CGAN generator
def generatorCGAN(generator, num_samples, class_label, style_label):
    # Create one-hot encoded class and style labels
    class_one_hot = to_one_hot(torch.LongTensor([class_label] *
        num_samples), class_dim).to(device)
    style_one_hot = to_one_hot(torch.LongTensor([style_label] *
        num_samples), style_dim).to(device)

    # Generate random noise vectors
    noise = torch.randn(num_samples, latent_dim).to(device)

    # Generate images using the CGAN generator
    generated_images = generator(noise, class_one_hot, style_one_hot)

    # Display the generated images
    show_images(generated_images.cpu().detach(), class_label, style_label
        )

# Generate and display 10 images for class 0, style 0
generatorCGAN(generator, num_samples=10, class_label=0, style_label=0)
# Generate and display 10 images for class 0, style 1
generatorCGAN(generator, num_samples=10, class_label=0, style_label=1)

# Generate and display 10 images for class 10, style 0
generatorCGAN(generator, num_samples=10, class_label=10, style_label=0)
# Generate and display 10 images for class 10, style 1
generatorCGAN(generator, num_samples=10, class_label=10, style_label=1)
```

# 5 Model Comparison

The performance of the two models is compared by discussing their strengths and weaknesses and evaluating the quality of the generated samples using both qualitative (visual) and quantitative measures.

## 5.1 C-VAE output

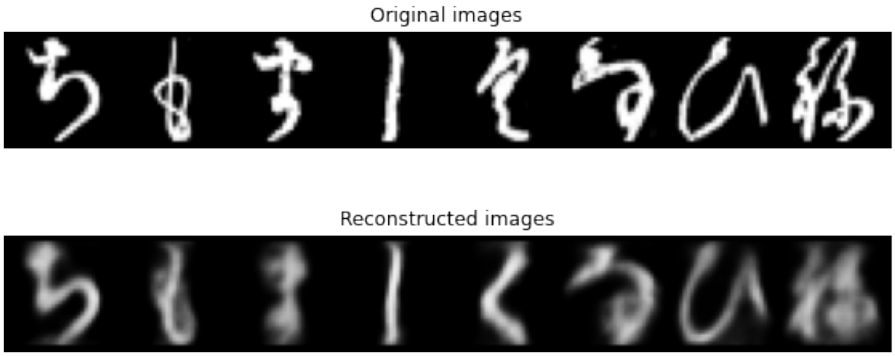



Figure 8: CVAE testing output

### 5.1.1 Testing loop output

The image displays a comparison between the original images and the reconstructed images produced by the Conditional Variational Autoencoder (CVAE) model for the Kuzushiji-49 dataset (See Figure 8). The top row illustrates the original images of Japanese characters, showcasing clear and distinct shapes with fine details. In contrast, the bottom row presents the reconstructed images generated by the CVAE model. While the reconstructed images maintain the general structure and form of the original characters, they appear significantly blurrier and lack the sharpness and detail present in the originals. This indicates that while the CVAE model captures the overall shape and pattern of the characters, it struggles to reproduce the fine details with high fidelity. Improving the model architecture or training parameters might help enhance the quality of the reconstructed images.

The main reason for blurred images can be explained by the compression of high dimensional data to low dimension in the latent dimension or bottle neck, which makes it hard to reconstruct the high dimensional image from the low dimensional representation.

### 5.1.2 Generated images

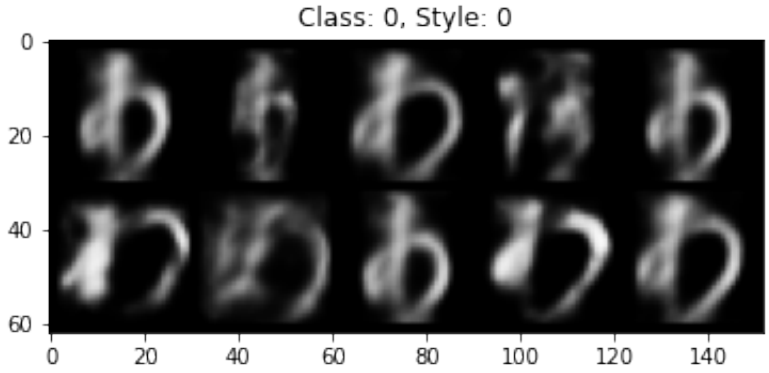



Figure 9: Generated image: Class 0, Style thin(0)

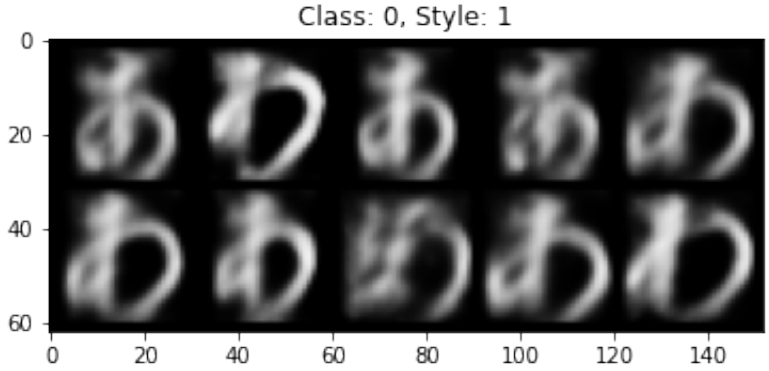



Figure 10: Generated image: Class 0, Style thick(1)

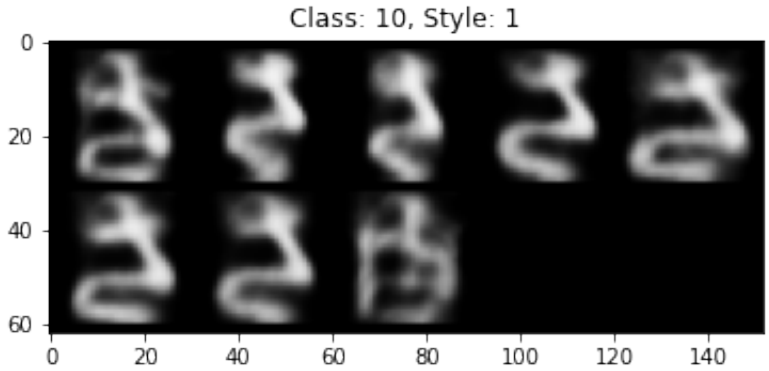



Figure 11: Generated image: Class 10, Style thick(1)

The provided images illustrate the generative capabilities of a Conditional Variational Autoencoder (CVAE) model applied to the Kuzushiji-49 dataset, focusing on different classes and styles of Japanese characters. The first set (See Figure 9) (Class: 0, Style: 0) and the second set (See Figure 10) (Class: 0, Style: 1) both demonstrate the model's ability to capture the core structure of Class 0 while exhibiting variations consistent with different styles, though the generated images are noticeably blurry and lack fine details. The third set (See Figure 11) (Class: 10, Style: 1) showcases characters from a different class, maintaining distinct shapes characteristic of Class 10 with variations in Style 1, yet still suffering from blurriness. These observations indicate that while the CVAE model effectively captures the general shapes and style variations within and across different classes, there is a consistent issue with image clarity and detail reproduction that necessitates further refinement in the model's architecture or training process to improve the fidelity of the generated images.

## 5.2 C-GAN output

### 5.2.1 Training loop output

The images generated from the training loop presents a comparison between real images from the Kuzushiji-49 dataset and fake images generated by the model at different epochs. The real images, displayed on the left, are clear and detailed depictions of Japanese characters, each with distinct and recognizable features. These images serve as the ground truth for the model to learn from. On the right, the fake images, generated by the model at this early stage of training, appear as undifferentiated noise, lacking any discernible patterns or recognizable structures as observed in epoch 0 (See Figure 12). This stark contrast underscores the model's initial incapacity to generate meaningful representations of the characters from the latent space. At epoch 0, the model's performance in synthesizing realistic images is notably poor, highlighting the necessity for continued training. As training progresses over successive epochs, the model improves its generative capabilities, producing images that increasingly resemble the real characters.

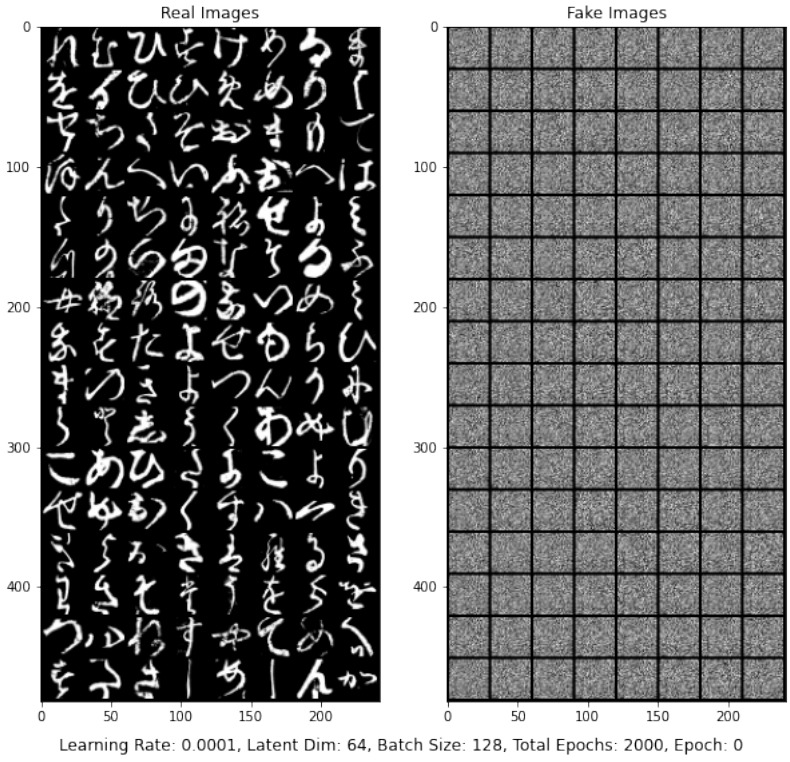



Figure 12: CGAN epoch 0

The 100th epoch shows recognizable characters similar to the real images, even though it lacks clarity and detail (See Figure 13). Where as huge improvements can be noticed at the 1900th epoch (See Figure 14). The generated class character start to look like the real images. There is certain improvement in the model after training for a longer time.

Key parameters such as learning rate, latent dimension, batch size, and the total number of epochs will significantly influence the model's learning process and its ultimate ability to generate high-quality, realistic images.

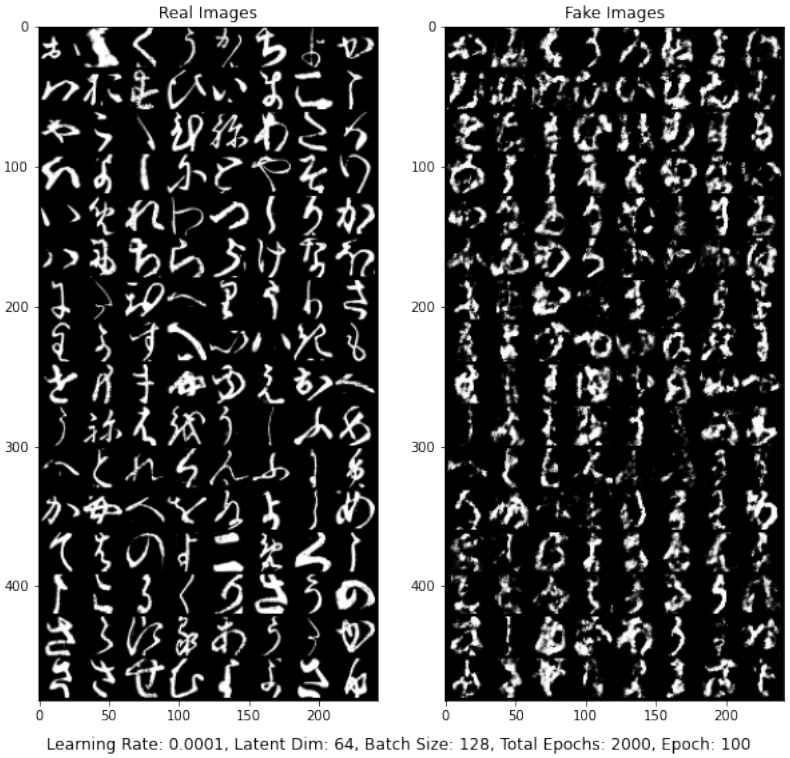



Figure 13: CGAN epoch 100

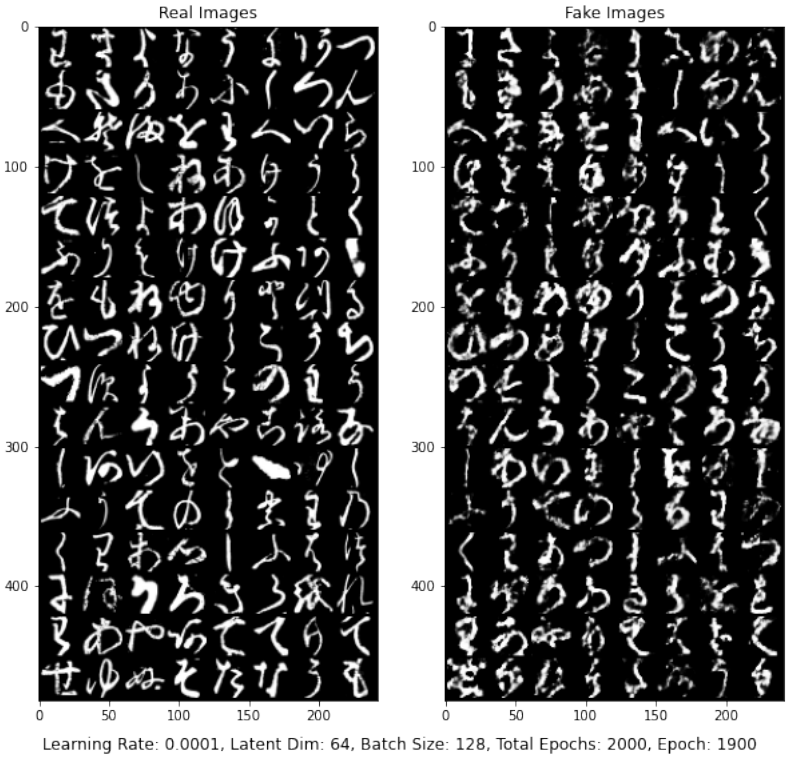



Figure 14: CGAN epoch 1900

### 5.2.2 Generated images

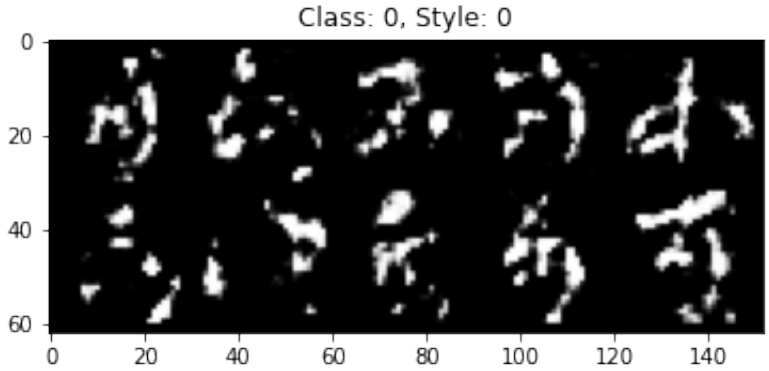



Figure 15: Generated image Class 0, Style 0 (thin)

The images show the generative performance of the Conditional Generative Adversarial Network (CGAN) model for different classes and styles of Japanese characters. The first image (See Figure 15)(Class: 0, Style: 0) and the second image (See Figure 16) (Class: 0, Style: 1) display characters from the same class but with different styles. Both sets are blurry and lack clear structure, indicating that the model fails to generate quality images and has not yet effectively captured the fine details and distinct features of the training dataset characters. Similarly, the third image (See Figure 17) (Class: 10, Style: 1) and the fourth image (See Figure 18)(Class: 10, Style: 0) show characters from a different class, again with different styles, but also suffer from the same issues of blurriness and lack of definition.

The generated images across all classes and styles appear as fragmented and indistinct, suggesting that the CGAN model needs further training and refinement to improve its ability to generate clear and accurate representations of the Japanese characters, even though the training output was able to generate high resembling fake images compared to the real images.

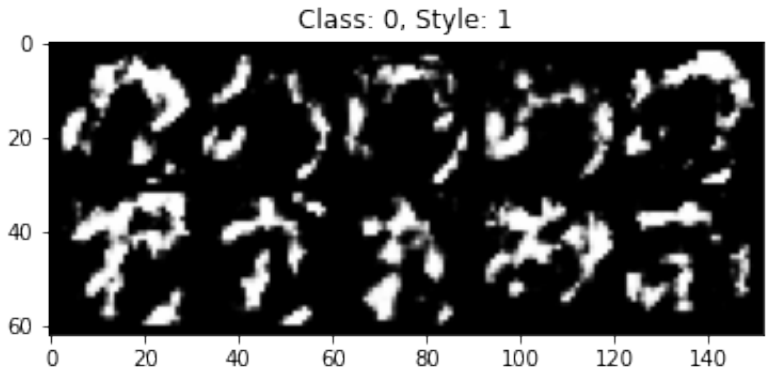



Figure 16: Generated image Class 0, Style 1 (thick)

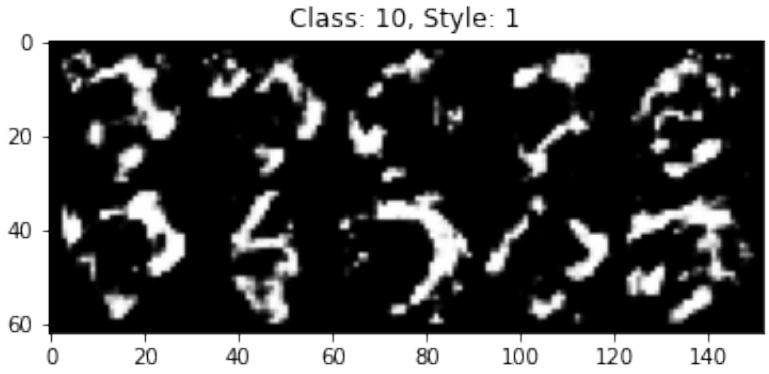



Figure 17: Generated image Class 10, Style 1 (thick)

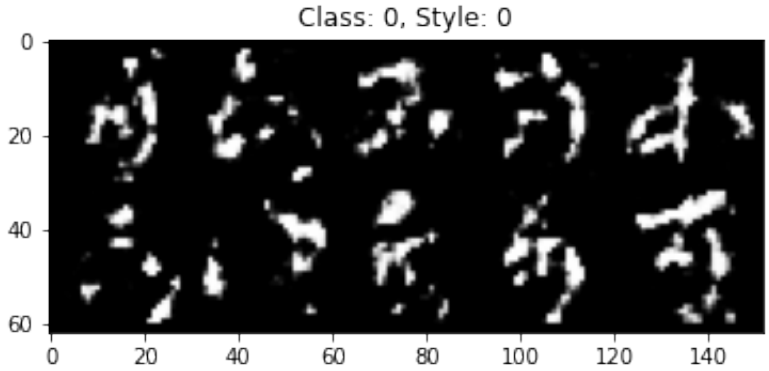



Figure 18: Generated image Class 10, Style 0 (thin)

### 5.3 Comparison

- The CVAE's output from the training loop after 2000 epochs was not high quality output, and missed the finer details even though the characters had high resemblance to the real images(See Figure 8). The quality might improve with increased latent dimension or by altering other parameters.
- The CGAN's output from training loop significantly improved over the epochs. In about 1900 epochs, the CGAN was able to generate fake images, very similar to the real images with above average image quality (See Figure 14)
- The images generated by CVAE (See Figure 9) has a higher quality and resemblance to real images compared to the CGAN (See Figure 15).
- CGAN generated appears less blurry compared to the CVAE generated images (See Figures 10 and 16).

### 5.3.1 CVAE & CGAN loss plots

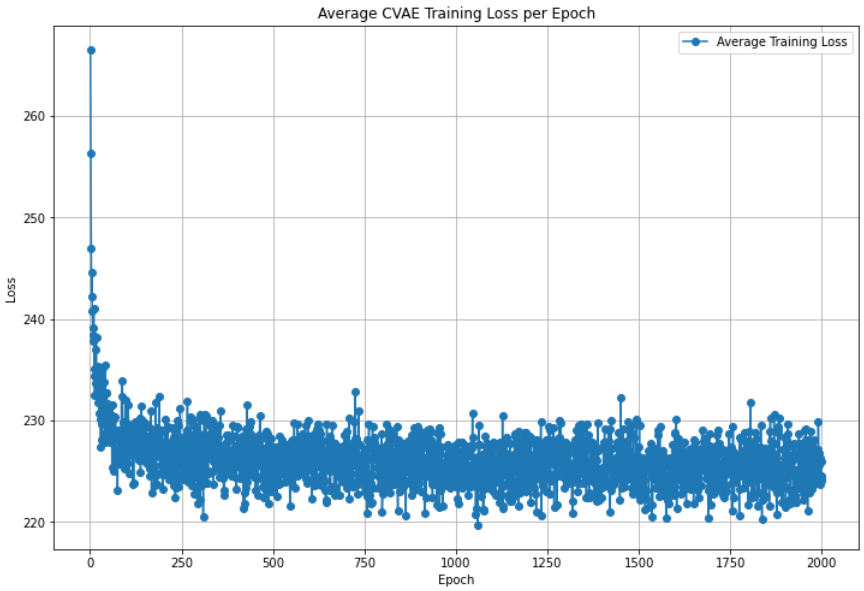



Figure 19: CVAE loss

The CVAE loss plot illustrates the average training loss of the Conditional Variational Autoencoder (CVAE) model over the course of 2000 epochs. Initially, the training loss is significantly high, peaking around 270 at the very first epoch. As training progresses, the loss decreases during the initial epochs, suggesting that the model is learning effectively in the early stages. However, after this initial reduction, the training loss stabilizes and fluctuates around the 220-230 range for the remaining epochs. Despite these fluctuations, the loss does not drop below 220, indicating a potential limitation in the model's capacity to minimize the training loss further. This persistent level of loss suggests that the model may require further tuning or modifications to achieve a lower training loss, which is critical for enhancing its performance (see Figure 19).

The CGAN loss plot illustrates the average training loss of the Conditional Generative Adversarial Network (CGAN) model over 2000 epochs, with separate lines for the discriminator and generator losses. Initially, both losses start high, with the generator loss peaking above 3.5 and the discriminator loss around 0.5. During the early epochs, both losses decrease, indicating that the model quickly adapts to the training data. As training progresses, the discriminator loss stabilizes around 0.4 to 0.5, while the generator loss fluctuates between 1.0 and 2.0, reflecting the ongoing adversarial interplay between the two components. Despite these fluctuations, the discriminator maintains consistent performance, while the generator shows more variability, suggesting that further tuning may be required to achieve lower and more stable losses for optimal model performance (See Figure 20).

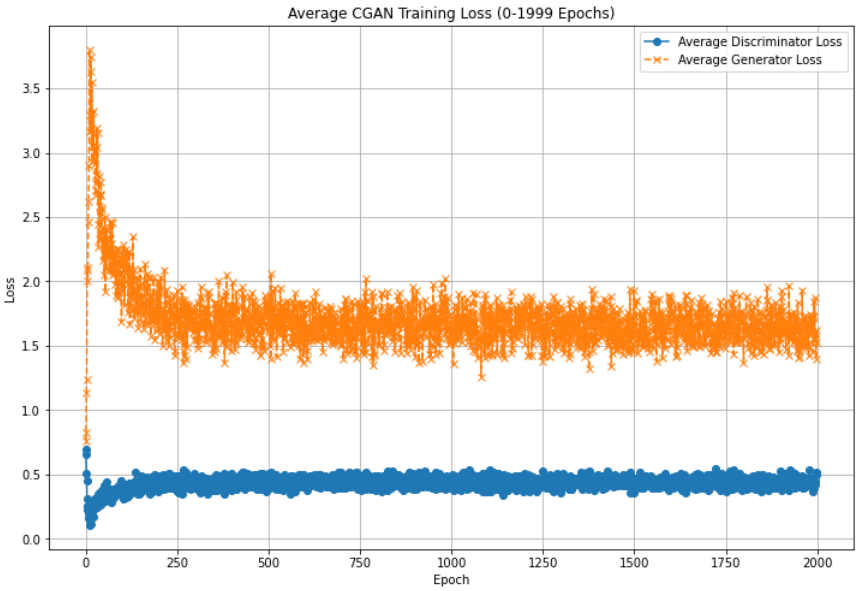


Figure 20: CGAN Loss

An optimal CGAN training loss graph would show both the discriminator and generator losses decreasing rapidly in the initial epochs, converging to low values, and stabilizing with minimal oscillations, indicating effective and balanced learning.

# 6 Experimentations

Different experiments with additional stylistic labels, different architectures, training strategies, and techniques with an objective improving the quality and diversity of the generated images are conducted. Findings and explanations for the observed results are documented.

All experimental architectures were run for 100 epochs, time for training the model was recorded. Also the loss plots, as well as generated images are documented.

### 6.0.1 CVAE Architectures

Three different architectures were employed for CVAE. Architecture one uses mean instead of median as classifier. The order of input dimension, class dimensions, and style dimension were changed in architecture 2. The third architecture uses additional layers in the neural network.

### 6.0.2 Architecture 1:

The first experimentation was to classify the style labels using mean of the foreground pixel values instead of median to see if there is any specific improvement in the model.

The training time is 688.14 seconds, with an average training loss of 228 and average validation loss of 230. The MSE is 0.06, meaning there is not much difference between the distribution of reconstructed images to the original images.

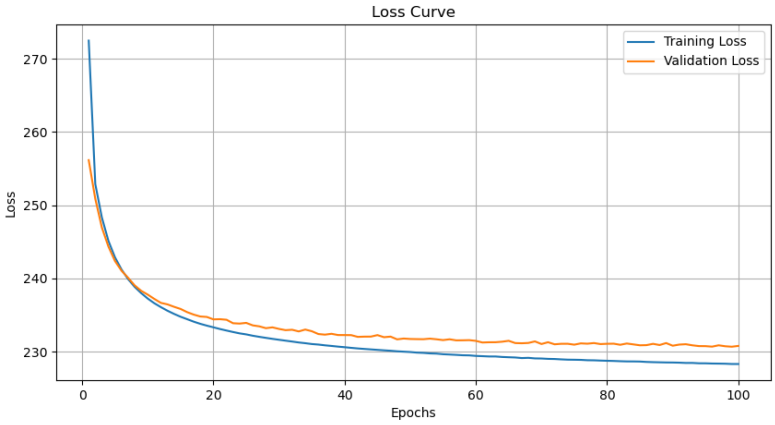



Figure 21: Loss curve for CVAE Architecture 1

The loss curve demonstrates that both the training and validation loss decrease steadily over the epochs, indicating effective learning and convergence of the model. The training loss decreases consistently, while the validation loss follows a similar trend, suggesting that the model is not overfitting and generalizes well to unseen data. The gradual plateauing of both losses towards the end implies that the model has reached a stable state with minimal further improvements in performance (See Figure 21).

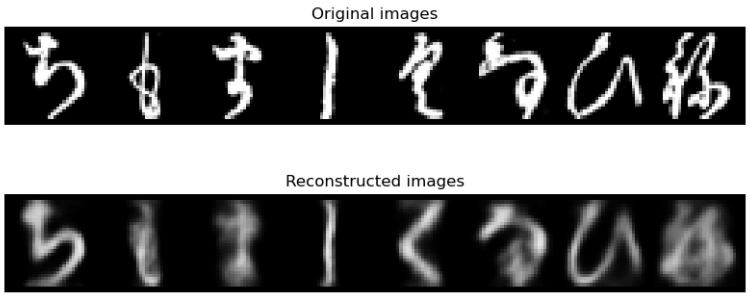



Figure 22: CVAE Architecture 1: Reconstructed Images

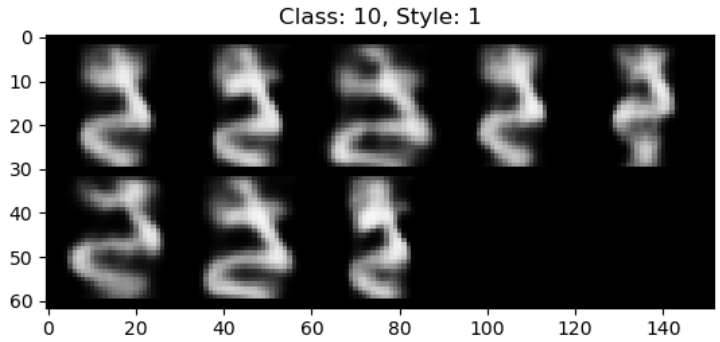



Figure 23: CVAE Architecture 1: Generated Images

There is no significant improvement in the reconstructed images even though the losses are more stable (See Figure 22). Even the generated images are blurry but the characters are identifiable (See Figure 23).

#### 6.0.3 Architecture 2:

The CVAE architecture 2 modifies the arrangement format of the final dataset used for training the model. Instead of concatenating one hot encoded class labels and style labels to the image pixel values, the image pixel of 784 is concatenated to one hot encoded class labels. The style labels are then concatenated.

This architecture resulted in a training time of 681 seconds which is same as architecture 1 training time.

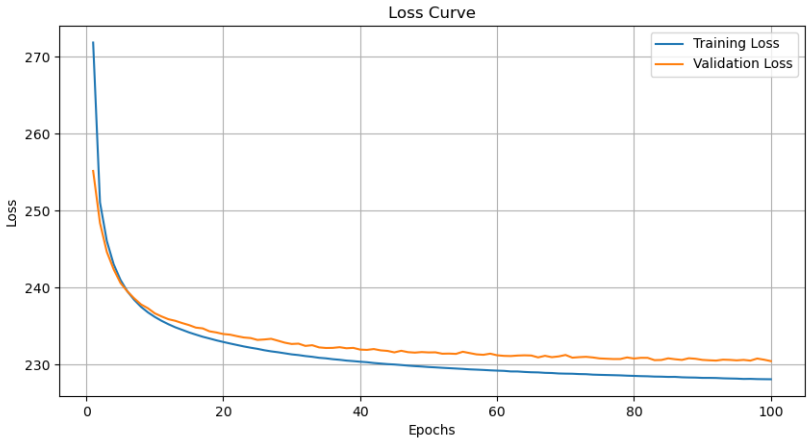



Figure 24: CVAE Architecture 2: Loss curve

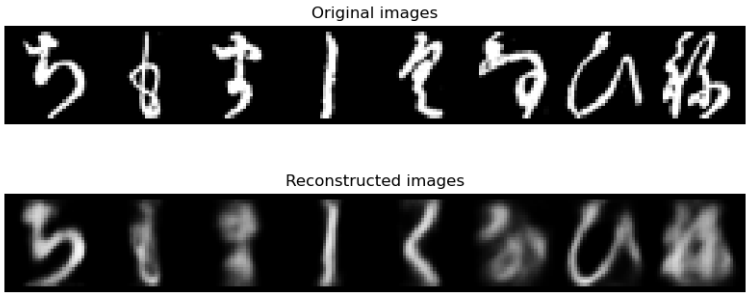



Figure 25: CVAE Architecture 2: Reconstructed images

This architecture hasn't provided much improvement compared to the previous architecture. (See Figure 26 & 25)

### 6.0.4 Architecture 3:

The third architecture for CVAE includes additional neural network layers. The additional layers in Architecture 3's encoder and decoder enhance the model's capacity and stability. The encoder includes two extra linear layers that reduce the dimensionality to 256 and then to 128, each followed by batch normalization and ReLU activation functions. This structure improves the network's ability to learn robust features by normalizing the input distributions and introducing non-linearities. Similarly, the decoder mirrors this structure with linear layers that increase dimensionality back to 256 and then to the hidden dimension, also incorporating batch normalization and ReLU activations to ensure stable and effective reconstruction of the input images.

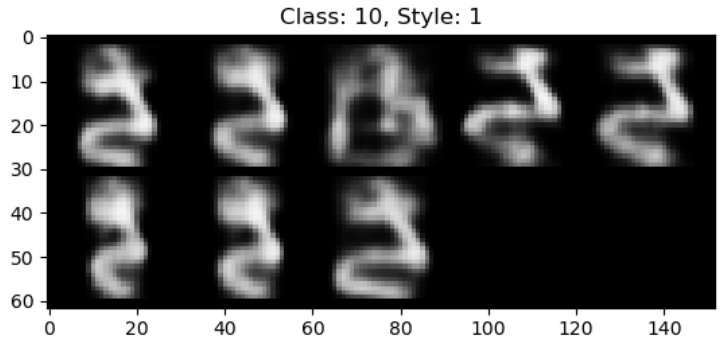



Figure 26: CVAE Architecture 2: Generated images

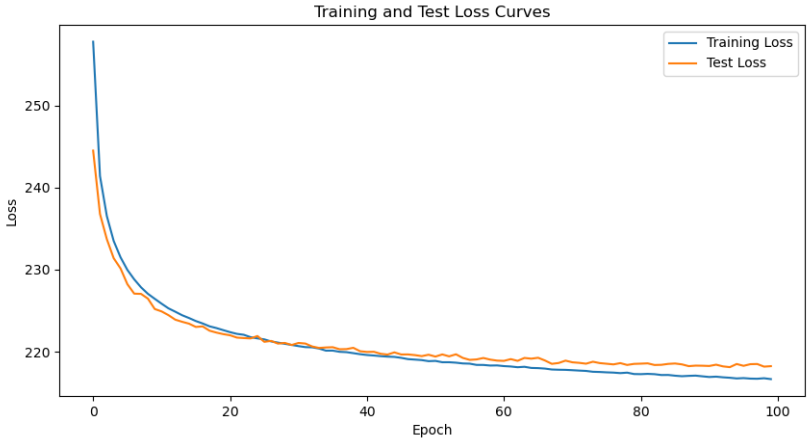



Figure 27: CVAE Architecture 3: Generated images

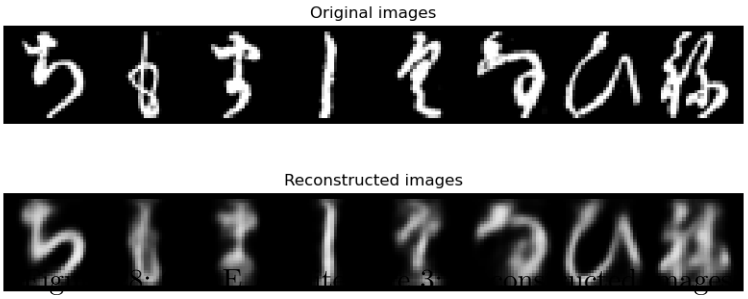



Figure 28: CVAE Architecture 3: Reconstructed images

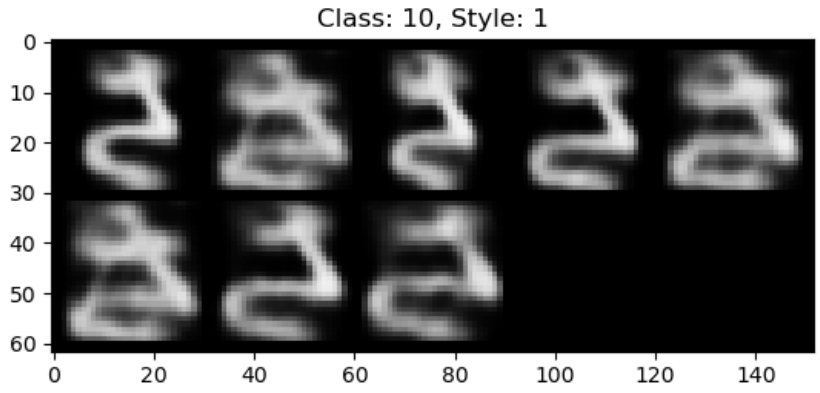



Figure 29: CVAE Architecture 3: Loss curves

Architecture 3 also gives similar output like the previous two architectures, comparing the image outputs (See Figure 29, 27 & 27) even though it had a higher training time of 1303 seconds - the highest training time in the CVAE architectures.

## 6.1 CGAN Architectures

Two different architectures were employed for CGAN. The first architecture exploits the potential of dataset concatenation order, similar to CVAE architecture 2. The image pixels are concatenated between the onehot encoded class and style labels.

### 6.1.1 Architecture 1:

The graph illustrates the training loss trajectories for both the generator and discriminator of a GAN over 100 epochs (See Figure 30). Initially, the generator loss starts low but rapidly increases, peaking around epoch 20, and then fluctuates while generally decreasing, indicating the generator's improving ability to produce realistic data as training progresses. Conversely, the discriminator loss remains relatively stable and low throughout the training process, suggesting that the discriminator consistently performs well in distinguishing between real and generated data. This interplay highlights the typical adversarial learning process, where the generator improves its output quality over time in response to the discriminator's feedback.

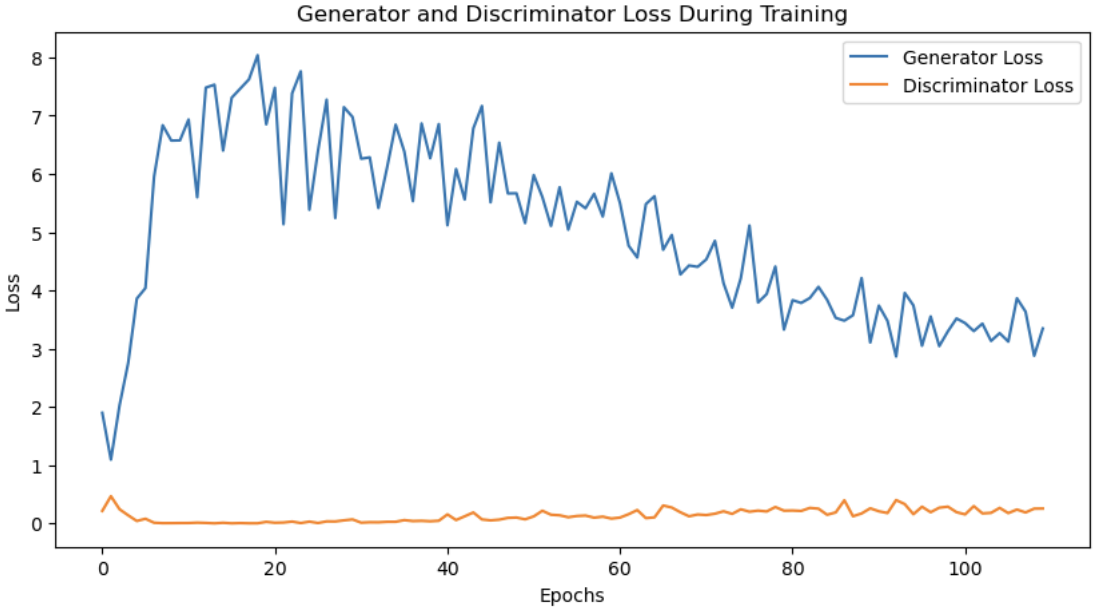
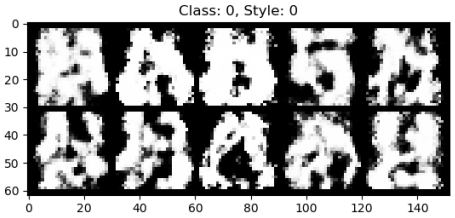
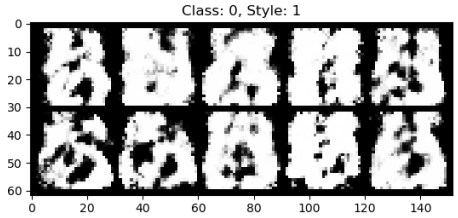



Figure 30: CGAN Architecture 1: Loss curve

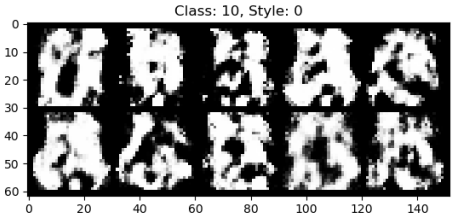
The change in architecture of the CGAN model has improvement in terms of image generation because the images appear to have more pixels and complete, even though they are missing fine detailing (See Figure 31a, 31b, 32a & 32b). . The original CGAN model outputs were incomplete compared to CGAN Architecture 1 model. The images appear better in the training (See Figure 33b, 33a). The model needs more training epochs before finalising the generation quality of this architecture.



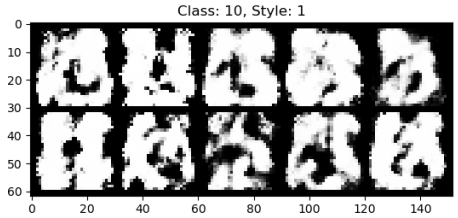

(a) CGAN Architecture 1: Class 0, Style 0



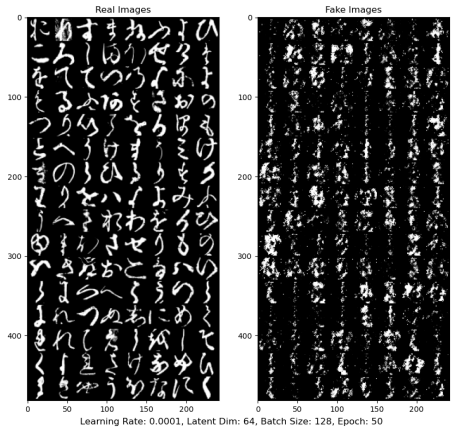

(b) CGAN Architecture 1: Class 0, Style 1



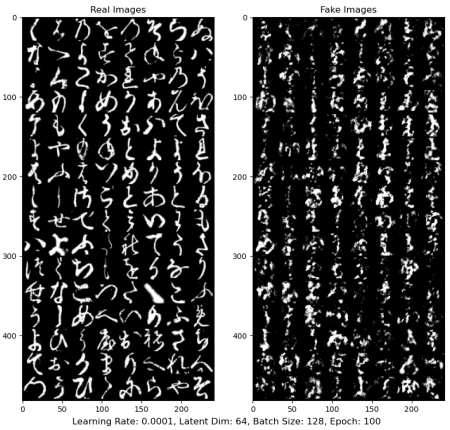

(a) CGAN Architecture 1: Class 10, Style 0




(b) CGAN Architecture 1: Class 10, Style 1




(a) CGAN Architecture 1: Epoch 50




(b) CGAN Architecture 1: Epoch 100

### 6.1.2 Architecture 2:

The second architecture for CGANs use Sigmoid actiavation function on the generator instead of Tanh activation function. The dataset is not renormalized between -1 and 1. The dataset is normalised between 0 and 1, with one hot encoded class labels and style labels.

This model performs better than the original CGAN model trained for 2000 epochs. The generated images are whole and resembles the real character even better at 100 epochs compared to the original model. This model has potential for room for improvement.


Figure 34: Generated images at epoch 50, batch 0.

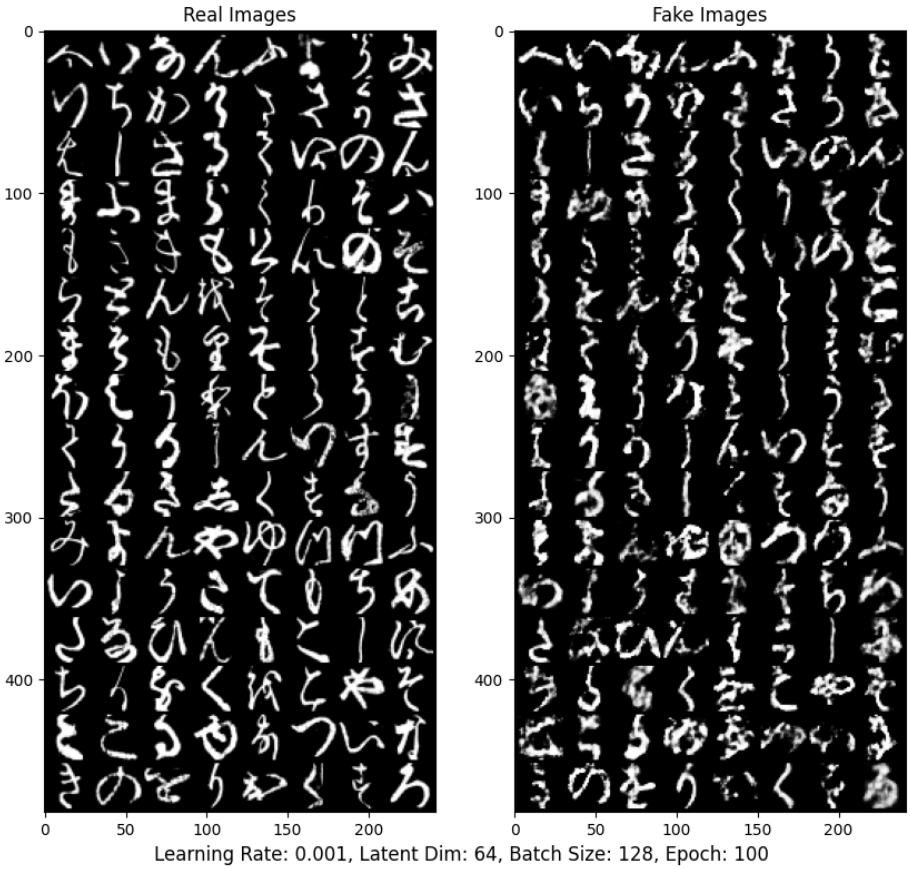



Figure 35: Generated images at epoch 100, batch 0.

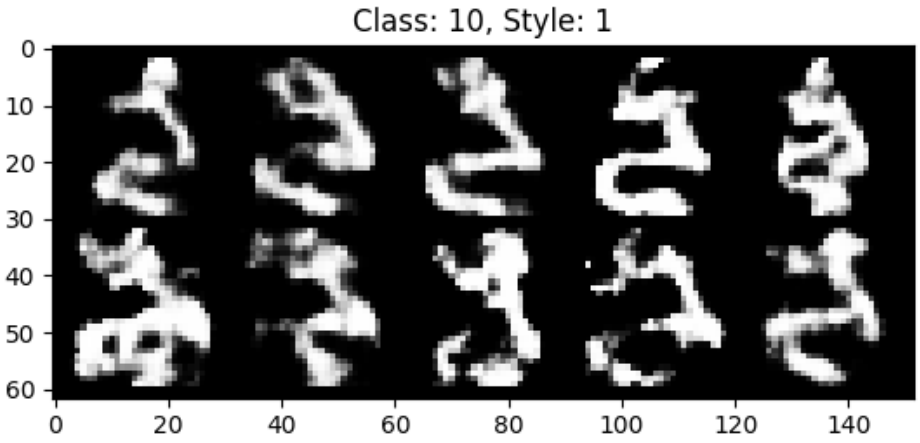



Figure 36: Comparison of generated images from different models.

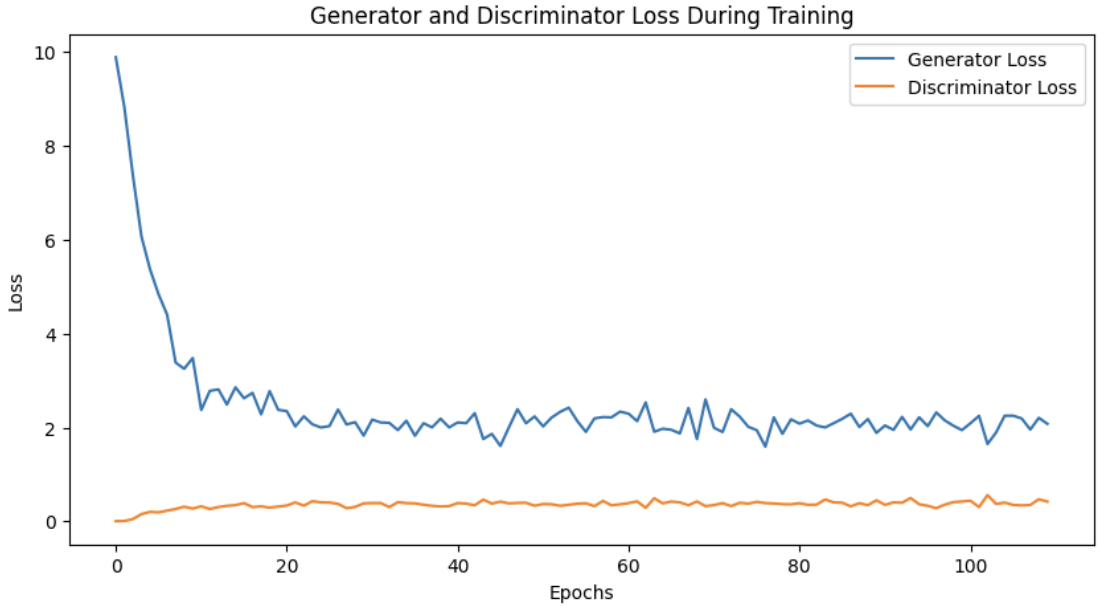



Figure 37: CGAN Architecture 2: Loss Curve

The CGAN Architecture 2 graph of loss curves depicts the training dynamics over 120 epochs. Initially, the generator loss is high, starting around 10, and decreases sharply in the early epochs, stabilizing below 2 after about epoch 40 (See Figure 37). This trend indicates that the generator quickly improves its ability to produce realistic outputs. Conversely, the discriminator loss remains very low and stable throughout the training, consistently hovering around zero, suggesting that it effectively distinguishes between real and generated data from the beginning. The stabilization of both losses over time indicates that the CGAN has reached a state of equilibrium, where the generator and discriminator are learning effectively from each other, leading to improved performance in generating realistic data conditioned on specific inputs.

# 7 Conclusion

In this project, we explored the conditional generation of images using two prominent generative models: Conditional Variational Autoencoders (CVAEs) and Conditional Generative Adversarial Networks (CGANs). Both models were applied to the Kuzushiji-49 dataset, which consists of handwritten Japanese characters, aiming to generate new images based on specific class and style labels.

## 7.1 Key Findings

- **CVAE Performance:**
  - Capable of generating images that capture the general structure and form of characters.
  - Consistently produced blurry images lacking fine detail.
  - Limitation due to compression of high-dimensional data to lower dimensions in the latent space.
- **CGAN Performance:**
  - Significant improvements in generating realistic images over the epochs.
  - Second architecture with sigmoid activation and data normalization between 0 and 1 produced the most realistic images.
  - Generator and discriminator exhibited typical adversarial learning dynamics.
  - Less blurry images compared to CVAEs.

- **Model Comparison:**
  - CVAE struggled with image clarity and detail.
  - CGAN generated images with higher resemblance to real data.
  - CGAN had more consistent performance in generating high-quality images.
- **Experimentation Insights:**
  - Importance of the arrangement of input data and additional layers to enhance model capacity and stability.
  - Various stylistic labels and training strategies provided insights for improving image quality and diversity.
  - Room for improvement in achieving sharper and more detailed outputs.

## 7.2 Challenges

- **Computation Time:** Training models, especially CGANs, required significant computational resources and time.
- **Image Clarity:** Both models faced challenges in producing sharp and detailed images.
- **Model Stability:** CGANs exhibited fluctuations in generator loss, indicating the need for further tuning.

## 7.3 Future Work

- Further refine latent space representations in CVAEs to improve image reconstruction quality.
- Explore advanced training techniques and loss functions, such as Wasserstein GANs, to address issues like mode collapse and training instability in CGANs.
- Leverage larger and more diverse datasets to enhance the generalization capabilities of both models.

In conclusion, this project successfully demonstrated the potential of conditional generative models in producing images of handwritten Japanese characters. While challenges remain, particularly with image clarity and detail, the advancements made provide

a solid foundation for future explorations and improvements in the field of generative modeling.

# References


Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875*.

Bagheri, M. (2019). A tutorial on conditional generative adversarial nets + keras implementation. Medium.

Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., and Ha, D. (2018). Deep learning for classical japanese literature.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27:2672–2680.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM*, 63(11):139–144.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134.

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT press.

Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., et al. (2017). Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690.

Mirza, M. and Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

Sofeikov, F. (2019). Implementing conditional variational auto-encoders (cvae) from scratch. Medium.

Sofeikov, K. (2023). Implementing conditional variational auto-encoders (cvae) from scratch. *Medium*.