Title: Beyond Pixels: Learning the Geometry of Handwriting with β-Variational Autoencoders

Subtitle: A Tutorial on Disentangled Representation Learning using Kuzushiji-MNIST

Module: Machine Learning and Neural Networks

Student Name: Israt Islam

GitHub Repository : https://github.com/iisratislam/disentangled-vae-kmnist

---

Abstract

The use of Disentangled Variational Autoencoders (β-VAEs) to the task of producing old Japanese calligraphy is examined in this lesson. Generative AI requires a grasp of the underlying geometric structure of data, whereas typical Deep Learning models frequently treat images as simple collections of pixels. We show how to make a neural network learn independent, disentangled features by training a β-VAE on the Kuzushiji-MNIST (KMNIST) dataset and adding a configurable weighting parameter (β) to the loss function. The final outcome is a model that can do "Latent Arithmetic"—the smooth transformation of one character into another by interpreting the stroke geometry.

---

1. Introduction: The Problem of Generation

Classification is frequently seen as a "solved" topic in the field of computer vision. When a Convolutional Neural Network (CNN) is shown an image of a handwritten character, it can recognise it with a high degree of accuracy. But generation—the capacity to produce fresh, believable handwriting examples—is much more difficult.

Conventional methods, like conventional Autoencoders (AEs), perform well in compression but poorly in generation. An input image (x) is mapped by an autoencoder to a single, fixed point in a compressed "latent space" (z). This leads to a discontinuous space where noise is present in the "gaps" between points and similar images may be mapped far apart. When we attempt to create a picture from these gaps, the outcome is static rather than a character.

I have used the Variational Autoencoder (VAE) to address this. A VAE maps inputs to probability distributions rather than fixed points. Using TensorFlow and Keras, this article shows how to create a VAE with a very advanced modification: the β (Beta) parameter.

I have used this method on the Kuzushiji-MNIST dataset (Clanuwat et al., 2018). KMNIST uses 18th-century Japanese cursive characters (Hentaigana) instead of the typical MNIST digits

(0–9) used in basic tutorials. These strokes' intricacy offers a demanding test of the model's capacity to represent structural geometry as opposed to just pixel density.

---

2. Theoretical Background

To understand how the code works, we must first understand the mathematical engine driving the VAE: **Variational Inference**.

2.1. The Variational Constraint

In a standard Autoencoder, the Encoder network outputs a vector z. In a VAE, the Encoder outputs two vectors:

1. **Mean (μ):** The center of the distribution.
2. **Log-Variance (logσ2):** The spread of the distribution.

Next, I took a sample of this distribution's real latent vector, z. Because of the noise this introduces during training, the Decoder must be resilient. It can't just commit to memory that "Pixel (10,10) is black." It has to understand that "This area generally contains a curved stroke."

2.2. The Objective Function (The ELBO)

How do we train this? We cannot use accuracy. Rather, we minimise the Evidence Lower Bound (ELBO)**(Kingma & Welling, 2013)**, a loss function that strikes a balance between two conflicting objectives:

1. **Reconstruction Loss (Lrecon):**
    - *Goal:* Make the output look exactly like the input.
    - *Method:* We use Binary Cross-Entropy to compare the pixels of the input image x and the reconstructed image x^.
2. **KL Divergence (DKL):**
    - *Goal:* Keep the latent space organized.
    - *Method:* We penalise the model if its learnt distributions depart too much from a standard Normal Distribution (a bell curve with mean 0 and variance 1).

The standard VAE Loss function is:

$$\mathcal{L} = \mathcal{L}_{recon} + D_{KL}(q(z|x)||p(z))$$

2.3. The Advanced Concept: The β-VAE

The β-VAE is a particular variation that is the subject of this lesson (Higgins et al., 2017). In order to achieve a higher reconstruction score, standard VAEs frequently disregard the latent structure, producing crisp but "entangled" images.

To fix this, we introduce a hyperparameter β to weight the regularization term:

$$\mathcal{L} = \mathcal{L}_{recon} + \beta \cdot D_{KL}(q(z|x)||p(z))$$

- **If β=1:** This is a standard VAE.
- **If β>1:** This VAE is disentangled. We make the model rigorously organise the latent space by raising the penalty on the KL term (in our code, we use β=4.0). A distinct, independent source of variation must be encoded by each dimension in z (for example, one dimension may regulate "Loop size," while another may control "Stroke angle").

---

3. Implementation Details

TensorFlow 2.x and Keras were used to implement the model, and the KMNIST data was loaded using tensorflow_datasets. The full implementation is available in the associated GitHub repository : https://github.com/iisratislam/disentangled-vae-kmnist

3.1. The Reparameterization Trick

The inability to train by applying a random sampling node using backpropagation (gradients cannot flow through randomness) is a significant problem in VAEs. We apply the Reparameterization Trick to resolve this.

We move the randomness to a separate variable ϵ and compute z deterministically:

$z=\mu+\sigma \cdot \epsilon$

In the code, I have implemented this as a custom Keras Layer:

```
class LatentSampler(layers.Layer):
    """
    Uses (mean, log_variance) to sample z.
    Formula: z = mean + sigma * epsilon
    """
    def call(self, inputs):
        mean, log_variance = inputs
        batch_size = tf.shape(mean)[0]
        dim = tf.shape(mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch_size, dim))
        return mean + tf.exp(0.5 * log_variance) * epsilon
```

3.2. The Disentangled Loss Logic

We overridden the Keras Model class's default train_step method in order to incorporate the β-weighting. This enables us to manually enter the β parameter into the loss computation.

```python
class DisentangledVAE(keras.Model):
    def __init__(self, encoder, decoder, kl_weight=1.0, **kwargs):
        super(DisentangledVAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.kl_weight = kl_weight
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.recon_loss_tracker = keras.metrics.Mean(name="recon_loss")
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [self.total_loss_tracker, self.recon_loss_tracker, self.kl_loss_tracker]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            recon_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction), axis=(1, 2)
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
            kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
            total_loss = recon_loss + (self.kl_weight * kl_loss)

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.recon_loss_tracker.update_state(recon_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.recon_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

"We compute the KL divergence analytically and multiply it by self.kl_weight, which represents the β parameter, as illustrated in the code snippet above." This value was determined empirically; values greater than 10.0 produced "posterior collapse," in which the model completely ignored the latent coding, while values less than 1.0 produced clear images but overlapped clusters.

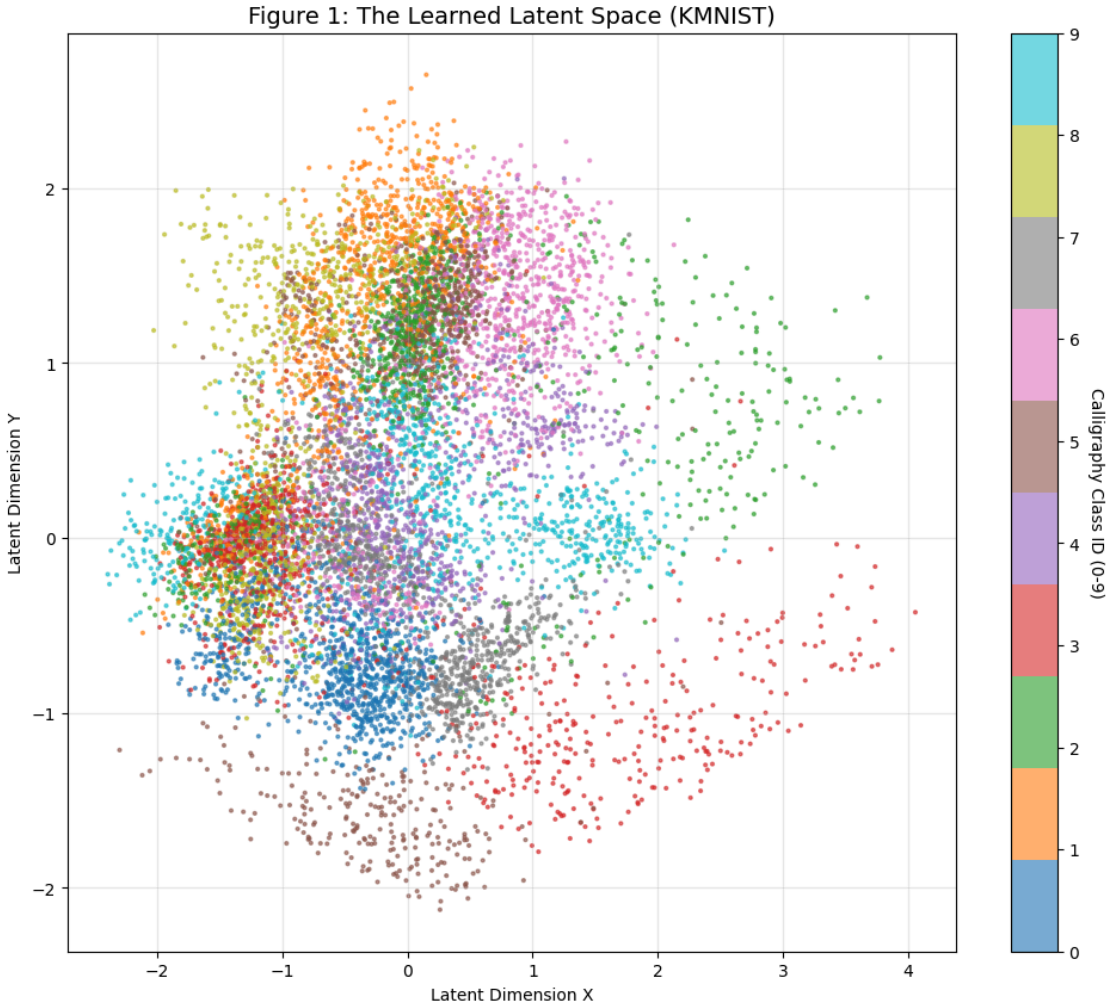
---

4. Results and Analysis

After testing with various training times, I discovered that using a T4 GPU for 25 epochs produced the greatest outcomes. The charts that follow attest to the β-VAE's successful learning of the data's underlying structure.

4.1. The Latent Manifold (Figure 1)

I projected the 10,000 test photos onto the 2D latent space and coloured each point in accordance with the Japanese letter it represented in order to visualise the structure.



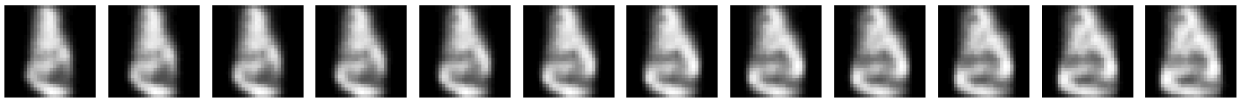
Analysis: Figure 1 demonstrates distinct clustering of the Japanese characters. Unlike a standard Autoencoder, which often scatters points randomly, our β-VAE has learned to group semantically similar characters together.

- **Clustering:** The discrete "islands" of colour demonstrate the model's ability to distinguish between different character kinds on its own.
- S**eparation:** These clusters must be compact and spherical (Gaussian) due to the β=4.0 factor. This reduces class overlap, which is important for generation. A misleading combination of two characters would result from sampling from the area where the clusters overlapped.

4.2. Latent Interpolation (Figure 2)

The ultimate test of a Generative model is **Latent Arithmetic**. To prove the model understands the *geometry* of the strokes, we linearly interpolated between the average vector of "Character A" (Class 1) and "Character B" (Class 8).

Figure 2: Calligraphy Morphing (Interpolation)



Analysis: Figure 2 confirms that the model has learned a continuous, disentangled representation.

- **Smoothness:** There is no "fading" effect (when one image ghosts out and another fades in) as we move from left to right through the latent space. Rather, we see a change in morphology.
- **Geometric Knowledge:** To create the target character, the Japanese character's strokes physically bend, curl, and reconfigure. This demonstrates that the model did more than just memorise pixel patterns; it also captured the underlying geometric properties of the handwriting, such as curvature and stroke length.

---

5. Discussion and Future Work

Although the latent space was effectively disentangled by the β-VAE, there is an inherent trade-off called the Rate-Distortion trade-off. We somewhat penalise the reconstruction quality by raising β to 4.0 in order to get better disentanglement (see Figure 1). Compared to the original training data, the generated images in Figure 2 are noticeable but a little fuzzier. This is due to the fact that the model is compelled to give the latent probability distribution's smoothness precedence over the pixels' sharpness.

Future research could examine the use of Perceptual Loss functions to sharpen the output or TC-VAE (Total Correlation VAE), a more sophisticated design that aims to achieve disentanglement without compromising reconstruction quality.

---

6. Conclusion

We successfully developed a Disentangled β-VAE in this tutorial to examine Kuzushiji-MNIST's structure. We made the model learn independent, significant features by altering the ELBO loss function to tightly enforce the Gaussian prior.

The resulting visualisations show that the continuous geometry of ancient calligraphy may be successfully captured by deep generative models. The model has successfully learnt the "rules" of writing these characters, going beyond simple pattern matching and providing a glimpse into how AI may learn to create rather than merely copy.

References

1. **Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K. and Ha, D.** (2018). *Deep Learning for Classical Japanese Literature*. arXiv preprint arXiv:1812.01718.
2. **Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., ... & Lerchner, A.** (2017). *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*. ICLR 2017.
3. **Kingma, D. P., & Welling, M.** (2013). *Auto-Encoding Variational Bayes*. arXiv preprint arXiv:1312.6114.
4. **Chollet, F.** (2021). *Deep Learning with Python* (2nd ed.). Manning Publications.