Student Name: Mervan Kaya
Email: mckaya@kth.se

DD2434 Machine Learning Advanced

Project 1.3 - Variational Autoencoders

Mervan Kaya

mckaya@kth.se

Abstract

Variational autoencoders (VAEs) are a popular and powerful approach for learning compact, latent representations of high-dimensional data. In this report, the performance of VAEs on the MNIST dataset of handwritten digits are evaluated. Results are compared to those obtained using a traditional sequential model and discuss the strengths and weaknesses of each approach. The experiments show that VAEs are able to achieve good reconstruction accuracy while also allowing for sampling and interpolation in the latent space, capabilities that are not available in the sequential model. However, the sequential model outperforms the VAE in terms of training speed and sample generation quality. Overall, the results suggest that VAEs and sequential models can complement each other and offer different trade-offs depending on the task at hand.

Introduction

Variational Bayesian (VB) methods allows for approximate inference and learning for directed probabilistic models with intractable posterior distributions for continuous latent variables. VB methods commonly use the mean-field approximation to analytically derive the expectations of the approximate posteriors. However, in the general case, this analytical derivation of the approximate posterior distribution will also be intractable. To combat this problem, the model proposed in the paper by Kingma and Welling [3] utilizes a reparametrization of the lower bound, which allows for the usage of gradient descent methods to infer the approximate posterior distribution.

Implementation of the Variational Auto-Encoding model

We assume that we have a dataset $\mathbf{X}$ with $\mathbf{N}$ i.i.d samples of a variable x. The data is generated from an unobserved process where a value $\mathbf{z}$ is generated from a prior distribution p_{θ^*} and that the data x_i is generated from the conditional distribution $p_{\theta^*}(\mathbf{x}|\mathbf{z})$.

The model revolves around introducing the *recognition* model, $q_\phi(\mathbf{z}|\mathbf{x})$ which is an approximation of the true posterior $p_\theta(\mathbf{z}|\mathbf{x})$. The recognition model parameters are learned jointly with the generative model parameters.

The latent variable $\mathbf{z}$ in coding terms is called a *code*. Thus, the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ is called a probabilistic *encoder*, as it allows us to generate a distribution over the latent variables $\mathbf{z}$ given an observation $\mathbf{x}$. Conversely, the probability distribution $p_\theta(\mathbf{x}|\mathbf{z})$ is called a probabilistic *decoder*, as it produces values of $\mathbf{x}$ given a latent variable $\mathbf{z}$.

The Encoder

The encoder represents the latent space from which we sample the model parameters from that we later on use to probabilistically generate data samples. The encoder is implemented using a convolutional 2D network using TensorFlow Keras. An intuitive image of how the Auto-Encoder model is structured can be seen in 1.

We begin by first creating an Input object which will handle our 28x28 sized images of handwritten digits. We then continue by creating our convolutional layers, whose size we can determine ourselves. Different sizes may yield in better or worse performance of the model. We then flatten the model and add a fully connected layer to tie the encoder up.

Sampling in the Latent Space

As seen in 1 the model consists of actively sampling the latent space $\mathbf{z}$. We can create a differentiable transformation by utilizing the parameterization trick presented in Kingma and Welling [3]. We select approach number two, which defines the transformation as:

$$g(\cdot) = location + scale \times \epsilon \tag{1}$$

where $\epsilon \sim \mathcal{N}(0, 1)$. The location and scale of a Gaussian family is the mean and variance, respectively. In Python, we can create our custom sampling layer, which we can define our parameterization trick within. We can use this layer in conjunction with two Dense layers, which we use to retrain the means and variances with. These are then fed into the encoder to finish the encoder.

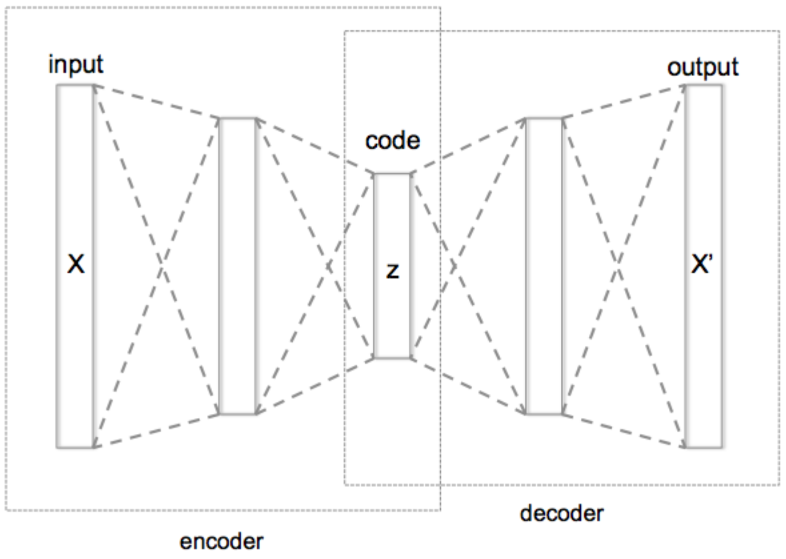


Figure 1: Schematic structure of an autoencoder with 3 fully connected hidden layers. The code (z) is the most internal layer.

[2]

The Decoder

For intuition's sake, we can see the decoder as a reverser of the encoder. The encoder allows us to sample for the latent space, whilst the decoder allows us to sample for the posterior distribution given a latent variable. The implementation of the decoder thus utilizes *transposed convolution layers*, which is also called *Deconvolution*.

The decoder takes the latent vector **z** as input, with the shape of the latent space. We then add a Dense layer with the same shape as our convolutional network, which will be $7 \times 7 \times 64$. We do this so that we can remap back to the original image. We then reshape that layer to our desired dimensions. Afterward, we add our deconvolutional layers to decode the latent variable. Finally, we add one last deconvolutional layer with 1 as filter because the output is binary in image channel (black and white). The code for this and summary output of the model is shown in listing 4 in the Appendix.

Training the model with Gradient Descent

The model Kingma and Welling [3] have proposed allows for reduction of the losses using gradient descent. The parameterization trick allows for the estimator of the model to be the sum of the *reconstruction loss* and the *KL-loss*, which is defined in Eq. 10 in their paper. The estimator is as follows:

$$\mathcal{L}(\theta, \phi; x^i) \approx \frac{1}{2}\sum_{j=1}^{J}\left(1 + \log((\sigma_j^i)^2) - (\mu_j^i)^2 - (\sigma_j^i)^2\right) + \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x^i|z^{(i,l)}) \tag{2}$$

The reconstruction loss measures the difference between the input data and the reconstructed data, while the KL-loss measures the difference between the latent space learned by the VAE and the prior distribution assumed for the latent space. The reconstruction loss encourages accurate reconstruction, and the KL-loss helps to regularize the latent space and prevent overfitting.

The left sum is the KL-loss and the right sum is the reconstruction loss. Training the model to better approximate the true posterior is the same as minimizing the "incorrectness" which is the same as minimizing the loss of the estimator. This can be done using Gradient Descent, where the minimal loss will be found at the minima.

Minimizing the reconstruction loss can be done by minimizing the sum of the binary cross entropy losses, as this is a binary classification task of should the pixel be dark or not. This is the reason we used a sigmoid activation function in the last decoder layer, as it allows us to use the binary cross entropy loss function.

The total loss of the estimator will be the mean of the sum of the KL-loss and the reconstruction loss. Therefore, we can reduce the means of the losses separately and then return the total loss as the sum of these two means.

We get our $\mathbf{z}$, σ and μ from the encoder, which we then utilize for the losses. The latent variable is used for the reconstruction loss with the decoder, and the mean and variance is used for the KL-loss.

We utilize TensorFlow's GradientTape class that allows us to compute the gradient on a given function. Then, utilizing the chosen optimizer (Adam in this case as it is a reliable go-to optimizer), the gradient can be optimized to reduce the loss. We also define metrics that allow us to track the losses and progress of the model.

The code for the custom model was largely based on the guide by TensorFlow on how to customize the Model.Fit() [4]. The only changes needed to be made were to implement the logic for the losses and to integrate the encoder and decoder in the class.

Results and Analysis

Posterior Sampling

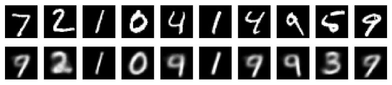
We can sample from the finished VAE model by decoding the data generated from the encoder. Below are images showing comparisons between sampled data from the latent space and the MNIST input into the model for models with different latent space dimensionality.

It is quite evident from the comparisons in 2, that the 6-Dimensional Latent Space model performs the best. The dimensionality of the model affects how well the model is able to capture information regarding the nuances of the input. However, from the curse of dimensionality, too high dimension may cause a decrease in performance as there might be data sparsity. The model may also be overfitting to the data as the model complexity and variance increases. Too small of a dimension may result in the opposite, where some information that may have been captured by increasing dimension is missed out.

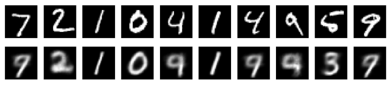
As always, the model performance will be determined by the amount of layers, epochs, batch-sizes, etc. that the model is training on. More complex models utilizing functions such as Batch Normalization and Dropout may yield in better models. However, for this project, a simple model was used to show the general implementation. It can always be extended.

Latent Space Visualization

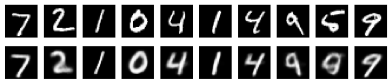
We can plot how the model clusters the different numbers to visualize how it decides the output. In 3, we see clear clusters that are also colored that symbolizes a certain number given a latent variable



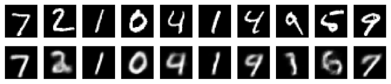
2 Dimensional Latent Space Sampled Data



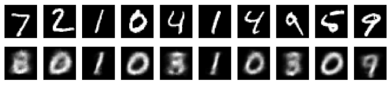
4 Dimensional Latent Space Sampled Data



6 Dimensional Latent Space Sampled Data



8 Dimensional Latent Space Sampled Data



10 Dimensional Latent Space Sampled Data

Figure 2: Sampled Latent Space Data

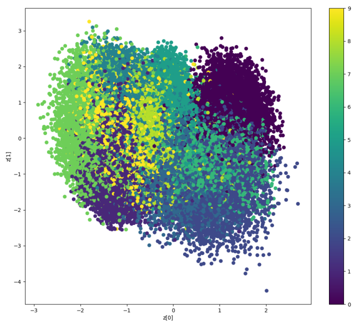
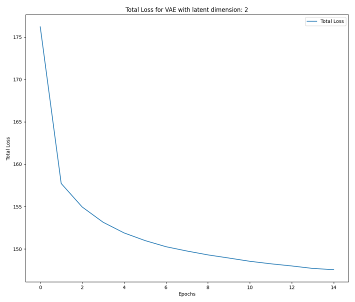


Figure 3: 6 Dimensional VAE Latent Space

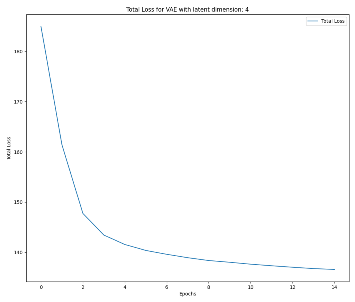
input. Some insights to draw is that around the borders between clusters, the output may look "morphed" as the model has a hard time to decide which number to generate. Furthermore, should one sample from gaps between clusters, the model will not be able to generate comprehensible results. The latent space generated by the 6-dimensional VAE model has quite cohesive clusters with little gaps between them. This means that if we select a random latent variable, the output will most likely be comprehensible.

The Variational Lower Bound

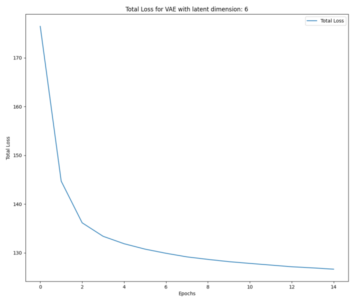
As can be seen in figure 8 below, the loss decreases as expected for the model. As also discussed above, the loss will change differently depending on the amount of dimensions used for the latent space. We can clearly see that 6D is the preferred amount for this model. The performance gets boosted around 6 dimensions, but then declines as the model has difficulty to deal with higher dimensions with the amount of data and the complexity of the model.



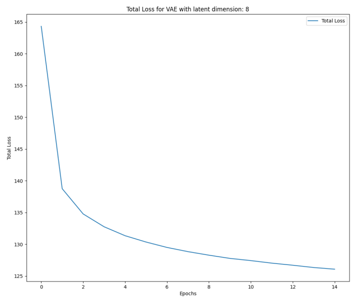
2D



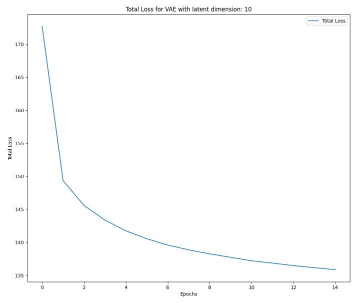
4D



6D



8D



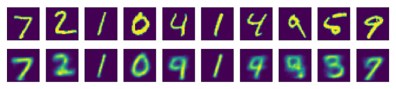
10D

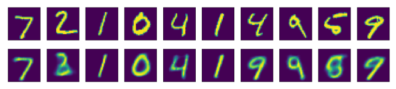
Figure 4: The Variational Lower Bound as the Loss over Epochs

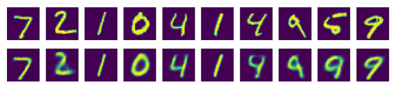
Comparing with a Vanilla Sequential Model

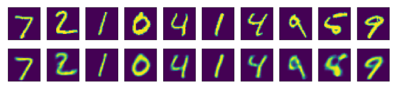
A simple Sequential convolutional network is made with two layers. The model is implemented using Keras modules, with binary cross entropy loss and the Adam optimizer. The Sequential model utilizes simply a list of layers and is not a generative model. A VAE can be used to generate new data, while a Sequential Autoencoder can only reconstruct the input data that it was trained on. Also, by inspecting the latent space of the vanilla autoencoder shown in the article by Anwar [1], picking random latent variables may result in garbage output as the latent space contains significant empty areas which are not clustered. Therefore, the vanilla model is limited in its output and input.

From testing, the Sequential model benefits greatly from dimension increasing. However, this gain quickly flattens out as I believe the model runs into issues with data sparsity in the higher dimensional space.


2 Dimensional Latent Space Sampled Data


4 Dimensional Latent Space Sampled Data


6 Dimensional Latent Space Sampled Data


8 Dimensional Latent Space Sampled Data

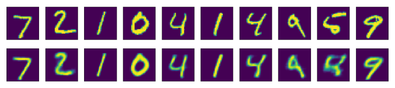

10 Dimensional Latent Space Sampled Data

Figure 5: Sampled Latent Space Data

Comparing the results from the VAE model in 2 with the results for the vanilla model in 5, we can clearly see that the vanilla model performs better for higher dimensions than the VAE model. However, for the six dimensional VAE model, the results are quite comparable to the ones generated by the sequential model. The performance of the sequential model may be because the data which was used contained little nuances. The strength of the VAE model is that it is highly customizable and can be designed to be adapted to data which is highly complicated and nuanced. Therefore, should the model be tested with other data, for example the ”Frey” data used in the paper by Kingma and Welling [3], then the results may be very different. However, for this application, it may be enough to simply use the vanilla model.

References

[1] Aqeel Anwar. Difference between autoencoder (ae) and variational autoencoder (vae), 11 2021.

[2] Chervinskii. Schematic structure of an autoencoder with 3 fully connected hidden layers., 12 2015.

[3] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[4] TensorFlow. Customize what happens in model.fit — tensorflow core, 01 2022.

Appendix

Listing 1: Encoder

```python
encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
```

Listing 2: Sampler

```python
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Listing 3: Full Encoder

```python
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
```

Listing 4: Decoder

```python
latent_inputs = keras.Input(shape=(latent_dim))
x = layers.Dense(units=np.prod(shape_before_flattening), activation="relu")(latent_inputs
    )
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, (3,3), activation="sigmoid", padding="same")(
    x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

Listing 5: Custom Keras Model for VAE with custom loss function

```python
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_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)
            reconstruction_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 = reconstruction_loss + kl_loss

        trainable_vars = self.trainable_variables
        trainable_weights = self.trainable_weights

        gradients = tape.gradient(total_loss, trainable_vars)
        self.optimizer.apply_gradients(zip(gradients, trainable_weights))

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {
            "Total Loss": self.total_loss_tracker.result(),
            "Reconstruction Loss": self.reconstruction_loss_tracker.result(),
            "KL-Loss": self.kl_loss_tracker.result(),
        }
```

Listing 6: Vanilla Sequential Auto Encoder Model

```python
encoder = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(32, 3, strides=2, activation="relu", padding="same"),
        layers.Conv2D(64, 3, strides=2, activation="relu", padding="same"),
        layers.Flatten(),
        layers.Dense(16, activation="relu"),
        layers.Dense(latent_dim, name="latent_vector"),
    ],
    name="encoder",
)

decoder = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(7 * 7 * 32, activation="relu"),
        layers.Reshape((7, 7, 32)),
        layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same"),
        layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same"),
        layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same"),
    ],
    name="decoder",
)

ae = keras.Sequential([encoder, decoder], name="ae")

ae.compile(
    loss="binary_crossentropy",
    optimizer=tf.keras.optimizers.Adam(),
    metrics=["accuracy"]
)

history = ae.fit(
    X_train, X_train,
    epochs=5,
    validation_data=(X_test, X_test)
)
```