∨ Licensed under the Apache License, Version 2.0 (the "License");

```
1  #@title Licensed under the Apache License, Version 2.0 (the "License");
2  # you may not use this file except in compliance with the License.
3  # You may obtain a copy of the License at
4  #
5  # https://www.apache.org/licenses/LICENSE-2.0
6  #
7  # Unless required by applicable law or agreed to in writing, software
8  # distributed under the License is distributed on an "AS IS" BASIS,
9  # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
10 # See the License for the specific language governing permissions and
11 # limitations under the License.
```

∨ Convolutional Variational Autoencoder

[View on TensorFlow.org](#)   [Run in Google Colab](#)   [View source on GitHub](#)   [Download notebook](#)

This notebook demonstrates how to train a Variational Autoencoder (VAE) ([1](#), [2](#)) on the MNIST dataset. A VAE is a probabilistic take on the autoencoder, a model which takes high dimensional input data and compresses it into a smaller representation. Unlike a traditional autoencoder, which maps the input onto a latent vector, a VAE maps the input data into the parameters of a probability distribution, such as the mean and variance of a Gaussian. This approach produces a continuous, structured latent space, which is useful for image generation.
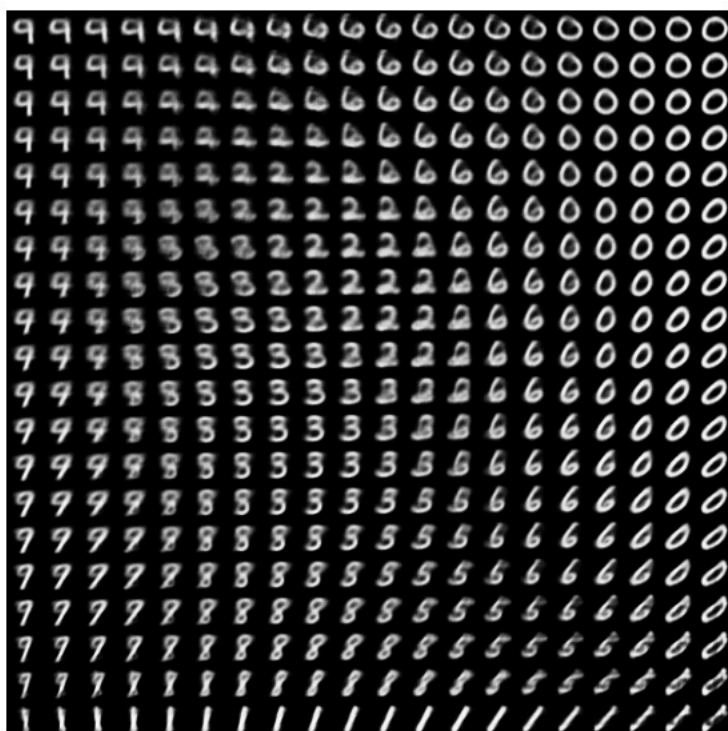
∨ Setup

```
1  !pip install tensorflow-probability
2
3  # to generate gifs
4  !pip install imageio
5  !pip install git+https://github.com/tensorflow/docs
```

```
1 from IPython import display
2
3 import glob
4 import imageio
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import PIL
8 import tensorflow as tf
9 import tensorflow_probability as tfp
10 import time
```

Load the MNIST dataset

Each MNIST image is originally a vector of 784 integers, each of which is between 0-255 and represents the intensity of a pixel. Model each pixel with a Bernoulli distribution in our model, and statically binarize the dataset.

```
1 (train_images, _), (test_images, _) = tf.keras.datasets.mnist.load_data()
```

```
1 def preprocess_images(images):
2   images = images.reshape((images.shape[0], 28, 28, 1)) / 255.
3   return np.where(images > .5, 1.0, 0.0).astype('float32')
4
5 train_images = preprocess_images(train_images)
6 test_images = preprocess_images(test_images)
```

```
1 train_size = 60000
2 batch_size = 32
3 test_size  = 10000
```

Use *tf.data* to batch and shuffle the data

```
1 train_dataset = (tf.data.Dataset.from_tensor_slices(train_images)
2                  .shuffle(train_size).batch(batch_size))
3 test_dataset = (tf.data.Dataset.from_tensor_slices(test_images)
4                  .shuffle(test_size).batch(batch_size))
```

Define the encoder and decoder networks with *tf.keras.Sequential*

In this VAE example, use two small ConvNets for the encoder and decoder networks. In the literature, these networks are also referred to as inference/recognition and generative models respectively. Use `tf.keras.Sequential` to simplify implementation. Let x and z denote the observation and latent variable respectively in the following descriptions.

Encoder network

This defines the approximate posterior distribution $q(z|x)$, which takes as input an observation and outputs a set of parameters for specifying the conditional distribution of the latent representation z. In this example, simply model the distribution as a diagonal Gaussian, and the network outputs the mean and log-variance parameters of a factorized Gaussian. Output log-variance instead of the variance directly for numerical stability.

Decoder network

This defines the conditional distribution of the observation $p(x|z)$, which takes a latent sample z as input and outputs the parameters for a conditional distribution of the observation. Model the latent distribution prior $p(z)$ as a unit Gaussian.

Reparameterization trick

To generate a sample z for the decoder during training, you can sample from the latent distribution defined by the parameters outputted by the encoder, given an input observation x. However, this sampling operation creates a bottleneck because backpropagation cannot flow through a random node.

To address this, use a reparameterization trick. In our example, you approximate z using the decoder parameters and another parameter ϵ as follows:

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

where μ and σ represent the mean and standard deviation of a Gaussian distribution respectively. They can be derived from the decoder output. The ϵ can be thought of as a random noise used to maintain stochasticity of z. Generate ϵ from a standard normal distribution.

The latent variable z is now generated by a function of μ, σ and ϵ, which would enable the model to backpropagate gradients in the encoder through μ and σ respectively, while maintaining stochasticity through ϵ.

Network architecture

For the encoder network, use two convolutional layers followed by a fully-connected layer. In the decoder network, mirror this architecture by using a fully-connected layer followed by three convolution transpose layers (a.k.a. deconvolutional layers in some contexts). Note, it's common practice to avoid using batch normalization when training VAEs, since the additional stochasticity due to using mini-batches may aggravate instability on top of the stochasticity from sampling.

```python
class CVAE(tf.keras.Model):
  """Convolutional variational autoencoder."""

  def __init__(self, latent_dim):
    super(CVAE, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential(
        [
            tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
            tf.keras.layers.Conv2D(
                filters=32, kernel_size=3, strides=(2, 2), activation='relu'),
            tf.keras.layers.Conv2D(
                filters=64, kernel_size=3, strides=(2, 2), activation='relu'),
            tf.keras.layers.Flatten(),
            # No activation
            tf.keras.layers.Dense(latent_dim + latent_dim),
        ]
    )

    self.decoder = tf.keras.Sequential(
        [
            tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
            tf.keras.layers.Dense(units=7*7*32, activation=tf.nn.relu),
            tf.keras.layers.Reshape(target_shape=(7, 7, 32)),
            tf.keras.layers.Conv2DTranspose(
                filters=64, kernel_size=3, strides=2, padding='same',
                activation='relu'),
            tf.keras.layers.Conv2DTranspose(
                filters=32, kernel_size=3, strides=2, padding='same',
                activation='relu'),
            # No activation
            tf.keras.layers.Conv2DTranspose(
                filters=1, kernel_size=3, strides=1, padding='same'),
        ]
    )

  @tf.function
  def sample(self, eps=None):
    if eps is None:
      eps = tf.random.normal(shape=(100, self.latent_dim))
    return self.decode(eps, apply_sigmoid=True)

  def encode(self, x):
    mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
    return mean, logvar

  def reparameterize(self, mean, logvar):
    eps = tf.random.normal(shape=mean.shape)
    return eps * tf.exp(logvar * .5) + mean

  def decode(self, z, apply_sigmoid=False):
    logits = self.decoder(z)
    if apply_sigmoid:
      probs = tf.sigmoid(logits)
      return probs
    return logits
```

⌄ Define the loss function and the optimizer

VAEs train by maximizing the evidence lower bound (ELBO) on the marginal log-likelihood:

$$\log p(x) \geq \text{ELBO} = \mathbb{E}_{q(z|x)}\left[\log \frac{p(x, z)}{q(z|x)}\right].$$

In practice, optimize the single sample Monte Carlo estimate of this expectation:

$$\log p(x|z) + \log p(z) - \log q(z|x),$$

where z is sampled from $q(z|x)$.

Note: You could also analytically compute the KL term, but here you incorporate all three terms in the Monte Carlo estimator for simplicity.

```python
1 optimizer = tf.keras.optimizers.Adam(1e-4)
2
3
4 def log_normal_pdf(sample, mean, logvar, raxis=1):
5   log2pi = tf.math.log(2. * np.pi)
6   return tf.reduce_sum(
7       -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
8       axis=raxis)
9
10
11 def compute_loss(model, x):
12   mean, logvar = model.encode(x)
13   z = model.reparameterize(mean, logvar)
14   x_logit = model.decode(z)
15   cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
16   logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
17   logpz = log_normal_pdf(z, 0., 0.)
18   logqz_x = log_normal_pdf(z, mean, logvar)
19   return -tf.reduce_mean(logpx_z + logpz - logqz_x)
20
21
22 @tf.function
23 def train_step(model, x, optimizer):
24   """Executes one training step and returns the loss.
25
26   This function computes the loss and gradients, and uses the latter to
27   update the model's parameters.
28   """
29   with tf.GradientTape() as tape:
30     loss = compute_loss(model, x)
31   gradients = tape.gradient(loss, model.trainable_variables)
32   optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

∨ Training

- Start by iterating over the dataset
- During each iteration, pass the image to the encoder to obtain a set of mean and log-variance parameters of the approximate posterior $q(z|x)$
- then apply the *reparameterization trick* to sample from $q(z|x)$
- Finally, pass the reparameterized samples to the decoder to obtain the logits of the generative distribution $p(x|z)$
- Note: Since you use the dataset loaded by keras with 60k datapoints in the training set and 10k datapoints in the test set, our resulting ELBO on the test set is slightly higher than reported results in the literature which uses dynamic binarization of Larochelle's MNIST.

Generating images

- After training, it is time to generate some images
- Start by sampling a set of latent vectors from the unit Gaussian prior distribution $p(z)$
- The generator will then convert the latent sample z to logits of the observation, giving a distribution $p(x|z)$
- Here, plot the probabilities of Bernoulli distributions

```python
1 epochs = 100
2 # set the dimensionality of the latent space to a plane for visualization later
3 latent_dim = 2
4 num_examples_to_generate = 16
5
6 # keeping the random vector constant for generation (prediction) so
7 # it will be easier to see the improvement.
8 random_vector_for_generation = tf.random.normal(
9     shape=[num_examples_to_generate, latent_dim])
10 model = CVAE(latent_dim)
```

```python
1  def generate_and_save_images(model, epoch, test_sample):
2    mean, logvar = model.encode(test_sample)
3    z = model.reparameterize(mean, logvar)
4    predictions = model.sample(z)
5    fig = plt.figure(figsize=(4, 4))
6
7    for i in range(predictions.shape[0]):
8      plt.subplot(4, 4, i + 1)
9      plt.imshow(predictions[i, :, :, 0], cmap='gray')
10     plt.axis('off')
11
12   # tight_layout minimizes the overlap between 2 sub-plots
13   plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
14   plt.show()
```

```python
1  # Pick a sample of the test set for generating output images
2  assert batch_size >= num_examples_to_generate
3  for test_batch in test_dataset.take(1):
4    test_sample = test_batch[0:num_examples_to_generate, :, :, :]
```

```python
1  generate_and_save_images(model, 0, test_sample)
2
3  for epoch in range(1, epochs + 1):
4    start_time = time.time()
5    for train_x in train_dataset:
6      train_step(model, train_x, optimizer)
7    end_time = time.time()
8
9    loss = tf.keras.metrics.Mean()
10   for test_x in test_dataset:
11     loss(compute_loss(model, test_x))
12   elbo = -loss.result()
13   display.clear_output(wait=False)
14   print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'
15         .format(epoch, elbo, end_time - start_time))
16   generate_and_save_images(model, epoch, test_sample)
```

⌄ Display a generated image from the last training epoch

```python
1  def display_image(epoch_no):
2    return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```

```python
1  plt.imshow(display_image(epoch))
2  plt.axis('off')  # Display images
```

⌄ Display an animated GIF of all the saved images

```python
1  anim_file = 'cvae.gif'
2
3  with imageio.get_writer(anim_file, mode='I') as writer:
4    filenames = glob.glob('image*.png')
5    filenames = sorted(filenames)
6    for filename in filenames:
7      image = imageio.imread(filename)
8      writer.append_data(image)
9    image = imageio.imread(filename)
10   writer.append_data(image)
```

```python
1  import tensorflow_docs.vis.embed as embed
2  embed.embed_file(anim_file)
```

⌄ Display a 2D manifold of digits from the latent space

Running the code below will show a continuous distribution of the different digit classes, with each digit morphing into another across the 2D latent space. Use [TensorFlow Probability](#) to generate a standard normal distribution for the latent space.

```python
1 def plot_latent_images(model, n, digit_size=28):
2   """Plots n x n digit images decoded from the latent space."""
3
4   norm = tfp.distributions.Normal(0, 1)
5   grid_x = norm.quantile(np.linspace(0.05, 0.95, n))
6   grid_y = norm.quantile(np.linspace(0.05, 0.95, n))
7   image_width = digit_size*n
8   image_height = image_width
9   image = np.zeros((image_height, image_width))
10
11  for i, yi in enumerate(grid_x):
12    for j, xi in enumerate(grid_y):
13      z = np.array([[xi, yi]])
14      x_decoded = model.sample(z)
15      digit = tf.reshape(x_decoded[0], (digit_size, digit_size))
16      image[i * digit_size: (i + 1) * digit_size,
17            j * digit_size: (j + 1) * digit_size] = digit.numpy()
18
19  plt.figure(figsize=(10, 10))
20  plt.imshow(image, cmap='Greys_r')
21  plt.axis('Off')
22  plt.show()
```

```python
1 plot_latent_images(model, 20)
```

Next steps

This tutorial has demonstrated how to implement a convolutional variational autoencoder using TensorFlow.

As a next step, you could try to improve the model output by increasing the network size. For instance, you could try setting the `filter` parameters for each of the `Conv2D` and `Conv2DTranspose` layers to 512. Note that in order to generate the final 2D latent image plot, you would need to keep `latent_dim` to 2. Also, the training time would increase as the network size increases.

You could also try implementing a VAE using a different dataset, such as CIFAR-10.

VAEs can be implemented in several different styles and of varying complexity. You can find additional implementations in the following sources:

- Variational AutoEncoder (keras.io)
- VAE example from "Writing custom layers and models" guide (tensorflow.org)
- TFP Probabilistic Layers: Variational Auto Encoder

If you'd like to learn more about the details of VAEs, please refer to An Introduction to Variational Autoencoders.