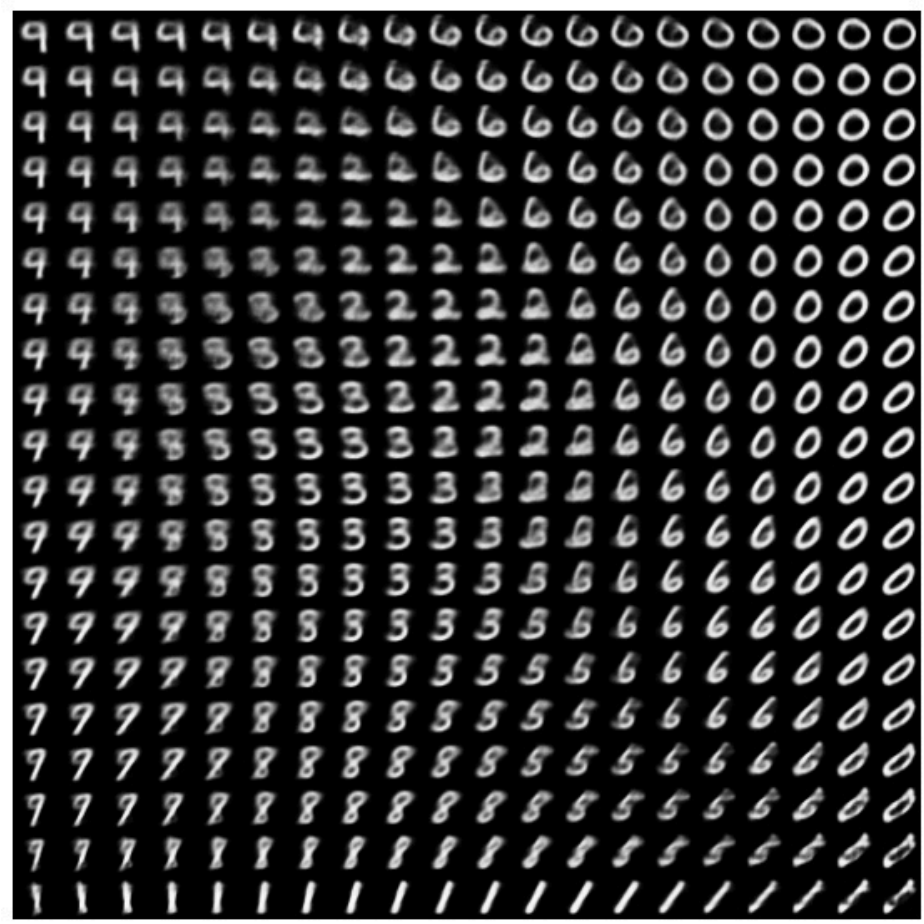
Codificador automático variacional convolucional

Este cuaderno demuestra cómo entrenar un codificador automático variacional (VAE) (1 (https://arxiv.org/abs/1312.6114) , 2 (https://arxiv.org/abs/1401.4082)) en el conjunto de datos MNIST. Un VAE es una versión probabilística del codificador automático, un modelo que toma datos de entrada de alta dimensión y los comprime en una representación más pequeña. A diferencia de un codificador automático tradicional, que asigna la entrada a un vector latente, un VAE asigna los datos de entrada a los parámetros de una distribución de probabilidad, como la media y la varianza de una Gaussiana. Este enfoque produce un espacio latente estructurado continuo, que es útil para la generación de imágenes.



Configuración

```
$ pip install tensorflow-probability
$
$ # to generate gifs
$ pip install imageio
$ pip install git+https://github.com/tensorflow/docs
```

```
from IPython import display

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf
import tensorflow_probability as tfp
import time
```

Cargue el conjunto de datos MNIST

Cada imagen MNIST es originalmente un vector de 784 números enteros, cada uno de los cuales está entre 0 y 255 y representa la intensidad de un píxel. Modele cada píxel con una distribución de Bernoulli en nuestro modelo y binarice estáticamente el conjunto de datos.

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

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

train_images = preprocess_images(train_images)
test_images = preprocess_images(test_images)
```

```
train_size = 60000
batch_size = 32
test_size = 10000
```

Use *tf.data* para procesar por lotes y mezclar los datos

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

Defina las redes de codificador y decodificador con *tf.keras.Sequential*

En este ejemplo de VAE, use dos ConvNet pequeños para las redes de codificador y decodificador. En la literatura, estas redes también se conocen como inferencia/reconocimiento y modelos generativos, respectivamente. Utilice `tf.keras.Sequential` (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential?hl=es-419) para

simplificar la implementación. Deje que x y z denoten la observación y la variable latente respectivamente en las siguientes descripciones.

Red de codificador

Esto define la distribución posterior aproximada $q(z|x)$, que toma como entrada una observación y genera un conjunto de parámetros para especificar la distribución condicional de la representación latente z. En este ejemplo, simplemente modele la distribución como una gaussiana diagonal, y la red generará los parámetros de media y de varianza logarítmica de una gaussiana factorizada. Salida log-varianza en lugar de la varianza directamente para la estabilidad numérica.

Red decodificador

Esto define la distribución condicional de la observación $p(x|z)$, que toma una muestra latente z como entrada y genera los parámetros para una distribución condicional de la observación. Modele la distribución latente anterior $p(z)$ como una unidad gaussiana.

Truco de reparametrización

Para generar una muestra z para el decodificador durante el entrenamiento, puede tomar muestras de la distribución latente definida por los parámetros emitidos por el codificador, dada una observación de entrada x. Sin embargo, esta operación de muestreo crea un cuello de botella porque la retropropagación no puede fluir a través de un nodo aleatorio.

Para abordar esto, use un truco de reparametrización. En nuestro ejemplo, aproxima z usando los parámetros del decodificador y otro parámetro ϵ de la siguiente manera:

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

donde μ y σ representan la media y la desviación estándar de una distribución gaussiana, respectivamente. Se pueden derivar de la salida del decodificador. El ϵ se puede considerar como un ruido aleatorio que se utiliza para mantener la estocasticidad del z. Genere ϵ a partir de una distribución normal estándar.

La variable latente z ahora se genera mediante una función de μ, σ y ϵ, lo que permitiría que el modelo propagara hacia atrás los gradientes en el codificador a través μ y σ respectivamente, manteniendo la estocasticidad a través de ϵ.

Red de arquitectura

Para la red del codificador, utilice dos capas convolucionales seguidas de una capa completamente conectada. En la red del decodificador, refleje esta arquitectura mediante el uso de una capa totalmente conectada seguida de tres capas de transposición de convolución (también conocidas como capas deconvolucionales en algunos contextos). Tenga en cuenta que es una práctica común evitar el uso de la normalización por lotes al entrenar VAE, ya que la estocasticidad adicional debido al uso de minilotes puede agravar la inestabilidad además de la estocasticidad del muestreo.

```
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
```

Definir la función de pérdida y el optimizador

Los VAE se entrenan maximizando el límite inferior de evidencia (ELBO) en el log-verosimilitud marginal:

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

En la práctica, optimice la estimación de Monte Carlo de muestra única de esta expectativa:

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

donde z se muestra a partir de $q(z|x)$.

Nota: También puede calcular analíticamente el término KL, pero aquí incorpora los tres términos en el estimador de Monte Carlo por simplicidad.

```
optimizer = tf.keras.optimizers.Adam(1e-4)


def log_normal_pdf(sample, mean, logvar, raxis=1):
  log2pi = tf.math.log(2. * np.pi)
  return tf.reduce_sum(
      -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
      axis=raxis)


def compute_loss(model, x):
  mean, logvar = model.encode(x)
  z = model.reparameterize(mean, logvar)
  x_logit = model.decode(z)
  cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
  logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
  logpz = log_normal_pdf(z, 0., 0.)
  logqz_x = log_normal_pdf(z, mean, logvar)
  return -tf.reduce_mean(logpx_z + logpz - logqz_x)


@tf.function
def train_step(model, x, optimizer):
  """Executes one training step and returns the loss.

  This function computes the loss and gradients, and uses the latter to
  update the model's parameters.
```

```
  """
  with tf.GradientTape() as tape:
    loss = compute_loss(model, x)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Capacitación

- Comience iterando sobre el conjunto de datos
- Durante cada iteración, pase la imagen al codificador para obtener un conjunto de parámetros de varianza logarítmica y media $q(z|x)$posterior aproximado29
- luego aplique el *truco de reparametrización* a la muestra de $q(z|x)$
- Finalmente, pase las muestras reparametrizadas al decodificador para obtener los logits de la distribución generativa $p(x|z)$
- Nota: Dado que utiliza el conjunto de datos cargado por keras con 60 000 puntos de datos en el conjunto de entrenamiento y 10 000 puntos de datos en el conjunto de prueba, nuestro ELBO resultante en el conjunto de prueba es ligeramente superior a los resultados informados en la literatura que utiliza la binarización dinámica del MNIST de Larochelle.

Generando imágenes

- Después del entrenamiento, es hora de generar algunas imágenes.
- Comience muestreando un conjunto de vectores latentes de la unidad Gaussiana de distribución previa $p(z)$
- El generador luego convertirá la muestra latente z a logits de la observación, dando una distribución $p(x|z)$
- Aquí, trace las probabilidades de las distribuciones de Bernoulli

```
epochs = 10
# set the dimensionality of the latent space to a plane for visualization later
latent_dim = 2
num_examples_to_generate = 16

# keeping the random vector constant for generation (prediction) so
# it will be easier to see the improvement.
random_vector_for_generation = tf.random.normal(
    shape=[num_examples_to_generate, latent_dim])
model = CVAE(latent_dim)
```

```
def generate_and_save_images(model, epoch, test_sample):
  mean, logvar = model.encode(test_sample)
  z = model.reparameterize(mean, logvar)
  predictions = model.sample(z)
  fig = plt.figure(figsize=(4, 4))

  for i in range(predictions.shape[0]):
    plt.subplot(4, 4, i + 1)
    plt.imshow(predictions[i, :, :, 0], cmap='gray')
    plt.axis('off')

  # tight_layout minimizes the overlap between 2 sub-plots
  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()
```

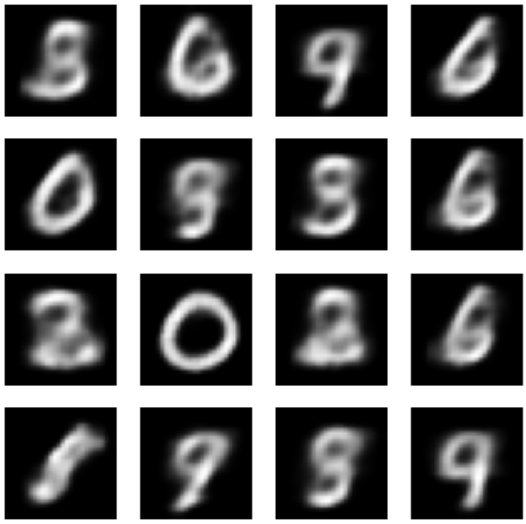
```
# Pick a sample of the test set for generating output images
assert batch_size >= num_examples_to_generate
for test_batch in test_dataset.take(1):
  test_sample = test_batch[0:num_examples_to_generate, :, :, :]
```

```
generate_and_save_images(model, 0, test_sample)

for epoch in range(1, epochs + 1):
  start_time = time.time()
  for train_x in train_dataset:
    train_step(model, train_x, optimizer)
  end_time = time.time()

  loss = tf.keras.metrics.Mean()
  for test_x in test_dataset:
    loss(compute_loss(model, test_x))
  elbo = -loss.result()
  display.clear_output(wait=False)
  print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'
        .format(epoch, elbo, end_time - start_time))
  generate_and_save_images(model, epoch, test_sample)
```

```
Epoch: 10, Test set ELBO: -156.4964141845703, time elapse for current epoch: 4.854437351226807
```



Mostrar una imagen generada a partir de la última época de entrenamiento

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

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

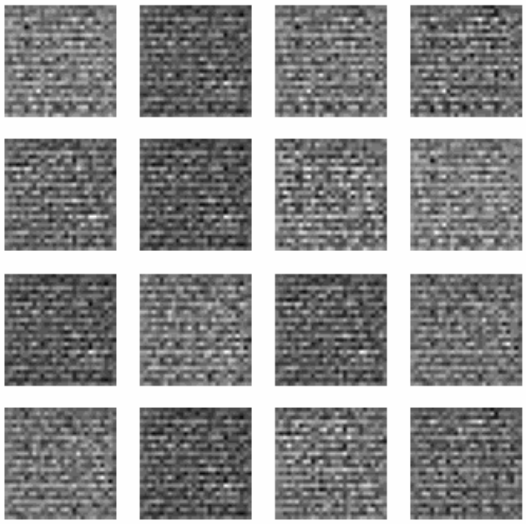
```
(-0.5, 287.5, 287.5, -0.5)
```

Muestra un GIF animado de todas las imágenes guardadas

```
anim_file = 'cvae.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  image = imageio.imread(filename)
  writer.append_data(image)
```

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```



Mostrar una variedad 2D de dígitos del espacio latente

Ejecutar el siguiente código mostrará una distribución continua de las diferentes clases de dígitos, con cada dígito transformándose en otro a través del espacio latente 2D. Usa TensorFlow Probability (https://www.tensorflow.org/probability?hl=es-419) para generar una distribución normal estándar para el espacio latente.

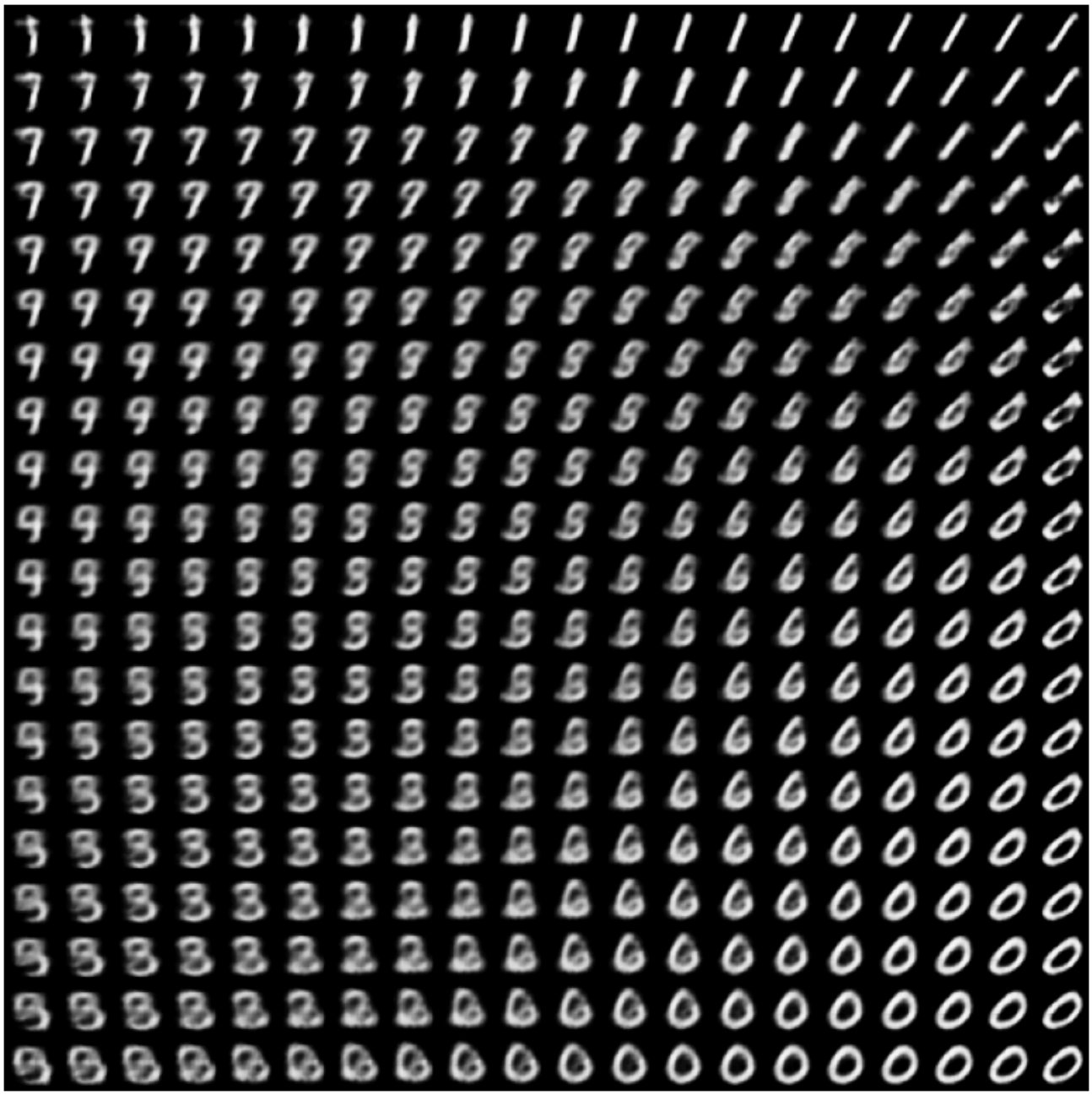
```
def plot_latent_images(model, n, digit_size=28):
  """Plots n x n digit images decoded from the latent space."""

  norm = tfp.distributions.Normal(0, 1)
  grid_x = norm.quantile(np.linspace(0.05, 0.95, n))
  grid_y = norm.quantile(np.linspace(0.05, 0.95, n))
  image_width = digit_size*n
  image_height = image_width
  image = np.zeros((image_height, image_width))

  for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
      z = np.array([[xi, yi]])
      x_decoded = model.sample(z)
      digit = tf.reshape(x_decoded[0], (digit_size, digit_size))
      image[i * digit_size: (i + 1) * digit_size,
            j * digit_size: (j + 1) * digit_size] = digit.numpy()

  plt.figure(figsize=(10, 10))
  plt.imshow(image, cmap='Greys_r')
  plt.axis('Off')
  plt.show()
```

```
plot_latent_images(model, 20)
```



Próximos pasos

Este tutorial ha demostrado cómo implementar un codificador automático variacional convolucional usando TensorFlow.

Como siguiente paso, podría intentar mejorar el resultado del modelo aumentando el tamaño de la red. Por ejemplo, podría intentar establecer los parámetros de `filter` para cada una de las capas `Conv2D` y `Conv2DTranspose` en 512. Tenga en cuenta que para generar el gráfico de imagen latente 2D final, deberá mantener `latent_dim` en 2. Además, el tiempo de entrenamiento aumentaría medida que aumenta el tamaño de la red.

También podría intentar implementar un VAE utilizando un conjunto de datos diferente, como CIFAR-10.

Los VAE se pueden implementar en varios estilos diferentes y de diversa complejidad. Puede encontrar implementaciones adicionales en las siguientes fuentes:

- Codificador automático variacional (keras.io) (https://keras.io/examples/generative/vae/)
- Ejemplo de VAE de la guía "Escribir capas y modelos personalizados" (tensorflow.org) (https://www.tensorflow.org/guide/keras/custom_layers_and_models?hl=es-419#putting_it_all_together_an_end-to-end_example)
- Capas probabilísticas TFP: codificador automático variacional (https://www.tensorflow.org/probability/examples/Probabilistic_Layers_VAE?hl=es-419)

Si desea obtener más información sobre los detalles de VAE, consulte Introducción a los codificadores automáticos variacionales (https://arxiv.org/abs/1906.02691) .