VAE: Code

We discuss the highlights of the code in this notebook (vae.ipynb)

- derived from the Keras examples (https://keras.io/examples/generative/vae/)

Deriving a new Model via sub-classing

Similar to the code for the `Autoencoder`, the `VAE` class

- is sub-classed from `Model`
- *contains* an Encoder `self.encoder` and a Decoder `self.decoder`
    - but these are defined *external* to the class
    - rather than within the class code

Custom `train_step`

Unlike the code for `Autoencoder`, the model for `VAE` implements the model behavior

- **not** by overriding the `call` method
- but by *overriding* the training step `train_step` method

So

- we can't actually "call" this model (e.g., `x = m(x)`)
- but, during training, we can make it behave in the desired manner
    - call encoder
    - call decoder

```
z_mean, z_log_var, z = self.encoder(data)
reconstruction = self.decoder(z)
```

Custom training step vs custom loss

The reason for overriding `train_step` rather than `call` may not be obvious at first glance.

The VAE has a *complex loss function* of two parts

$$\begin{aligned}\mathcal{L} &= -\log(p_\Theta(\mathbf{x})) + \mathbf{KL}(q_\Phi(\mathbf{z}|\mathbf{x}) \mid\mid q(\mathbf{z}|\mathbf{x})) \\ &= \mathcal{L}_R + \mathcal{L}_D\end{aligned}$$

- Reconstruction Loss $\mathcal{L}_R$
- KL loss $\mathcal{L}_D$

One could (in theory) create a custom loss

- an the ordinary training mechanism
- would compute the loss
- and the gradients of the loss

By overriding the `train_step`, *our code* becomes responsible for the gradient computation

```
with tf.GradientTape() as tape:
           z_mean, z_log_var, z = self.encoder(data)
           reconstruction = self.decoder(z)

           reconstruction_loss = ...
           kl_loss = ...
           total_loss = reconstruction_loss + kl_loss

       grads = tape.gradient(total_loss, self.trainable_weights)
       self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
```

The computation of `total_loss`

- is performed within the scope of `tf.GradientTape()`
- which allows automatic differentiation of the loss

We then manually compute the gradients

```
grads = tape.gradient(total_loss, self.trainable_weights)
```

and update the weights through the gradients

```
self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
```

So why go through this effort (as opposed to a custom loss) ?

The model *also* provides *custom metrics*

- values gathered during training
    - we are used to seeing Loss and Validation Loss

These metrics are

- total loss
- reconstruction loss
- KL loss

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

The reason for overriding `train_step` is so that *we can update the custom metrics* during training

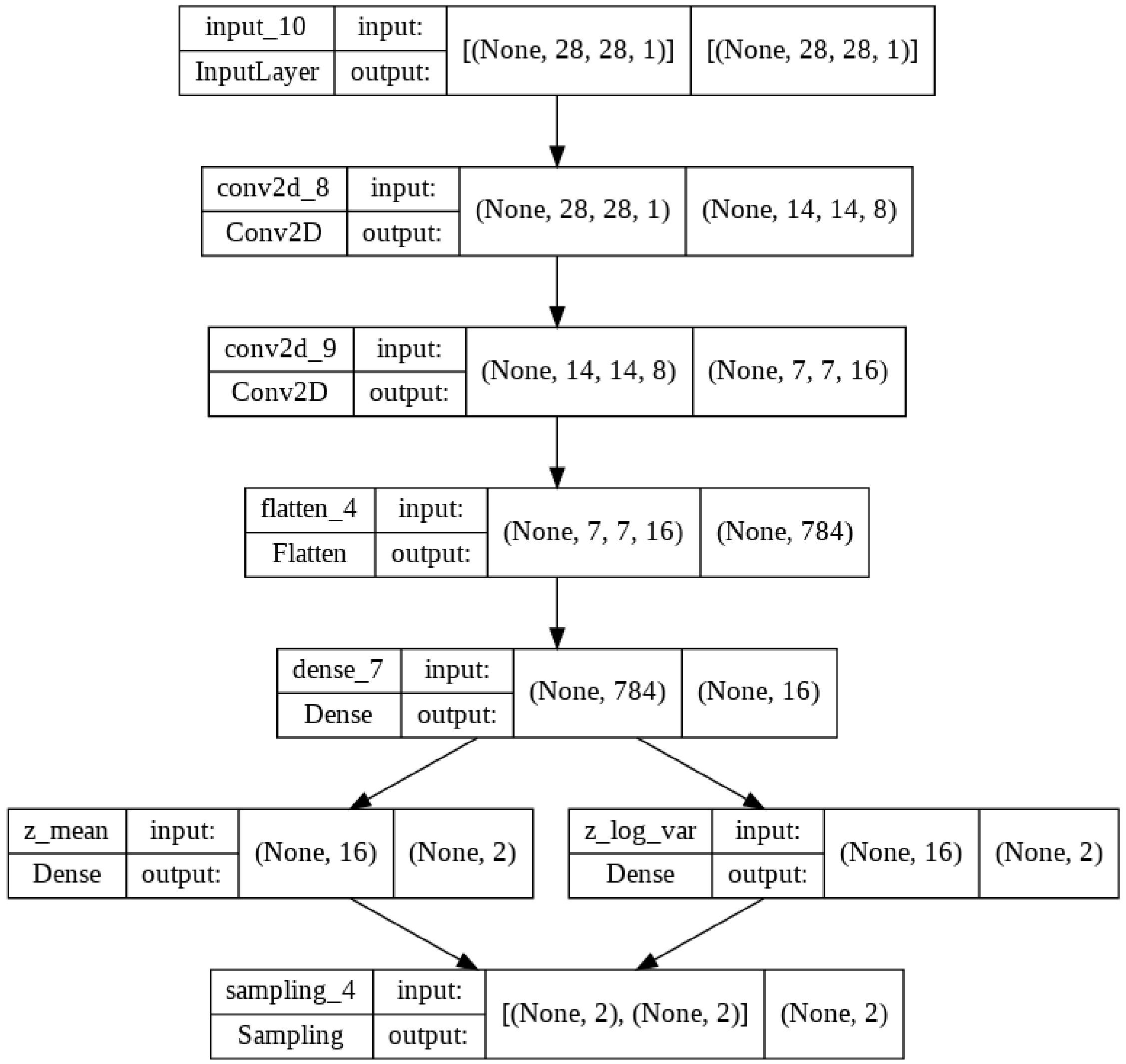
```
self.total_loss_tracker.update_state(total_loss)
self.reconstruction_loss_tracker.update_state(reconstruction_loss)
self.kl_loss_tracker.update_state(kl_loss)
```

Here is the complete code for the method

```
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_va
r))
        kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
        total_loss = reconstruction_loss + 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.reconstruction_loss_tracker.update_state(reconstruction_loss)
    self.kl_loss_tracker.update_state(kl_loss)
    return {
```

Encoder architecture

We can see the Encoder architecture



input_10 | InputLayer | input: | output: | [(None, 28, 28, 1)] | [(None, 28, 28, 1)]
conv2d_8 | Conv2D | input: | output: | (None, 28, 28, 1) | (None, 14, 14, 8)
conv2d_9 | Conv2D | input: | output: | (None, 14, 14, 8) | (None, 7, 7, 16)
flatten_4 | Flatten | input: | output: | (None, 7, 7, 16) | (None, 784)
dense_7 | Dense | input: | output: | (None, 784) | (None, 16)
z_mean | Dense | input: | output: | (None, 16) | (None, 2)
z_log_var | Dense | input: | output: | (None, 16) | (None, 2)
sampling_4 | Sampling | input: | output: | [(None, 2), (None, 2)] | (None, 2)

The input image $(28 \times 28 \times 1)$

- is processed by 2 Convolutional layers
- creating a representation with the *same spatial dimensions* as the input
- but with many more (16 versus 1) features

It is likely that this representation is richer than the alternative

- of initial flattening
- processing by `Dense` layers

You can see (in components `z_mean` and `z_log_var`)

- that the final representation of the input
- is used to derive the moments ($\mu^{(\mathbf{i})}$ and $\sigma_{(\mathbf{i})}$) of the *distribution* for example i

The final layer samples from this *multivariate* distribution

- to produce the latent representation
- a *vector* of length `latent_dim`

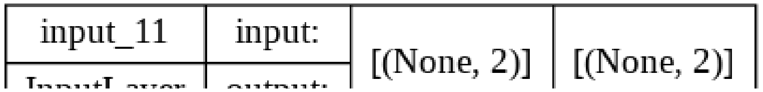
Note

In our code `latent_dim = 2`

- this is a sample from a *bivariate* distribution with mean $\mu^{(\mathbf{i})}$ and standard deviation $\sigma^{(\mathbf{i})}$
- don't confuse the length of the sample vector with the pair of moments

Decoder Architecture

We can see the Encoder architecture


input_11
input:
[(None, 2)]
[(None, 2)]

You can see that the Decoder

- takes a latent vector of length `latent_dim` as input
- Inverts the Encoder's Convolutional layers (`Conv2DTranspose`)

These steps are *almost* exactly the inverse of

- reverse of the Encoder's operation sequence

Kernel size of 1

The final `Conv2DTranspose` layer

- is an example of a Convolution with kernel size **one** discussed in Intro course (CNN_Space_and_Time.ipynb#Kernel-size-1)
- whose sole purpose is to "re-size" the channel dimension
    - in this case: to 1 channel, just like the input

Deriving a new Layer via sub-classing

Sampling from the multivariate distribution comes from a *new layer type* `Sampling`

- sub-class of the generic `Layer` class

Similar to sub-classing a `Model` the work of sub-classing a `Layer`

- comes from overriding the `call` method

There is one sample drawn from *each example* in the batch

- each with it's own $\mu^{(\mathbf{i})}$ and $\sigma^{(\mathbf{i})}$

```
  class Sampling(layers.Layer):
      """Uses (z_mean, z_log_var) to sample z, the vector encoding a dig
it."""

      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
```

Exploring the latent space

The notebook runs some experiments

- explore the latent space (vae.ipynb#Display-a-grid-of-sampled-digits)

Conditional VAE

The [code for the Conditional VAE (vae.ipynb#Conditional-VAE)](vae.ipynb#Conditional-VAE) is very similar to that of the unconditional VAE.

The main difference is that both the Encoder and Decoder inputs are now *pairs* where

- the second element of the pair is the desired label
- the first element is the same as the unconditional VAE

The label arguments

- are implemented as One Hot Encoded vectors
    - for both the Encoder and Decoder
    - look for the `InputLayer` of shape `(None, 10)`
        - this is the OHE of 10 possible classes (digits 0 through 9)
- not categorical constants

The way that an image with a *specific* label gets generated is not obvious

- puzzling in its simplicity

The Encoder

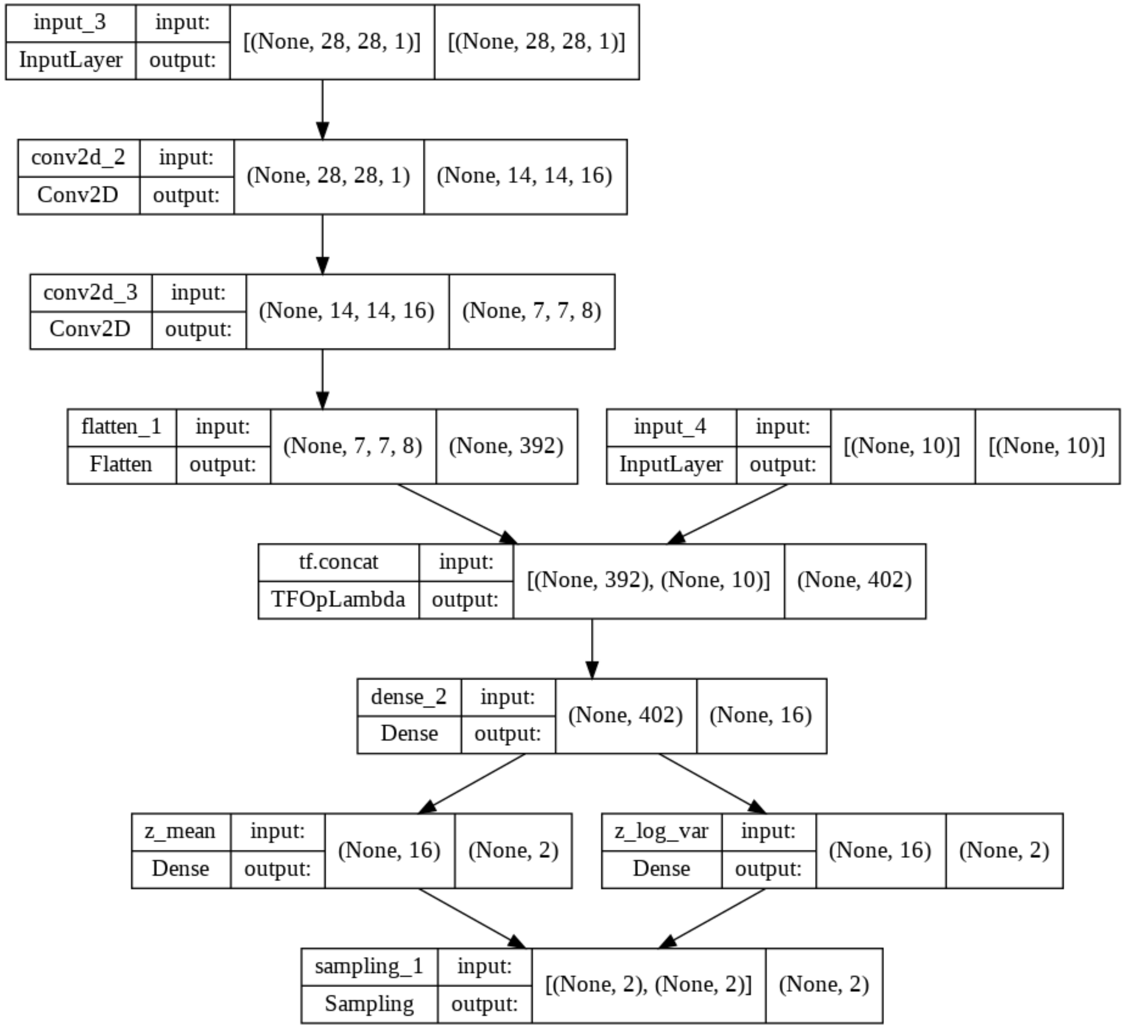
- ignores the label for most of its processing of the input
- uses the label
    - to modify the alternate representation of the input
    - immediately before creating $\mu^{(\mathbf{i})}$ and $\sigma^{(\mathbf{i})}$

Thus, $\mu^{(\mathbf{i})}$ and $\sigma^{(\mathbf{i})}$ are *conditioned* on both

- the alternate representation that the unconditional VAE would produce
- **and** the label

Here is the Encoder

- notice the two distinct `Input` layers


input_3 | InputLayer | input: | output: | [(None, 28, 28, 1)] | [(None, 28, 28, 1)]
conv2d_2 | Conv2D | input: | output: | (None, 28, 28, 1) | (None, 14, 14, 16)
conv2d_3 | Conv2D | input: | output: | (None, 14, 14, 16) | (None, 7, 7, 8)
flatten_1 | Flatten | input: | output: | (None, 7, 7, 8) | (None, 392)
input_4 | InputLayer | input: | output: | [(None, 10)] | [(None, 10)]
tf.concat | TFOpLambda | input: | output: | [(None, 392), (None, 10)] | (None, 402)
dense_2 | Dense | input: | output: | (None, 402) | (None, 16)
z_mean | Dense | input: | output: | (None, 16) | (None, 2)
z_log_var | Dense | input: | output: | (None, 16) | (None, 2)
sampling_1 | Sampling | input: | output: | [(None, 2), (None, 2)] | (None, 2)

The Decoder

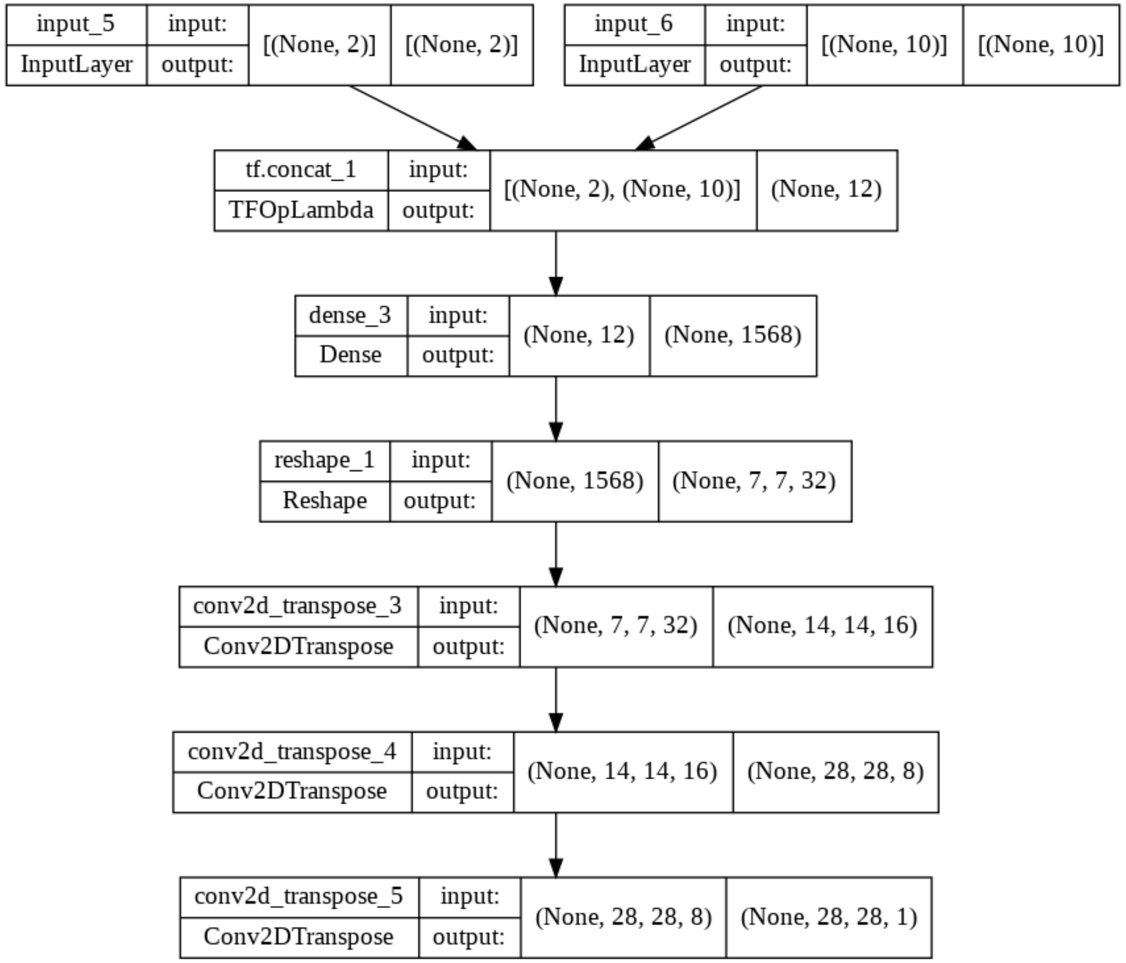
- concatenates the OHE of the label to the latent created by the Encoder
- to create a "longer" latent representation
    - length is: `latent_dim` + `number of classes`
    - versus `latent_dim` for the unconditional VAE

The CVAE Decoder is almost exactly the same as the unconditional VAE Decoder

- just with a longer latent

Here is the Decoder

- notice the two distinct `Input` layers


input_5 | InputLayer | input: [(None, 2)] | output: [(None, 2)]
input_6 | InputLayer | input: [(None, 10)] | output: [(None, 10)]
tf.concat_1 | TFOpLambda | input: [(None, 2), (None, 10)] | output: (None, 12)
dense_3 | Dense | input: (None, 12) | output: (None, 1568)
reshape_1 | Reshape | input: (None, 1568) | output: (None, 7, 7, 32)
conv2d_transpose_3 | Conv2DTranspose | input: (None, 7, 7, 32) | output: (None, 14, 14, 16)
conv2d_transpose_4 | Conv2DTranspose | input: (None, 14, 14, 16) | output: (None, 28, 28, 8)
conv2d_transpose_5 | Conv2DTranspose | input: (None, 28, 28, 8) | output: (None, 28, 28, 1)

So how does the Decoder produce an image with the chosen label ?

During training it learns

- the "meaning" of the label part of the elongated latent
- by observing a larger representation loss
    - when it creates an output that doesn't match the label

That is, consider training example i with label C

- the Decoder reconstruction loss is large if its output is very different than the input
- So the Decoder learns (i.e., its weights associate) a relationship between
    - the OHE of the label
    - the desired output

This is the mantra of Deep Learning

- you don't guide or program the model with specific instructions on **how** to achieve a task
- it **learns** the association between input and output
    - through Loss minimization

```
In [2]: print("Done")
```

```
Done
```