The Autoencoder: Code

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

- derived from the Tensorflow tutorial (https://www.tensorflow.org/tutorials/generative/autoencoder)

Deriving a new Model via sub-classing

A Model object in Keras provides a consistent API to all sorts of models.

This consistency makes is easier to deal to build, train, and use Neural Networks.

For example, all Models provide methods

- for `fit` and `predict`
- as well as saving their architecture, weights, and training state
    - can re-use a pre-trained model

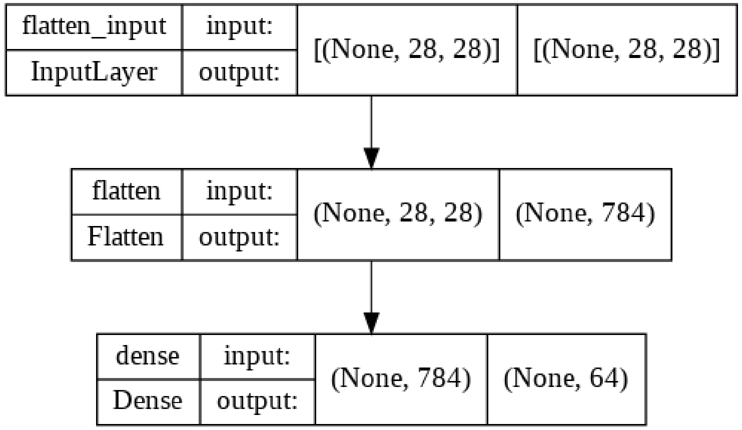
In our notebook we can [see a Basic Autoencoder (autoencoder.ipynb#First-example:-Basic-autoencoder)](autoencoder.ipynb#First-example:-Basic-autoencoder) implemented as a sub-class of `Model` :

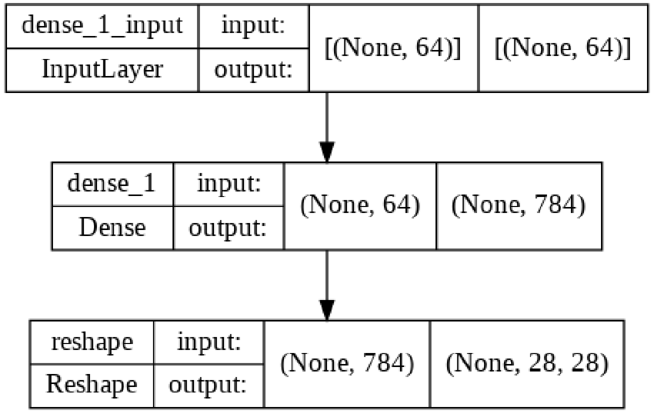
```
latent_dim = 64

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='sigmoid'),
      layers.Reshape((28, 28))
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded
```

This example implements a simple architecture for the Encoder and Decoder

- Encoder and Decoder don't need to be symmetric
- Can be more complex

Simple Autoencoder: Components





The Encoder (on the left)

- flattens the 2D input image (28×28)
- Uses a single `Dense` layer to create a latent vector of length `latent_dim`

The Decoder (on the right)

- takes a latent vector of length `latent_dim`
- Uses a single `Dense` layer to create a (flattened 2D) 1D vector
- Reshapes the 1D vector to a 2D image (28×28)

The `__init__` initialized method

- creates each Neural Network sub-component
- stores each as an attribute in the Model instance
    - `self.encoder`: `Sequential` Model for Encoder
    - `self.decoder`: `Sequential` Model for Decoder

So, an instance of an `Autoencoder` object *contains* two `Sequential` models

The heart of deriving a `Model` subclass is *overriding* the `call` method

- When actual parameters are applied (via the parentheses operator) to an instance of the `Model` object (e.g., `m`)

```
x = m(x)
```

- the `call` method is invoked

In the case of the `Autoencoder`

- the contained `encoder` and `decoder` models
- are retrievied
- and invoked

Just a reminder as to why the Neural Network sub-components are

- defined and saved in `__init__`
- rather than instantiated in `call`

By doing so in `__init__`

- the `Autoencdoer` object has a *single* instance of each sub-component
    - so the weights are preserved across `calls`
    - for example: across mini-batches
- had the objects been created in `call`
    - they (and their weights) would disappear after the call ended

We can instantiate an instance of the `Autoencoder Model` object

```
autoencoder = Autoencoder(latent_dim)
```

and then train it just like any other `Model` (e.g., `Sequential`)

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

autoencoder.fit(x_train, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test, x_test))
```

We can also apply all other `Model` methods like plotting its architecture

```
plot_model(autoencoder.encoder,
           to_file=os.path.join(tempdir, "autoencoder_simple_encoder.png"),
           show_shapes=True)
```

and saving and restoring a trained model.

Here, we reference the contained sub-models (`encoder` , `decoder`)

- and save them separately
- as typically, they are used separately **after** training
    - `encoder` is often used to create alternate (reduced dimension) representation of input
    - `decoder` is often used to create synthetic examples (from latent "noise" input)

```
autoencoder.encoder.save(ae_encoder_dir)

autoencoder.decoder.save(ae_decoder_dir)
```

Exploring the latent space

Clustering

The notebook [continues (autoencoder.ipynb#Examine-the-latent-representations-of-the-test-dataset)](autoencoder.ipynb#Examine-the-latent-representations-of-the-test-dataset) with code to examine the "latent" space

- i.e., the output of the Encoder

For example

- we find the latent representation of each test example

```
encoded_imgs = autoencoder.encoder(x_test).numpy()
```

and [plot (autoencoder.ipynb#Project-the-high-dimensionality-latents-into-2D)](autoencoder.ipynb#Project-the-high-dimensionality-latents-into-2D) them to see whether the representations form clusters

Since we can't easily visualize higher dimensional plots

- we use Principal Components to for dimensionality reduction

```
pca = PCA_fit(encoded_imgs, n_components=10)
X_proj = pca.transform(encoded_imgs)
```

- and plot the first two components
- coloring each example according to its label (type of clothing)
    - do examples of the same clothing type cluster ?

Synthetic examples by altering a latent

Once we observe that images of the same clothing type cluster in latent space

- we might be able to create a new, synthetic image
- by perturbing the latent representation of an example image
- using the Decoder to translate the perturbed latent back into the space of Images

We run one experiment (autoencoder.ipynb#Explore-the-latents-in-a-small-radius-of-the-latent-of-a-single-input)

- where we add random noise to a latent and Decode the result

A second experiment (autoencoder.ipynb#Interpolate-between-the-latents-of-two-inputs) examines whether there is a smooth transition between images of different clothing types

- by interpolating (linear combination) of the latents of two Images

We run a [third experiment (autoencoder.ipynb#Examine-the-2D-projections-obtained-by-PCA-on-the-high-dimensionality-latents)](autoencoder.ipynb#Examine-the-2D-projections-obtained-by-PCA-on-the-high-dimensionality-latents)

- trying to visualize the top Components of the PCA
- an actual image is a linear combination of the components
    - property of PCA
    - do components have a "natural" interpretation ?
        - expresses some commonality across multiple examples
        - perhaps it expresses a "concept" ("has arms", "has legs")

Denoising Autoencoder

We can also learn about the [Denoising Autoencoder (autoencoder.ipynb#Second-example:-Image-denoising)](autoencoder.ipynb#Second-example:-Image-denoising).

- using a simple `Dense` layer for the sub-components
- using a more complex Convolutional (`Conv2d`) layer for the sub-components

Anomaly detection

We [show (autoencoder.ipynb#Third-example:-Anomaly-detection)](autoencoder.ipynb#Third-example:-Anomaly-detection) how to use an Autoencoder for Anomaly Detection.

The basic idea is that the reduced dimension "latent" space is a *bottle-neck*

- to minimize reconstruction error *over a wide variety of training examples*
- the latent representation must focus on *commonality*
    - properties that are *shared across several* training examples

By passing an example through the bottle-neck and reconstructing it (via the Decoder)

- we "strip away" the non-essential properties of the example
- Reconstruction error is the difference between the original and reconstructed output

The theory is that an example with a *large Reconstruction Error*

- is an *anomaly*
- because it has a large element that is *not common* to many examples

In the example: the anomaly corresponds to an abnormal heart rhythm.

In [2]:

```python
print("Done")
```

Done