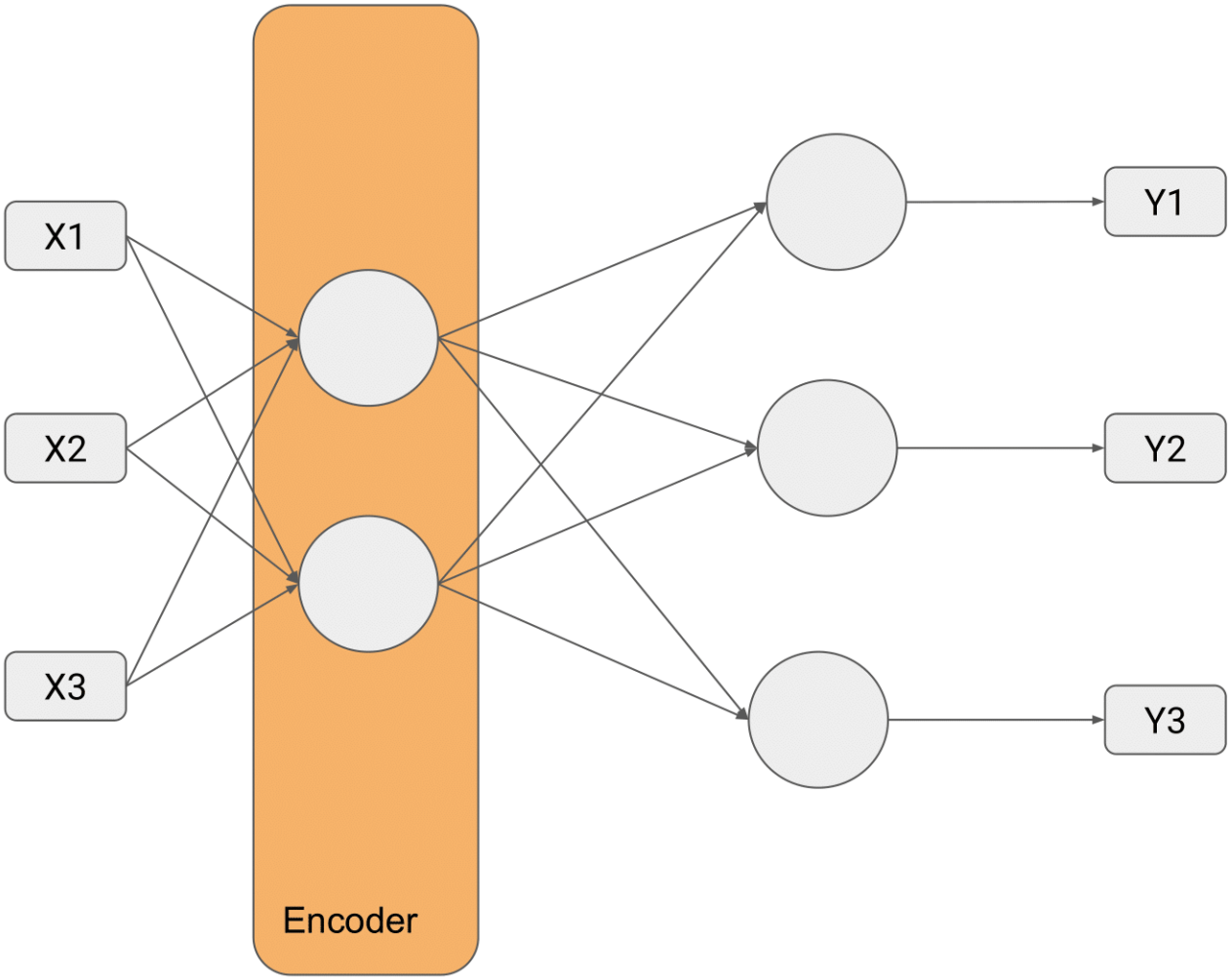
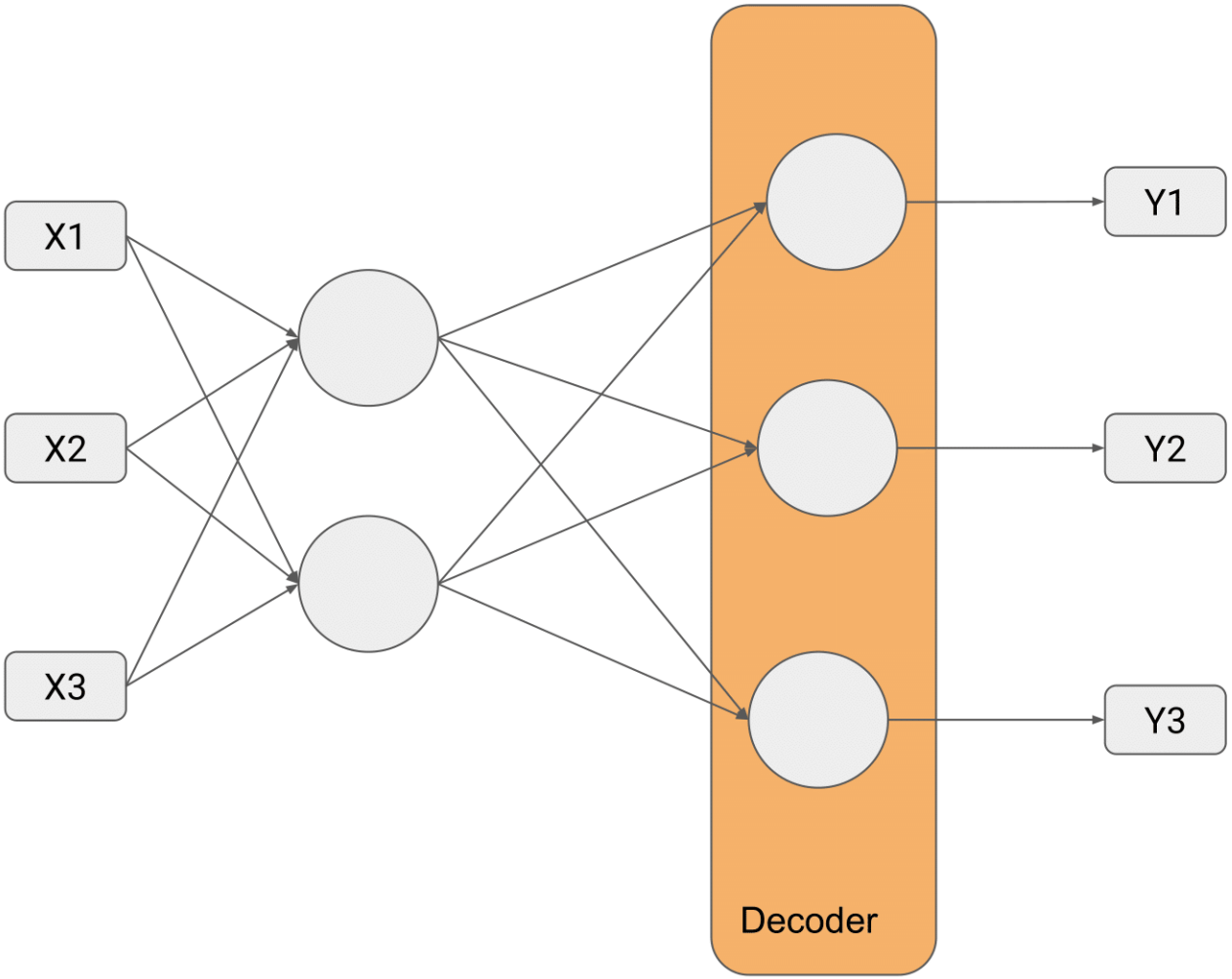
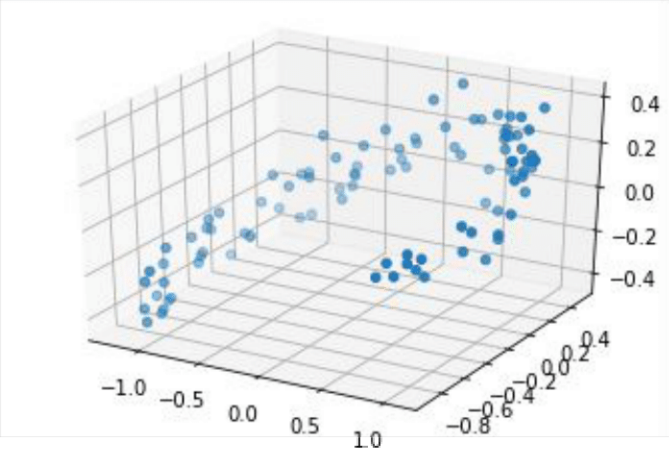
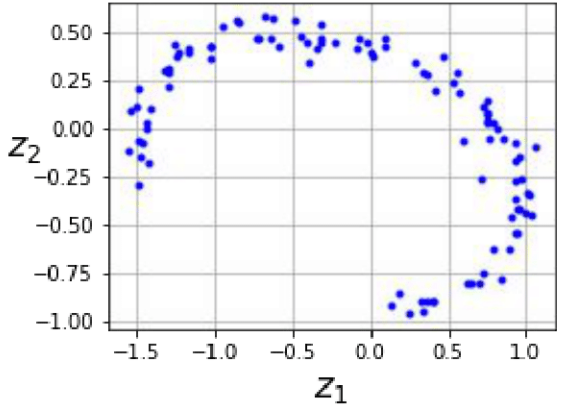
What are AutoEncoders?

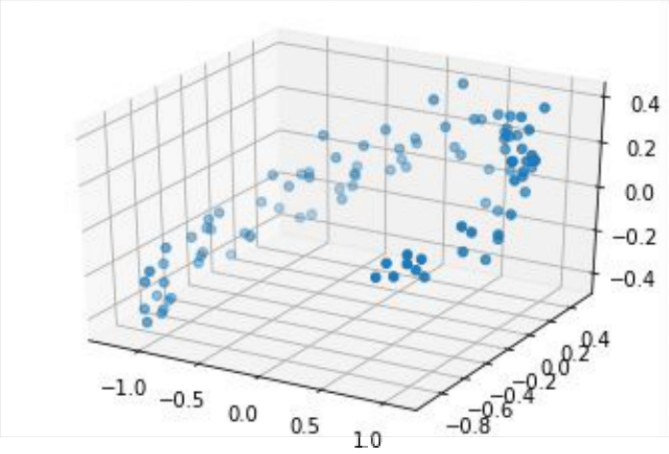
- Neural networks capable of learning dense representations of input data without supervision
  - Training data is not labelled
- Useful for dimensionality reduction and for visualization
- Can be used to generate new data that resembles input data
- In practice they
  - Copy input to output
  - They learn efficient ways to represent data

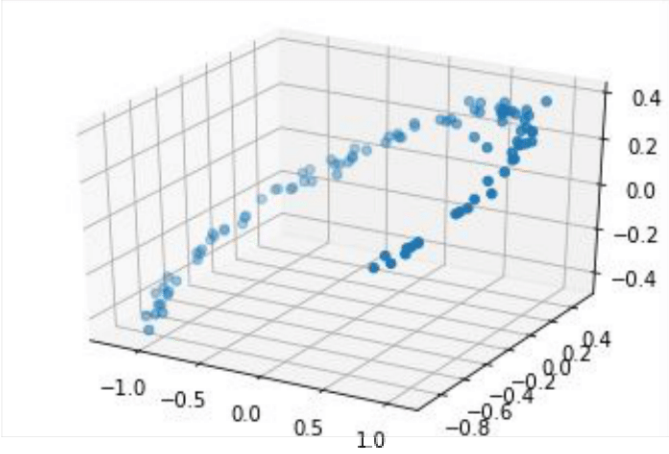

X1
X2
X3
Encoder
Y1
Y2
Y3


X1
X2
X3
Y1
Y2
Y3
Decoder

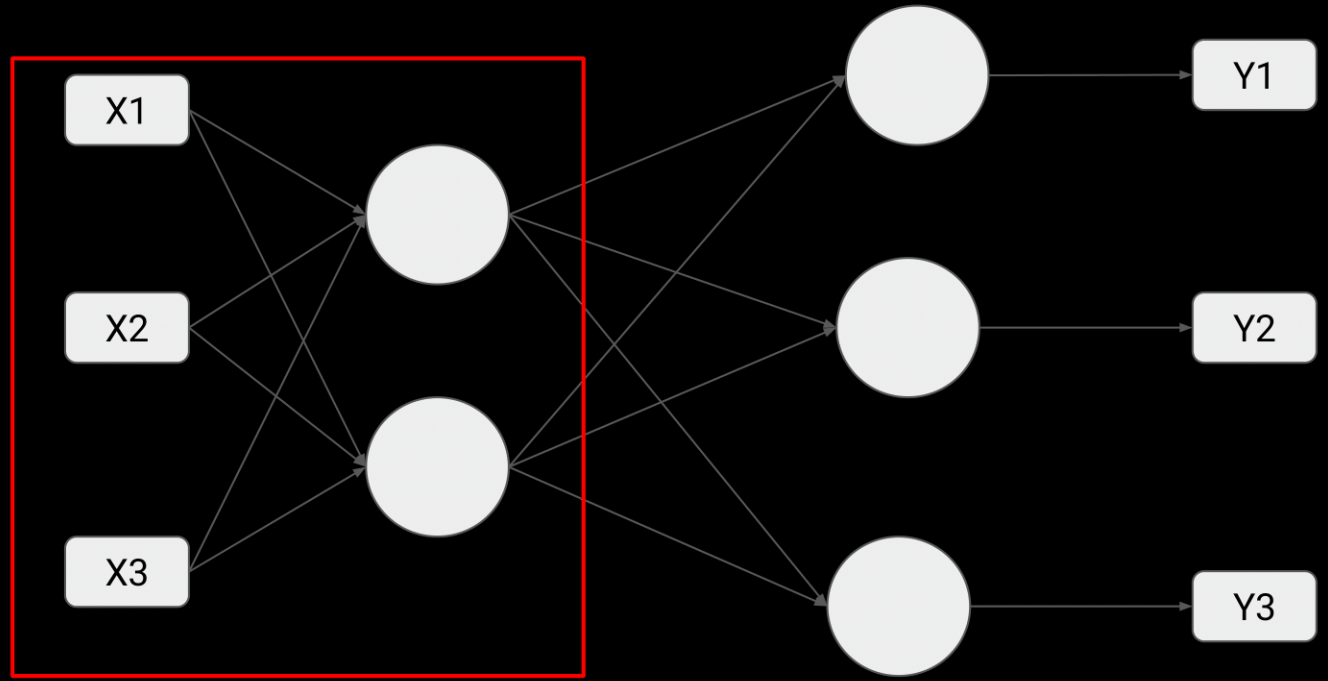

0.4
0.2
0.0
−0.2
−0.4
−1.0
−0.5
0.0
0.5
1.0
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8


0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
z_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
z_1

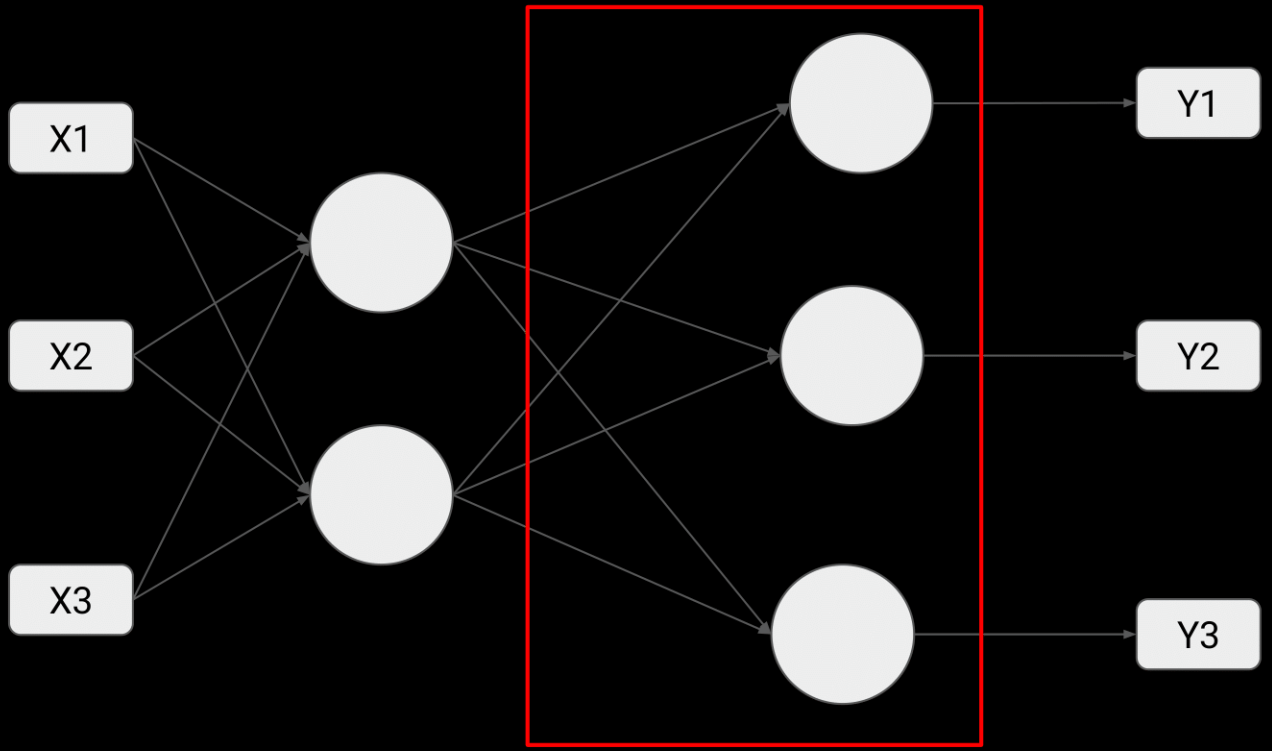

0.4
0.2
0.0
−0.2
−0.4
−1.0
−0.5
0.0
0.5
1.0
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8


0.4
0.2
0.0
−0.2
−0.4
−1.0
−0.5
0.0
0.5
1.0
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8

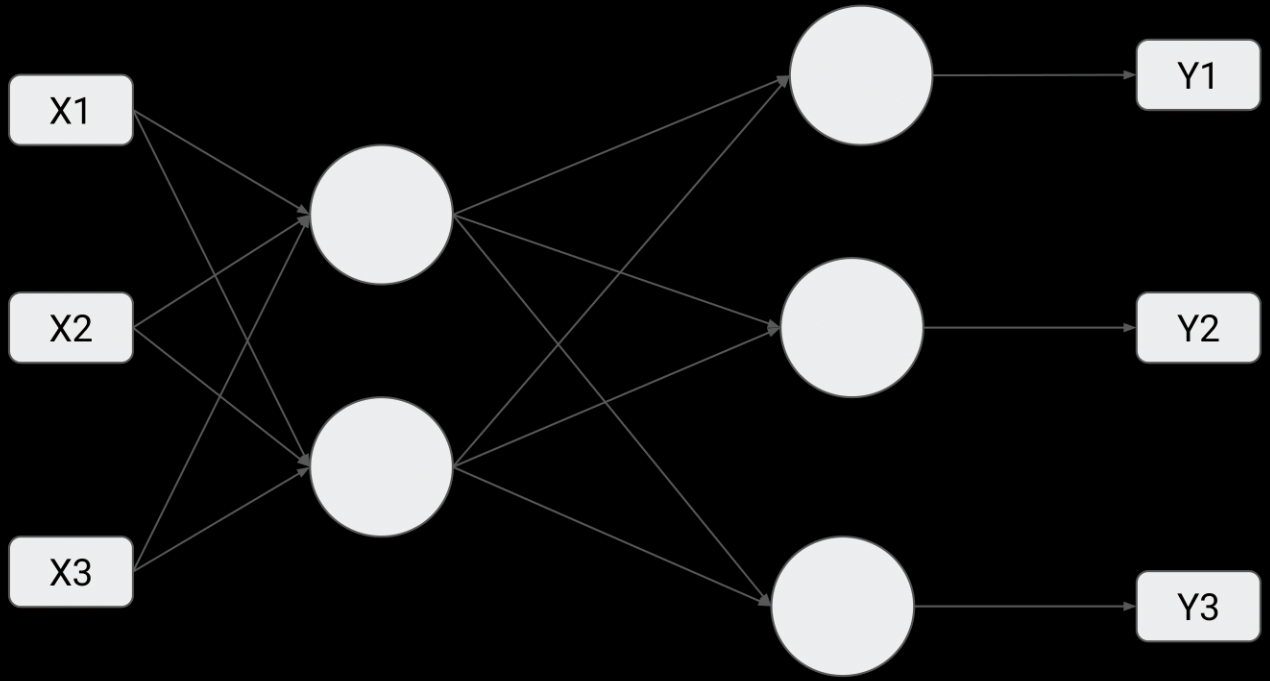
```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])

decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])

autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1.5))
```



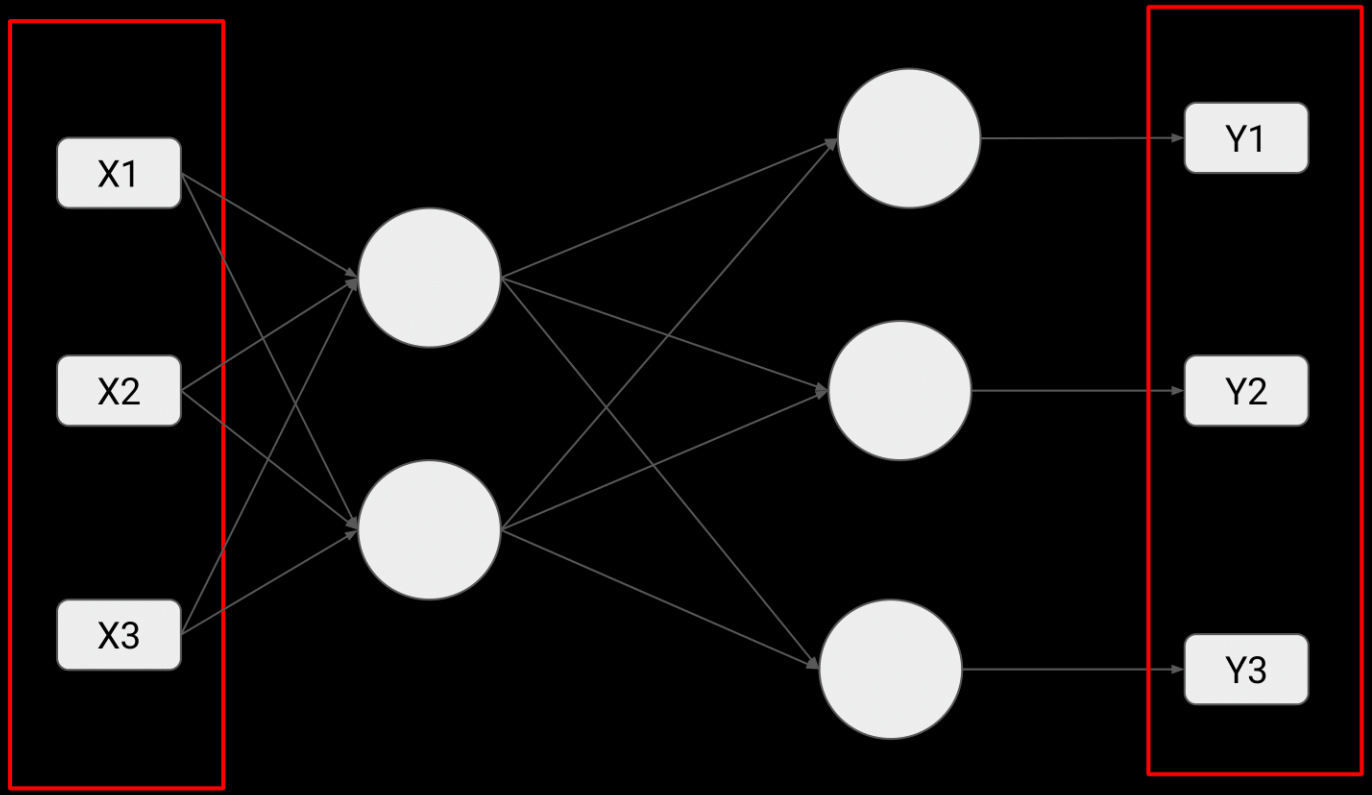
```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])
autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1.5))
```



```
encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])

decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])

autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1.5))
```

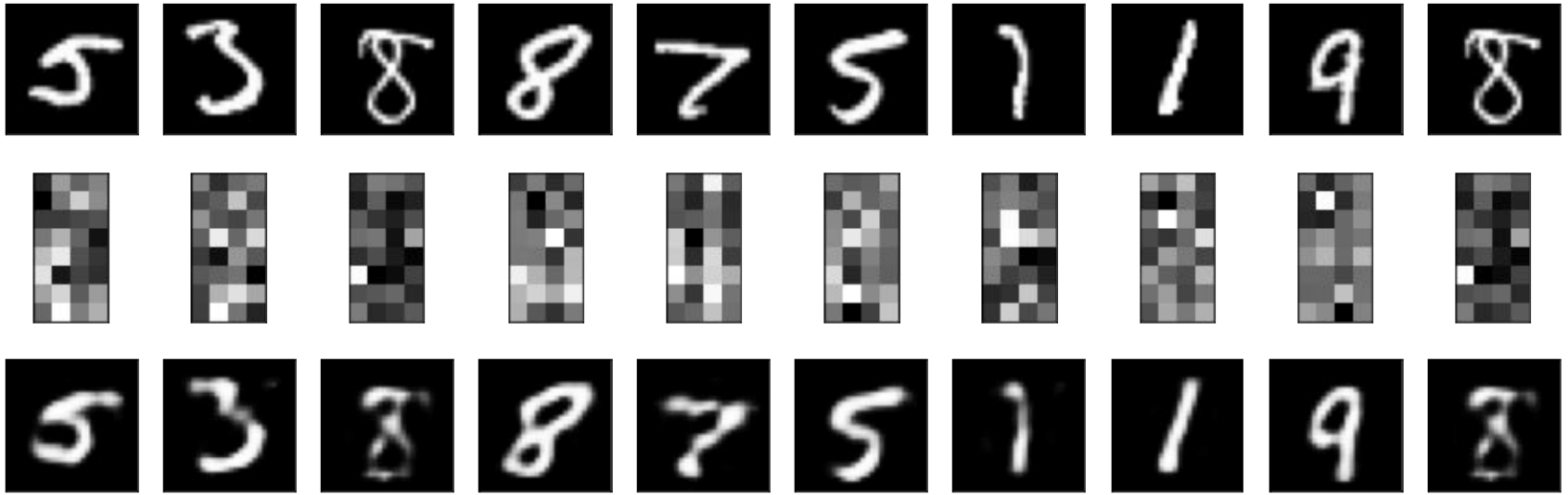


```
history = autoencoder.fit(X_train, X_train, epochs=200)
```



```
codings = encoder.predict(data)
```

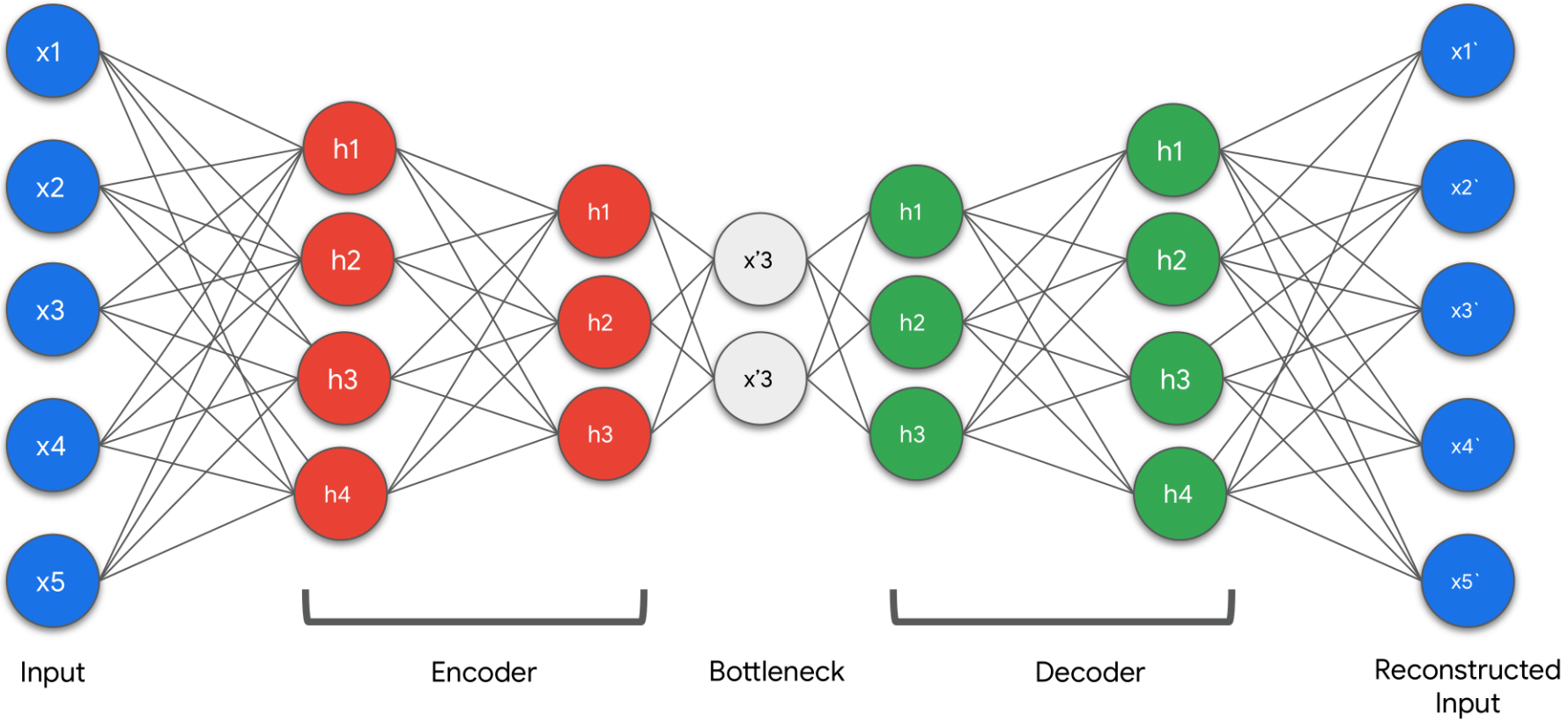
```
decodings = decoder.predict(codings)
```



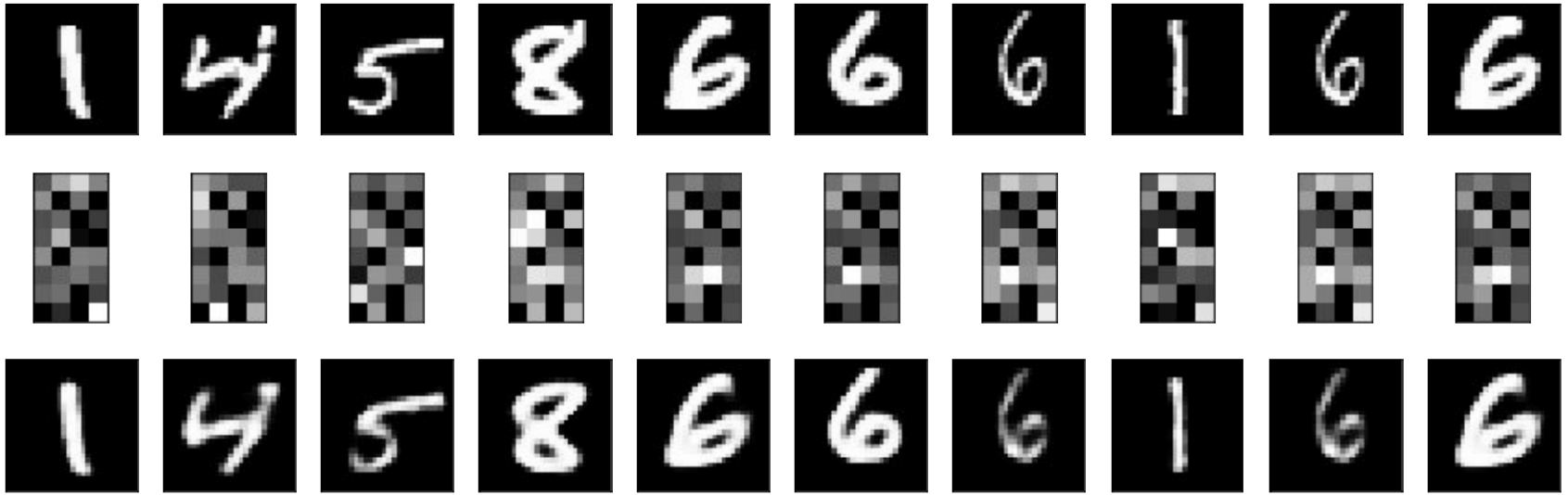
```
inputs =  tf.keras.layers.Input(shape=(784,))

def simple_autoencoder():
    encoder = tf.keras.layers.Dense(units=32, activation='relu')(inputs)
    decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(encoder)
    return encoder, decoder

encoder_output, decoder_output = simple_autoencoder()

encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_output)

autoencoder_model = tf.keras.Model(inputs=inputs, outputs=decoder_output)
```

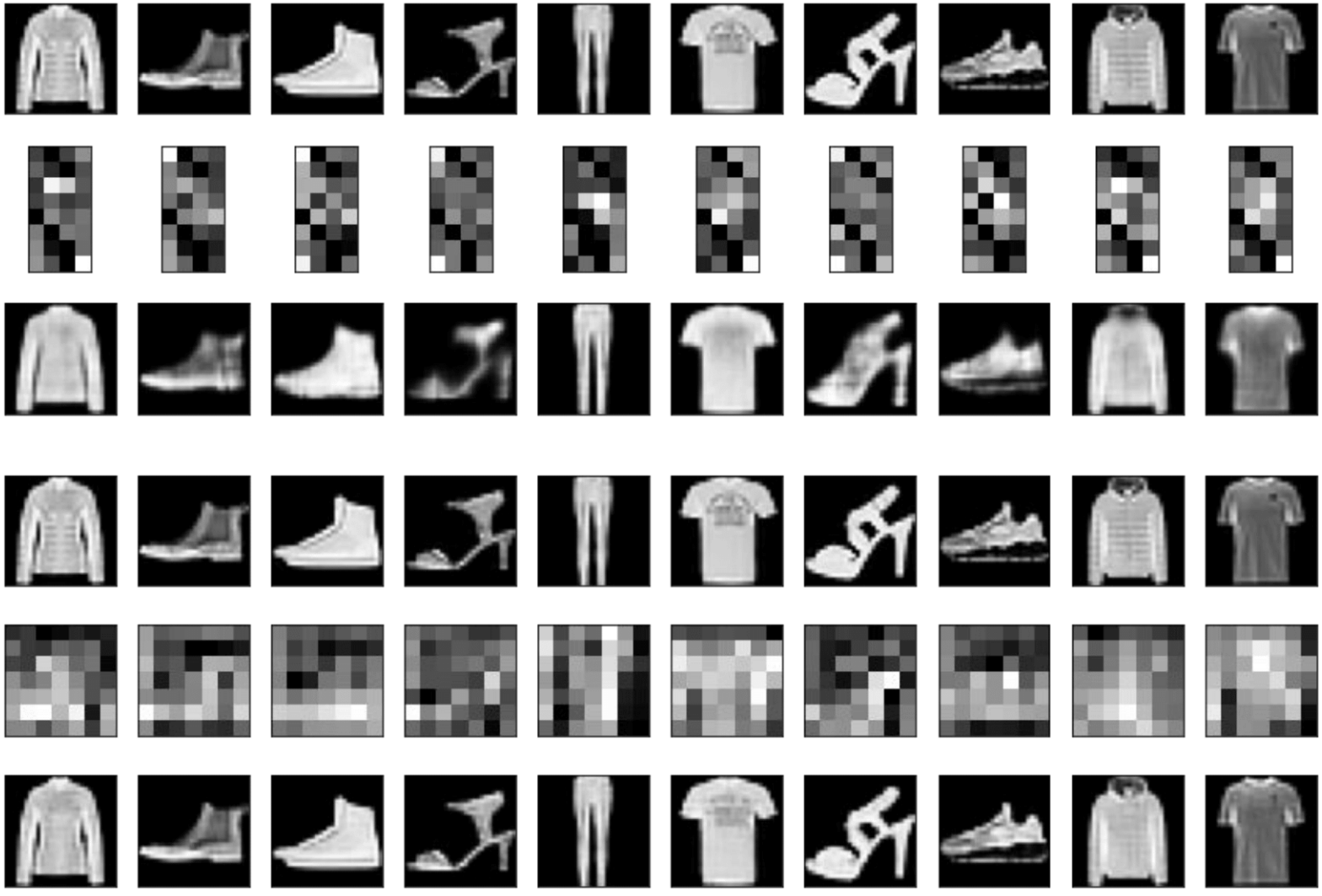
```
autoencoder_model.compile(
        optimizer=tf.keras.optimizers.Adam(),
        loss='binary_crossentropy')
```

Stacked Auto-Encoders

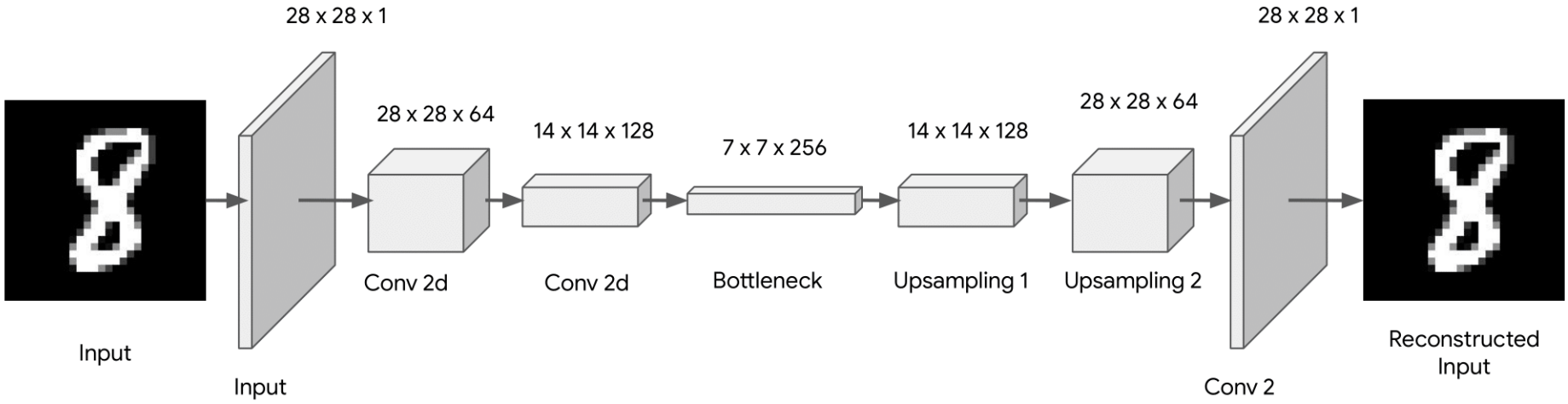


```
inputs =  tf.keras.layers.Input(shape=(784,))

def deep_autoencoder():
 encoder = tf.keras.layers.Dense(units=128, activation='relu')(inputs)
 encoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
 encoder = tf.keras.layers.Dense(units=32, activation='relu')(encoder)

 decoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
 decoder = tf.keras.layers.Dense(units=128, activation='relu')(decoder)
 decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(decoder)

 return encoder, decoder

deep_encoder_output, deep_autoencoder_output = deep_autoencoder()

deep_encoder_model = tf.keras.Model(inputs=inputs, outputs=deep_encoder_output)
deep_autoencoder_model = tf.keras.Model(inputs=inputs, outputs=deep_autoencoder_output)
```





Convolutional Auto-Encoders



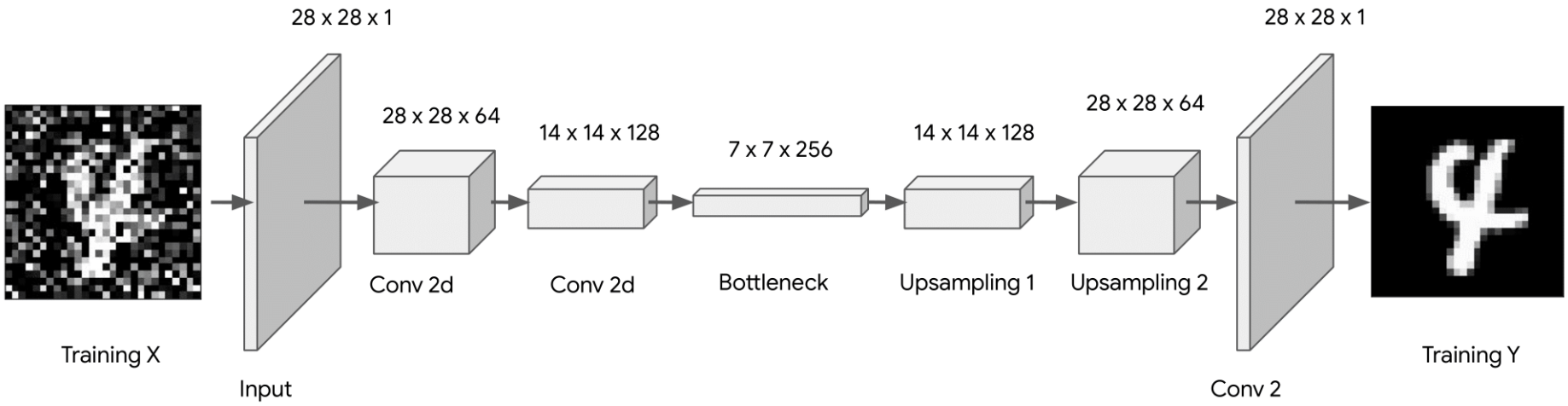
```
def encoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)

  max_pool_1 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(max_pool_1)

  max_pool_2 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_2)

  return max_pool_2
```

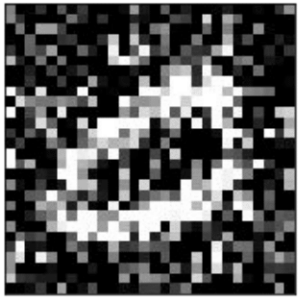
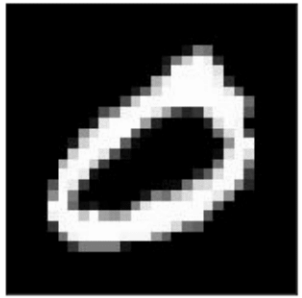
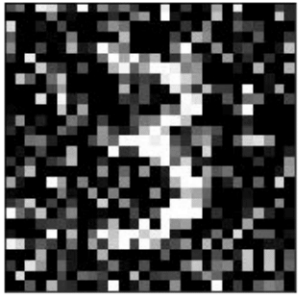
```
def bottle_neck(inputs):
  bottle_neck = tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3),
                                       activation='relu', padding='same')(inputs)



  encoder_visualization = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                                 activation='sigmoid',
                                                 padding='same')(bottle_neck)

  return bottle_neck, encoder_visualization
```

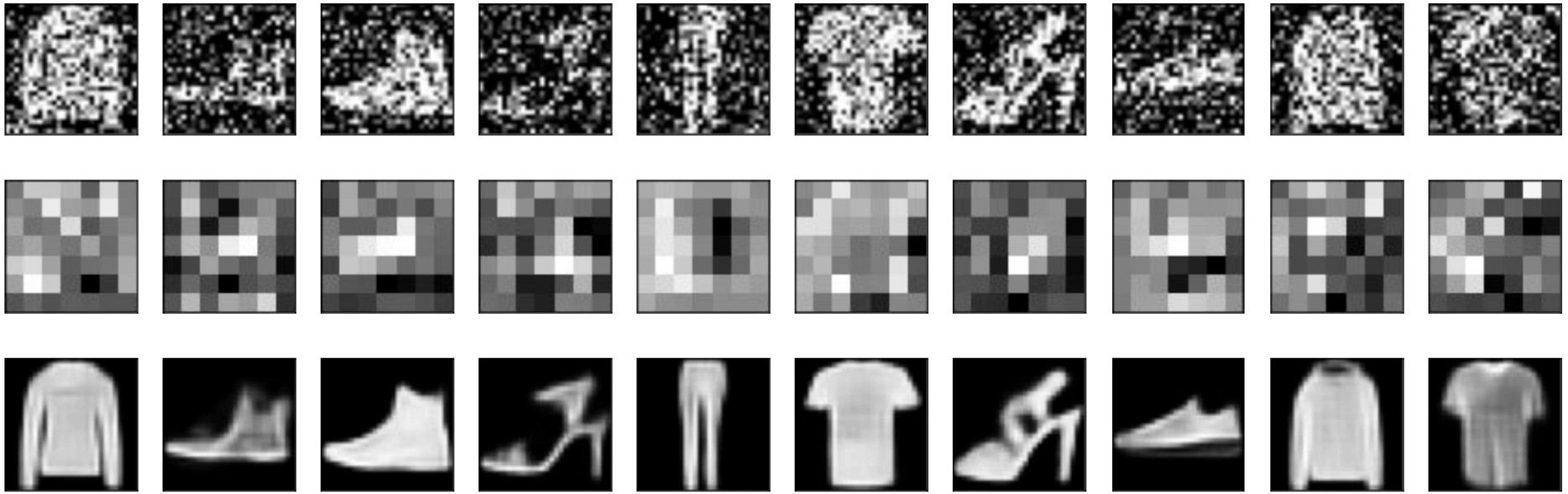
```
def decoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)
  up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(up_sample_1)
  up_sample_2 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_2)

  conv_3 = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                  activation='sigmoid',
                                  padding='same')(up_sample_2)

  return conv_3
```

```
def convolutional_auto_encoder():
  inputs = tf.keras.layers.Input(shape=(28, 28, 1,))

  encoder_output = encoder(inputs)

  bottleneck_output, encoder_visualization = bottle_neck(encoder_output)

  decoder_output = decoder(bottleneck_output)

  model = tf.keras.Model(inputs =inputs, outputs=decoder_output)
  encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_visualization)

  return model, encoder_model
```

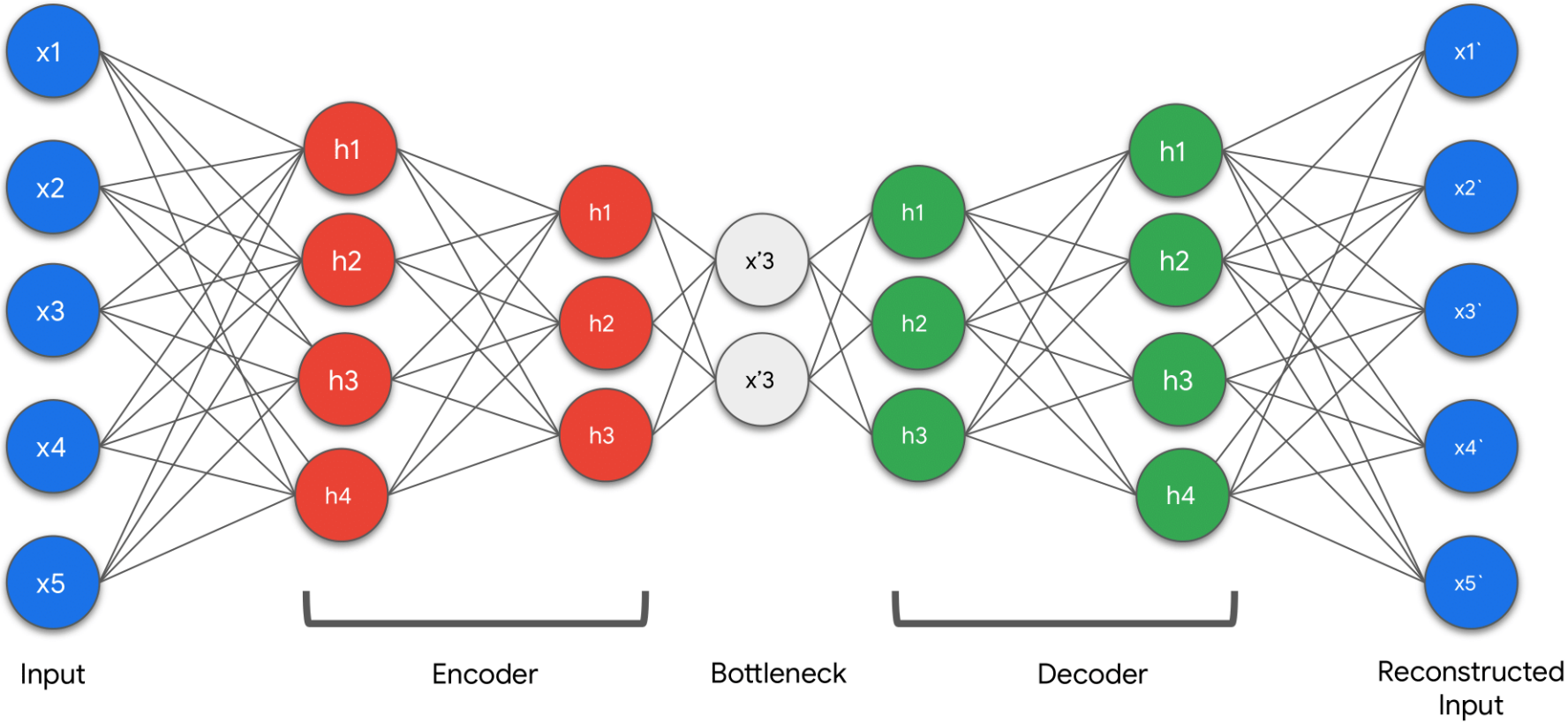
Convolutional Auto-Encoders

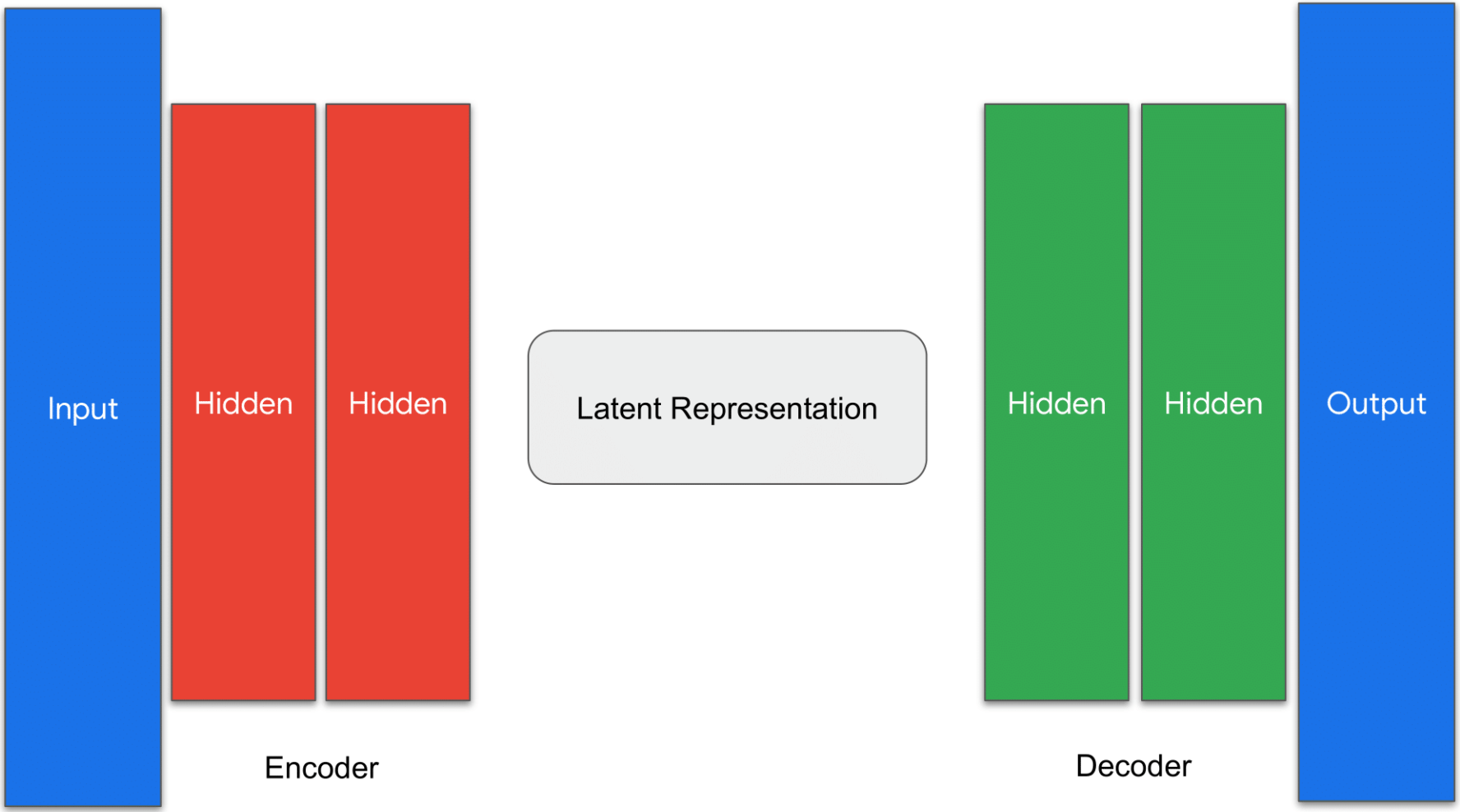


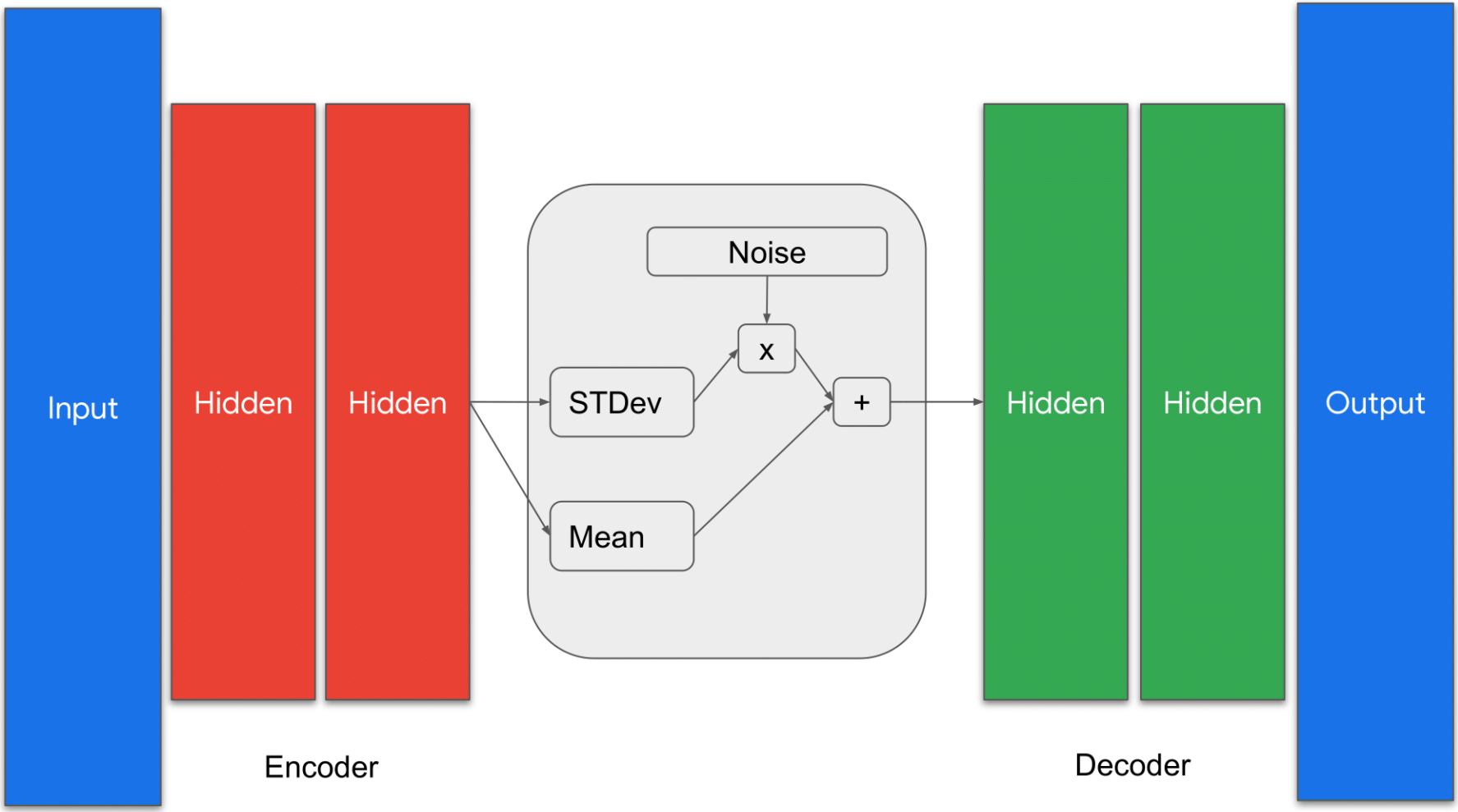




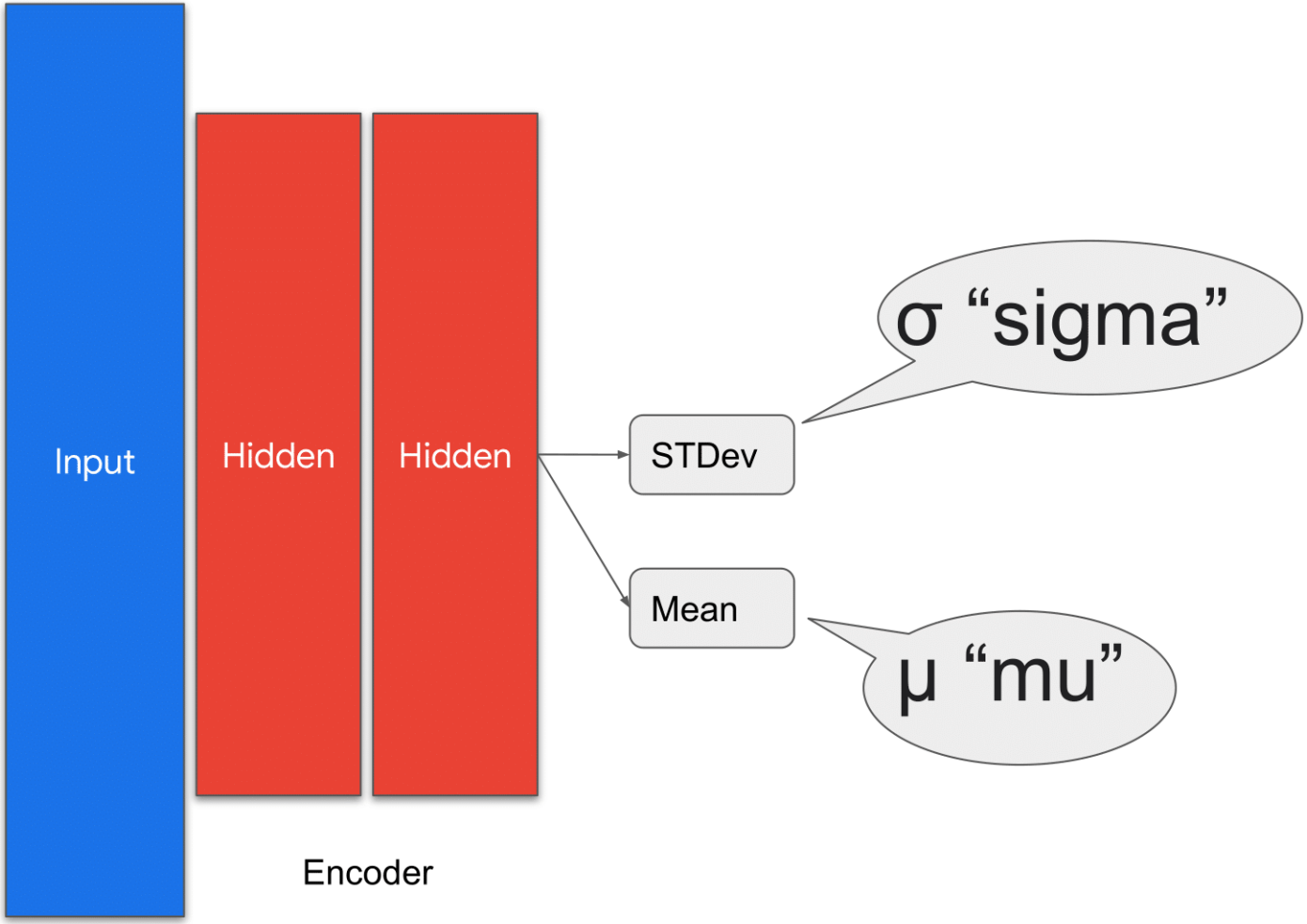



```
def map_image_with_noise(image, label):
  noise_factor = 0.5
  image = tf.cast(image, dtype=tf.float32)
  image = image / 255.0

  factor = noise_factor * tf.random.normal(shape=image.shape)
  image_noisy = image + factor
  image_noisy = tf.clip_by_value(image_noisy, 0.0, 1.0)

  return image_noisy, image
```


x1
x2
x3
x4
x5
h1
h2
h3
h4
h1
h2
h3
x'3
x'3
h1
h2
h3
h1
h2
h3
h4
x1`
x2`
x3`
x4`
x5`
Input
Encoder
Bottleneck
Decoder
Reconstructed Input


Input
Hidden
Hidden
Latent Representation
Hidden
Hidden
Output
Encoder
Decoder


Input
Hidden
Hidden
Noise
x
STDev
+
Mean
Hidden
Hidden
Output
Encoder
Decoder


Input
Hidden
Hidden
STDev
Mean
σ “sigma”
μ “mu”
Encoder

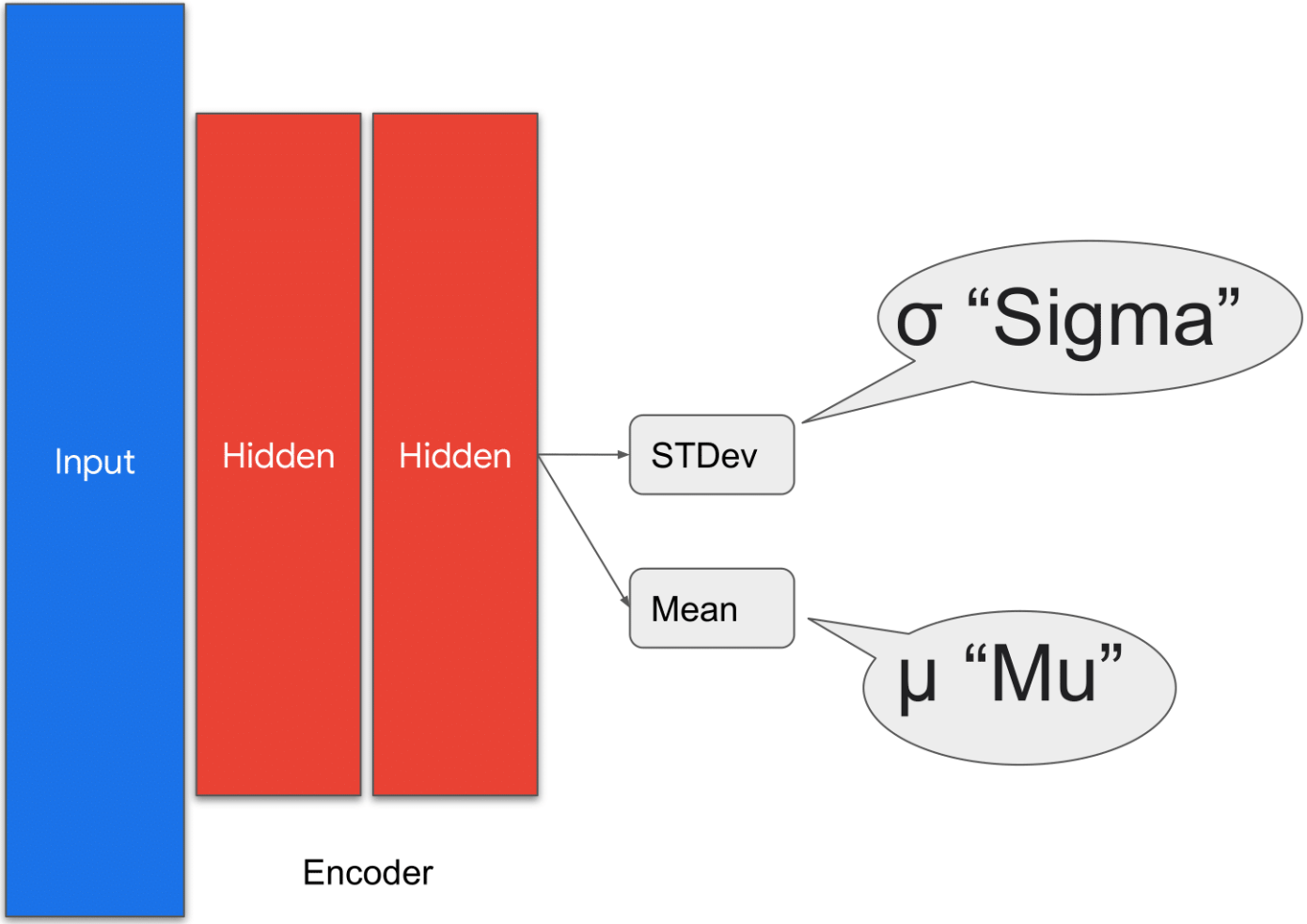
Probability Distribution

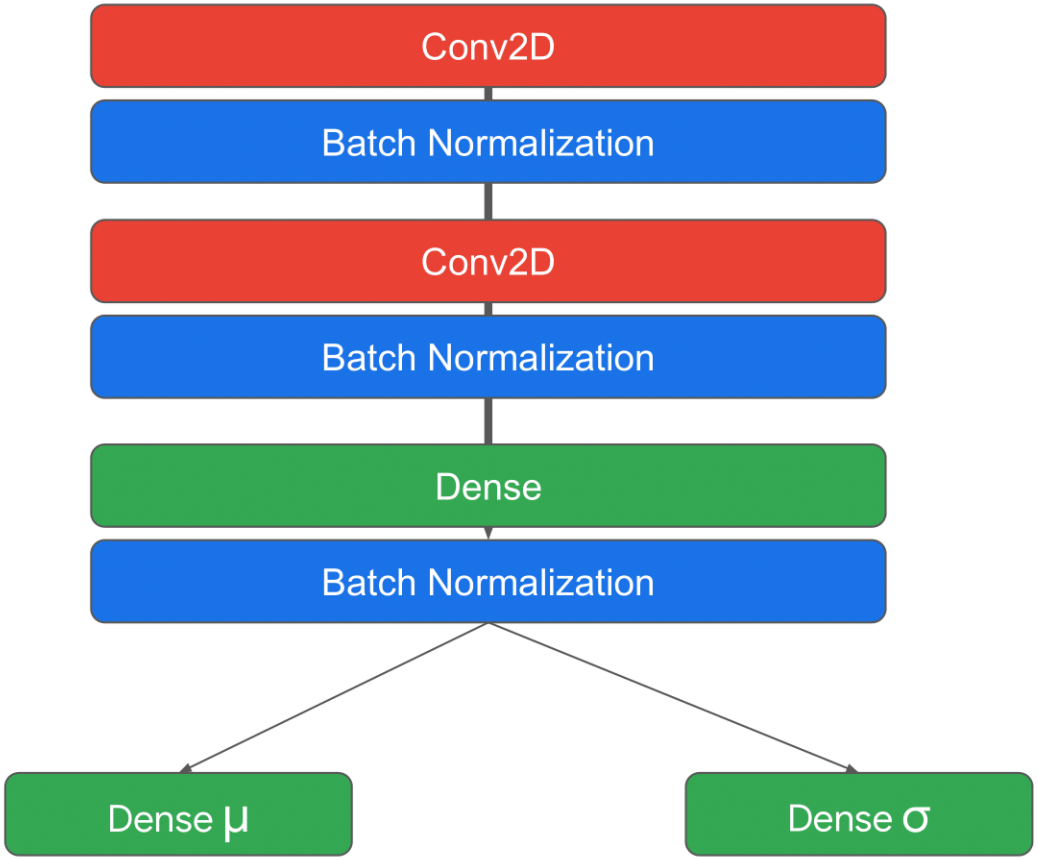
Gaussian probability density function or **Normal Distribution.**

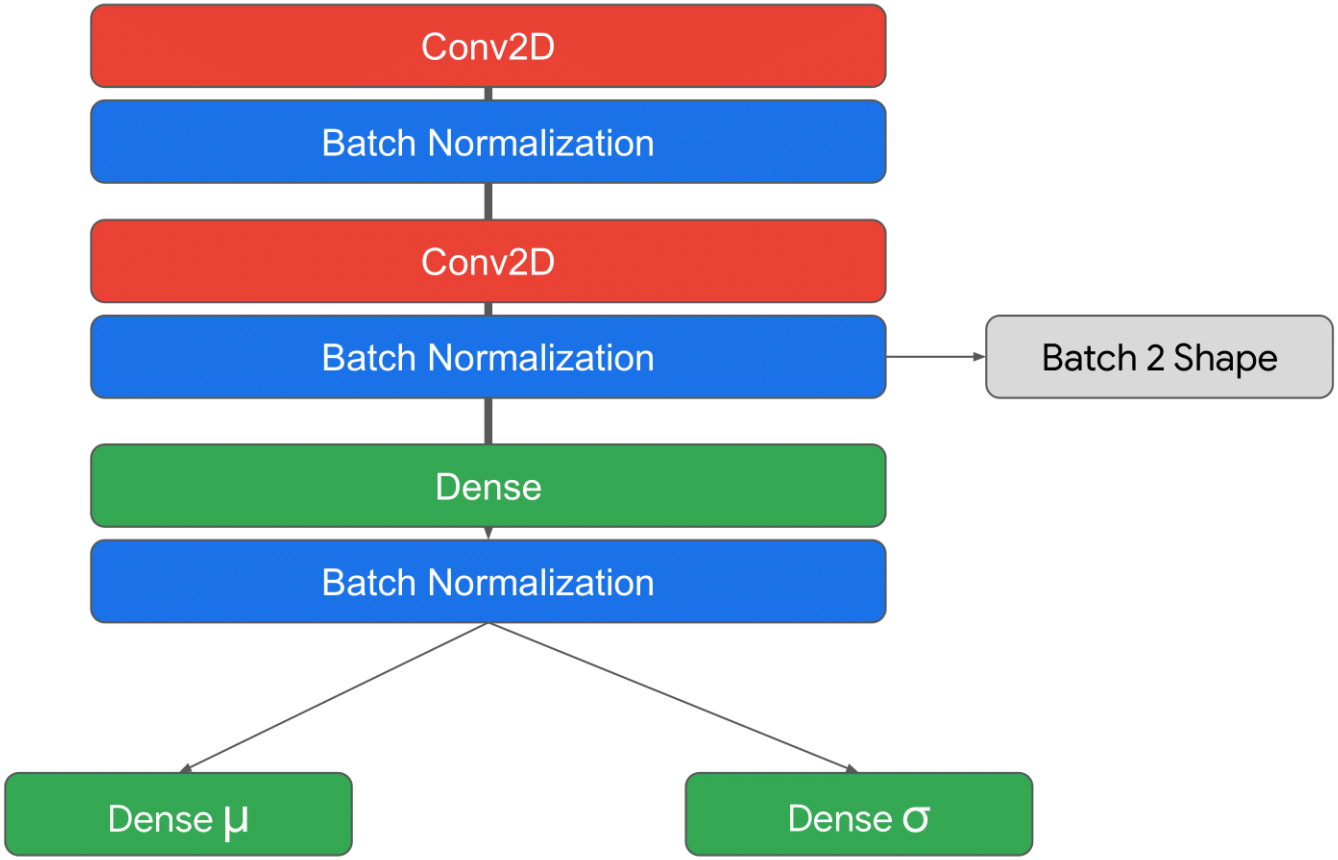
Normal Distribution is controlled by:

- **μ “mean”**
- **σ “standard deviation”**

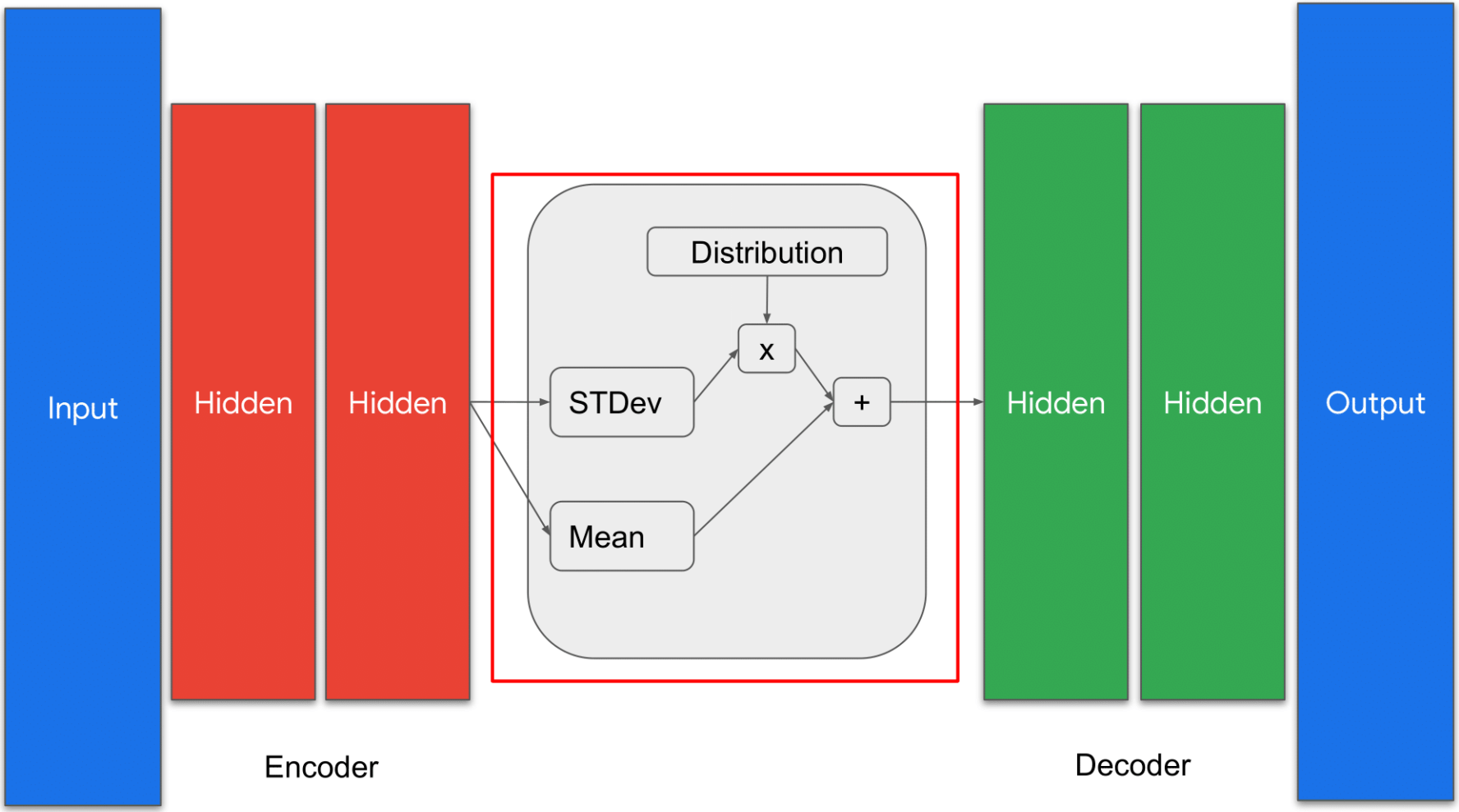
$$N(\mu, \sigma)$$

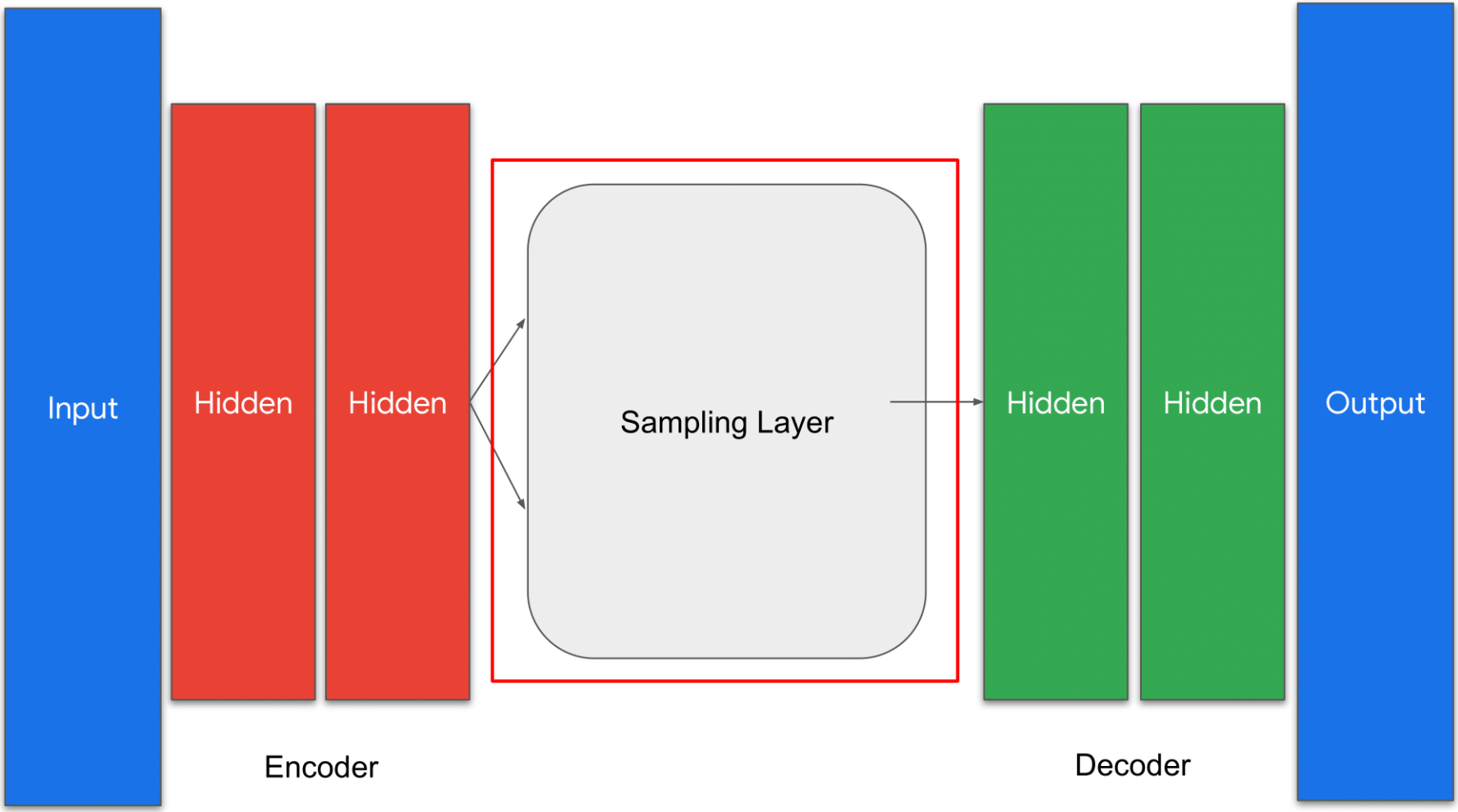

Input
Hidden
Hidden
STDev
Mean
σ “Sigma”
μ “Mu”
Encoder


Conv2D
Batch Normalization
Conv2D
Batch Normalization
Dense
Batch Normalization
Dense μ
Dense σ

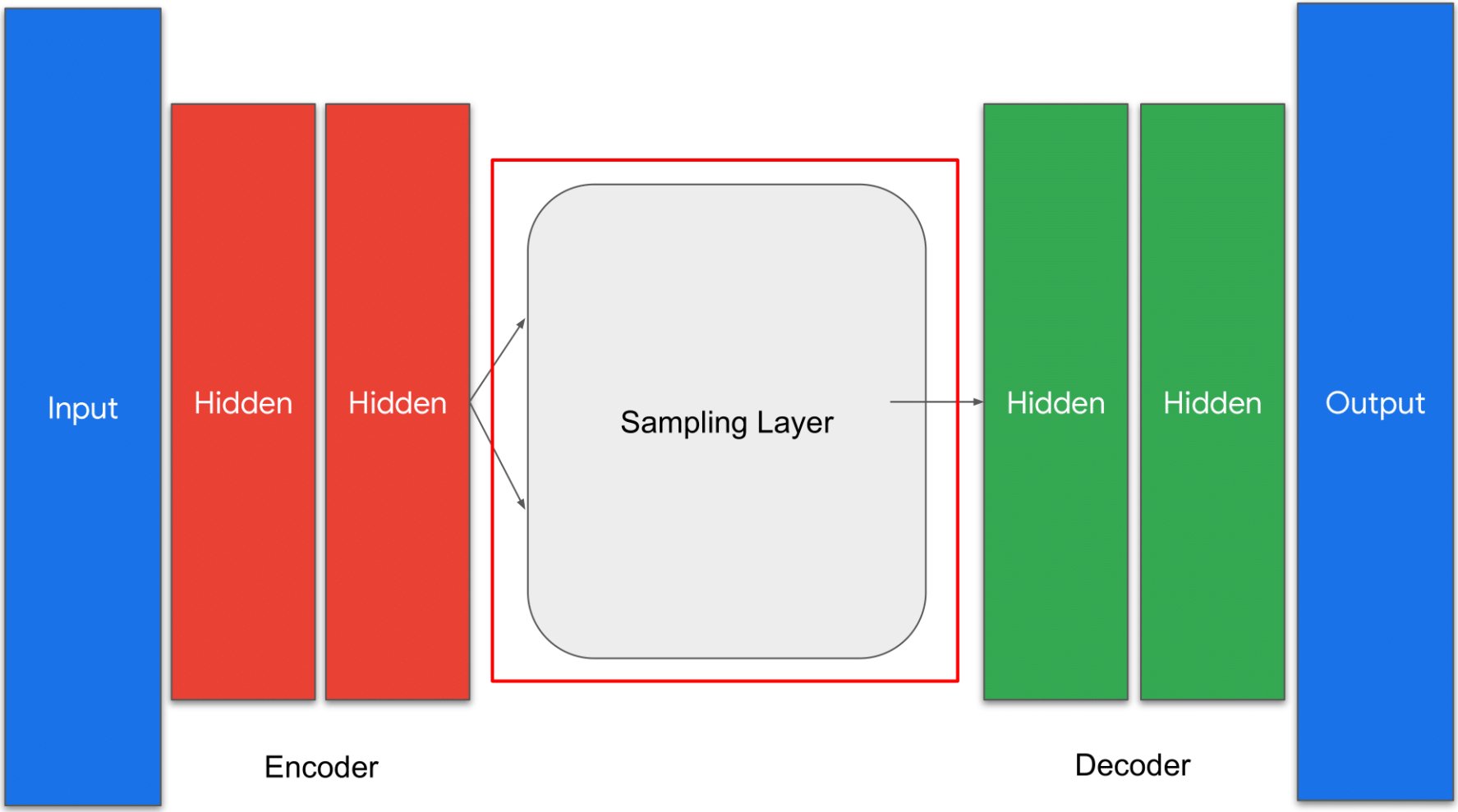

Conv2D
Batch Normalization
Conv2D
Batch Normalization
Batch 2 Shape
Dense
Batch Normalization
Dense μ
Dense σ

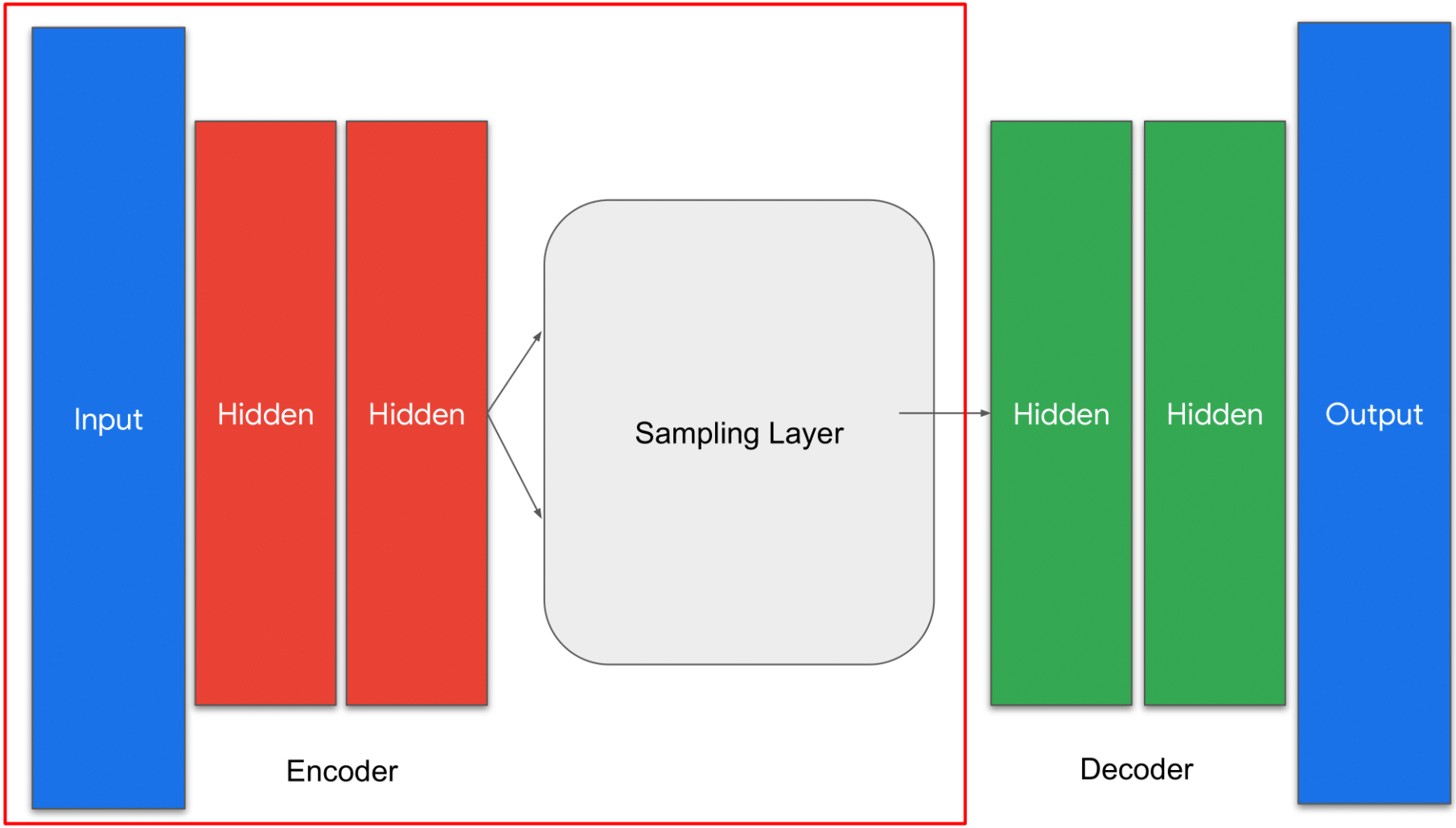
```python
# This function defines the encoder's layers
def encoder_layers(inputs, latent_dim):
  x = tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=2,
                             padding="same", activation='relu',
                             name="encode_conv1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=2,
                             padding='same', activation='relu',
                             name="encode_conv2")(x)
  batch_2 = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Flatten(name="encode_flatten")(batch_2)
  x = tf.keras.layers.Dense(20, activation='relu', name="encode_dense")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  mu = tf.keras.layers.Dense(latent_dim, name='latent_mu')(x)
  sigma = tf.keras.layers.Dense(latent_dim, name ='latent_sigma')(x)


  return mu, sigma, batch_2.shape
```


Input
Hidden
Hidden
Distribution
x
STDev
+
Mean
Hidden
Hidden
Output
Encoder
Decoder

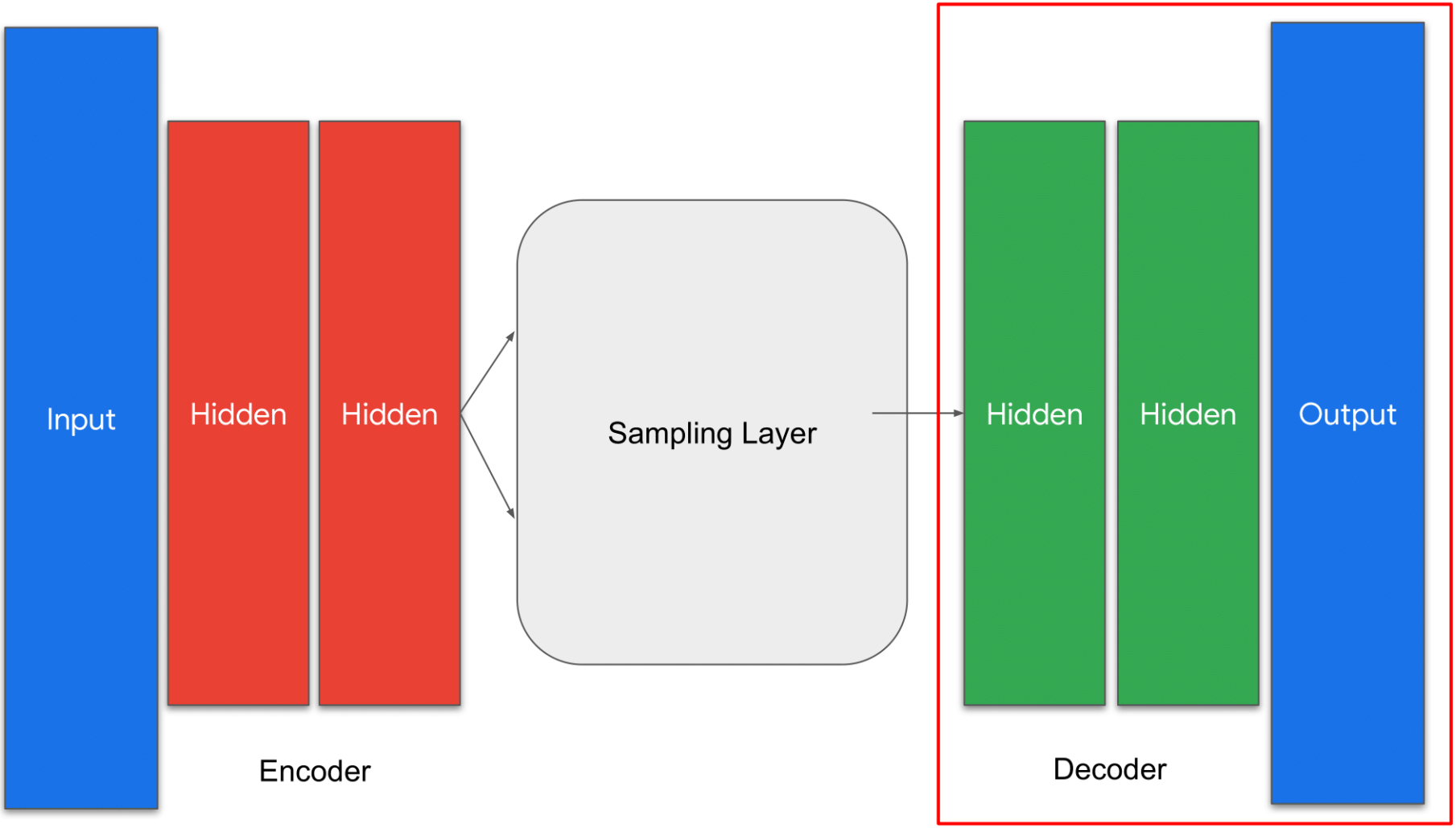

Input
Hidden
Hidden
Sampling Layer
Hidden
Hidden
Output
Encoder
Decoder

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


Input
Hidden
Hidden
Sampling Layer
Hidden
Hidden
Output
Encoder
Decoder

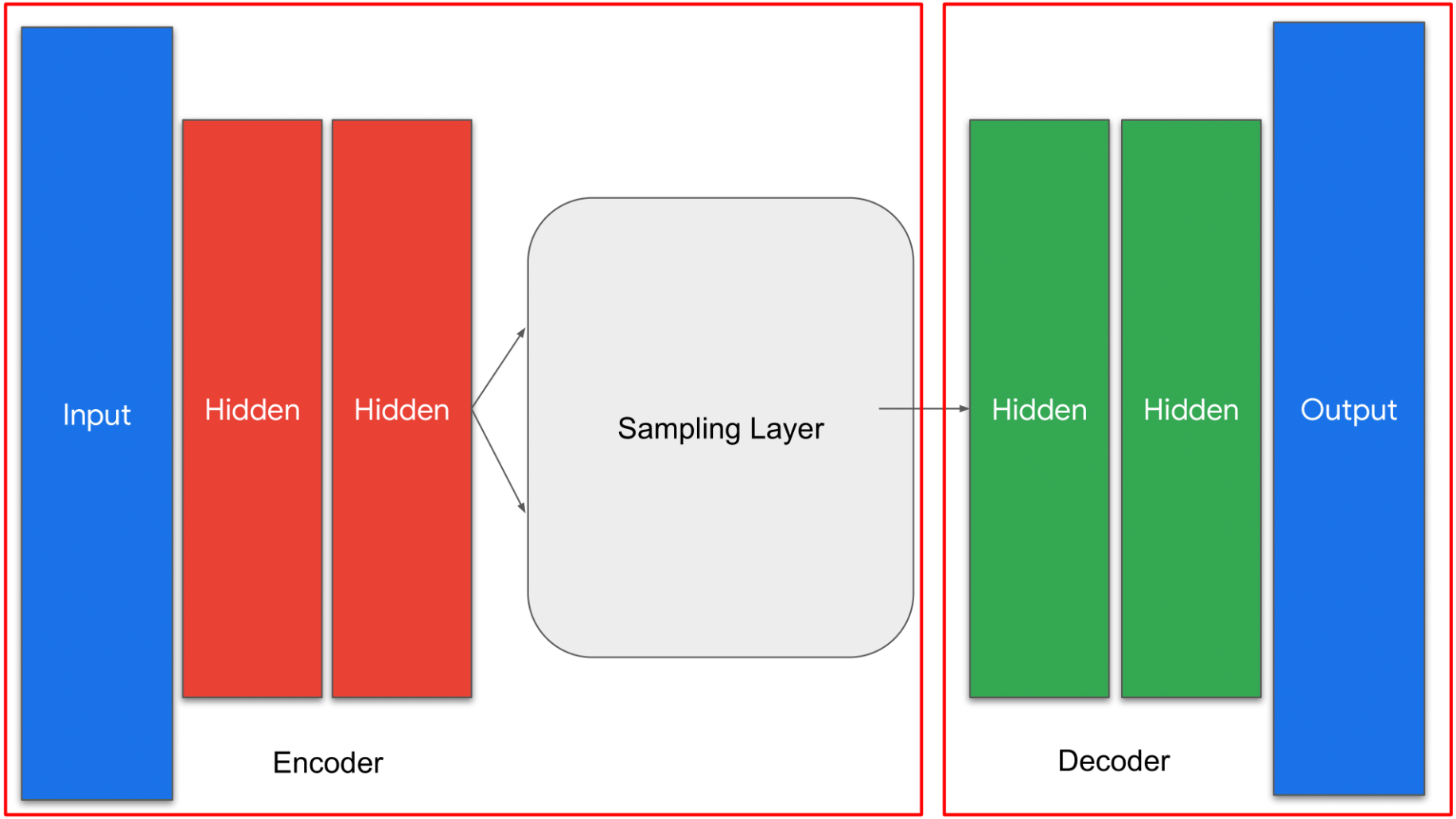

Input
Hidden
Hidden
Sampling Layer
Hidden
Hidden
Output
Encoder
Decoder

```
def encoder_model(LATENT_DIM, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu, sigma, conv_shape = encoder_layers(inputs, latent_dim=LATENT_DIM)
  z = Sampling()((mu, sigma))
  model = tf.keras.Model(inputs, outputs=[mu, sigma, z])
  return model, conv_shape
```


Input
Hidden
Hidden
Sampling Layer
Hidden
Hidden
Output
Encoder
Decoder

```
def decoder_layers(inputs, conv_shape):
  units = conv_shape[1] * conv_shape[2] * conv_shape[3]
  x = tf.keras.layers.Dense(units, activation = 'relu',
                            name="decode_dense1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Reshape((conv_shape[1], conv_shape[2], conv_shape[3]),
                              name="decode_reshape")(x)
  x = tf.keras.layers.Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d_2")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d3")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=1, kernel_size=3, strides=1, padding='same',
                                      activation='sigmoid', name="decode_final")(x)

  return x
```

```
def decoder_model(latent_dim, conv_shape):
  inputs = tf.keras.layers.Input(shape=(latent_dim,))
  outputs = decoder_layers(inputs, conv_shape)
  model = tf.keras.Model(inputs, outputs)
  return model
```


Input
Hidden
Hidden
Sampling Layer
Hidden
Hidden
Output
Encoder
Decoder

```python
# Define a kl reconstruction loss function
def kl_reconstruction_loss(inputs, outputs, mu, sigma):
  kl_loss = 1 + sigma - tf.square(mu) - tf.math.exp(sigma)
  return tf.reduce_mean(kl_loss) * -0.5
```

```
def vae_model(encoder, decoder, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu = encoder(inputs)[0]
  sigma = encoder(inputs)[1]
  z = encoder(inputs)[2]
  reconstructed = decoder(z)
  model = tf.keras.Model(inputs=inputs, outputs=reconstructed)
  loss = kl_reconstruction_loss(inputs, z, mu, sigma)
  model.add_loss(loss)
  return model
```

```
for epoch in range(epochs):
  for step, x_batch_train in enumerate(train_dataset):
    with tf.GradientTape() as tape:
      reconstructed = vae(x_batch_train)
      flattened_inputs = tf.reshape(x_batch_train, shape=[-1])
      flattened_outputs = tf.reshape(reconstructed, shape=[-1])
      loss = bce_loss(flattened_inputs, flattened_outputs) * 784
      loss += sum(vae.losses)  # Add KLD regularization loss

    grads = tape.gradient(loss, vae.trainable_weights)
    optimizer.apply_gradients(zip(grads, vae.trainable_weights))
```

epoch: 99, step: 400