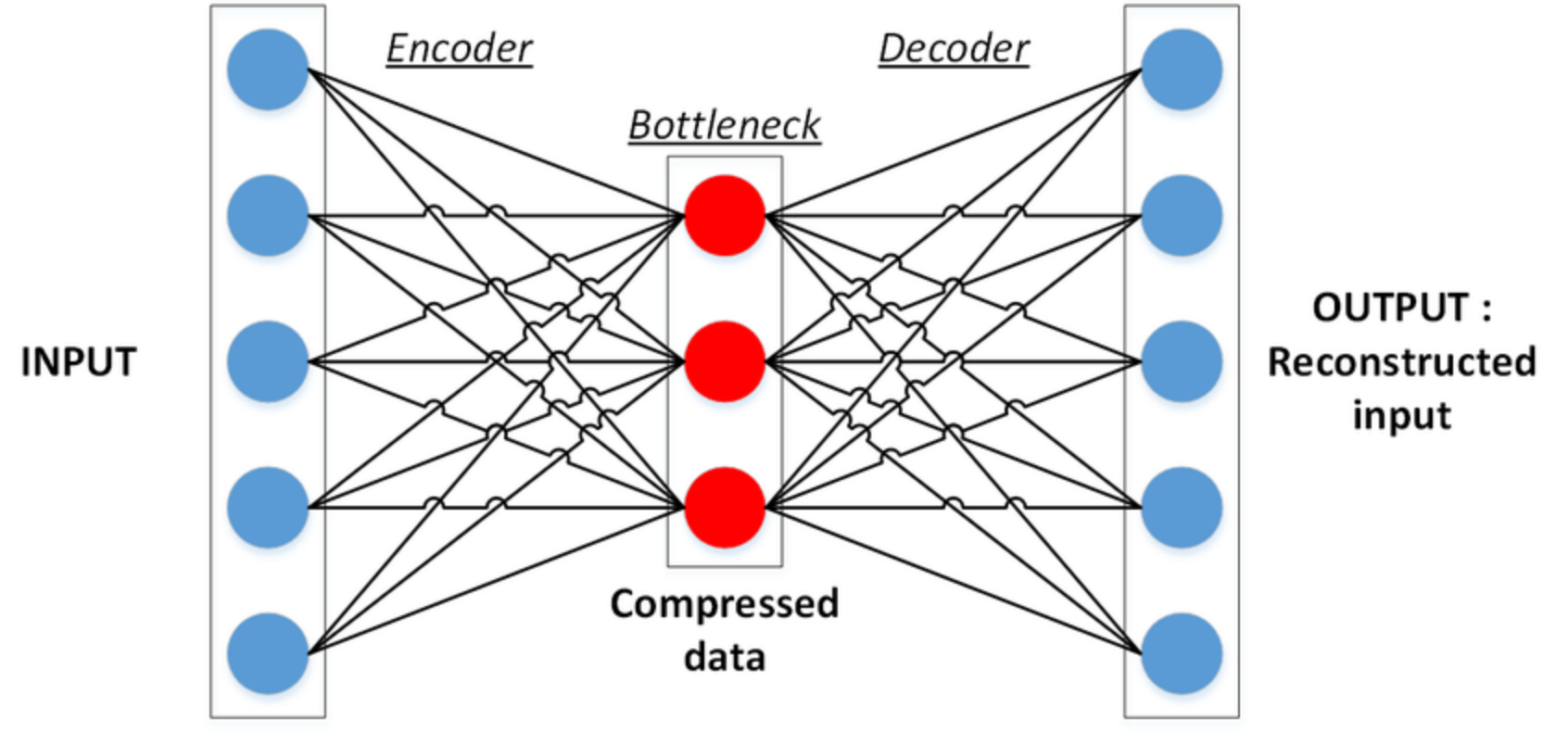
# Autocodificadores

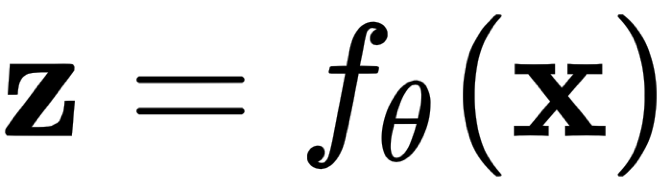
Un autocodificador es un tipo de FNN no supervisado que aprende a reconstruir datos de alta dimensión utilizando datos codificados latentes. Puede pensar en ello como intentar aprender una función de identidad (es decir, tomar x como entrada y luego predecir x).

Comencemos por echar un vistazo al siguiente diagrama, que muestra cómo se ve un codificador automático:

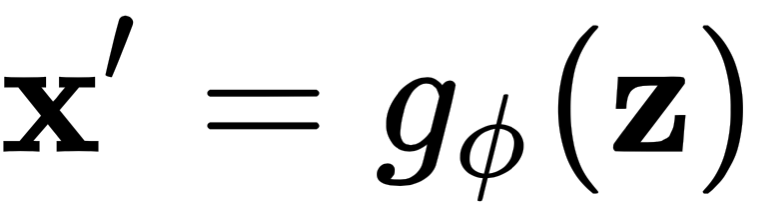


Como puede ver, la red está dividida en dos componentes, un **codificador** y un **descodificador**, que son imágenes especulares entre sí. Los dos componentes están conectados entre sí a través de una capa de **cuello de botella** (a veces denominada representación del **espacio latente o compresión**) que tiene *dimensiones mucho más pequeñas que la entrada*. Debe tener en cuenta que la arquitectura de la red es simétrica, pero eso no significa necesariamente que sus pesos deban serlo. ¿Pero por qué? ¿Qué aprende esta red y cómo lo hace? Echemos un vistazo a ambas redes y exploremos lo que están haciendo.

La red del codificador toma la entrada de alta dimensión y la reduce a código latente de dimensión inferior (es decir, aprende los patrones en los datos de entrada). Esto funciona de manera similar al *análisis de componentes principales* y la *factorización de matrices*. Funciona de la siguiente manera:



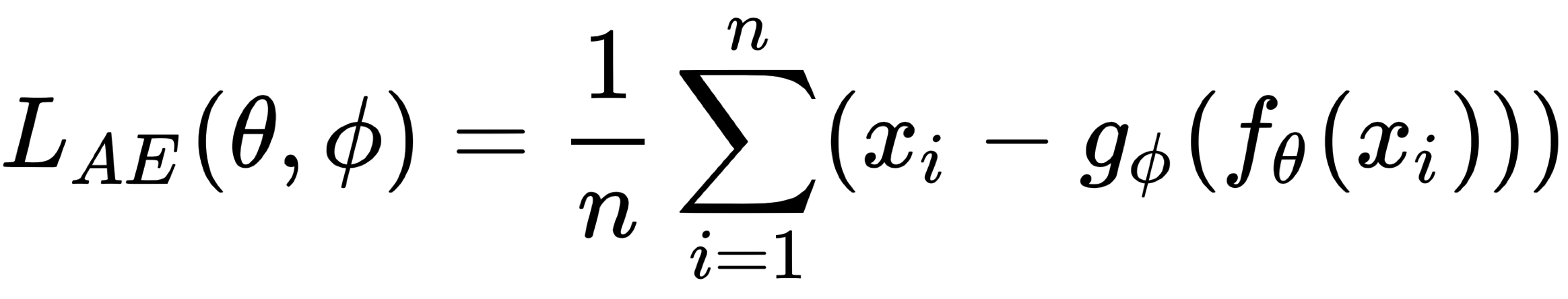
La red de decodificadores toma como entrada el código latente de menor dimensión (los patrones), que contiene toda la información principal sobre la entrada, y **reconstruye la entrada original** (o lo más cerca posible de la entrada original) a partir de ella. Funciona de la siguiente manera:



Podemos combinar las dos ecuaciones anteriores y expresar el codificador automático de la siguiente manera:

Nuestro objetivo es que la entrada original sea lo más cercana (idealmente, idéntica) a la salida reconstruida, es decir, .

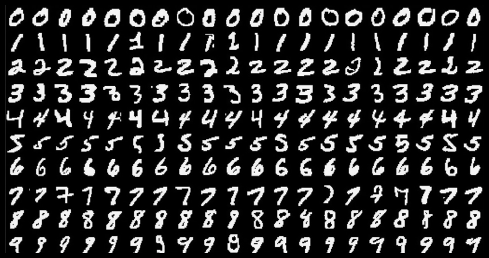
Tanto el codificador como el decodificador tienen pesos separados, pero aprendemos los parámetros juntos para generar los datos reconstruidos, que son casi idénticos a la entrada original. Durante el entrenamiento, podemos utilizar la pérdida de MSE:



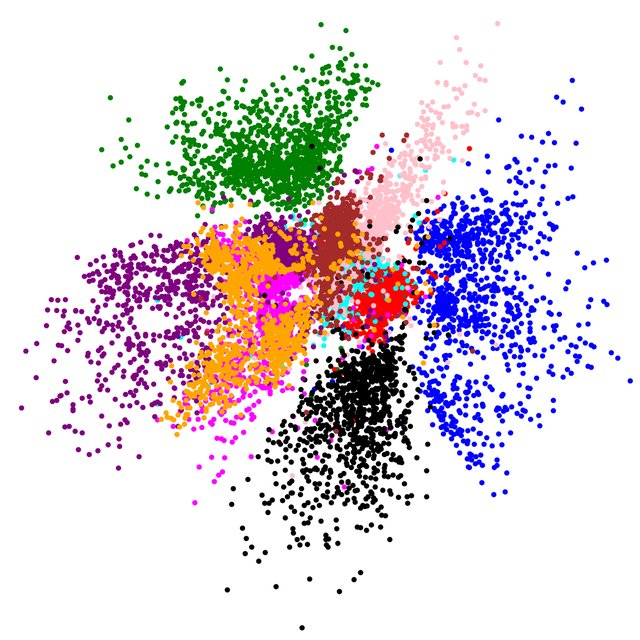
Este tipo de codificador automático se denomina comúnmente **codificador automático incompleto** porque la capa de cuello de botella es mucho más pequeña que las dimensiones de la capa de entrada y salida.

Pero, ¿qué sucede en esta capa de cuello de botella que permite al decodificador reconstruir la entrada a partir de ella? Esta codificación latente, que es un espacio de alta dimensión que se asigna a uno de dimensión inferior, aprende una (manifold) variedad, que es un espacio topológico que se asemeja al espacio euclidiano en cada punto (arrojaremos más luz sobre los espacios topológicos y las variedades en el Capítulo 12). , Aprendizaje profundo geométrico). Podemos representar esta variedad como un campo vectorial y visualizar los grupos de datos. Es este campo vectorial desde el que el codificador automático está aprendiendo a reconstruir las entradas. Cada punto de datos se puede encontrar en este colector y podemos proyectarlo de nuevo en un espacio de dimensiones superiores para reconstruirlo.

Supongamos que tenemos el conjunto de datos MNIST, que contiene imágenes de dígitos escritos a mano del 0 al 9. En la siguiente captura de pantalla, podemos ver algunas de las imágenes del conjunto de datos:



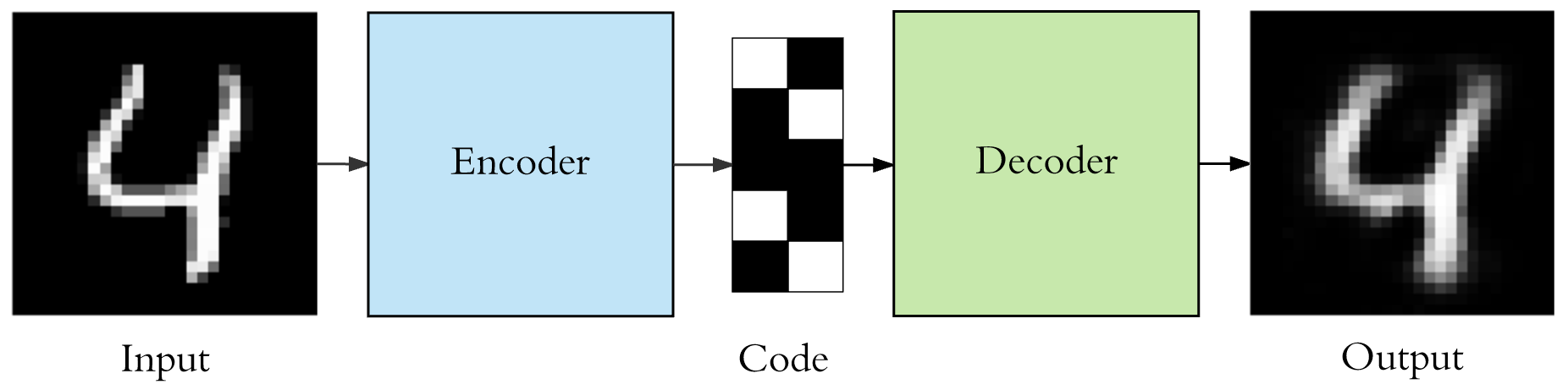
La red del codificador toma estos datos como entrada y los codifica en una capa latente de cuello de botella de menor dimensión, que contiene una representación comprimida de esta entrada de mayor dimensión y nos la muestra en dos dimensiones. Este espacio de incrustación tiene el siguiente aspecto, donde cada uno de los colores representa un dígito específico:



Por ahora, probablemente se esté preguntando para qué sirve una arquitectura como esta. ¿Qué podríamos ganar al entrenar un modelo para recrear y generar su propia entrada? Resulta que hay varias cosas: *podríamos usarlo para comprimir datos y almacenarlos para ahorrar espacio y reconstruirlos cuando necesitemos acceder a ellos*, podríamos eliminar el ruido de las imágenes o archivos de audio, o podríamos usarlo para la dimensionalidad reducción para visualización de datos.

Sin embargo, el hecho de que esta arquitectura se pueda utilizar para comprimir imágenes **no significa que sea similar a un algoritmo de compresión de datos como MP3 o JPEG.** Un codificador automático **solo puede comprimir los datos que ha visto durante el entrenamiento**, por lo que si se entrena con imágenes de automóviles, sería bastante ineficaz para comprimir imágenes de caballos, ya que las funciones que ha aprendido son específicas de los automóviles, que no generalizan bien a los caballos. Los algoritmos de compresión como MP3 y JPEG, por otro lado, no aprenden las características de las entradas que reciben; hacen suposiciones generales sobre sus insumos.

En el siguiente diagrama, puede ver un codificador automático comprimiendo una imagen en un espacio latente y reconstruyéndola en la salida:



Puede ver, en el diagrama, que el autoencoder ha logrado reconstruir la imagen de entrada y todavía se ve como el número cuatro, **pero no es una réplica exacta**; parte de la **información** se ha **perdido**. Esto no es un error de entrenamiento; esto es por diseño. Los codificadores automáticos están diseñados para tener pérdidas y solo copian aproximadamente los datos de entrada para que puedan extraer solo lo necesario al *priorizar lo que considera más útil*.

Como hemos visto hasta ahora en este libro, agregar capas y profundizar en los codificadores automáticos tiene sus ventajas; permite que nuestra red neuronal capture mayores complejidades y reduce el costo computacional requerido (en comparación con ir más amplia y menos profunda). Del mismo modo, podemos agregar capas adicionales a nuestro codificador y descodificador. Esto es particularmente cierto en el caso de tratar con imágenes porque sabemos que las capas convolucionales brindan mejores resultados que aplanar la imagen y usarla como entrada.

#### Construyendo un codificador con Keras

Ahora vamos a pasar a algo realmente emocionante, construir un codificador automático usando la biblioteca tf.keras. Para simplificar, usaremos el conjunto de datos MNIST para el primer conjunto de ejemplos. El autoencoder generará un vector latente a partir de los datos de entrada y recuperará la entrada usando el decodificador. El vector latente en este primer ejemplo es 16-dim.

En primer lugar, implementaremos el codificador automático compilando el codificador.

from tensorflow.keras.layers import Dense, Input

from tensorflow.keras.layers import Conv2D, Flatten

from tensorflow.keras.layers import Reshape, Conv2DTranspose

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import plot\_model

from tensorflow.keras import backend as K

import numpy as np

import matplotlib.pyplot as plt

# load MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# reshape to (28, 28, 1) and normalize input images

image\_size = x\_train.shape[1]

x\_train = np.reshape(x\_train, [-1, image\_size, image\_size, 1])

x\_test = np.reshape(x\_test, [-1, image\_size, image\_size, 1])

x\_train = x\_train.astype('float32') / 255

x\_test = x\_test.astype('float32') / 255

# network parameters

input\_shape = (image\_size, image\_size, 1)

batch\_size = 32

kernel\_size = 3

latent\_dim = 16

# encoder/decoder number of CNN layers and filters per layer

layer\_filters = [32, 64]

# build the autoencoder model

# first build the encoder model

inputs = Input(shape=input\_shape, name='encoder\_input')

x = inputs

# stack of Conv2D(32)-Conv2D(64)

for filters in layer\_filters:

x = Conv2D(filters=filters,

kernel\_size=kernel\_size,

activation='relu',

strides=2,

padding='same')(x)

# shape info needed to build decoder model

# so we don't do hand computation

# the input to the decoder's first

# Conv2DTranspose will have this shape

# shape is (7, 7, 64) which is processed by

# the decoder back to (28, 28, 1)

shape = K.int\_shape(x)

# generate latent vector

x = Flatten()(x)

latent = Dense(latent\_dim, name='latent\_vector')(x)

# instantiate encoder model

encoder = Model(inputs,

latent,

name='encoder')

encoder.summary()

plot\_model(encoder,

to\_file='encoder.png',

show\_shapes=True)

# build the decoder model

latent\_inputs = Input(shape=(latent\_dim,), name='decoder\_input')

# use the shape (7, 7, 64) that was earlier saved

x = Dense(shape[1] \* shape[2] \* shape[3])(latent\_inputs)

# from vector to suitable shape for transposed conv

x = Reshape((shape[1], shape[2], shape[3]))(x)

# stack of Conv2DTranspose(64)-Conv2DTranspose(32)

for filters in layer\_filters[::-1]:

x = Conv2DTranspose(filters=filters,

kernel\_size=kernel\_size,

activation='relu',

strides=2,

padding='same')(x)

# reconstruct the input

outputs = Conv2DTranspose(filters=1,

kernel\_size=kernel\_size,

activation='sigmoid',

padding='same',

name='decoder\_output')(x)

# instantiate decoder model

decoder = Model(latent\_inputs, outputs, name='decoder')

decoder.summary()

plot\_model(decoder, to\_file='decoder.png', show\_shapes=True)

# autoencoder = encoder + decoder

# instantiate autoencoder model

autoencoder = Model(inputs,

decoder(encoder(inputs)),

name='autoencoder')

autoencoder.summary()

plot\_model(autoencoder,

to\_file='autoencoder.png',

show\_shapes=True)

# Mean Square Error (MSE) loss function, Adam optimizer

autoencoder.compile(loss='mse', optimizer='adam')

# train the autoencoder

autoencoder.fit(x\_train,

x\_train,

validation\_data=(x\_test, x\_test),

epochs=1,

batch\_size=batch\_size)

# predict the autoencoder output from test data

x\_decoded = autoencoder.predict(x\_test)

# display the 1st 8 test input and decoded images

imgs = np.concatenate([x\_test[:8], x\_decoded[:8]])

imgs = imgs.reshape((4, 4, image\_size, image\_size))

imgs = np.vstack([np.hstack(i) for i in imgs])

plt.figure()

plt.axis('off')

plt.title('Input: 1st 2 rows, Decoded: last 2 rows')

plt.imshow(imgs, interpolation='none', cmap='gray')

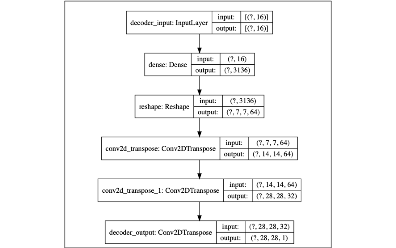
plt.savefig('input\_and\_decoded.png')

plt.show()

El decodificador del Listado 3.2.1 descomprime el vector latente para recuperar el dígito MNIST. La etapa de entrada del decodificador es una capa densa que aceptará el vector latente. El número de unidades es igual al producto de las dimensiones de salida de Conv2D guardadas del codificador. Esto se hace para que podamos cambiar fácilmente el tamaño de la salida de la capa Densa para Conv2DTranspose para finalmente recuperar las dimensiones originales de la imagen MNIST.

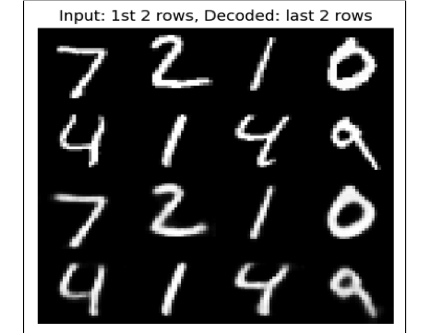
El decodificador está hecho de una pila de tres Conv2DTranspose. En nuestro caso, vamos a usar una CNN transpuesta (a veces llamada deconvolución), que se usa más comúnmente en decodificadores. Podemos imaginar la CNN transpuesta (Conv2DTranspose) como el proceso inverso de CNN.

En un ejemplo simple, si la CNN convierte una imagen en mapas de características, la CNN transpuesta producirá una imagen de mapas de características. La figura 3.2.2 muestra el modelo de decodificador:



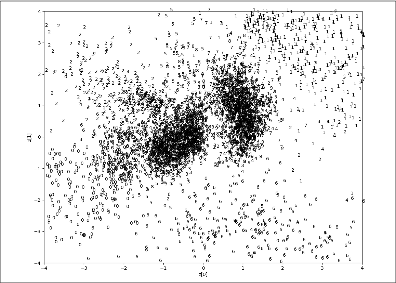
La salida del tensor del codificador también es la entrada a un decodificador que genera la salida del codificador automático. En este ejemplo, usaremos la función de pérdida MSE y el optimizador Adam. Durante el entrenamiento, la entrada es la misma que la salida, x\_train. Debemos tener en cuenta que en nuestro ejemplo, solo hay unas pocas capas que son suficientes para llevar la pérdida de validación a 0.01 en una época. Para conjuntos de datos más complejos, es posible que necesitemos un codificador y decodificador más profundos, así como más épocas de entrenamiento.

Después de entrenar el codificador automático durante una época con una pérdida de validación de 0.01, podemos verificar si puede codificar y decodificar los datos MNIST que no ha visto antes. La figura 3.2.4 nos muestra ocho muestras de los datos de prueba y las imágenes decodificadas correspondientes:



Excepto por un leve desenfoque en las imágenes, podemos reconocer fácilmente que el codificador automático puede recuperar la entrada con buena calidad. Los resultados mejorarán a medida que entrenemos para un mayor número de épocas.

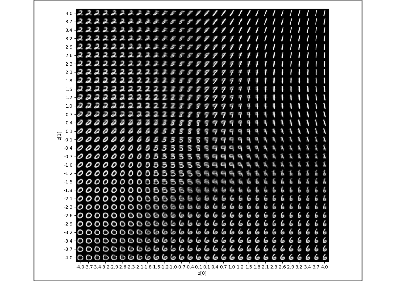
En este punto, podemos estar preguntándonos: ¿cómo podemos visualizar el vector latente en el espacio? Un método simple de visualización es forzar al codificador automático a aprender las características de los dígitos MNIST utilizando un vector latente de 2 atenuaciones. A partir de ahí, podemos proyectar este vector latente en un espacio bidimensional para ver cómo se distribuyen los vectores latentes MNIST. La Figura 3.2.5 y la Figura 3.2.6 muestran la distribución de los dígitos del MNIST en función de las dimensiones del código latente.



En la figura 3.2.5, podemos ver que los vectores latentes para un dígito específico se agrupan en una región del espacio. Por ejemplo, el dígito 0 está en el cuadrante inferior izquierdo, mientras que el dígito 1 está en el cuadrante superior derecho. Este agrupamiento se refleja en la figura. De hecho, la misma figura muestra el resultado de navegar o generar nuevos dígitos desde el espacio latente, como se muestra en la Figura 3.2.5.

Por ejemplo, comenzando desde el centro y variando el valor de un vector latente de 2 dim hacia el cuadrante superior derecho, esto nos muestra que el dígito cambia de 9 a 1. Esto se espera ya que, de la Figura 3.2.5, estamos capaz de ver que los valores de código latente para los grupos de dígitos 9 están cerca del centro, y los valores de código de dígito 1 se agrupan en el cuadrante superior derecho.

Para la Figura 3.2.5 y la Figura 3.2.6, solo hemos explorado las regiones entre -4.0 y +4.0 para cada dimensión vectorial latente:



Como se puede ver en la Figura 3.2.5, la distribución del código latente no es continua. Idealmente, debería verse como un círculo donde hay valores válidos en todas partes. Debido a esta discontinuidad, hay regiones donde, si decodificamos el vector latente, apenas se producirán dígitos reconocibles.

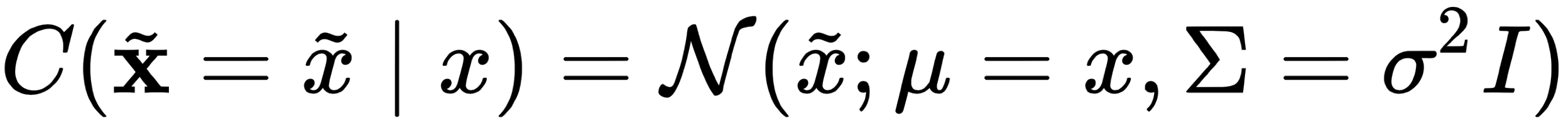
La Figura 3.2.5 y la Figura 3.2.6 se generaron después de 20 épocas de entrenamiento. El código autoencoder-mnist-3.2.1.py se modificó estableciendo latent\_dim = 2. La función plot\_ results () traza el dígito MNIST como una función del vector latente 2-dim. Por conveniencia, el programa se guarda como autoencoder-2dim- mnist-3.2.2.py con el código parcial que se muestra en el Listado 3.2.2. El resto del código es prácticamente similar al listado 3.2.1 y ya no se muestra aquí.

### El codificador automático de eliminación de ruido.

El codificador automático de eliminación de ruido (DAE) es una variación del codificador automático anterior, **ya que aprende a reconstruir entradas dañadas o ruidosas con casi certeza**. Supongamos que tenemos una imagen y, por alguna razón, está borrosa o algunos de los píxeles se han corrompido y nos gustaría mejorar la resolución de la imagen (algo así como lo hacen en las películas cuando pueden encontrar pistas en las imágenes). con resolución relativamente baja). Podemos pasarlo por nuestro DAE y obtener una imagen completamente reconstruida.

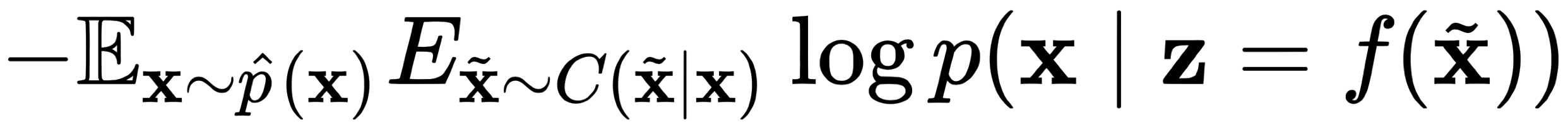
Comenzamos corrompiendo la entrada usando una distribución condicional, que es básicamente un mapeo estocástico, y nos devuelve las muestras corrompidas. Ahora que tenemos nueva entrada, nuestro codificador automático aprenderá a reconstruir los datos no corruptos, es decir, y para entrenar esto, nuestros datos serán los pares. Lo que queremos que aprenda el decodificador es , como antes, z era la salida del codificador.

La corrupción anterior funciona de la siguiente manera:

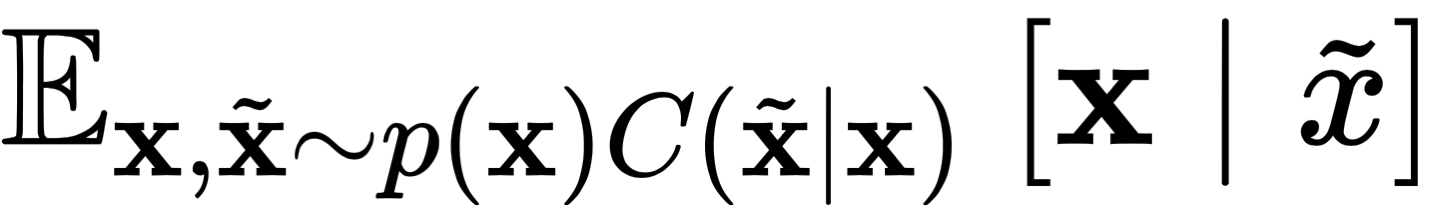
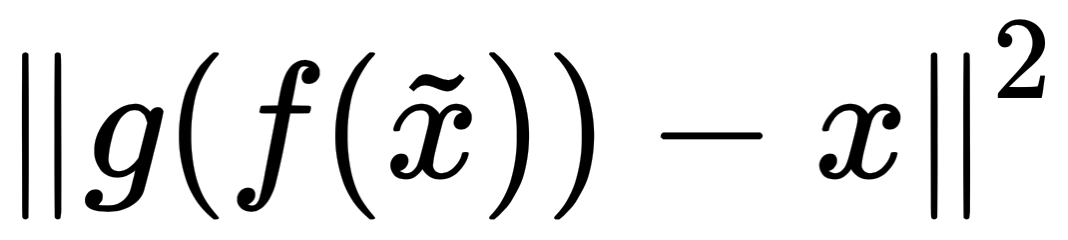


Aquí, es la varianza del ruido.

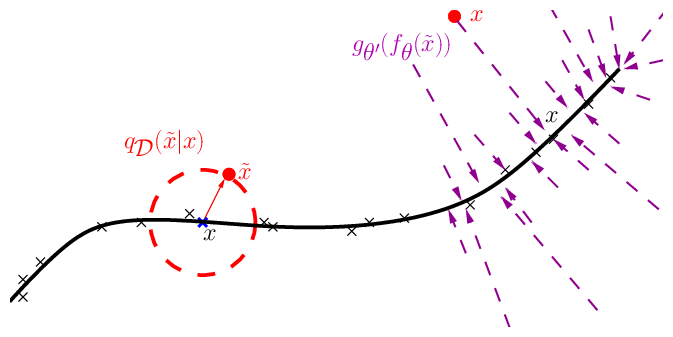
Podemos entrenar a nuestro DAE como cualquier otro FNN y realizar descenso de gradiente en el siguiente:



Aquí está la distribución de los datos de entrenamiento.

Como se mencionó, el codificador proyecta datos de alta dimensión en un espacio de menor dimensión, llamado **espacio latente** (latent space), y aprende la forma de la colección. Luego intenta mapear los datos corruptos en o cerca de este colector para averiguar qué podría ser y luego los junta en el proceso de reconstrucción para obtener x estimando  y minimizando el error cuadrado, .

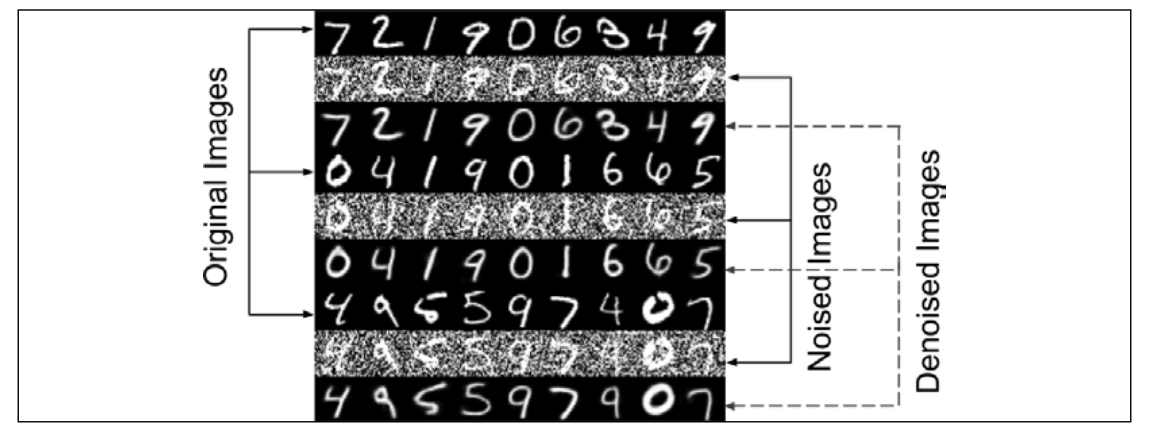
Podemos ver este proceso en el siguiente diagrama:



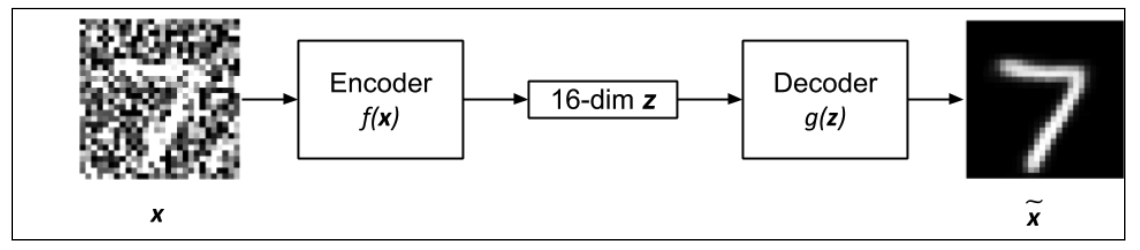
Aquí, la curva negra es la variedad aprendida en el espacio latente y puede ver los puntos ruidosos, se proyectan en el punto más cercano en la variedad para estimar lo que podría ser.

#### En Keras…

Ahora vamos a construir un codificador automático con una aplicación práctica. Primeramente, pintemos una imagen e imaginemos que las imágenes de dígitos del MNIST se corrompieron por el ruido, lo que dificultaría la lectura para los humanos. Podemos construir un codificador automático de eliminación de ruido (DAE) para eliminar el ruido de estas imágenes. La figura 3.3.1 nos muestra tres conjuntos de dígitos MNIST. Las filas superiores de cada conjunto (por ejemplo, los dígitos 7, 2, 1, 9, 0, 6, 3, 4 y 9 de MNIST) son las imágenes originales. Las filas del medio muestran las entradas al DAE, que son las imágenes originales corrompidas por el ruido. Como seres humanos, podemos encontrar que es difícil leer los dígitos MNIST corruptos. Las últimas filas muestran las salidas del DAE.



Como se muestra en la Figura anterior, el codificador automático de eliminación de ruido tiene prácticamente la misma estructura que el codificador automático para MNIST que presentamos en la sección anterior.



La entrada en la imagen anterior se define como:

En esta fórmula, x\_{orig} representa la imagen MNIST original corrompida por noise. El objetivo del codificador es descubrir cómo producir el vector latente, z, que permitirá al decodificador recuperarse como MSE, como se muestra a través de: x\_{orig} minimizando la función de pérdida de disimilitud:

En este ejemplo, m es la dimensión de salida (por ejemplo, en MNIST, m=ancho x altura x canal = 28 x 28 x1 = 784) y x\_i son elementos de x\_orig y x respectivamente.

Para implementar el DAE, necesitaremos realizar algunos cambios en el codificador automático presentado en la sección anterior. En primer lugar, los datos de entrada de entrenamiento deben ser dígitos MNIST corruptos. Los datos de salida de entrenamiento son los mismos dígitos MNIST limpios originales. Esto es como decirle al codificador automático cuáles deberían ser las imágenes corregidas o pedirle que averigüe cómo eliminar el ruido dada una imagen dañada. Por último, debemos validar el codificador automático en los datos de prueba MNIST corruptos.

El dígito 7 de MNIST que se muestra a la izquierda de la Figura 3.3.2 es una entrada de imagen corrupta real. El de la derecha es la salida de imagen limpia de un codificador automático de eliminación de ruido entrenado.

from tensorflow.keras.layers import Dense, Input

from tensorflow.keras.layers import Conv2D, Flatten

from tensorflow.keras.layers import Reshape, Conv2DTranspose

from tensorflow.keras.models import Model

from tensorflow.keras import backend as K

from tensorflow.keras.datasets import mnist

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

np.random.seed(1337)

# load MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# reshape to (28, 28, 1) and normalize input images

image\_size = x\_train.shape[1]

x\_train = np.reshape(x\_train, [-1, image\_size, image\_size, 1])

x\_test = np.reshape(x\_test, [-1, image\_size, image\_size, 1])

x\_train = x\_train.astype('float32') / 255

x\_test = x\_test.astype('float32') / 255

# generate corrupted MNIST images by adding noise with normal dist

# centered at 0.5 and std=0.5

noise = np.random.normal(loc=0.5, scale=0.5, size=x\_train.shape)

x\_train\_noisy = x\_train + noise

noise = np.random.normal(loc=0.5, scale=0.5, size=x\_test.shape)

x\_test\_noisy = x\_test + noise

# adding noise may exceed normalized pixel values>1.0 or <0.0

# clip pixel values >1.0 to 1.0 and <0.0 to 0.0

x\_train\_noisy = np.clip(x\_train\_noisy, 0., 1.)

x\_test\_noisy = np.clip(x\_test\_noisy, 0., 1.)

# network parameters

input\_shape = (image\_size, image\_size, 1)

batch\_size = 32

kernel\_size = 3

latent\_dim = 16

# encoder/decoder number of CNN layers and filters per layer

layer\_filters = [32, 64]

# build the autoencoder model

# first build the encoder model

inputs = Input(shape=input\_shape, name='encoder\_input')

x = inputs

# stack of Conv2D(32)-Conv2D(64)

for filters in layer\_filters:

x = Conv2D(filters=filters,

kernel\_size=kernel\_size,

strides=2,

activation='relu',

padding='same')(x)

# shape info needed to build decoder model so we don't do hand

computation

# the input to the decoder's first Conv2DTranspose will have this

shape

# shape is (7, 7, 64) which can be processed by the decoder back to

(28, 28, 1)

shape = K.int\_shape(x)

# generate the latent vector

x = Flatten()(x)

latent = Dense(latent\_dim, name='latent\_vector')(x)

# instantiate encoder model

encoder = Model(inputs, latent, name='encoder')

encoder.summary()

# build the decoder model

latent\_inputs = Input(shape=(latent\_dim,), name='decoder\_input')

# use the shape (7, 7, 64) that was earlier saved

x = Dense(shape[1] \* shape[2] \* shape[3])(latent\_inputs)

# from vector to suitable shape for transposed conv

x = Reshape((shape[1], shape[2], shape[3]))(x)

# stack of Conv2DTranspose(64)-Conv2DTranspose(32)

for filters in layer\_filters[::-1]:

x = Conv2DTranspose(filters=filters,

kernel\_size=kernel\_size,

strides=2,

activation='relu',

padding='same')(x)

# reconstruct the denoised input

outputs = Conv2DTranspose(filters=1,

kernel\_size=kernel\_size,

padding='same',

activation='sigmoid',

name='decoder\_output')(x)

# instantiate decoder model

decoder = Model(latent\_inputs, outputs, name='decoder')

decoder.summary()

# autoencoder = encoder + decoder

# instantiate autoencoder model

autoencoder = Model(inputs, decoder(encoder(inputs)),

name='autoencoder')

autoencoder.summary()

# Mean Square Error (MSE) loss function, Adam optimizer

autoencoder.compile(loss='mse', optimizer='adam')

# train the autoencoder

autoencoder.fit(x\_train\_noisy,

x\_train,

validation\_data=(x\_test\_noisy, x\_test),

epochs=10,

batch\_size=batch\_size)

# predict the autoencoder output from corrupted test images

x\_decoded = autoencoder.predict(x\_test\_noisy)

# 3 sets of images with 9 MNIST digits

# 1st rows - original images

# 2nd rows - images corrupted by noise

# 3rd rows - denoised images

rows, cols = 3, 9

num = rows \* cols

imgs = np.concatenate([x\_test[:num], x\_test\_noisy[:num], x\_

decoded[:num]])

imgs = imgs.reshape((rows \* 3, cols, image\_size, image\_size))

imgs = np.vstack(np.split(imgs, rows, axis=1))

imgs = imgs.reshape((rows \* 3, -1, image\_size, image\_size))

imgs = np.vstack([np.hstack(i) for i in imgs])

imgs = (imgs \* 255).astype(np.uint8)

plt.figure()

plt.axis('off')

plt.title('Original images: top rows, '

'Corrupted Input: middle rows, '

'Denoised Input: third rows')

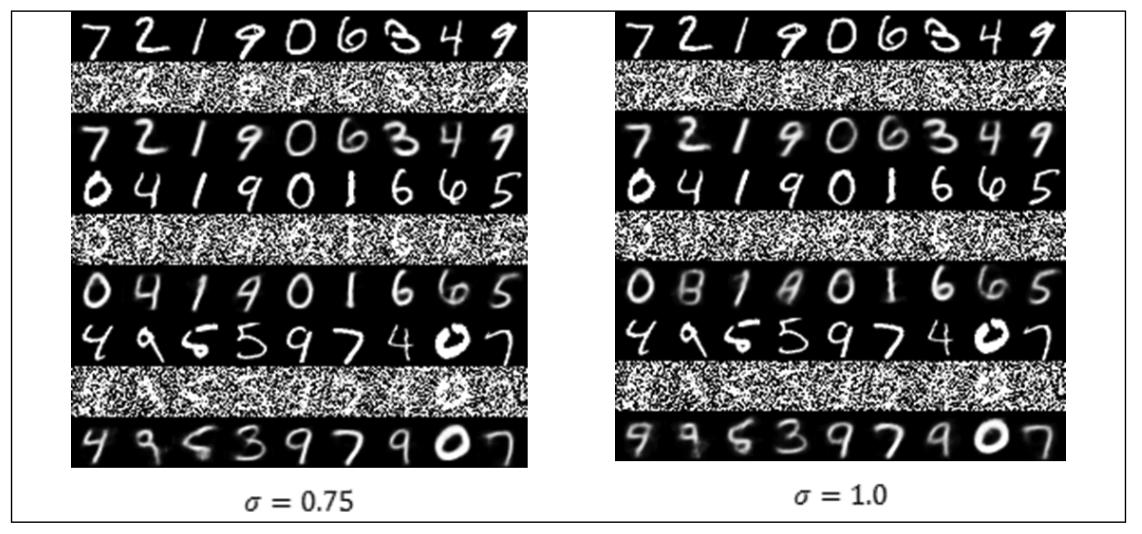
plt.imshow(imgs, interpolation='none', cmap='gray')

Image.fromarray(imgs).save('corrupted\_and\_denoised.png')

plt.show()

El listado 3.3.1 muestra el codificador automático de eliminación de ruido, que se ha contribuido a el repositorio oficial de Keras GitHub. Usando el mismo conjunto de datos MNIST, podemos simular imágenes corruptas agregando ruido aleatorio. El ruido agregado es una distribución gaussiana con una media de 𝜇𝜇 = 0.5 y una desviación estándar de 𝜎𝜎 = 0.5. Dado que la adición de ruido aleatorio puede empujar los datos de los píxeles a valores no válidos menores que 0 o mayores que 1, los valores de los píxeles se recortan en el rango [0.1, 1.0].

Todo lo demás permanecerá prácticamente igual que el autoencoder de la sección anterior. Usaremos la misma función de pérdida de MSE y el optimizador de Adam. Sin embargo, el número de épocas de entrenamiento ha aumentado a 10. Esto es para permitir una optimización suficiente de los parámetros.



La figura anterior muestra un cierto nivel de robustez del DAE a medida que aumenta el nivel de ruido de = 0,5 a = 0,75 y = 1,0. Con = 0,75, el DAE aún puede recuperar las imágenes originales. Sin embargo, en = 1.0, algunos dígitos, como 4 y 5 en el segundo y tercer conjuntos, ya no se pueden recuperar correctamente.

### El codificador automático variacional

El **autoencoder variacional (VAE)** es otro tipo de autoencoder, pero con algunas diferencias particulares. De hecho, en lugar de aprender las funciones f () y g (), aprende la función de densidad de probabilidad de los datos de entrada.

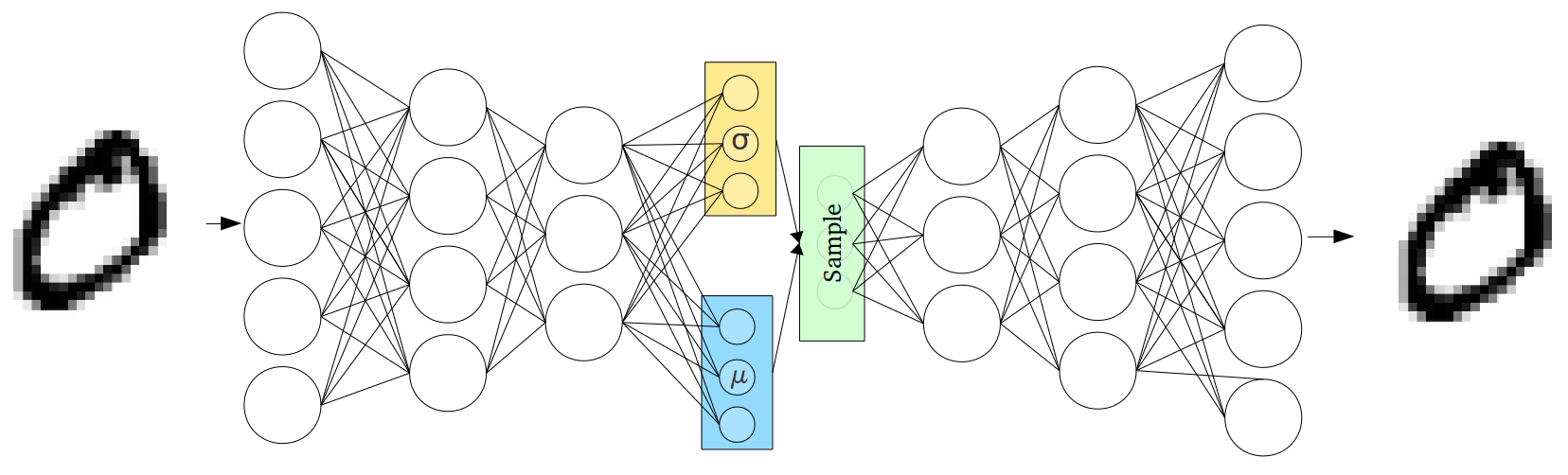
Supongamos que tenemos una distribución, pθ, y está parametrizada por θ. Aquí, podemos expresar la relación entre x y z de la siguiente manera:

* pθ (z): El anterior
* pθ (x | z): La probabilidad (la distribución de la entrada dado el espacio latente)
* pθ (z | x): La parte posterior (la distribución del espacio latente dado la entrada)

Las distribuciones antes mencionadas son parametrizadas por redes neuronales, lo que les permite capturar no linealidades complejas y, como sabemos, las entrenamos mediante descenso de gradiente.

Pero, ¿por qué los autores de este método decidieron desviarse del enfoque anterior para aprender una distribución? Hay algunas razones por las que esto es más efectivo. Para empezar, los datos con los que estaríamos tratando a menudo son ruidosos y, en cambio, modelar la distribución es mejor para nosotros. El objetivo aquí, como habrá adivinado, es generar datos que tengan una estadística similar a la de la entrada.

Antes de continuar, echemos un vistazo a cómo se ve un VAE:



Como puede ver, comparte algunas similitudes con el codificador automático pero, como mencionamos, en lugar de z = f (x) y x '= g (z), aprendemos p = (z | x) yp = (x | z), respectivamente. Sin embargo, debido a que ahora hay una variable aleatoria entre la entrada y la salida, esta arquitectura no se puede entrenar mediante la **propagación hacia atrás regular**; en su lugar, hacemos retropropagación a través de los parámetros de la distribución latente.

Una vez que conocemos las distribuciones previa y de probabilidad y los parámetros reales, θ \*, podemos generar muestras haciendo repetidamente lo siguiente: