TP5

Deep Learning

Juan Pablo Oriana - 60621
Tomás Cerdeira - 60051
Santiago Garcia Montagner - 60352

Autoencoders

Arquitectura de un Autoencoder

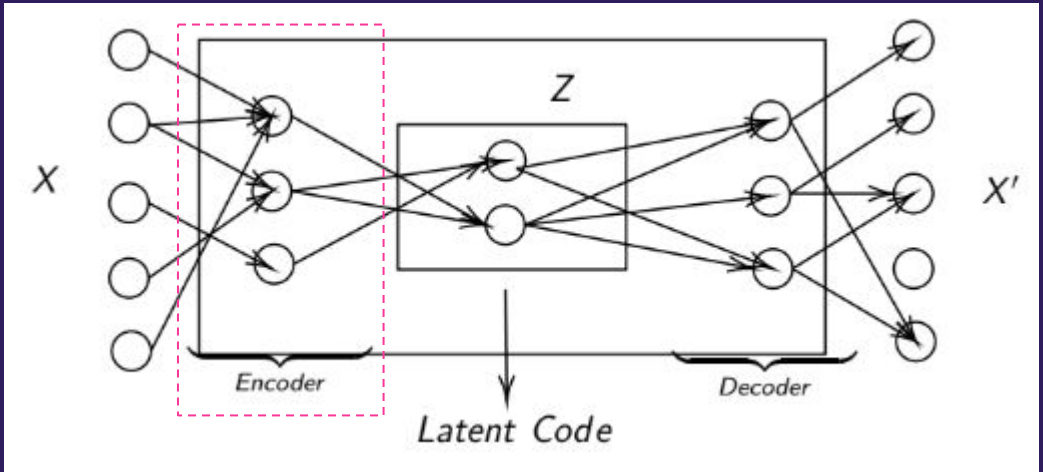
01 ENCODER

02 DECODER

03 ESPACIO LATENTE

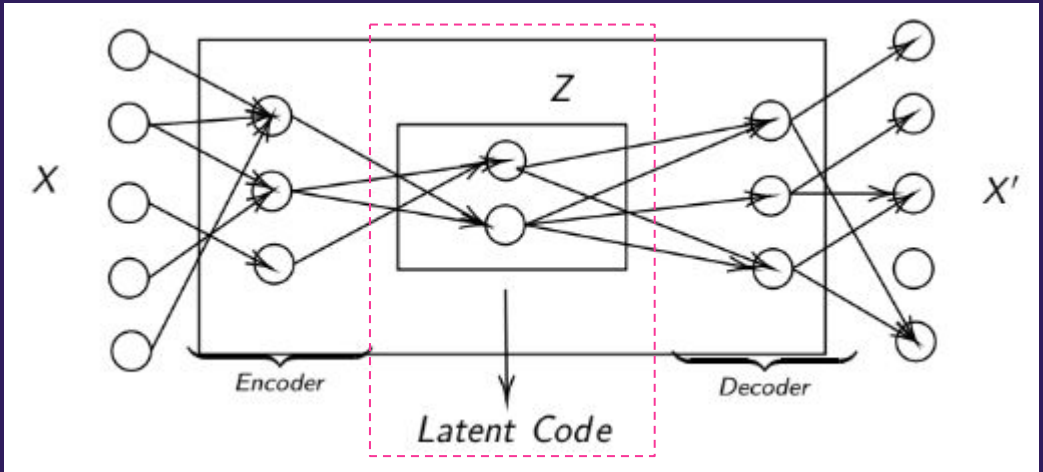
04 ACTIVACIONES Y ERROR

Encoder



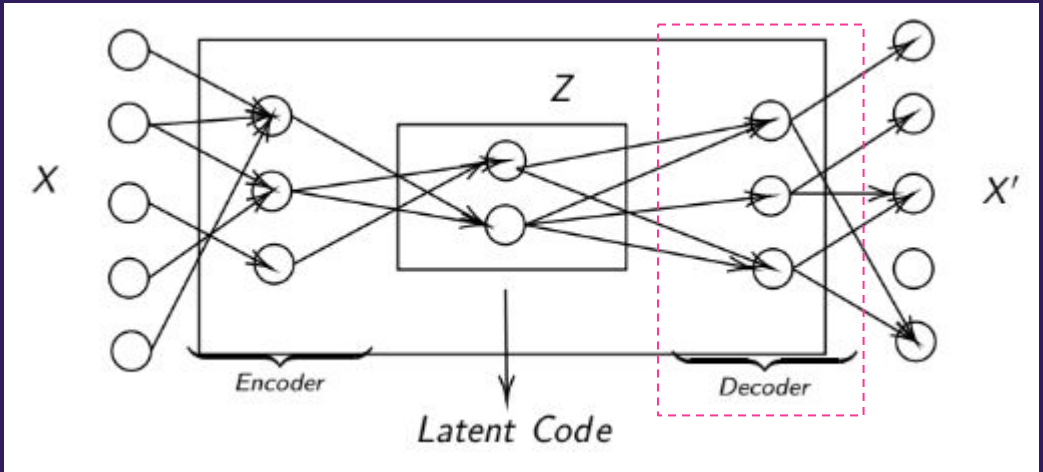
Reduce la dimensión de la entrada **al tamaño del espacio latente**

Especio Latente



Por el **Lemma: PCA**, los valores Z son las proyecciones de los datos en los **componentes principales**

Decoder

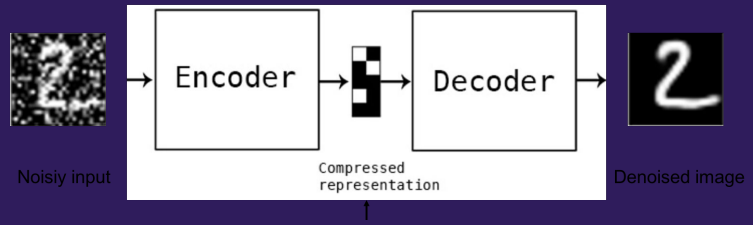


Aumenta la dimensión, descomprimiendo los datos para retornarlos **a su dimensión original**

Input (X) y Output (X')

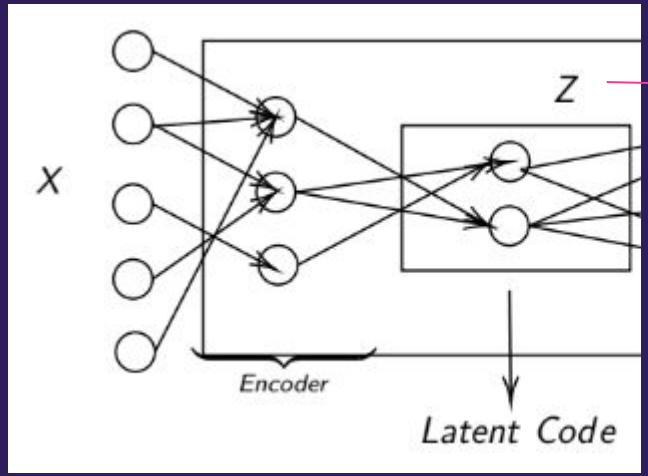
- En el **Entrenamiento**
  - X = X'
  - Se busca pasar por la red datos conocidos y obtener los mismos (o lo más parecido posible) en la salida.

- En la **Prueba**
  - e.j. Denoising
    - Y (X mutado/con ruido) = X' = X
    - Se busca pasar por la red datos con ruido/mutados y obtener sus correspondientes SIN dicha mutación/ruido



Activaciones

- En el **encoder**:

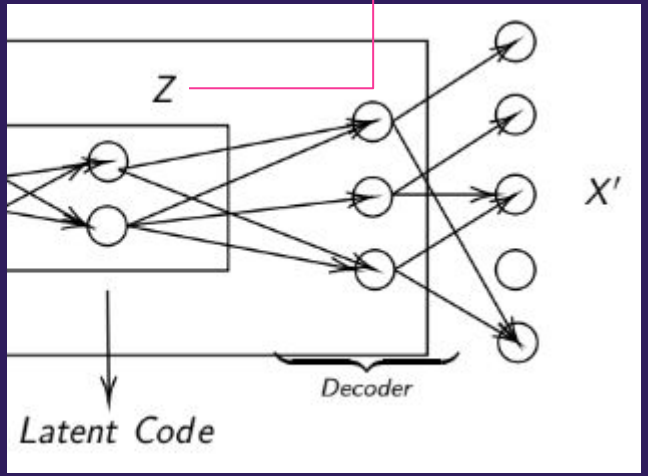


$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Z: valor en el espacio latente
σ: funcion de activacion
W: vector de pesos sinápticos
x: input
b: bias

Activaciones

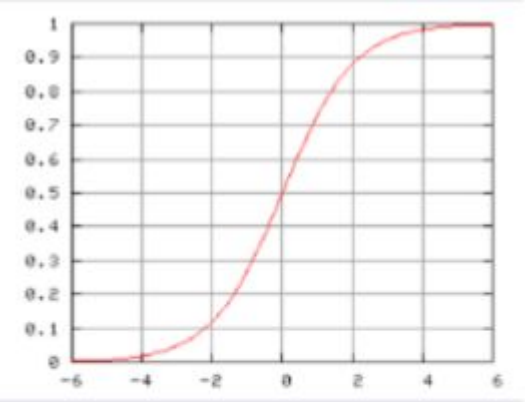
- En el **decoder**:



$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$$

Z: valor en el espacio latente
σ': funcion de activacion
W': vector de pesos sinápticos
x': output
b': bias

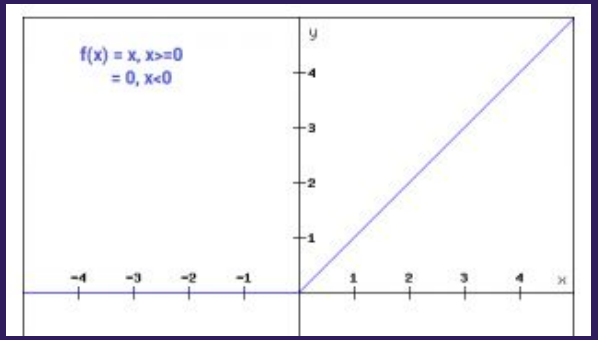
Función de activación



$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

- No Lineal
  - Sigmoide
    - **Logistica**

- También podríamos haber utilizado **ReLU**



PERO, tendríamos que haber normalizado la salida entre 0 y 1

Retropropagación del error

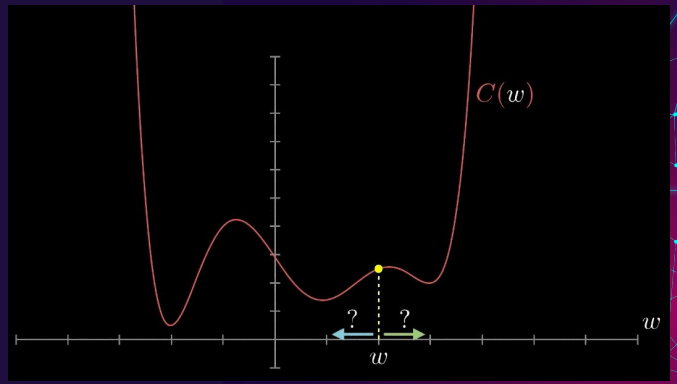
- **Pasos:**

1. Obtener todos los pesos sinápticos de la red en la iteración actual
2. Calcular la activación del input y compararla con el valor esperado que le corresponde
3. Utilizar el método de gradiente descendiente sobre 1. actualizando los mismo
4. Repetir desde 1. con la siguiente iteración

- **Condición de corte:** número de iteraciones máximas y/o un valor “mínimo” de error

$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2$$

Se busca **minimizar** esta función de “pérdida”



1.

a) Implementar un Autoencoder básico para las imágenes binarias de la lista de caracteres del archivo ”font.h”

1. Plantear una arquitectura de red para el Codificador y Decodificador que permita representar los datos de entrada en dos dimensiones.
2. Describan y estudien las diferentes técnicas de optimización que fueron aplicando para permitir que la red aprenda todo el set de datos o un subconjunto del mismo. En el caso de que sea un subconjunto mostrar porque no fue posible aprender el dataset completo.
3. Realizar el gráfico en dos dimensiones que muestre los datos de entrada en el espacio latente.
4. Mostrar cómo la red puede generar una nueva letra que no pertenece al conjunto de entrenamiento.

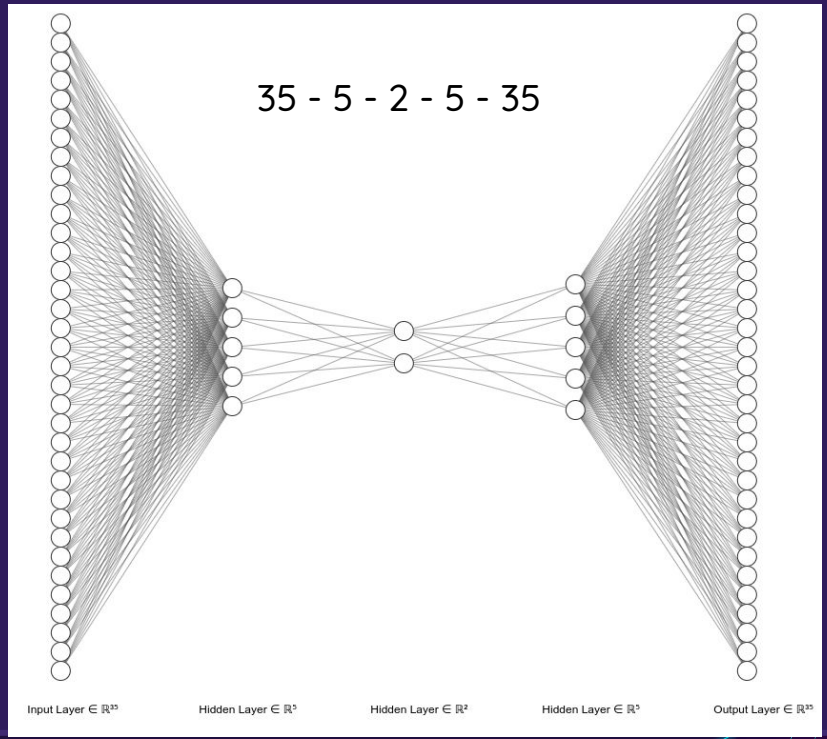
Conjunto de letras “fonts.h”

```
font1 = [
    [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],  # 0x20, space
    [0x04, 0x04, 0x04, 0x04, 0x04, 0x00, 0x04],  # 0x21, !
    [0x09, 0x09, 0x12, 0x00, 0x00, 0x00, 0x00],  # 0x22, "
    [0x0a, 0x0a, 0x1f, 0x0a, 0x1f, 0x0a, 0x0a],  # 0x23, #
    [0x04, 0x0f, 0x14, 0x0e, 0x05, 0x1e, 0x04],  # 0x24, $
    [0x19, 0x19, 0x02, 0x04, 0x08, 0x13, 0x13],  # 0x25, %
    [0x04, 0x0a, 0x0a, 0x0a, 0x15, 0x12, 0x0d],  # 0x26, &
    [0x04, 0x04, 0x08, 0x00, 0x00, 0x00, 0x00],  # 0x27, '
    [0x02, 0x04, 0x08, 0x08, 0x08, 0x04, 0x02],  # 0x28, (
    [0x08, 0x04, 0x02, 0x02, 0x02, 0x04, 0x08],  # 0x29, )
    [0x04, 0x15, 0x0e, 0x1f, 0x0e, 0x15, 0x04],  # 0x2a, *
    [0x00, 0x04, 0x04, 0x1f, 0x04, 0x04, 0x00],  # 0x2b, +
    [0x00, 0x00, 0x00, 0x00, 0x04, 0x04, 0x08],  # 0x2c, ,
    [0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00],  # 0x2d, -
    [0x00, 0x00, 0x00, 0x00, 0x00, 0x0c, 0x0c],  # 0x2e, .
    [0x01, 0x01, 0x02, 0x04, 0x08, 0x10, 0x10],  # 0x2f, /
    [0x01, 0x01, 0x02, 0x04, 0x08, 0x10, 0x10],  # 0x2f, /
    [0x0e, 0x11, 0x13, 0x15, 0x19, 0x11, 0x0e],  # 0x30, 0
    [0x04, 0x0c, 0x04, 0x04, 0x04, 0x04, 0x0e],  # 0x31, 1
    [0x0e, 0x11, 0x01, 0x02, 0x04, 0x08, 0x1f],  # 0x32, 2
    [0x0e, 0x11, 0x01, 0x06, 0x01, 0x11, 0x0e],  # 0x33, 3
    [0x02, 0x06, 0x0a, 0x12, 0x1f, 0x02, 0x02],  # 0x34, 4
    [0x1f, 0x10, 0x1e, 0x01, 0x01, 0x11, 0x0e],  # 0x35, 5
    [0x06, 0x08, 0x10, 0x1e, 0x11, 0x11, 0x0e],  # 0x36, 6
    [0x1f, 0x01, 0x02, 0x04, 0x08, 0x08, 0x08],  # 0x37, 7
    [0x0e, 0x11, 0x11, 0x0e, 0x11, 0x11, 0x0e],  # 0x38, 8
    [0x0e, 0x11, 0x11, 0x0f, 0x01, 0x02, 0x0c],  # 0x39, 9
    [0x00, 0x0c, 0x0c, 0x00, 0x0c, 0x0c, 0x00],  # 0x3a, :
    [0x00, 0x0c, 0x0c, 0x00, 0x0c, 0x04, 0x08],  # 0x3b, ;
    [0x02, 0x04, 0x08, 0x10, 0x08, 0x04, 0x02],  # 0x3c, <
    [0x00, 0x00, 0x1f, 0x00, 0x1f, 0x00, 0x00],  # 0x3d, =
    [0x08, 0x04, 0x02, 0x01, 0x02, 0x04, 0x08],  # 0x3e, >
    [0x0e, 0x11, 0x01, 0x02, 0x04, 0x00, 0x04]   # 0x3f, ?
]
```

Dimension Input (X) = 35 (en bits)

Arquitectura planteada 1.a.1

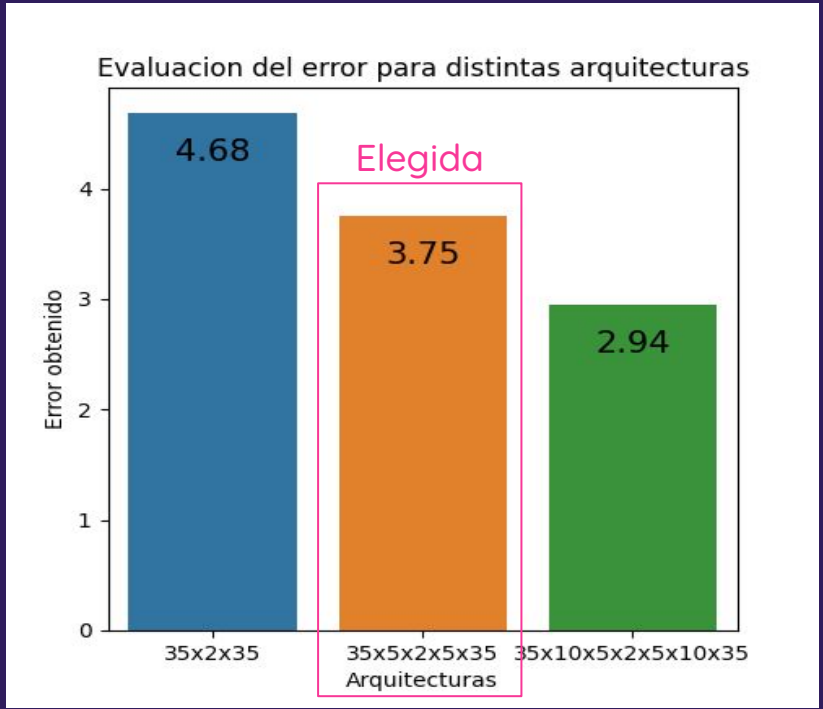
```
layer1 = Layer(5, 35)
layer2 = Layer(2, 5)
layer3 = Layer(5,2)
layer4 = Layer(35,5)
```



Evaluación de Arquitecturas 1.a.1

- Probamos **distintas arquitecturas** para encontrar una con el **error mínimo**
- Tradeoff de error mínimo contra tiempo de entrenamiento

- iters: 10.000
- learning rate: 0.1

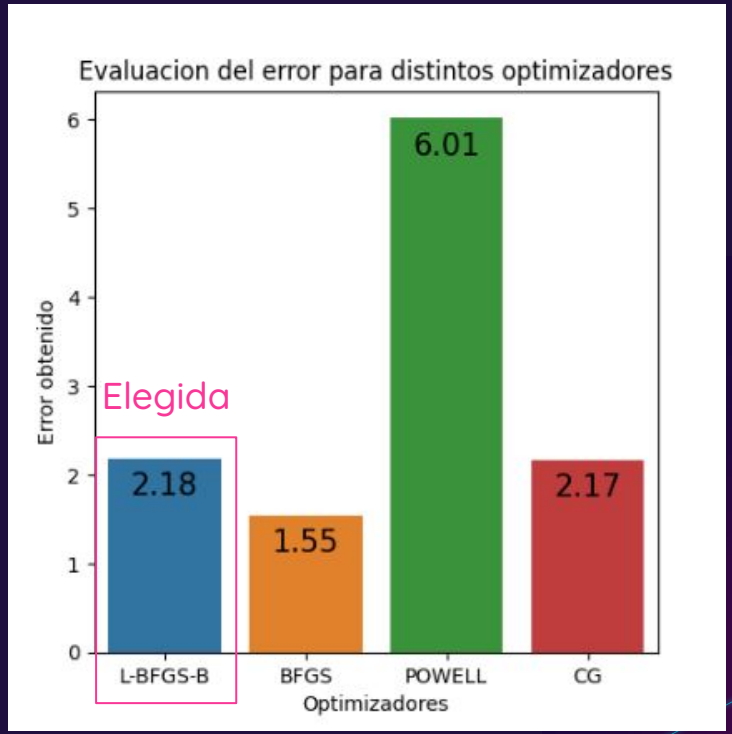


Funcionamiento de la red

Evaluación de los métodos Optimizadores 1.a.2

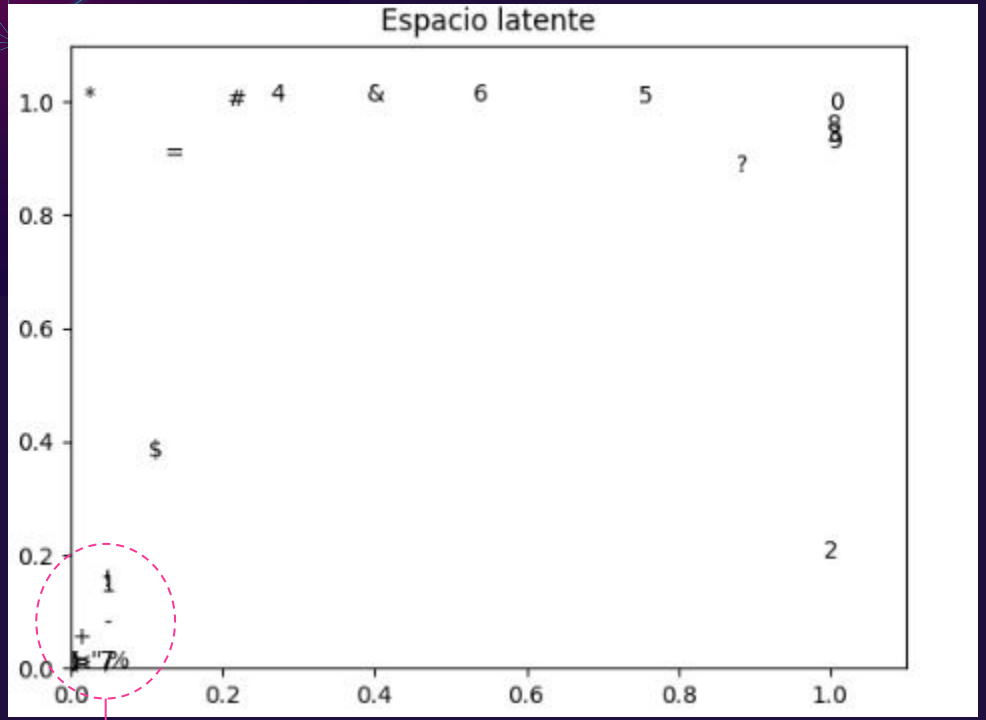
Subconjunto utilizado:

```
symbols1 = [
#, &, (, ),., /,0,
1, 2, 3, 4, 5, 6,
7, 8, 9,>,?
]
```



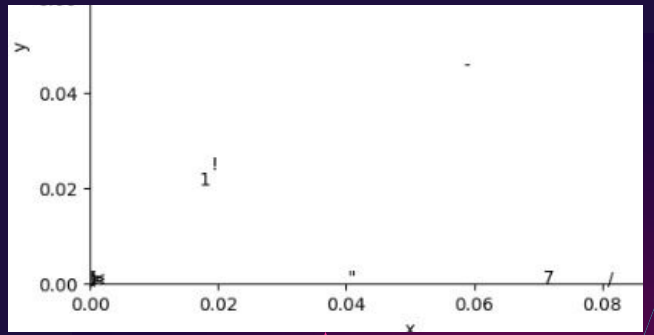
(50.000 iteraciones)

Resultados 1.a.3



Activación neurona 2

Activación neurona 1



ZOOM IN

Resultados 1.a.4

DEMO

1.

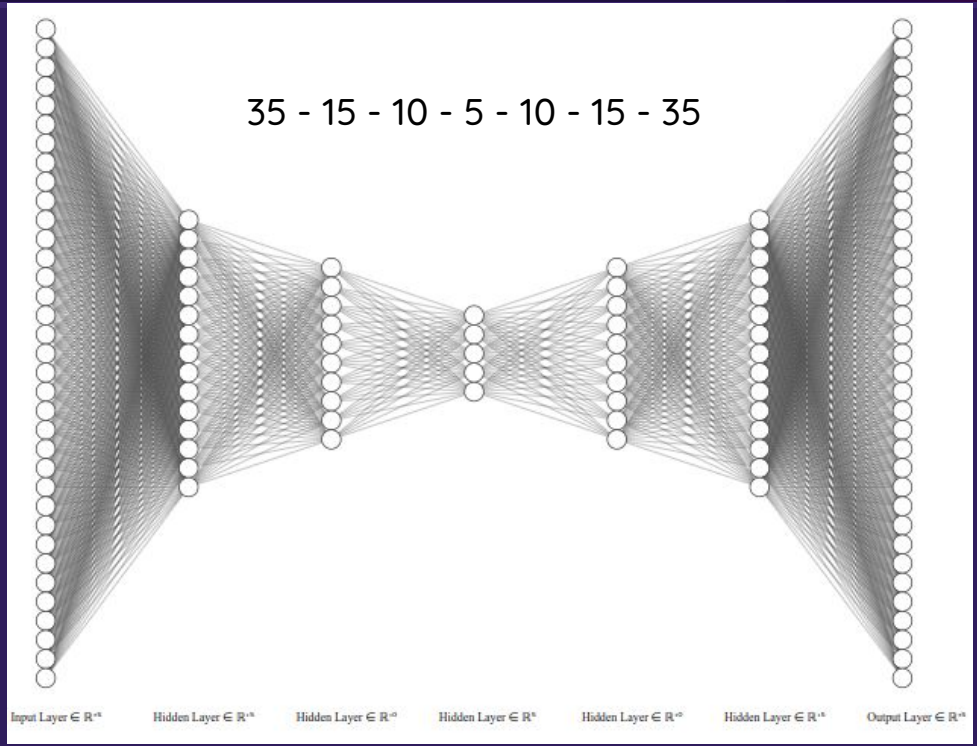
b) Implementar una variante que implemente un ”Denoising Autoencoder”.

1. Plantear una arquitectura de red conveniente para esta tarea. Explicar la elección.

2. Distorsionen las entradas en diferentes niveles y estudien la capacidad del Autoencoder de eliminar el ruido.

Arquitectura de la red 1.b.1

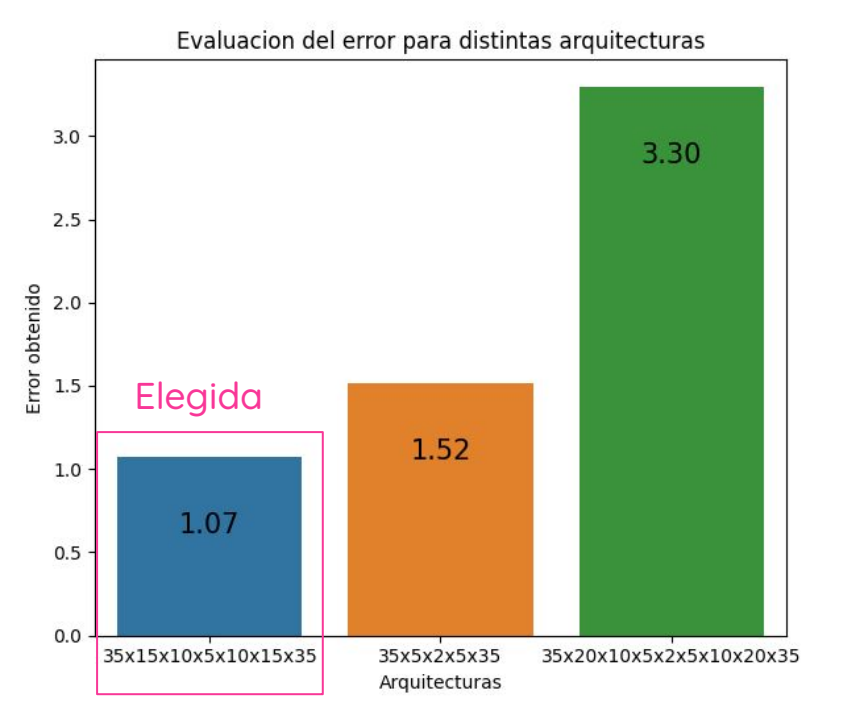
```
layer1 = Layer(15, 35)
layer2 = Layer(10, 15)
layer3 = Layer(5, 10)
layer4 = Layer(10, 5)
layer5 = Layer(15, 10)
layer6 = Layer(35, 15)
```



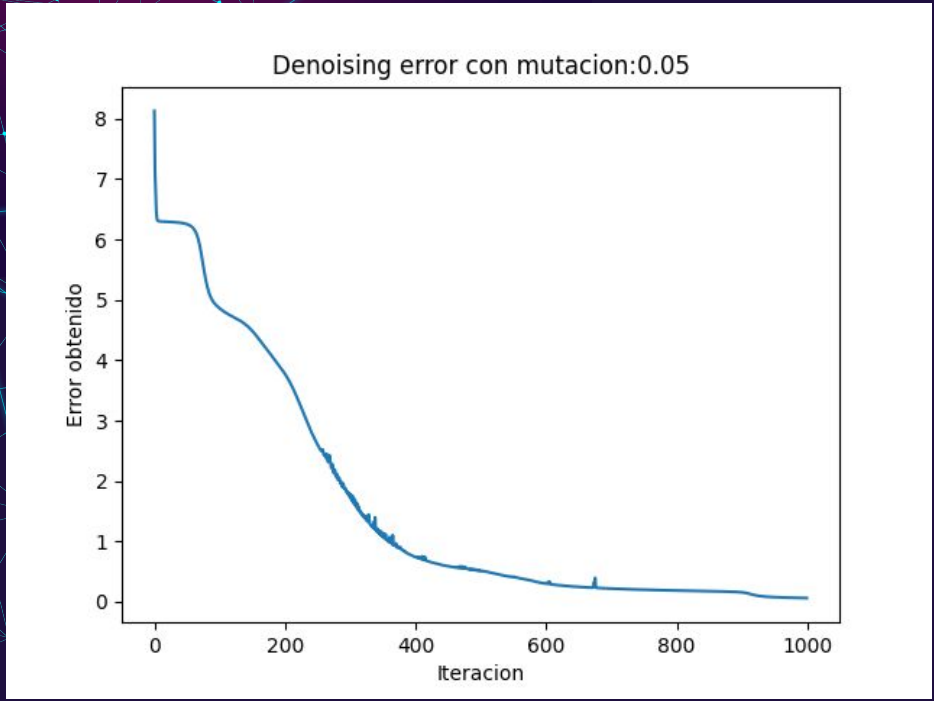
Evaluación de Arquitecturas 1.b.1

- Probamos **distintas arquitecturas** para encontrar una con el **error mínimo**
- Tradeoff de error mínimo contra tiempo de entrenamiento

- iters: 1000
- learning rate: 0.1

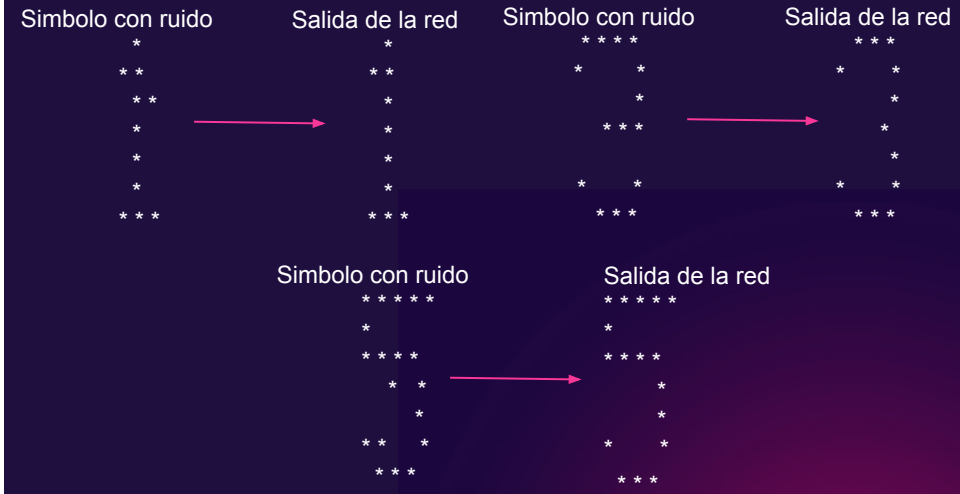


Resultados 1.b.2



Error min: 0.063

E.j. de prueba

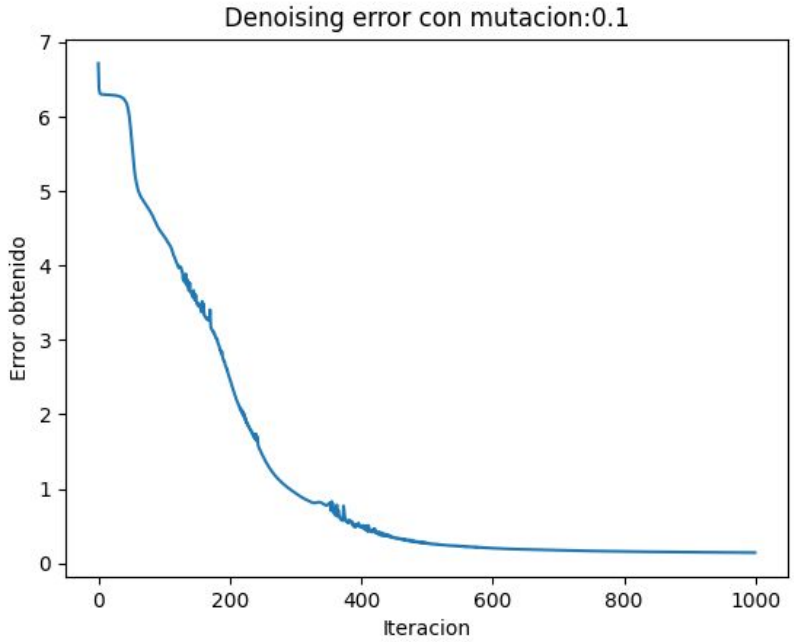


OBS: se entrenó a la red con 27 símbolos: 9 SIN ruido y 18 CON.

Resultados 1.b.2

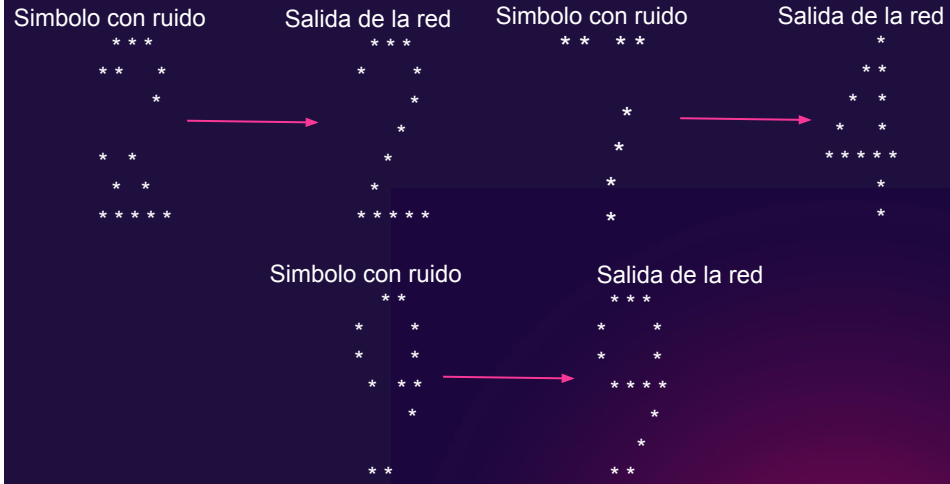
7/9

MAL! La salida esperada era un 7



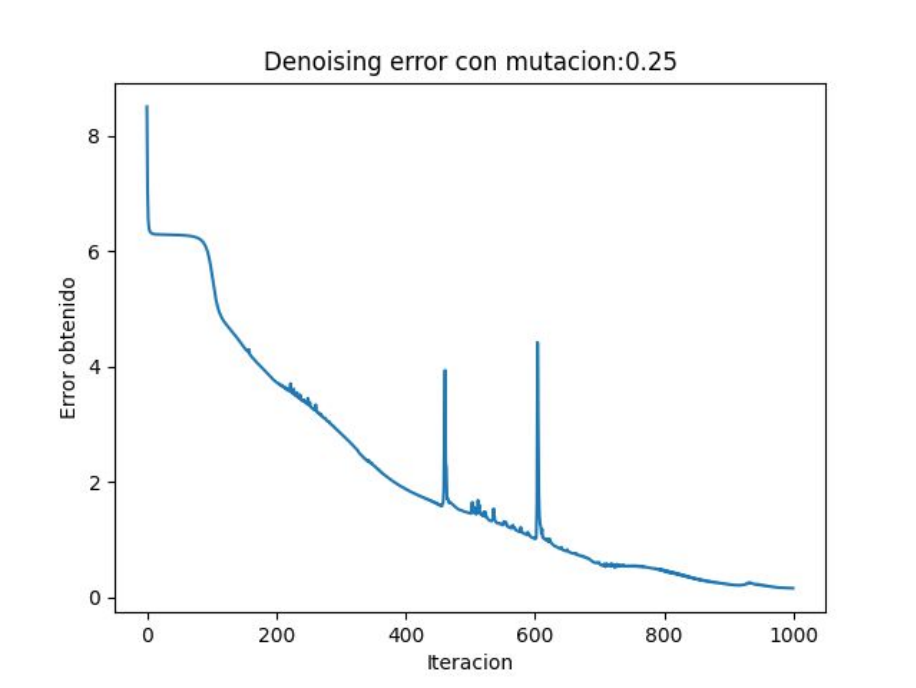
Error min: 0.159

E.j. de prueba



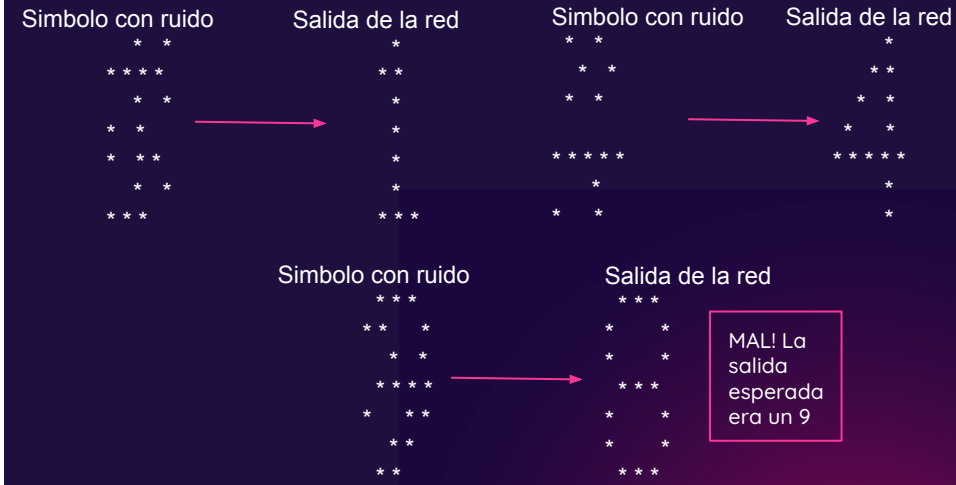
OBS: se entrenó a la red con 45 símbolos: 9 SIN ruido y 36 CON.

Resultados 1.b.2



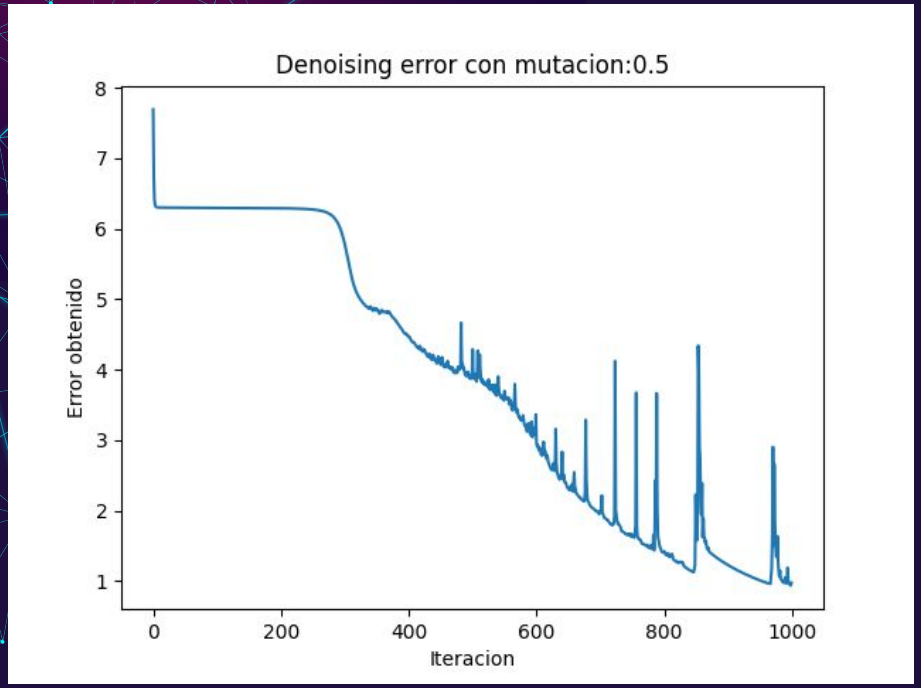
Error min: 0.274

E.j. de prueba



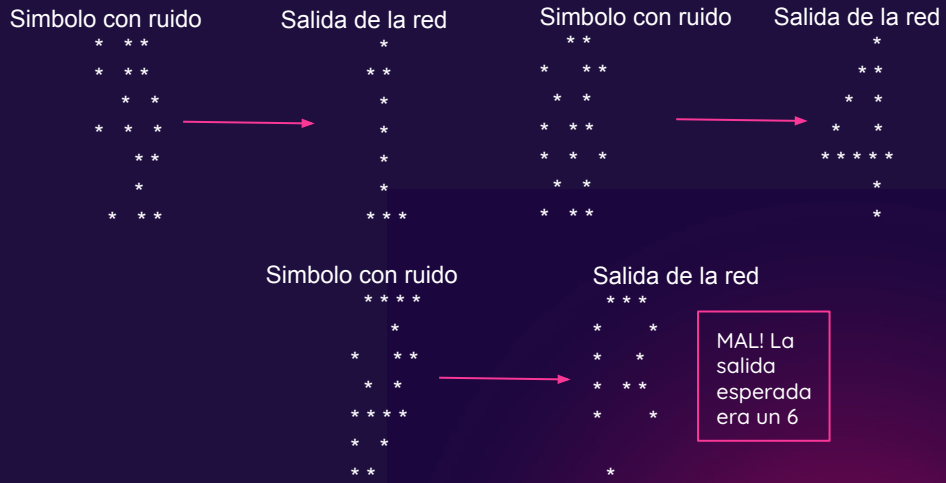
OBS: se entrenó a la red con 45 símbolos: 9 SIN ruido y 36 CON.

Resultados 1.b.2



Error min: 1.153

E.j. de prueba



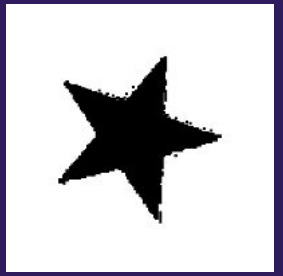
OBS: se entrenó a la red con 45 símbolos: 9 SIN ruido y 36 CON.

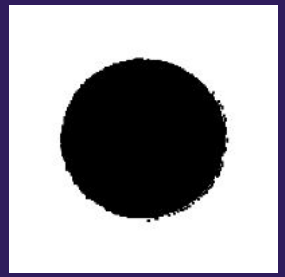
02 Generación de datos

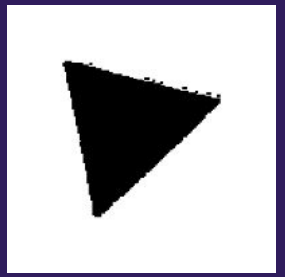
Con nuestro conjunto de datos

Conjunto de datos elegido

Para este punto se eligio el conjunto de dato Four Shapes de Kaggle. Este dataset incluye fotos de cuadrados, circulos, triangulos y estrellas. Aproximadamente 3750 de cada uno. Las figuras se encuentran en negro sobre blanco y sufren distintas transformaciones: rotacion, estiramiento, ruido, etc.







Conjunto de datos elegido

- 15000 imágenes de 200x200 en 3 canales (achicadas a 28x28 con un canal por limitaciones técnicas).
- Los valores de los pixeles se escalan entre 0 y 1 para facilitar el aprendizaje de la red
- Entrenado durante 50 epocas de a batches de a 64 individuos.
- Entrenado sobre un VAE diseñado en Keras.

Diseño de la red

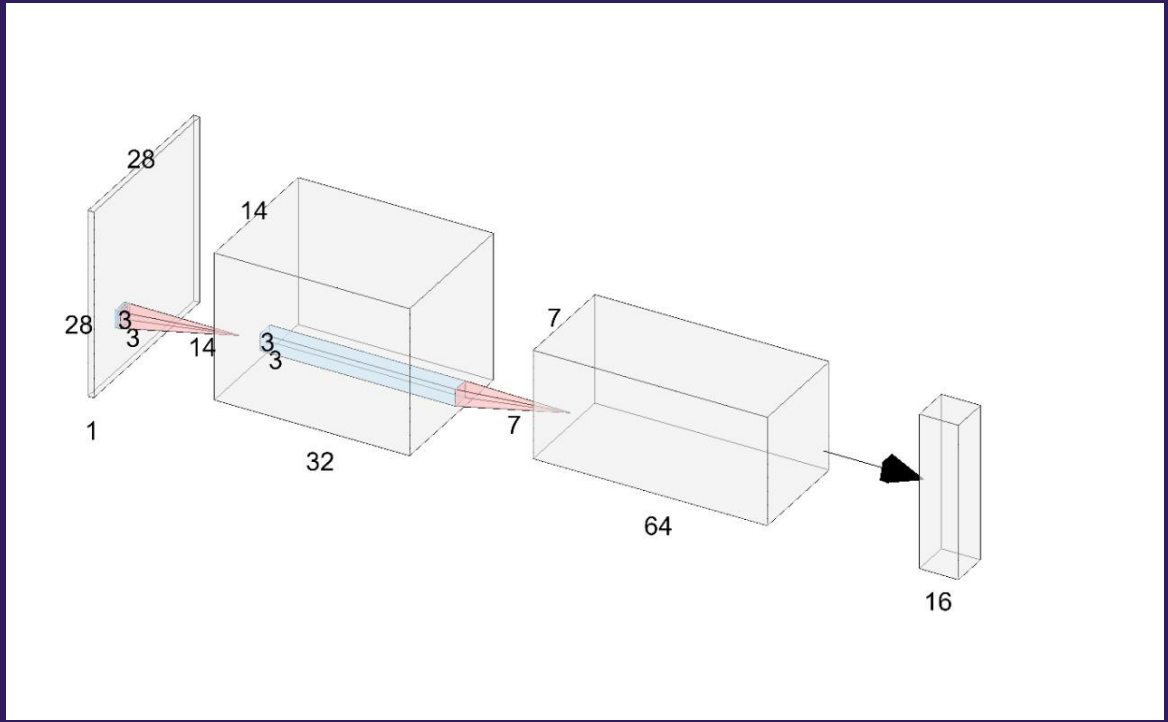
La red se diseño en Keras siguiendo su especificacion de una VAE para encodear números MNIST. Entre el decoder y el encoder se suman mas de 130000 parametros aprendibles!

https://keras.io/examples/generative/vae/

```
Model: "encoder"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 28, 28, 1)]  0           []

 conv2d (Conv2D)                (None, 14, 14, 32)   320         ['input_1[0][0]']

 conv2d_1 (Conv2D)              (None, 7, 7, 64)     18496       ['conv2d[0][0]']

 flatten (Flatten)              (None, 3136)         0           ['conv2d_1[0][0]']

 dense (Dense)                  (None, 16)           50192       ['flatten[0][0]']

 z_mean (Dense)                 (None, 2)            34          ['dense[0][0]']

 z_log_var (Dense)              (None, 2)            34          ['dense[0][0]']

 sampling (Sampling)            (None, 2)            0           ['z_mean[0][0]',
                                                                  'z_log_var[0][0]']

==================================================================================================
Total params: 69,076
Trainable params: 69,076
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 2)]               0

 dense_1 (Dense)             (None, 3136)              9408

 reshape (Reshape)           (None, 7, 7, 64)          0

 conv2d_transpose (Conv2DTra  (None, 14, 14, 64)       36928
 nspose)

 conv2d_transpose_1 (Conv2DT  (None, 28, 28, 32)       18464
 ranspose)

 conv2d_transpose_2 (Conv2DT  (None, 28, 28, 1)        289
 ranspose)

=================================================================
Total params: 65,089
Trainable params: 65,089
Non-trainable params: 0
_________________________________________________________________
```

Diseño de la red - Encoder

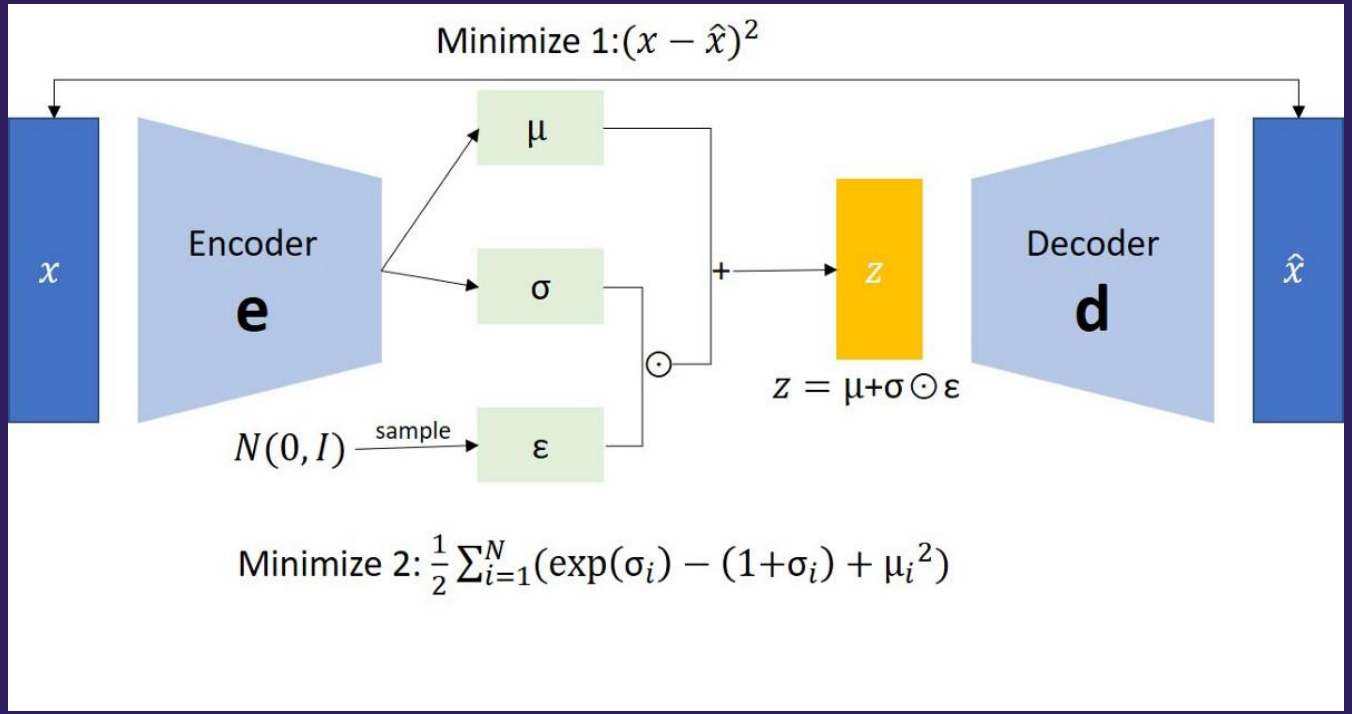


Diseño de la red

El encoder y el decoder se conectan por medio de un proceso de Sampling

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

    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
```

Diseño de la red



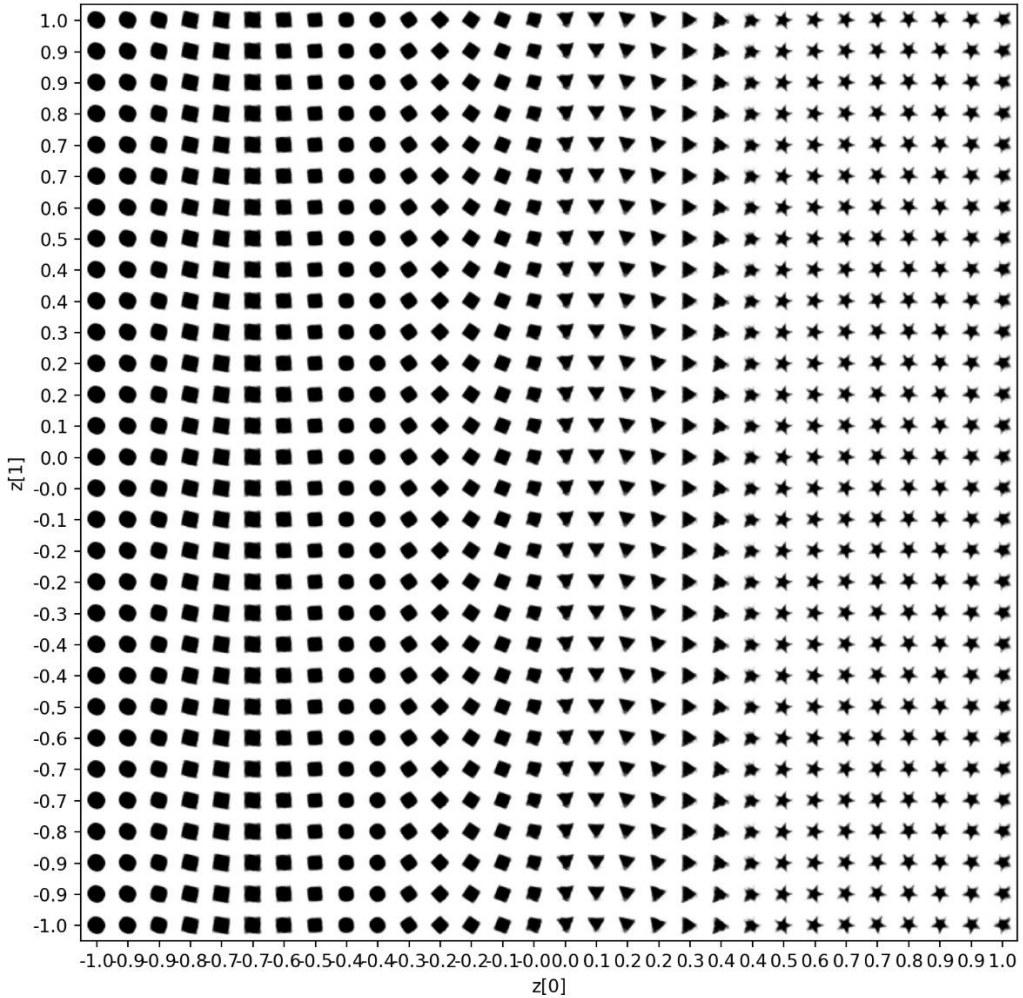
Diseño de la red

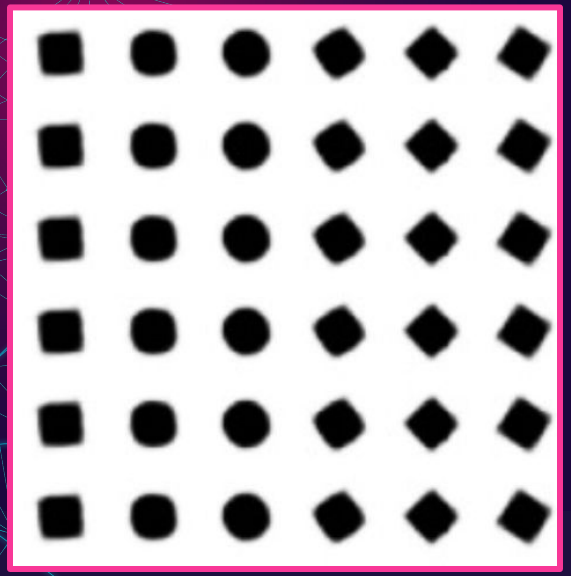
- Las capas convolucionales usan kernel de 3x3 con stride de 2 y padding SAME
- Todas las activaciones son por ReLU **excepto** la última capa del decoder que es sigmoide.
- Cada capa convolucional en la encoder se espeja con una capa convolucional transpuesta en el decoder

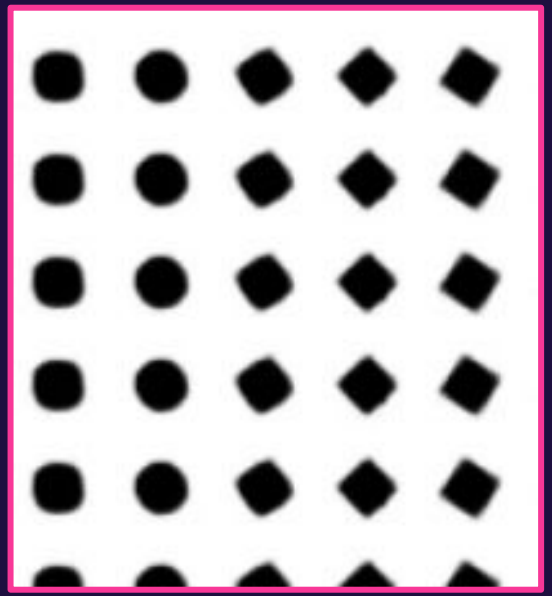
Diseño de la red - Metricas

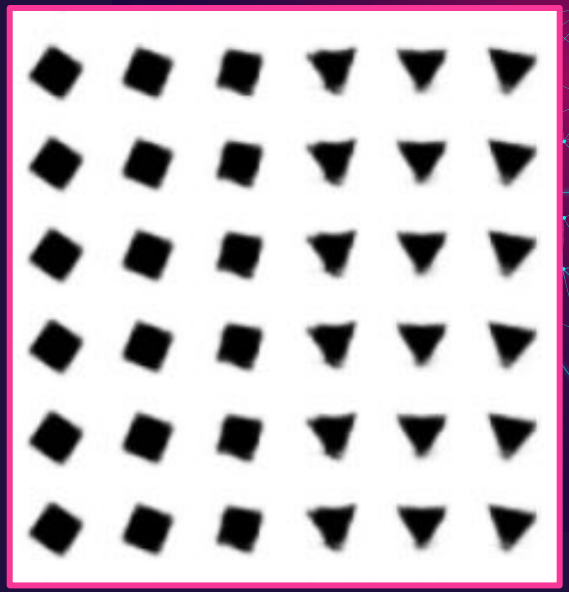
Epoca a epoca, se mide las siguientes metricas:

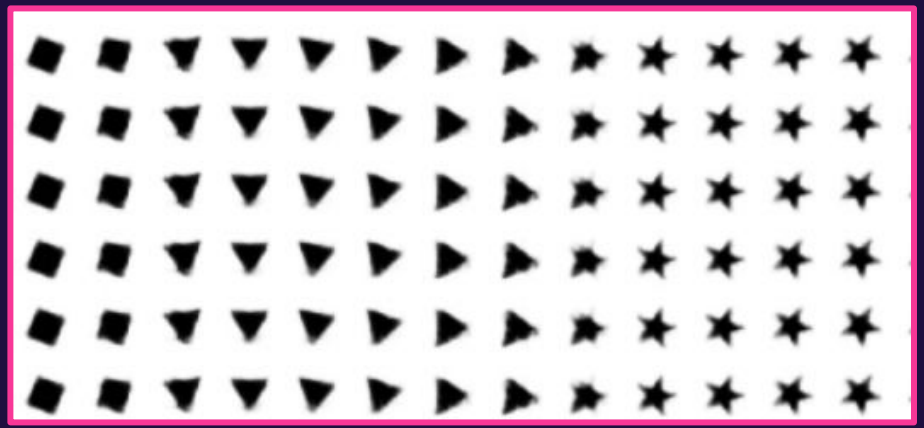
- **reconstruction_loss:** El calculo de loss de toda la vida, representa la media entre la diferencia de los x de entrada y los x' de salida.
- **kl_loss:** Perdida en base a la convergencia KL. Busca minimizar la diferencia entre las funciones de distribucion
- **loss** = recontruction_loss + kl_loss


z[1]
1.0
0.9
0.9
0.8
0.7
0.7
0.6
0.5
0.4
0.4
0.3
0.2
0.2
0.1
0.0
-0.0
-0.1
-0.2
-0.2
-0.3
-0.4
-0.4
-0.5
-0.6
-0.7
-0.7
-0.8
-0.9
-0.9
-1.0
-1.0 -0.9 -0.9 -0.8 -0.7 -0.7 -0.6 -0.5 -0.4 -0.4 -0.3 -0.2 -0.2 -0.1 -0.0 0.0 0.1 0.2 0.2 0.3 0.4 0.4 0.5 0.6 0.7 0.7 0.8 0.9 0.9 1.0
z[0]









Generación de datos

Gracias

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**