SIA - TP5
Deep Learning

Grupo 1

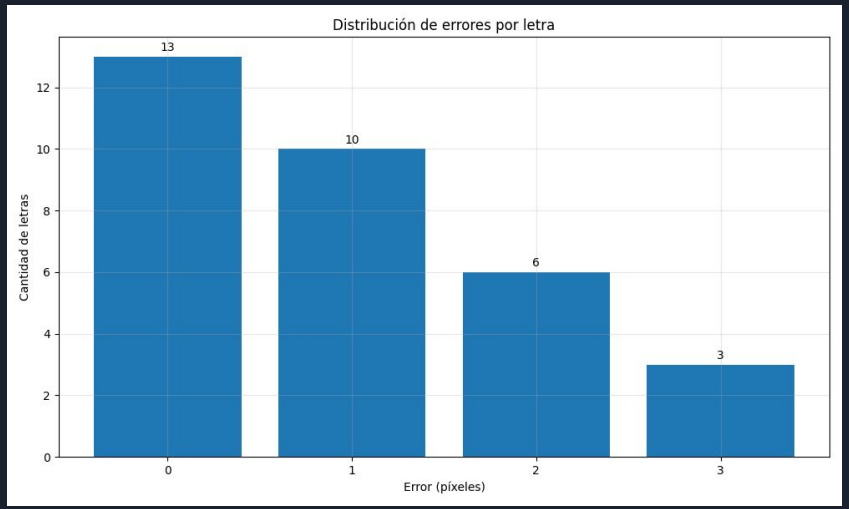
Alberto Bendayan
Tobias Ves Losada
Cristian Tepedino

Autoencoder

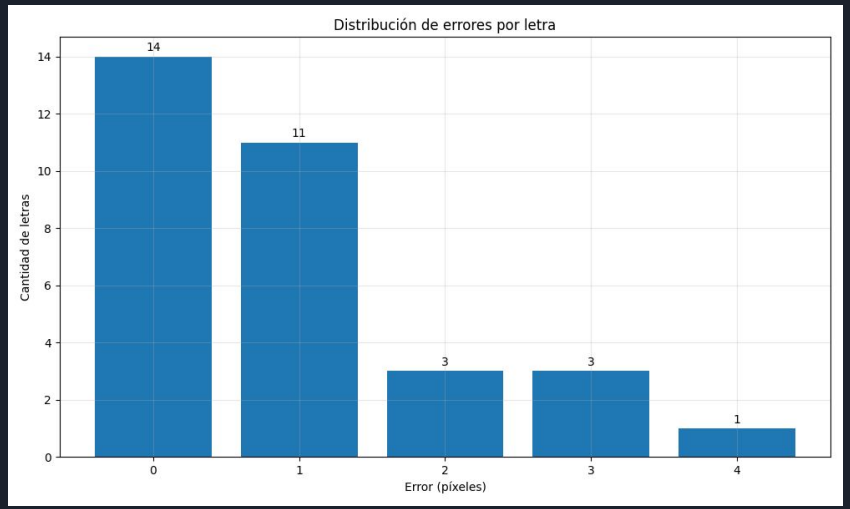
Primera etapa

- Perceptrón Multicapa
- Learning rate fijo
- Función de activación: sigmoid
- Épocas: 50.000 - 100.000
- Capas
  - [35, X, 2, X, 35]
  - [35, Z, Y, 2, Y, Z, 35]
- Optimizador: Adam
- Loss function: MSE
- Early Stopping: 8.000 épocas sin mejoras

Primeros resultados - 5 capas

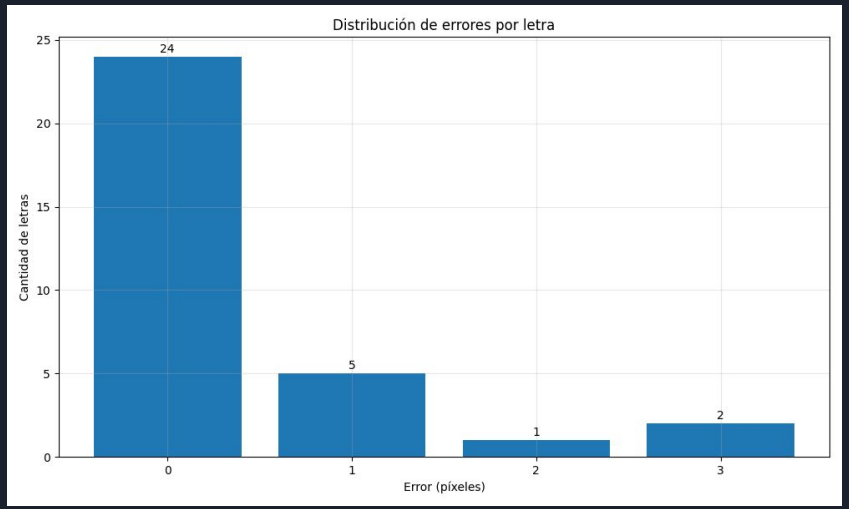


Capas: [35, 12, 2, 12, 35]
LR: 0.0024
Épocas: 50.000

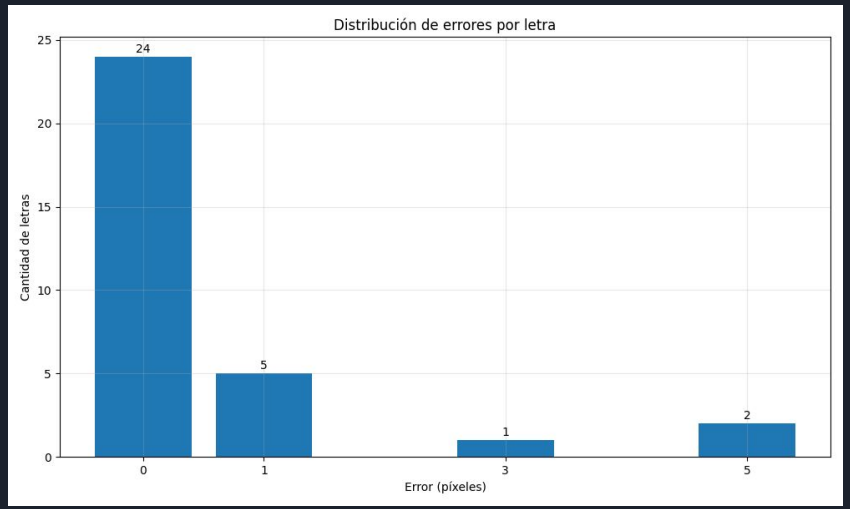


Capas: [35, 10, 2, 10, 35]
LR: 0.0033
Épocas: 50.000

Primeros resultados - 7 capas

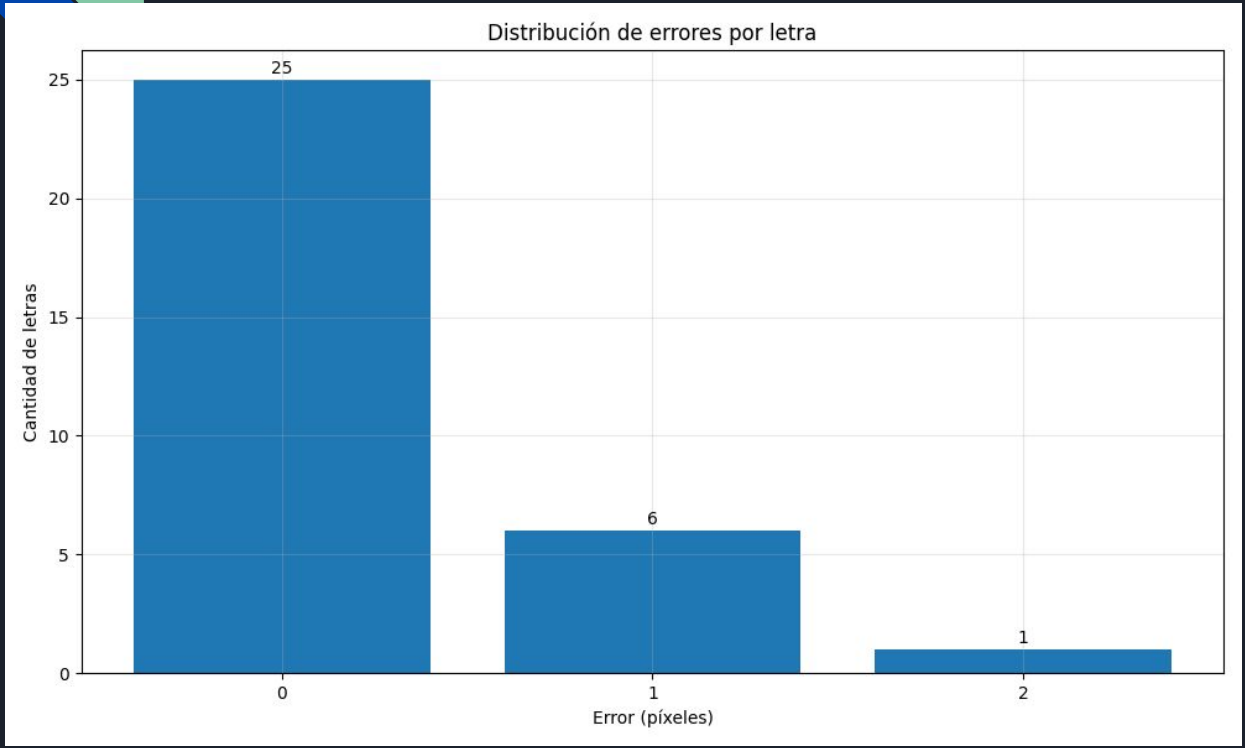


Capas: [35, 18, 6, 2, 6, 18, 35]
LR: 0.0038
Épocas: 50.000



Capas: [35, 14, 6, 2, 6, 14, 35]
LR: 0.0024
Épocas: 100.000

Primeros resultados - El mejor



Capas: [35, 18, 6, 2, 6, 18, 35]
LR: 0.0024
Épocas: 100.000

Cómo mejoramos esto?

El resultado anterior era bueno, pero, lo podemos mejorar?

Propuestas:

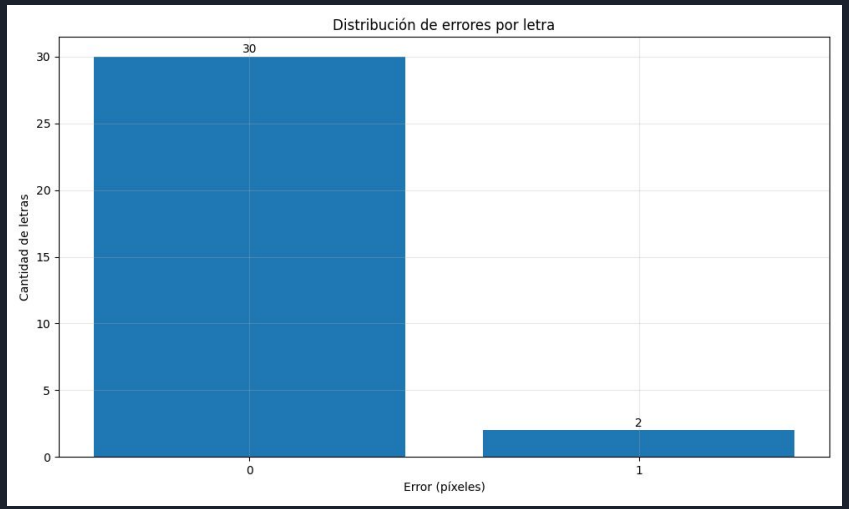
- Learning rate adaptativo
- Loss function: Binary Cross Entropy

Además, guardamos los pesos por capas para cierta configuración de capas para que la próxima vez que se corra igual, se inicialice con estos pesos.

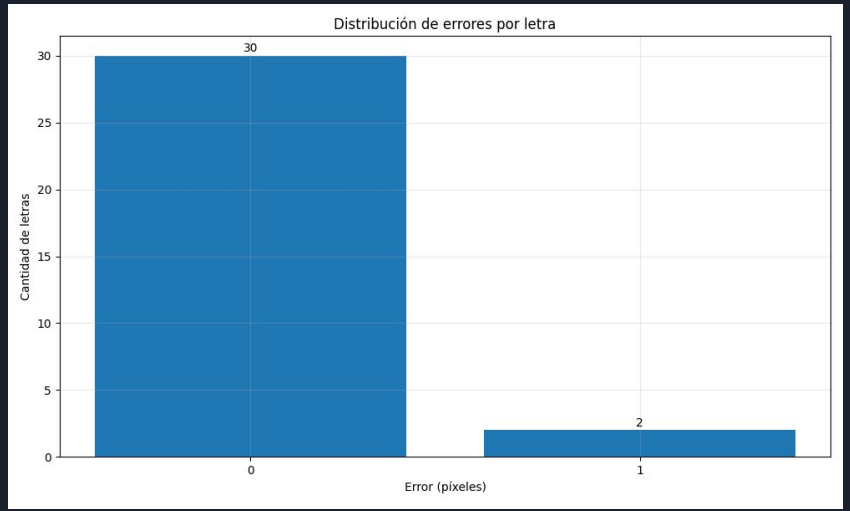
Segunda etapa

- Perceptrón Multicapa
- Learning rate adaptativo
- Función de activación: sigmoid
- Épocas: 5.000 (entre 10 y 20 veces menos!)
- Capas
  - [35, X, 2, X, 35]
- Optimizador: Adam
- Loss function: Binary Cross Entropy

Primeras pruebas - 5 capas



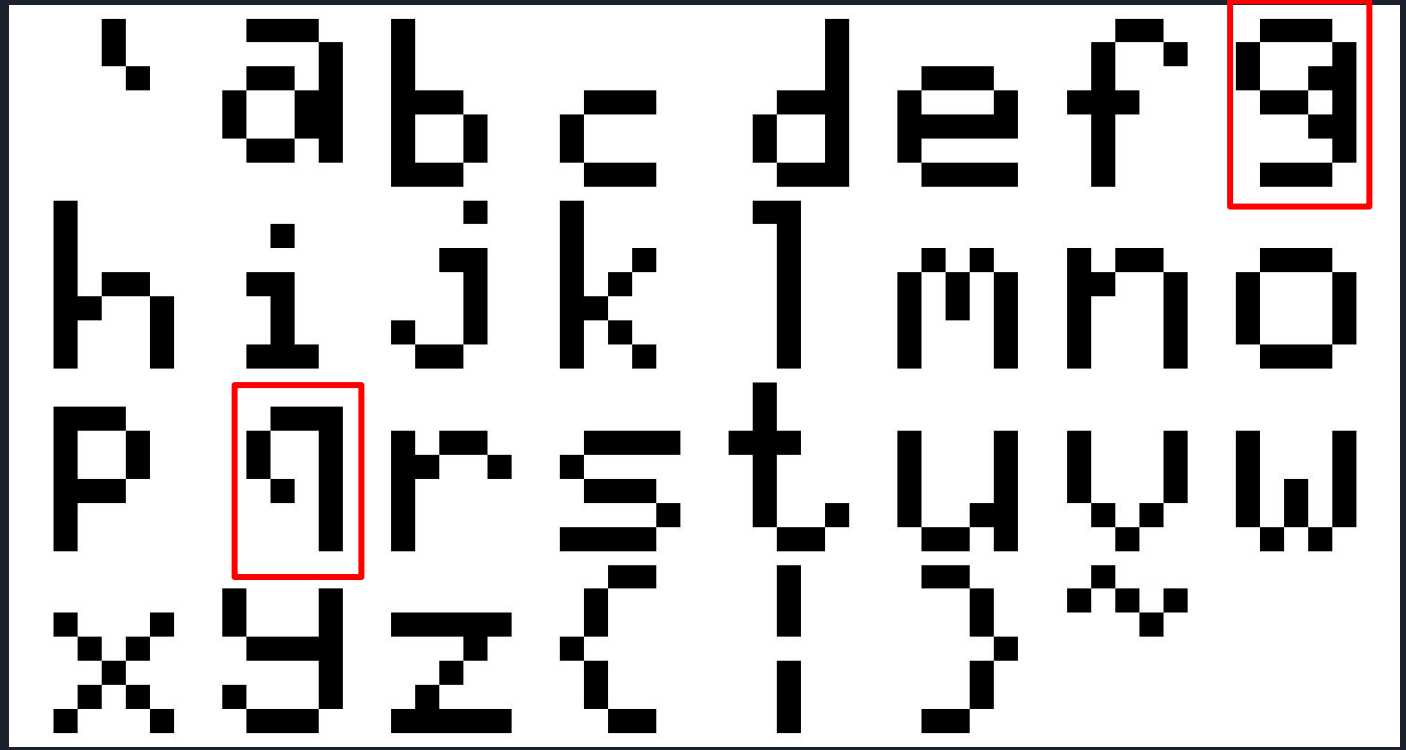
Capas: [35, 14, 2, 14, 35]
LR inicial: 0.0076
Pesos: aleatorios



Capas: [35, 18, 2, 18, 35]
LR inicial: 0.009
Pesos: aleatorios

Errores

Capas: [35, 18, 2, 18, 35]
LR inicial: 0.009
Pesos: aleatorios

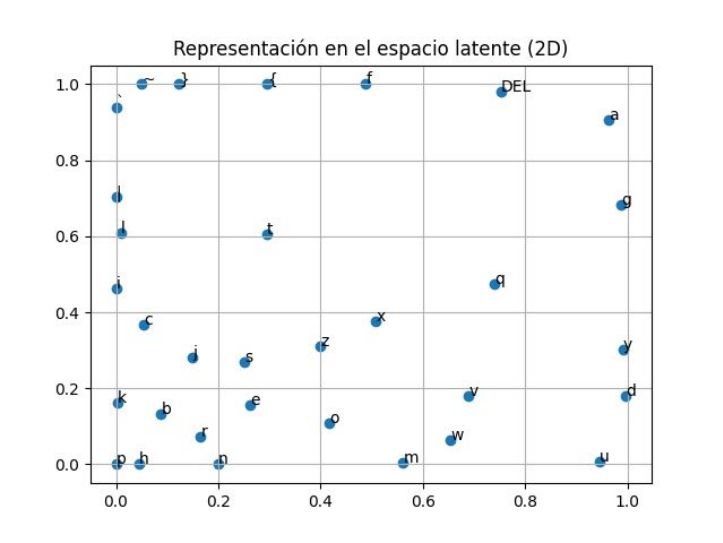


Ejecutando de vuelta con los pesos finales

Capas: [35, 18, 2, 18, 35]
LR: 0.009
Épocas: 5.000
Pesos: finales de la ejecución anterior

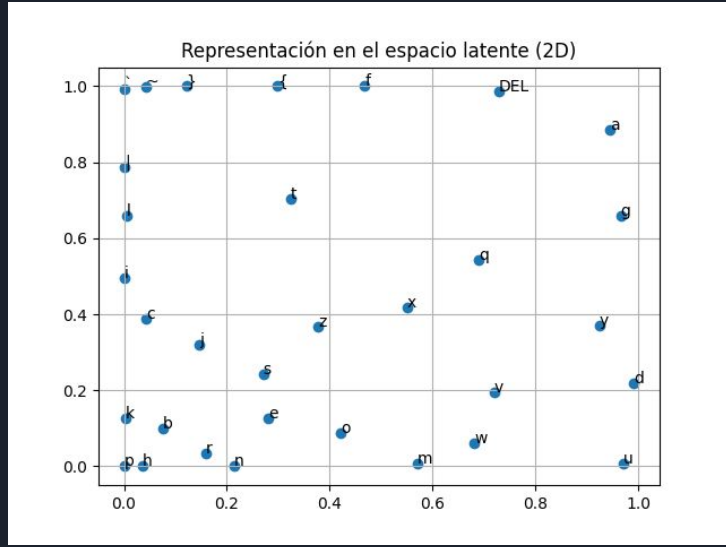
Espacios latentes

1ra ejecución



Capas: [35, 18, 2, 18, 35]
LR inicial: 0.009
Pesos: aleatorios

2da ejecución

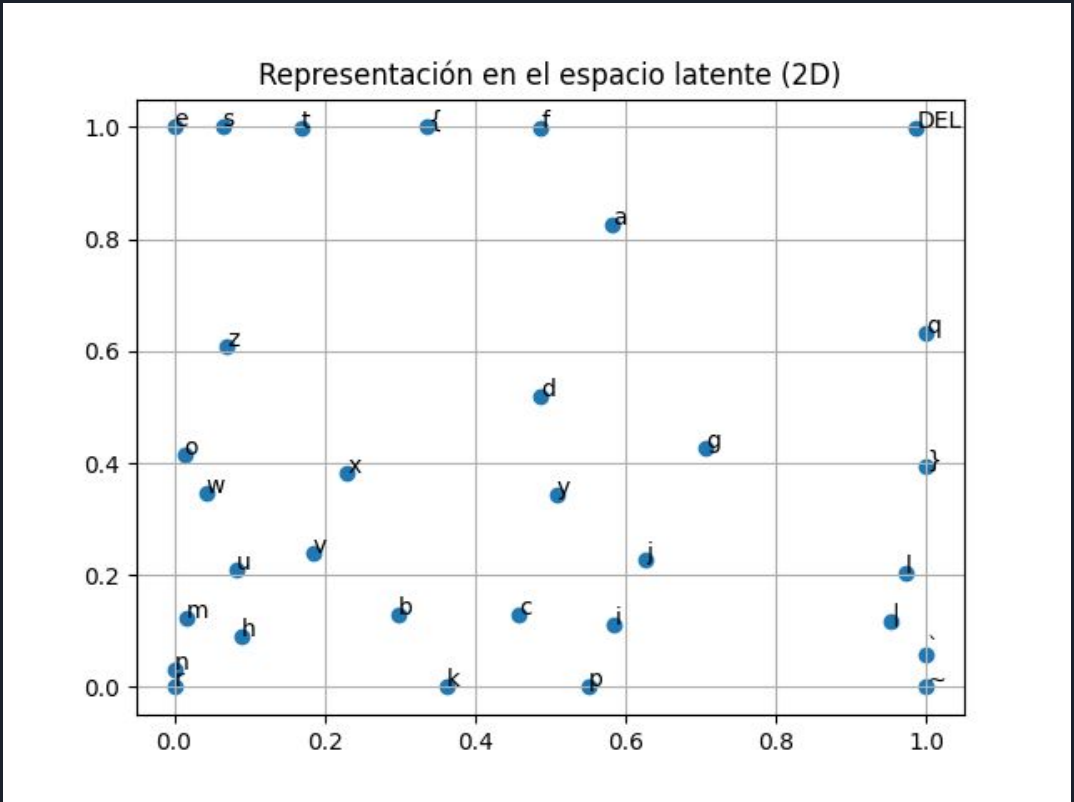


Capas: [35, 18, 2, 18, 35]
LR inicial: 0.009
Pesos: finales de la ejecución anterior

Prueba final - Distribución de errores

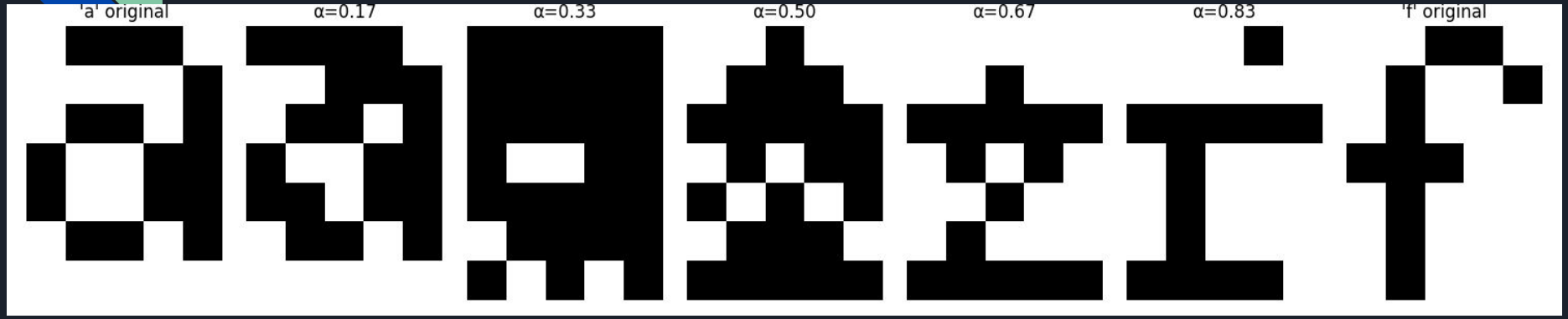
Capas: [35, 17, 2, 17, 35]
LR: 0.009
Épocas: 10.000
Pesos: aleatorios

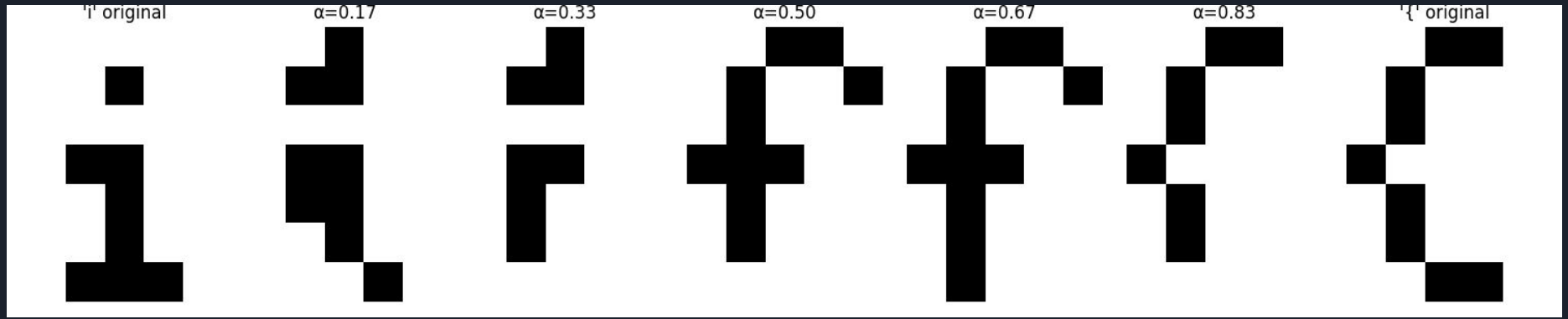
Prueba final - Espacio latente



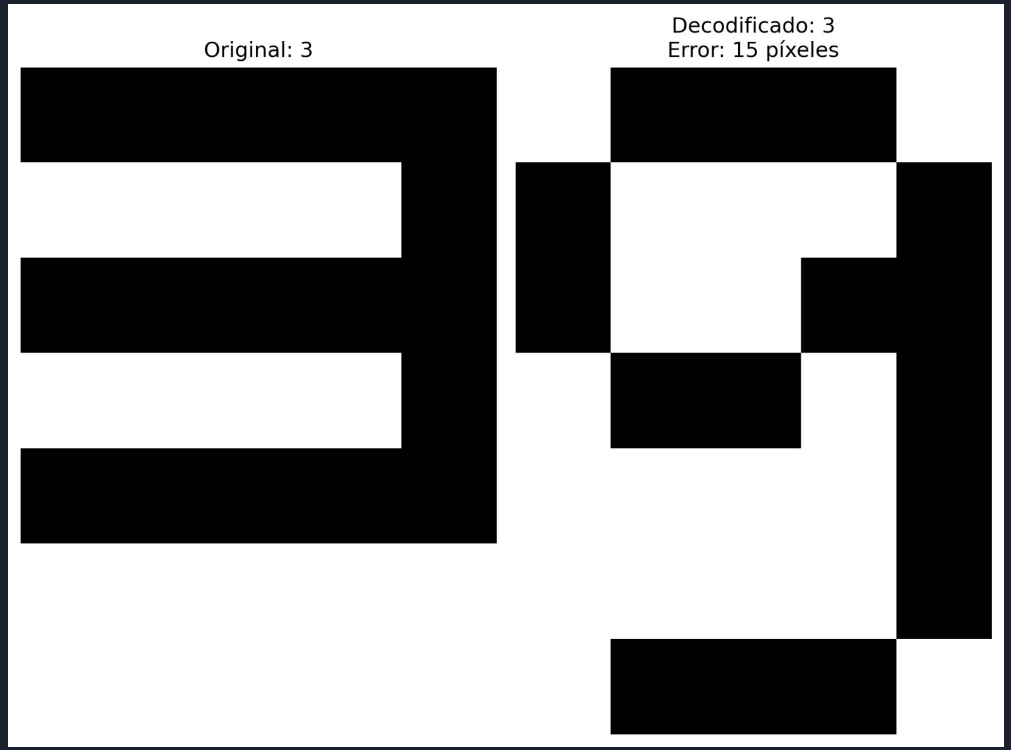
Capas: [35, 17, 2, 17, 35]
LR: 0.009
Épocas: 10.000
Pesos: aleatorios

Letras generadas por la red



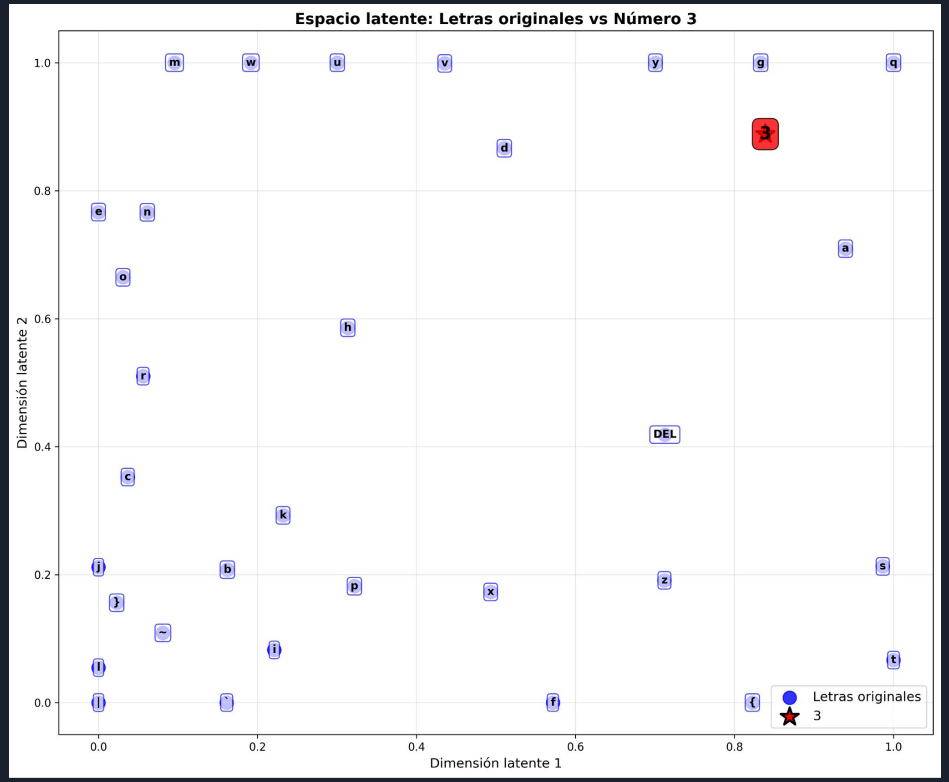


Detección de outliers



Capas: [35, 17, 2, 17, 35]

Detección de outliers



Conclusiones

- Variar el valor learning rate (fijo) no es la solución a todo

- Aumentar el número de capas puede ser perjudicial

- Aumentar las épocas tampoco es la mejor solución

Denoising Autoencoder

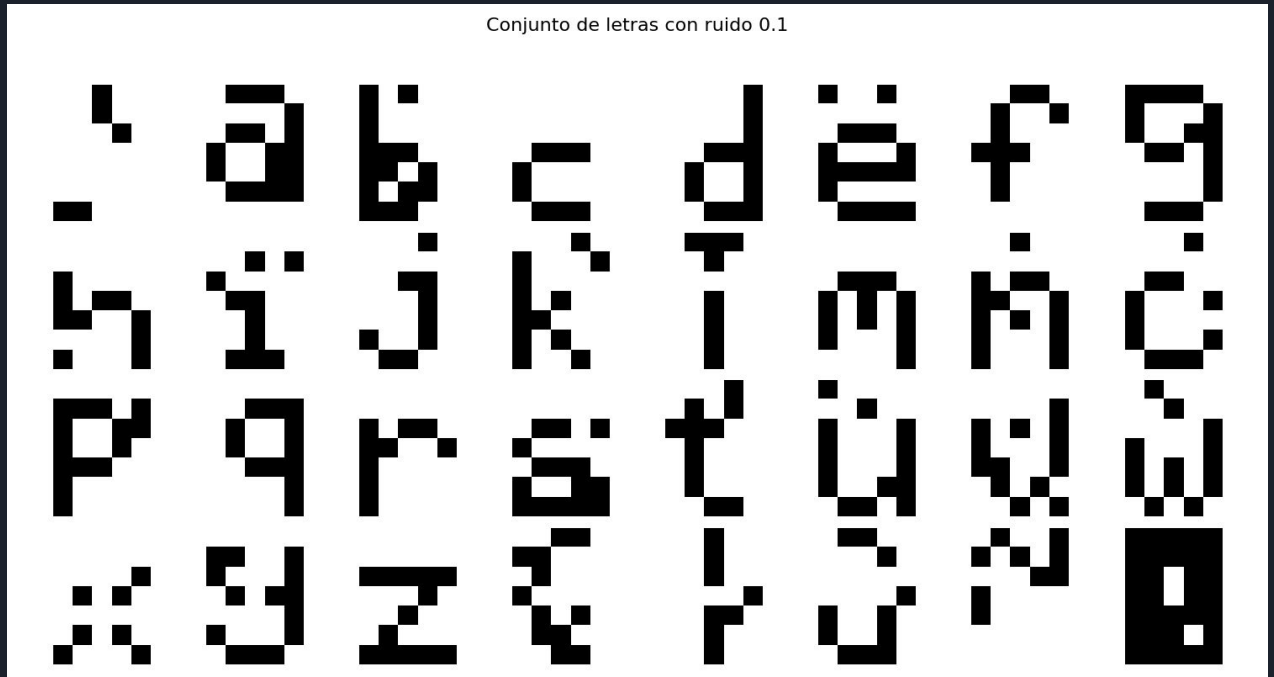
Parámetros

Seguimos con los mismos parámetros con los que el autoencoder aprendió sin errores:

- Capas:
  - Encoder: 35 -> 17
  - Espacio Latente: 2
  - Decoder: 17 -> 35
- Learning rate inicial 0.009
- 10000 epochs
- Pesos iniciales aleatorios

Ruido

- Salt And Pepper
- Variamos el porcentaje de probabilidad para ver cómo se desempeña

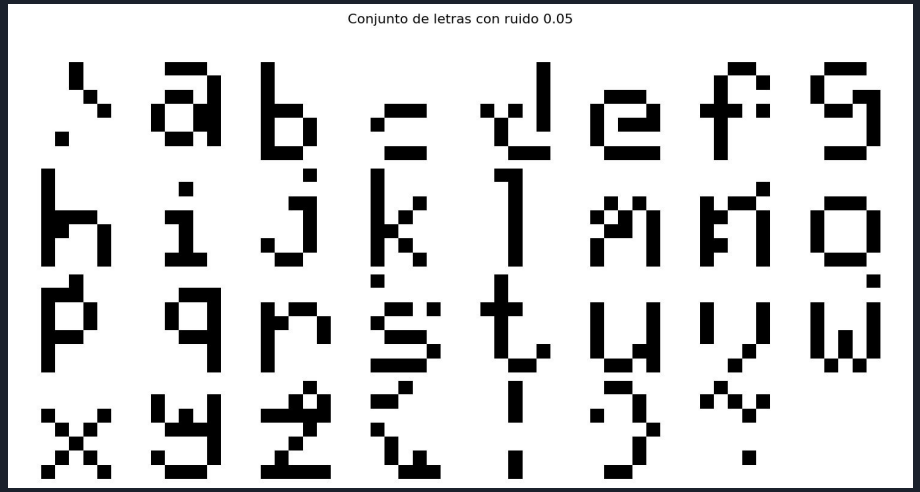


Entrenamiento

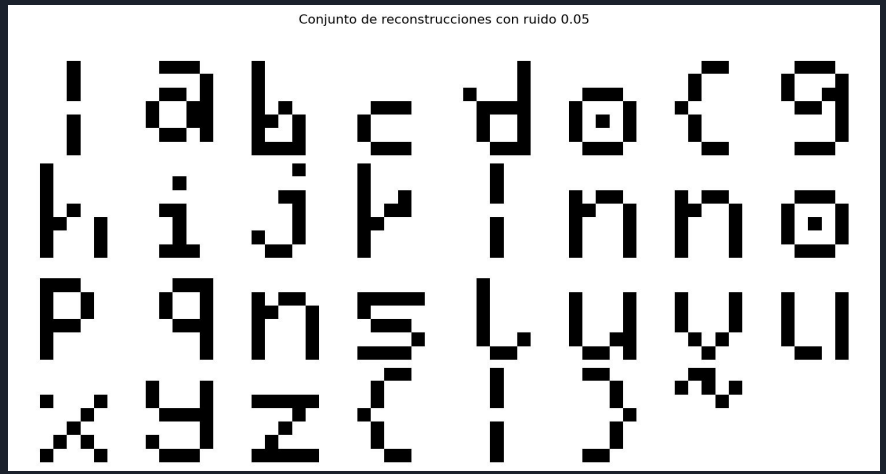
- Le pasamos el dataset X y la función de ruido al autoencoder
- Al entrenar, para cada epoch, obtiene X’ = ruido(X, p)
- Para esa epoch, usa los x’ ∈ X’ como input y el x ∈ X asociado como output esperado
- El valor de p se mantiene fijo durante todo el entrenamiento
- Luego, probamos con nuevos conjuntos de letras con ruido para ver que tan bien aprendió

Resultados del denoising

Entrenado con probabilidad de ruido 0.05



Se generó este alfabeto con ruido tras entrenar para probar…



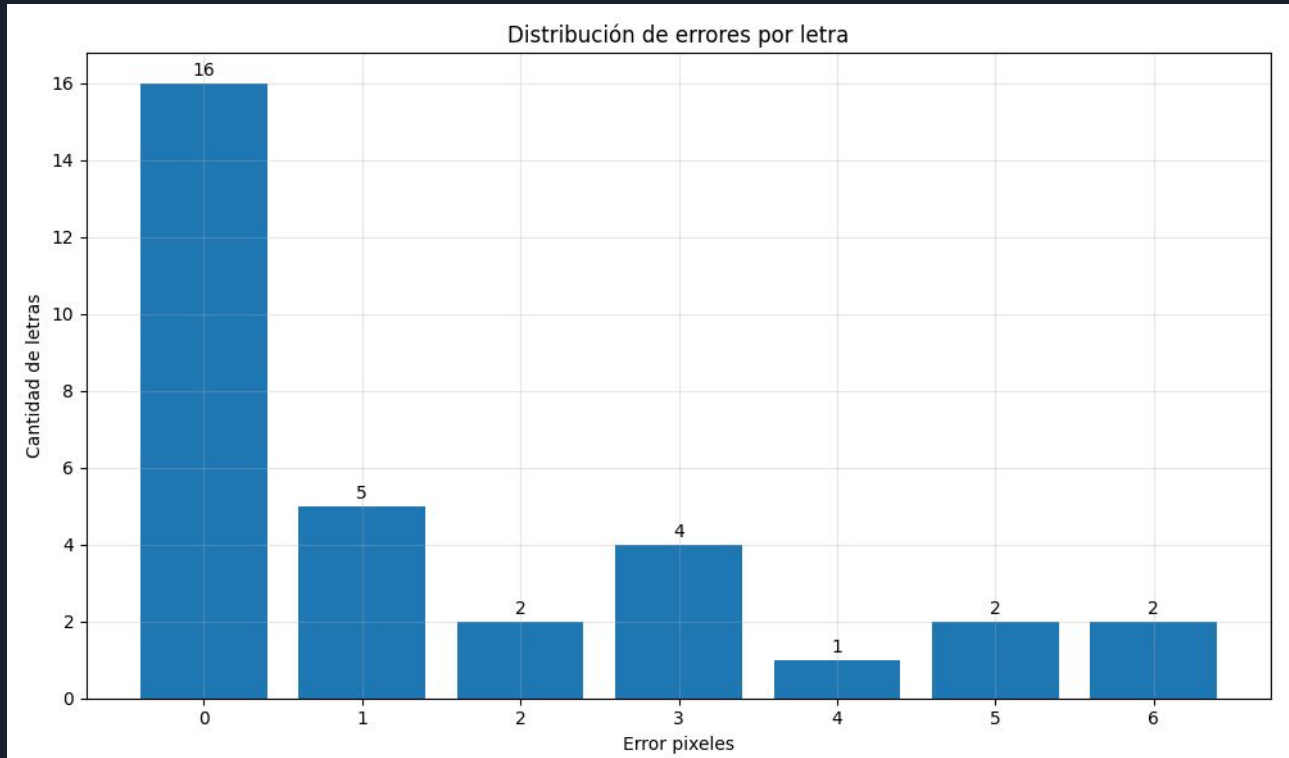
En general, las reconstrucciones no salieron muy bien

Evaluando el desempeño

Pudo reconstruir bien la mitad de las letras

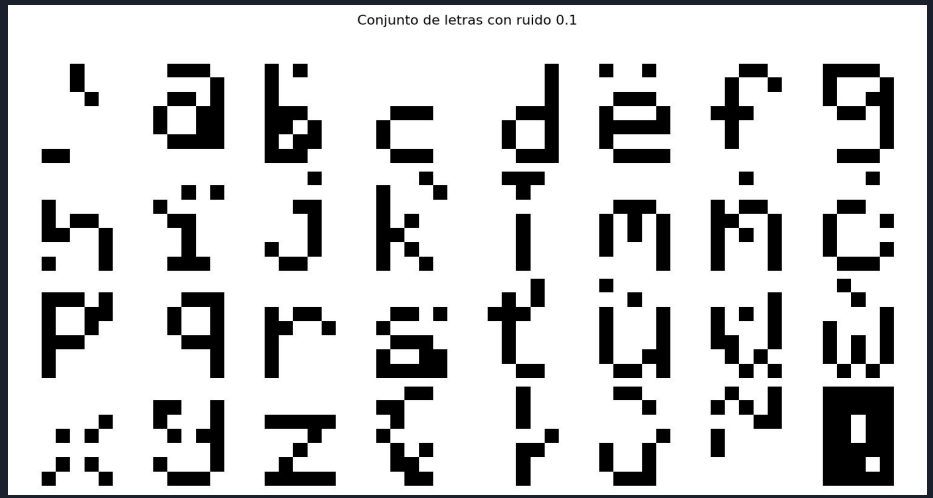
Sin embargo, algunas tienen bastantes pixeles de error

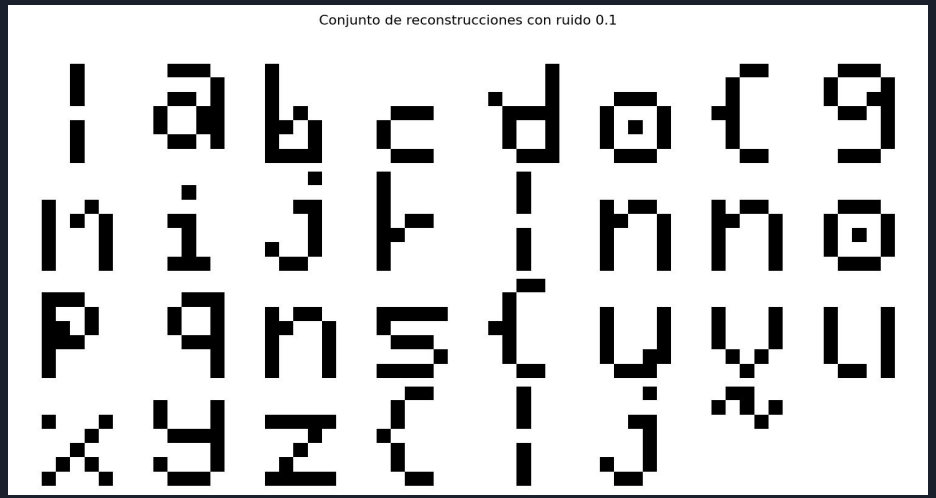
¿Qué pasa si entrenamos con más ruido?



Resultados del denoising

Entrenado con probabilidad de ruido 0.1

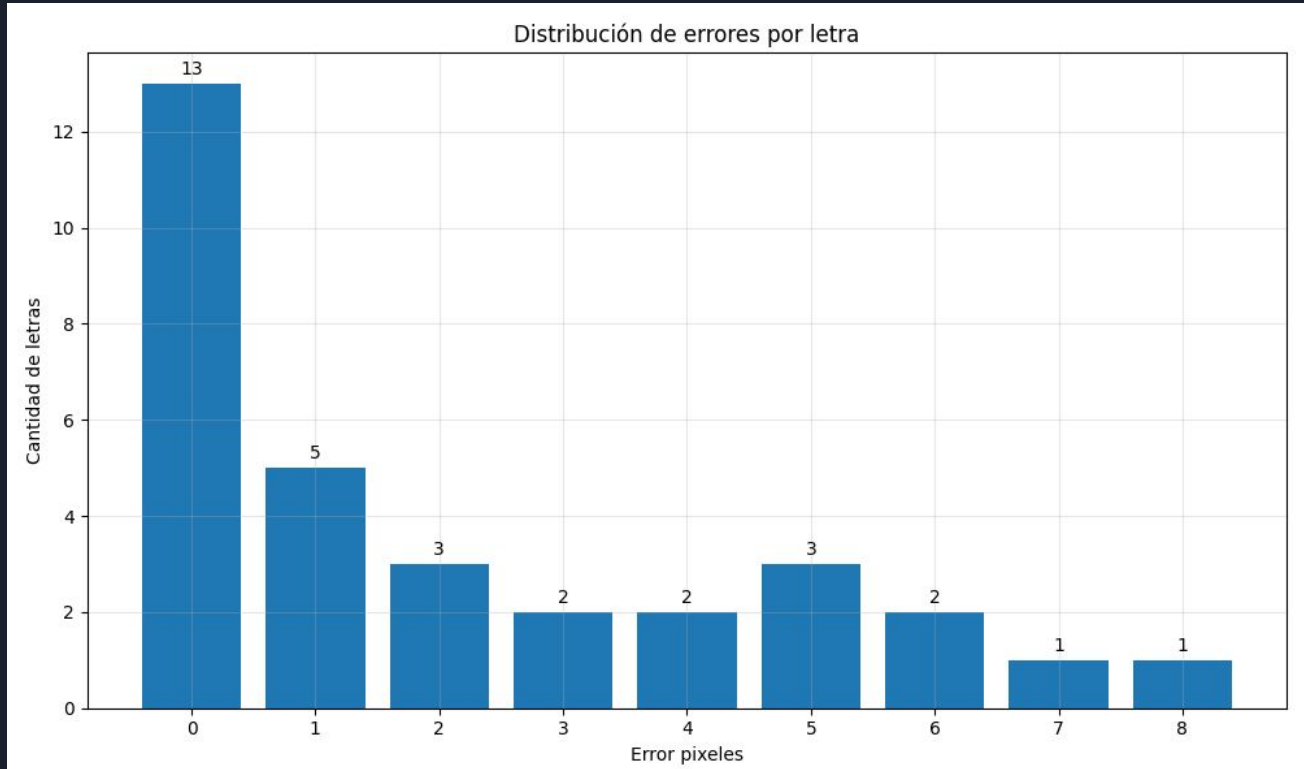




Evaluando el desempeño

Sigue con bastante error

Además, 8 pixeles es una diferencia importante



¿Cómo se puede mejorar?

Si bien puede reconstruir bien algunas letras, tiene muchos errores

A veces no termina entendiendo bien la forma original de la letra

Input:  Output: 

Otras veces, el ruido lo lleva a clasificarla como una letra distinta:

Input:  Output: 

(era una m)

Idea: Preentrenar el DAE

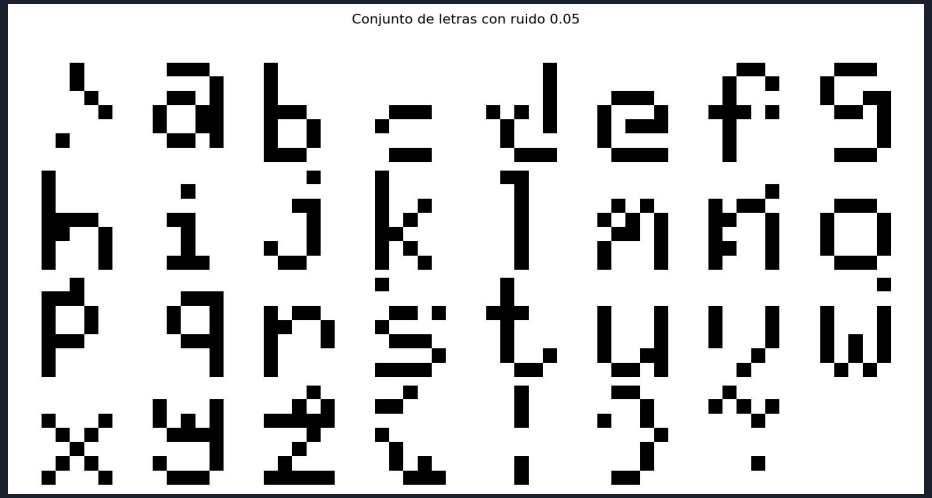
- Primero, entrenamos un autoencoder (normal) con el dataset sin ruido para que aprenda el dataset
- Después, guardamos los pesos de ese AE y usamos eso como pesos iniciales al entrenar el DAE

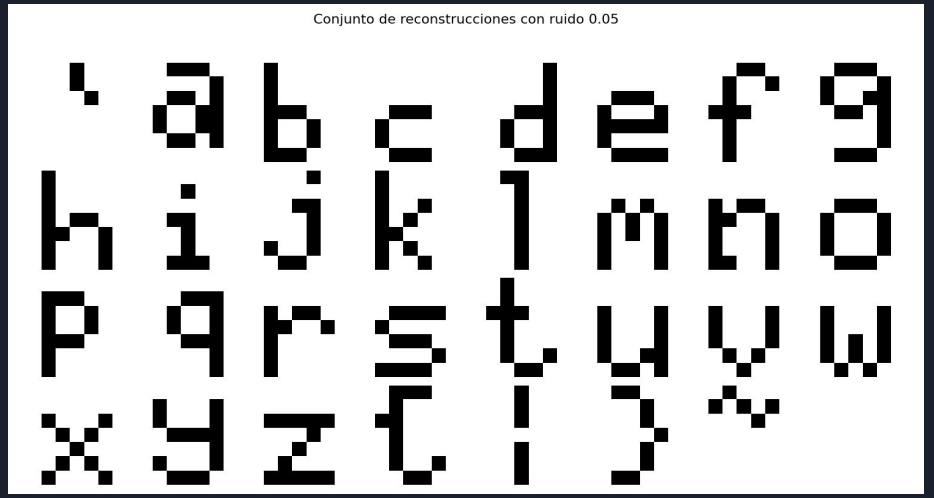
- Lo que esperamos es que así el DAE pueda entender mejor el dataset que tiene que reconstruir

- Mantuvimos los mismos parámetros que antes

Resultados del denoising

Entrenado con probabilidad de ruido 0.05 +
Preentrenamiento como AE

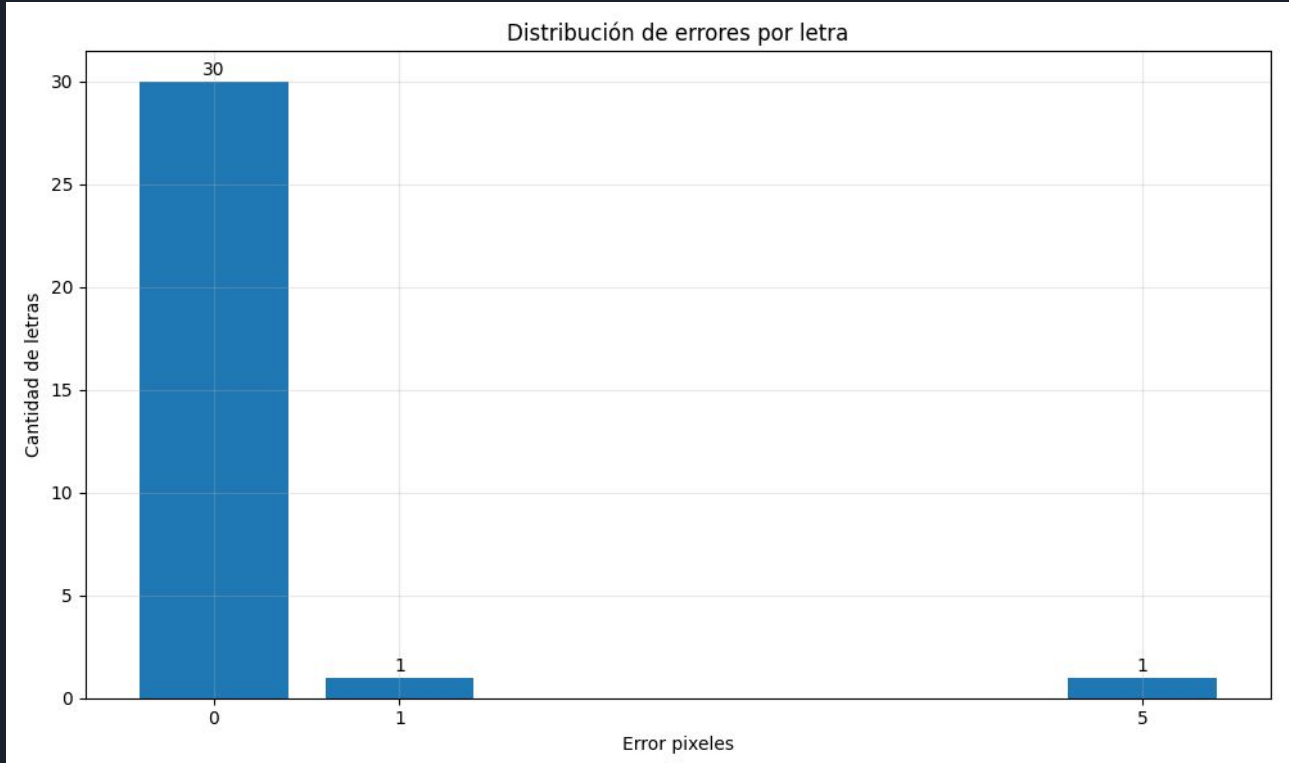




Evaluando el desempeño

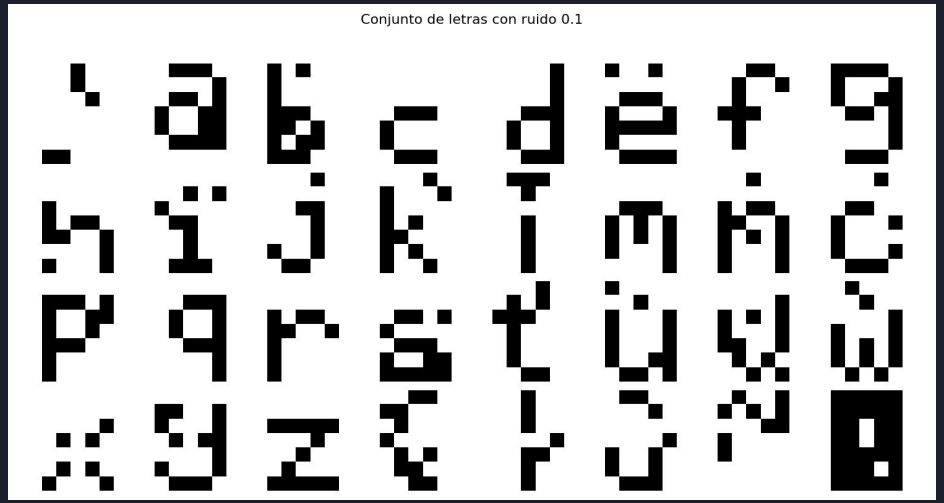
Mejoro bastante

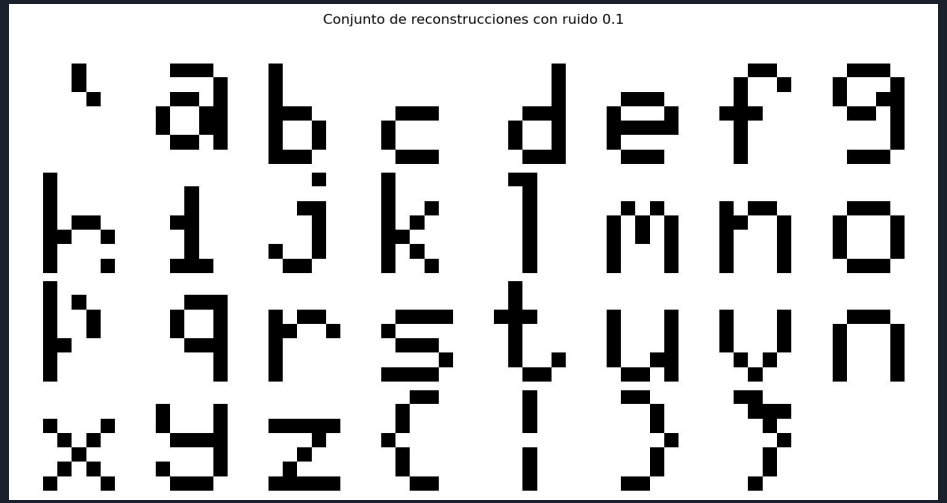
¿Se mantendrá ante mayor ruido?



Resultados del denoising

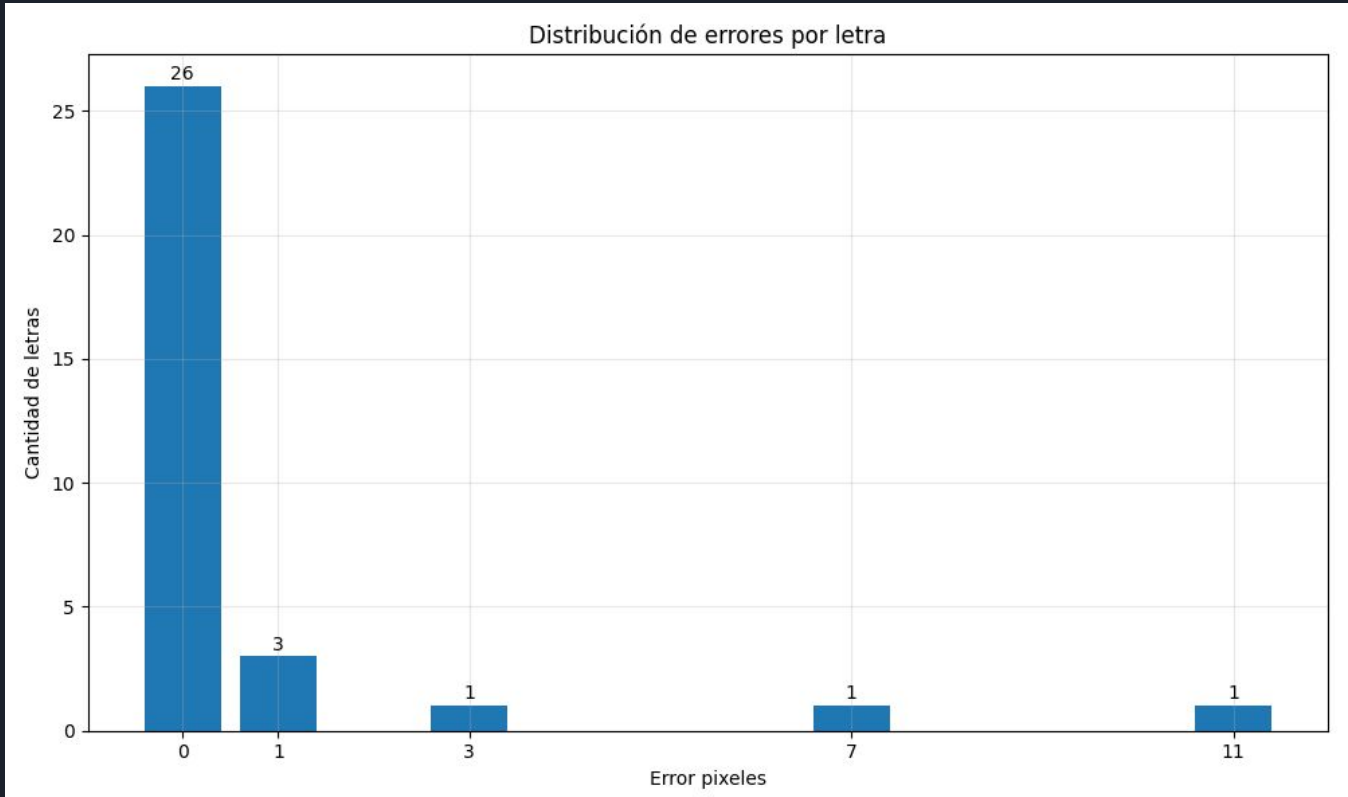
Entrenado con probabilidad de ruido 0.1 +
Preentrenamiento como AE





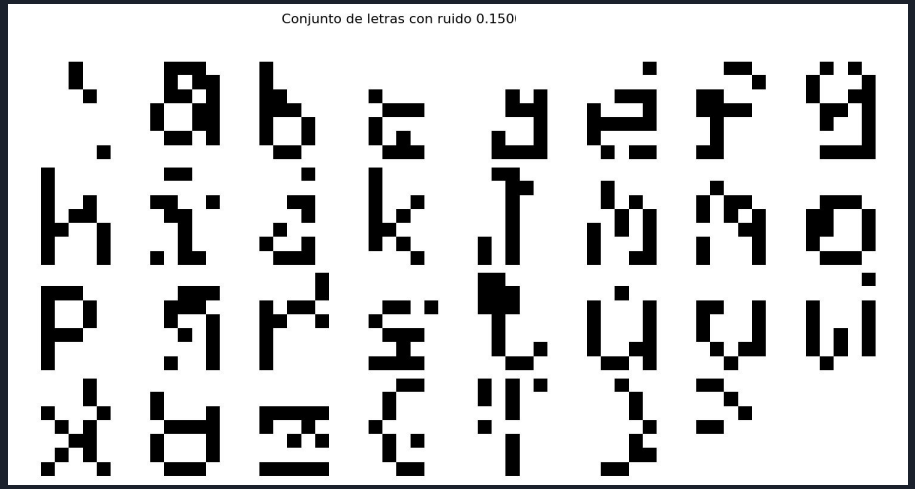
Evaluando el desempeño

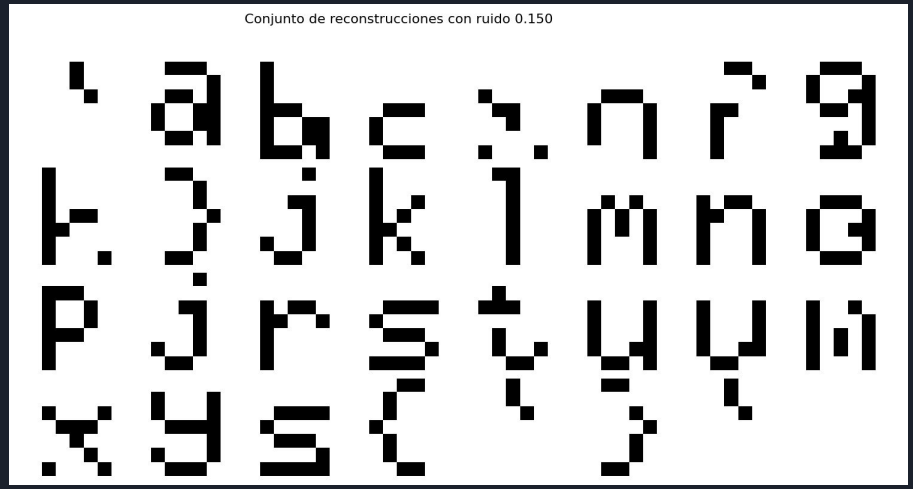
Nuevamente, aunque no es perfecto, se desempeñó bastante mejor



Resultados del denoising

Entrenado con probabilidad de ruido 0.15 +
Preentrenamiento como AE

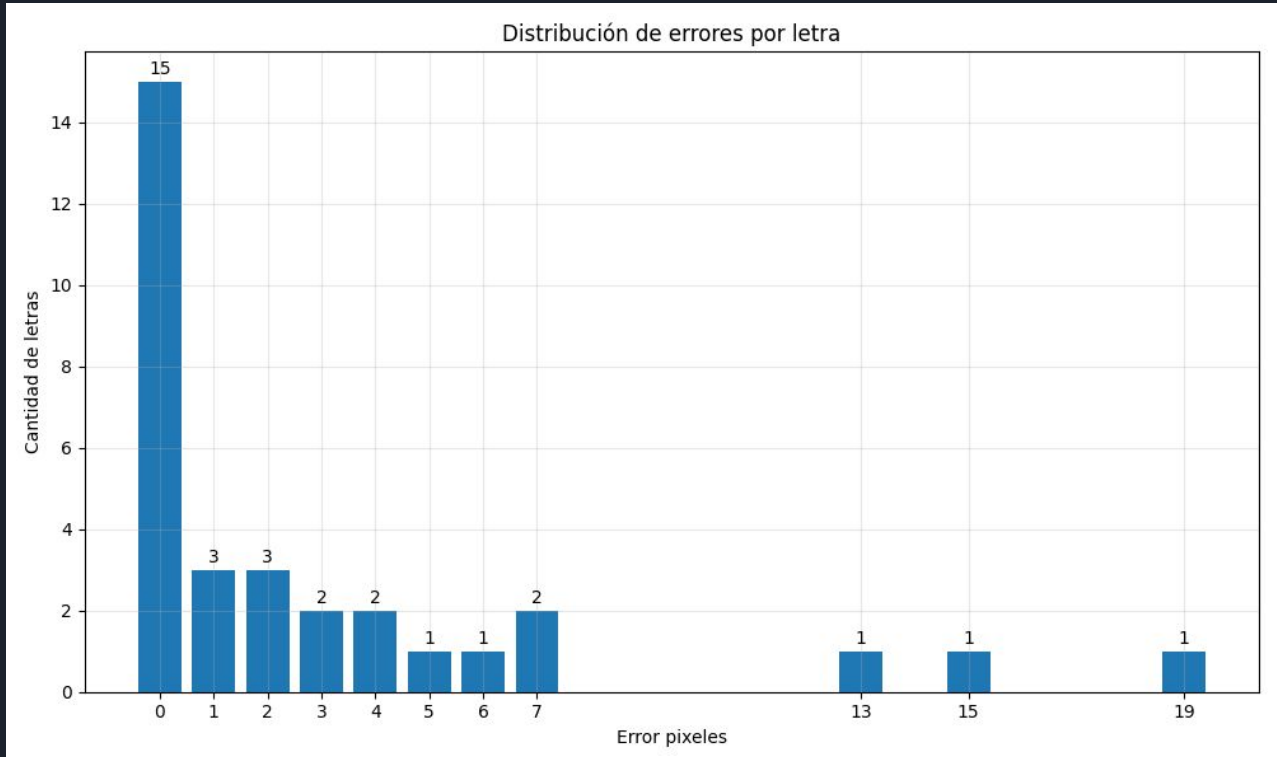




Evaluando el desempeño

Ante ruido de 0.15, vuelven a aparecer una cantidad de errores elevada

Creemos que es entendible, ya que las letras generadas con este nivel de ruido ya parecen estar muy distorsionadas como para poder reconstruirlas de forma consistente.

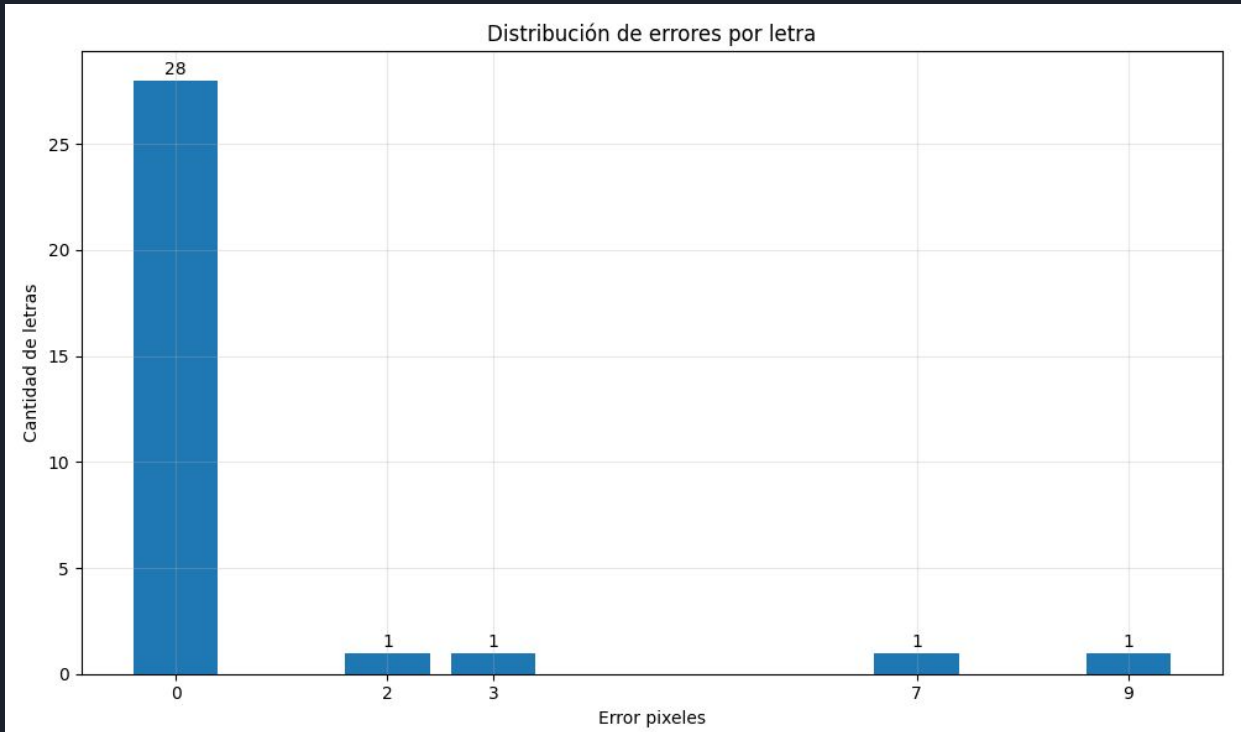


¿Como se desempeña con probabilidades distintas a las que fue entrenado?

Le dimos un alfabeto con ruido de p = 0.05 al DAE que entrenamos con AE + p = 0.15

Esperábamos que, ante un conjunto de menos ruido, tuviera una capacidad similar o incluso mejor

Resulta que suele terminar con errores de bastante más pixeles de los que tenía el DAE entrenado con AE + p = 0.05



¿Se puede hacer algo al respecto?

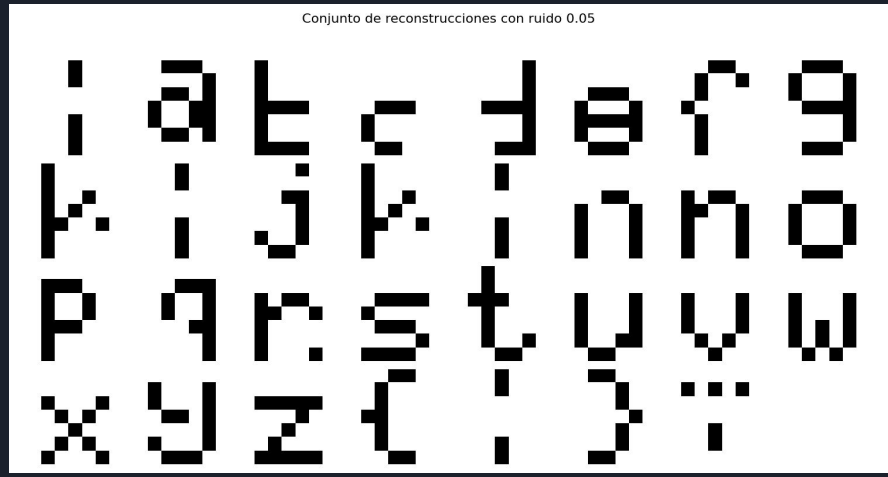
- Queríamos ver si podemos conseguir un DAE que performe bien ante diferentes niveles de ruido.

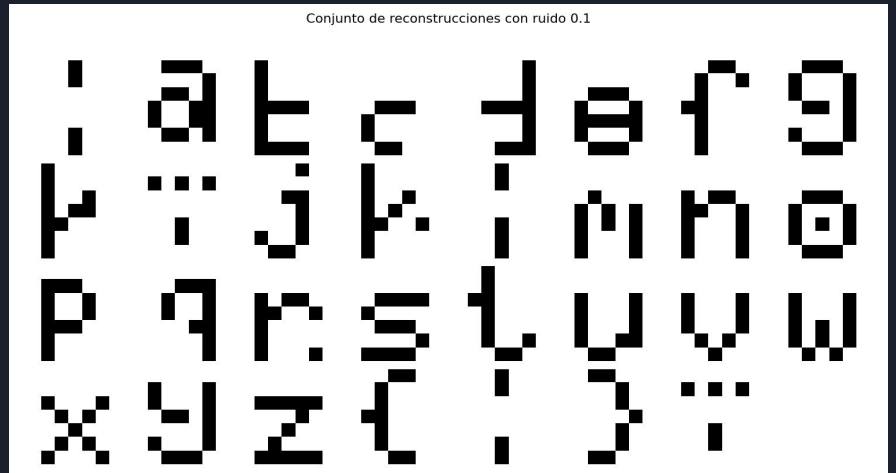
- Ya que entrenar con un ruido mayor no parece ser muy efectivo, se nos ocurrió entrenar con múltiples niveles de ruido

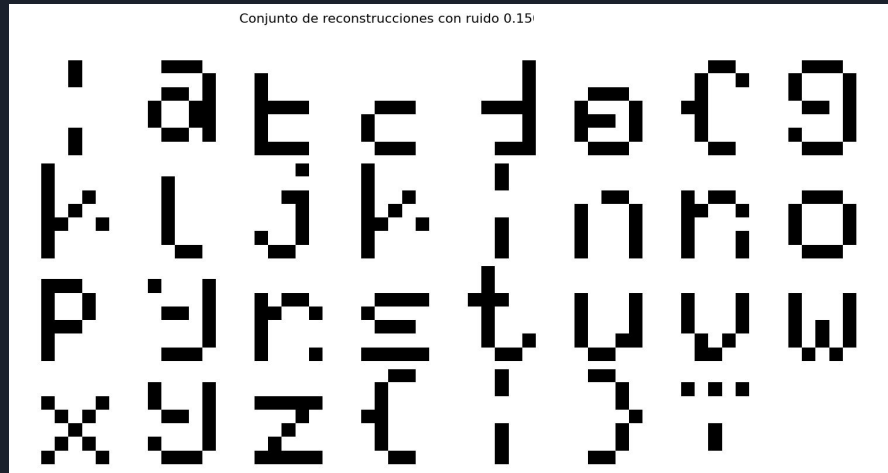
- En cada época, antes de aplicar el ruido, vamos a tomar un p entre [0, 0.15] de forma uniforme, y generar el ruido con ese valor de p.

Resultados del denoising

Entrenado con probabilidad de ruido aleatoria en [0, 0.15] + Preentrenamiento como AE

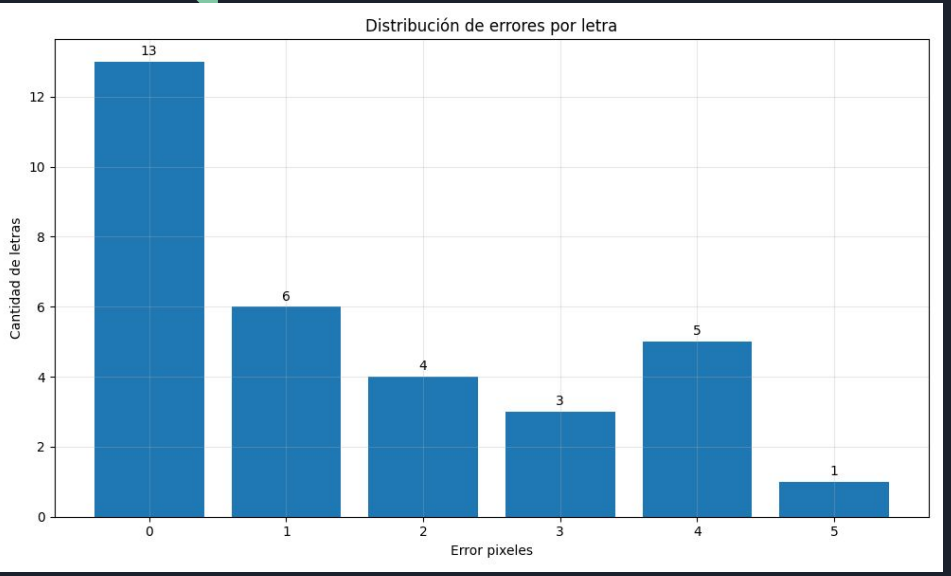




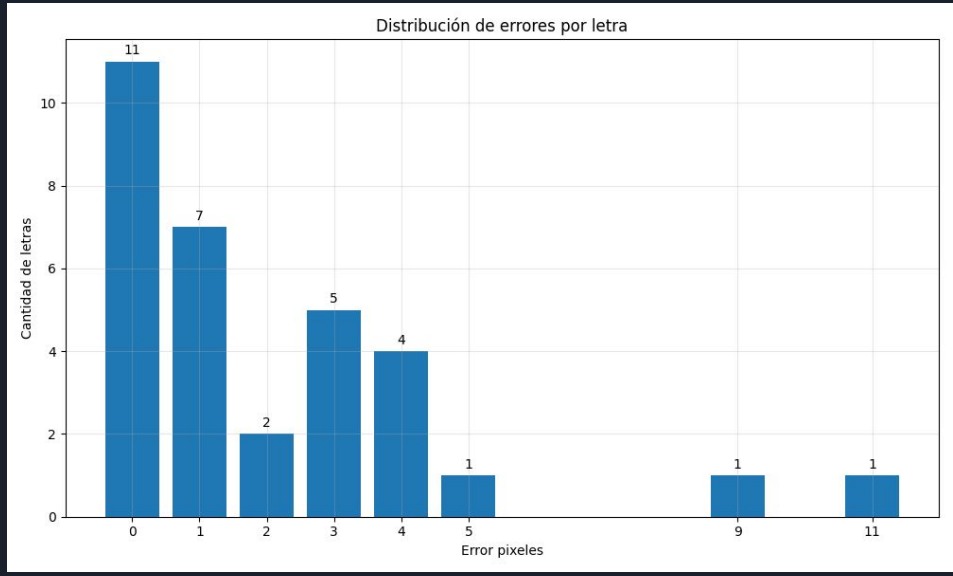


Evaluando el desempeño

Para p=0.05



Para p=0.15



Se obtienen valores similares para otros p, en general no performa muy bien en ninguno

Conclusiones

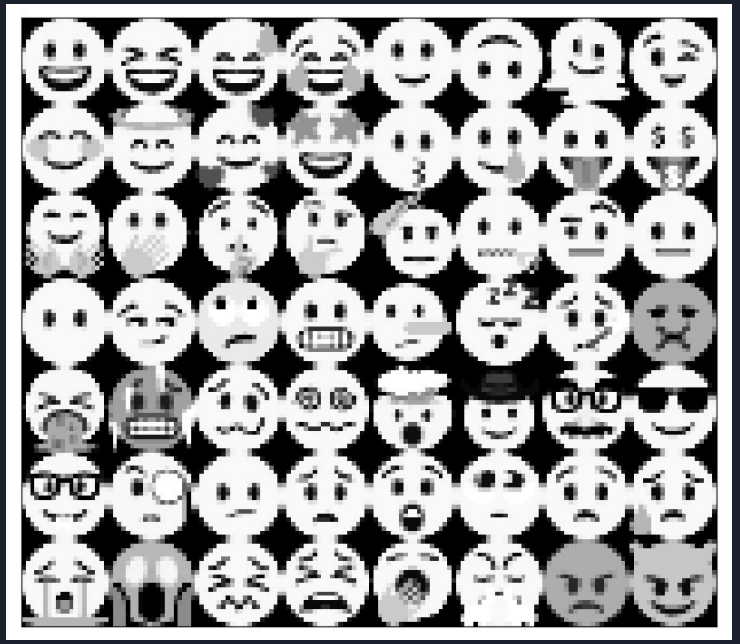
- El Denoising Autoencoder es capaz de eliminar ruido de un dataset, provisto que esté bien entrenado.
- Preentrenarlo con una versión limpia del dataset parece mejorar significativamente los resultados
- Sin embargo, a niveles altos de ruido, ya no puede distinguir entre las letras de forma consistente.

- Si bien en las pruebas que realizamos, entrenar con un valor de p aleatorio no dio buenos resultados, podría valer la pena intentar con más epochs (dado que esperamos que aprenda “más” ruidos), con más epochs, o con una dimensión mayor en el espacio latente.

Variational Autoencoder

Dataset - Emojis

- Usamos un dataset de 56 emojis
- Convertidos a escalas de grises
- 20x20 pixeles
- Mapeamos los colores en [0, 1]



Primer Intento

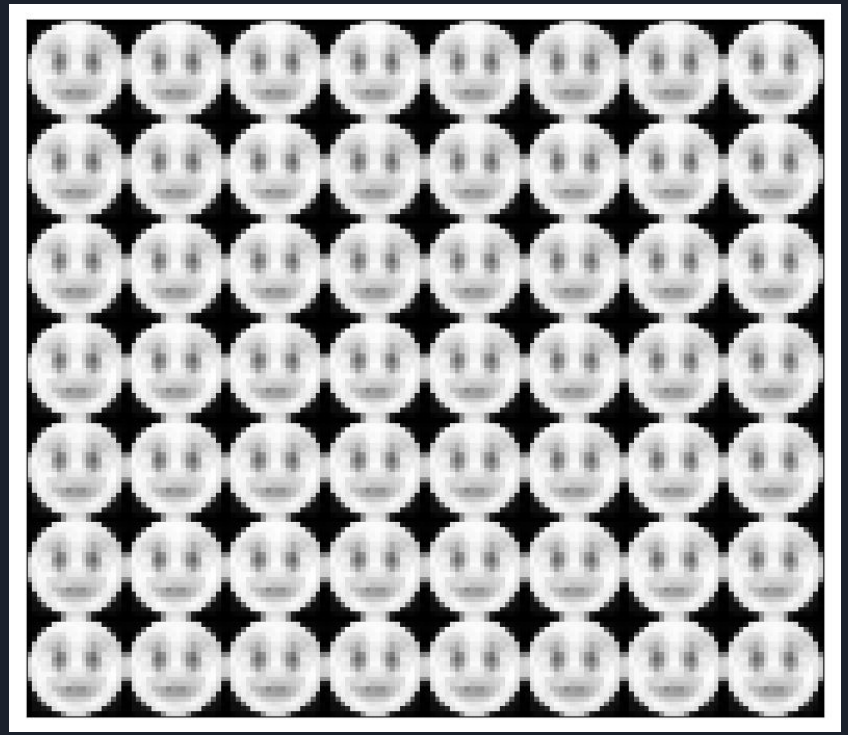
Capas:
- Encoder: 30 -> 20 -> 10
- Espacio latente: 2
- Decoder: 10 -> 20 -> 30

Learning rate: 0.0001

Epochs: 100

Función de activación: Sigmoidea

Optimizador: Adam



Segundo intento

Capas:

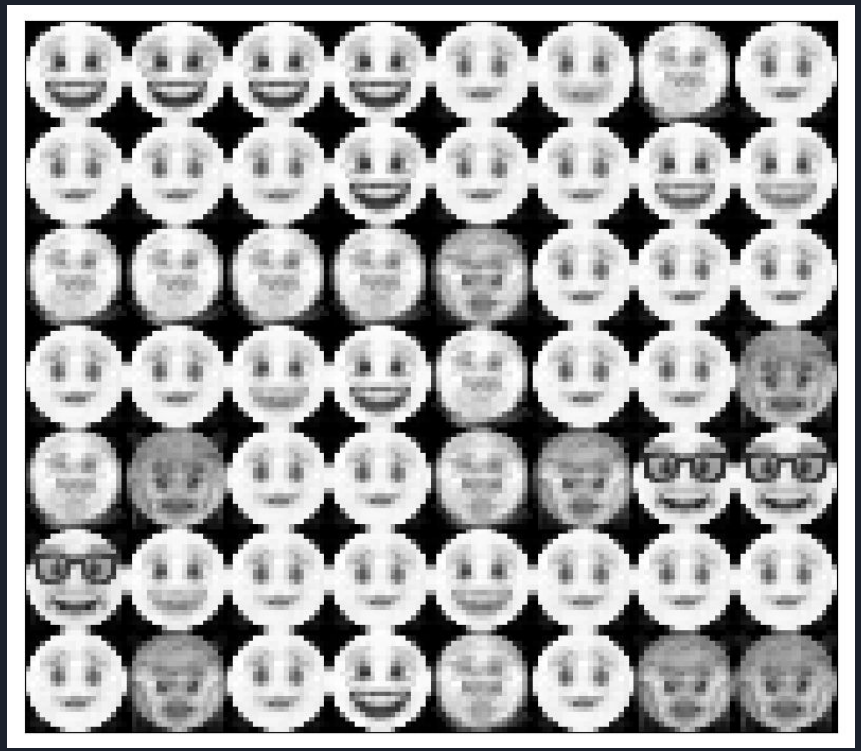
- Encoder: 30 -> 20 -> 10
- Espacio latente: 2
- Decoder: 10 -> 20 -> 30

Learning rate: 0.0001

Epochs: 1000

Función de activación: Sigmoidea

Optimizador: Adam



Tercer intento

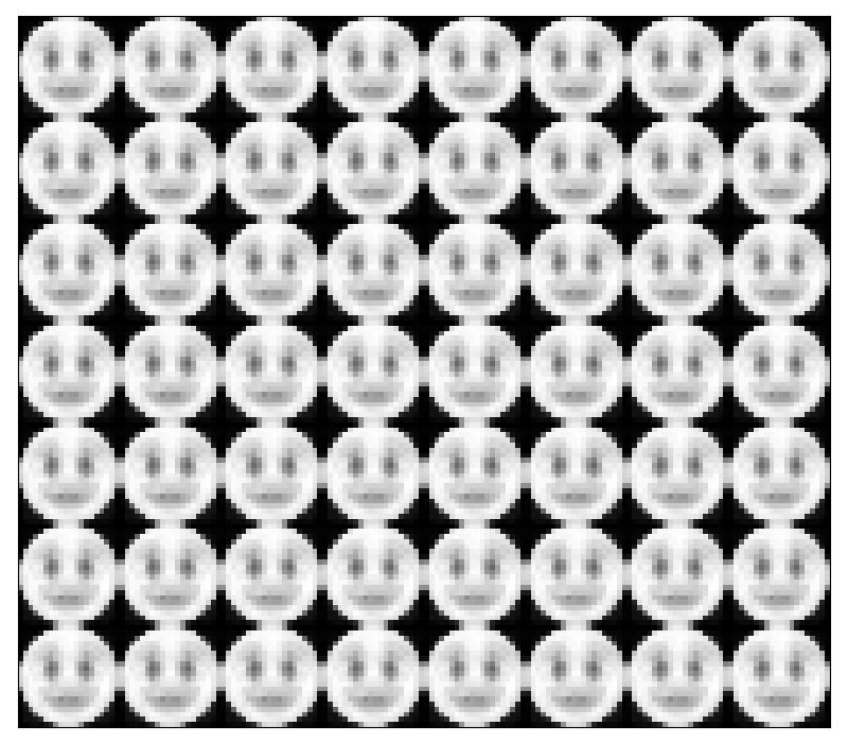
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 2
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

Epochs: 1000

Función de activación: Sigmoidea

Optimizador: Adam



Cuarto intento

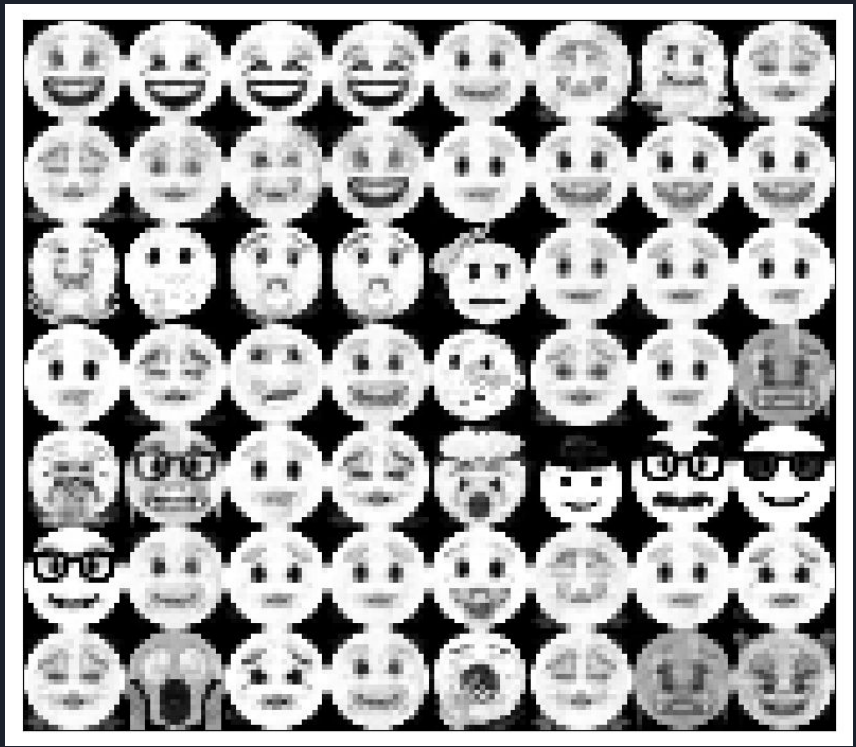
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 2
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

Epochs: 1000

Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Espacio latente 2D

Capas:

Encoder: 300 -> 200 -> 100
Espacio latente: 2
Decoder: 100 -> 200 -> 300

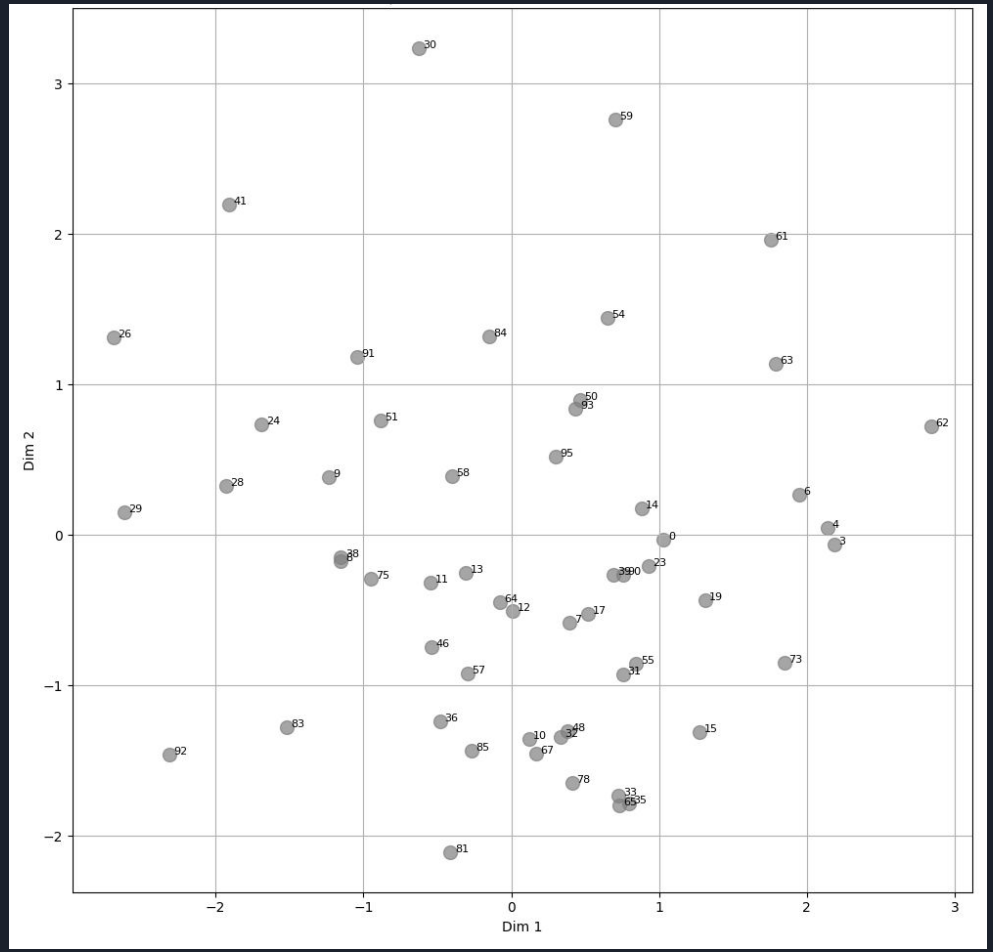
Learning rate: 0.0001

Epochs: 1000

Funciones de activación:

ReLU
Sigmoidea (sólo ultima capa)

Optimizador: Adam



Errores de reconstrucción

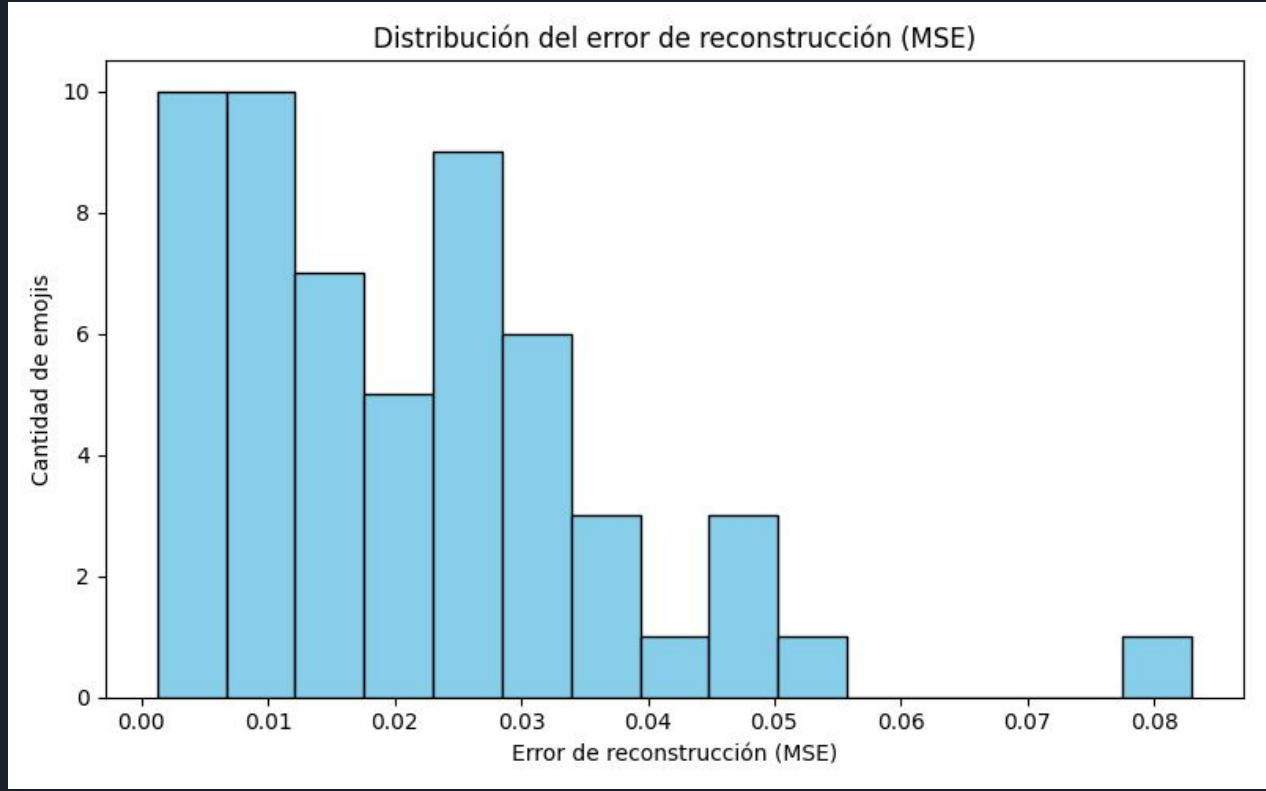
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 2
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

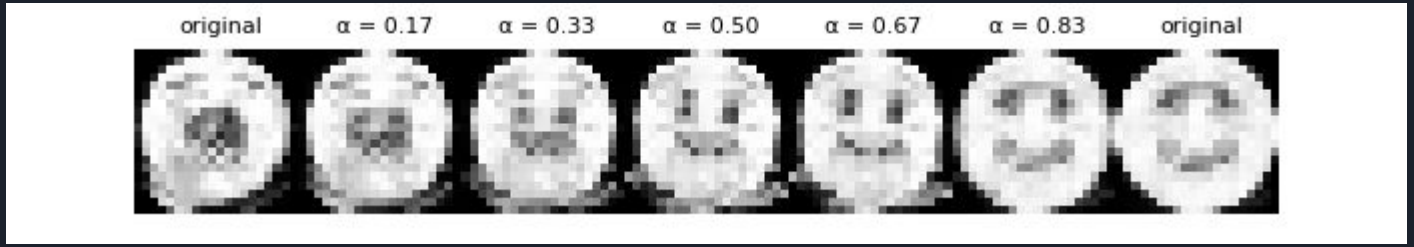
Epochs: 1000

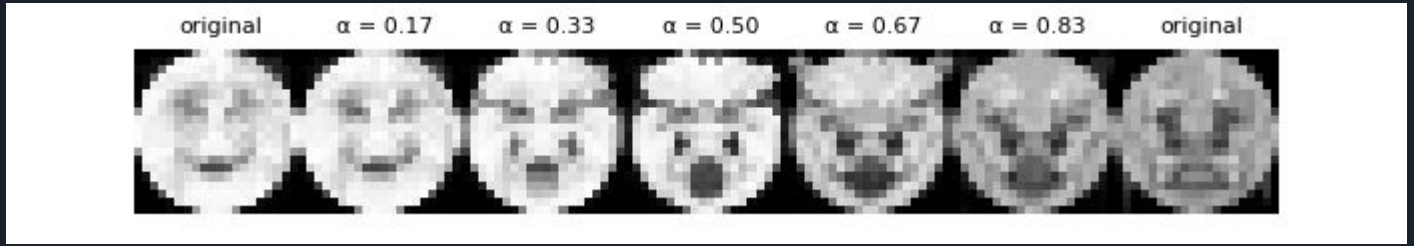
Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

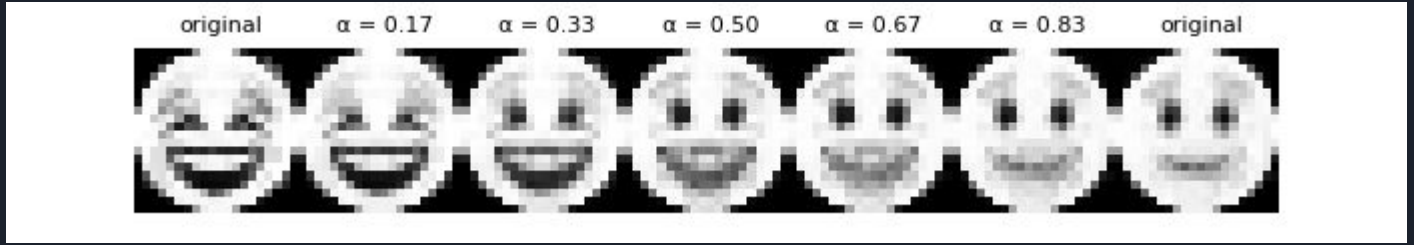
Optimizador: Adam



Generando elementos nuevos







Quinto intento

Capas:

- Encoder: 300 -> 200 -> 100
- Espacio latente: 3
- Decoder: 100 -> 200 -> 300

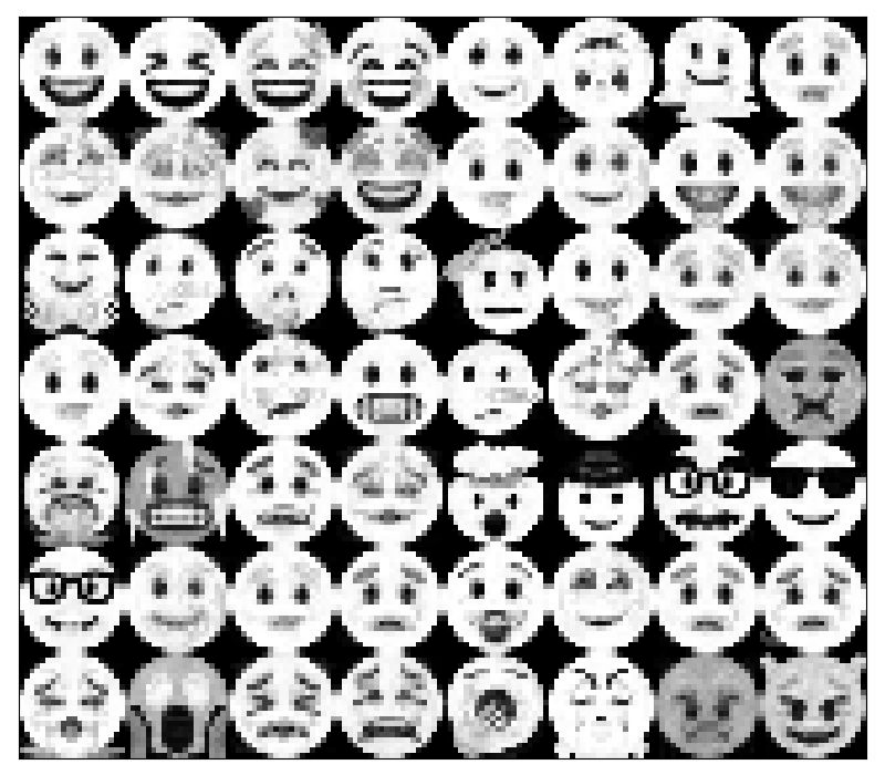
Learning rate: 0.0001

Epochs: 1000

Funciones de activación:

- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Espacio latente 3D

Capas:

- Encoder: 300 -> 200 -> 100
- Espacio latente: 3
- Decoder: 100 -> 200 -> 300

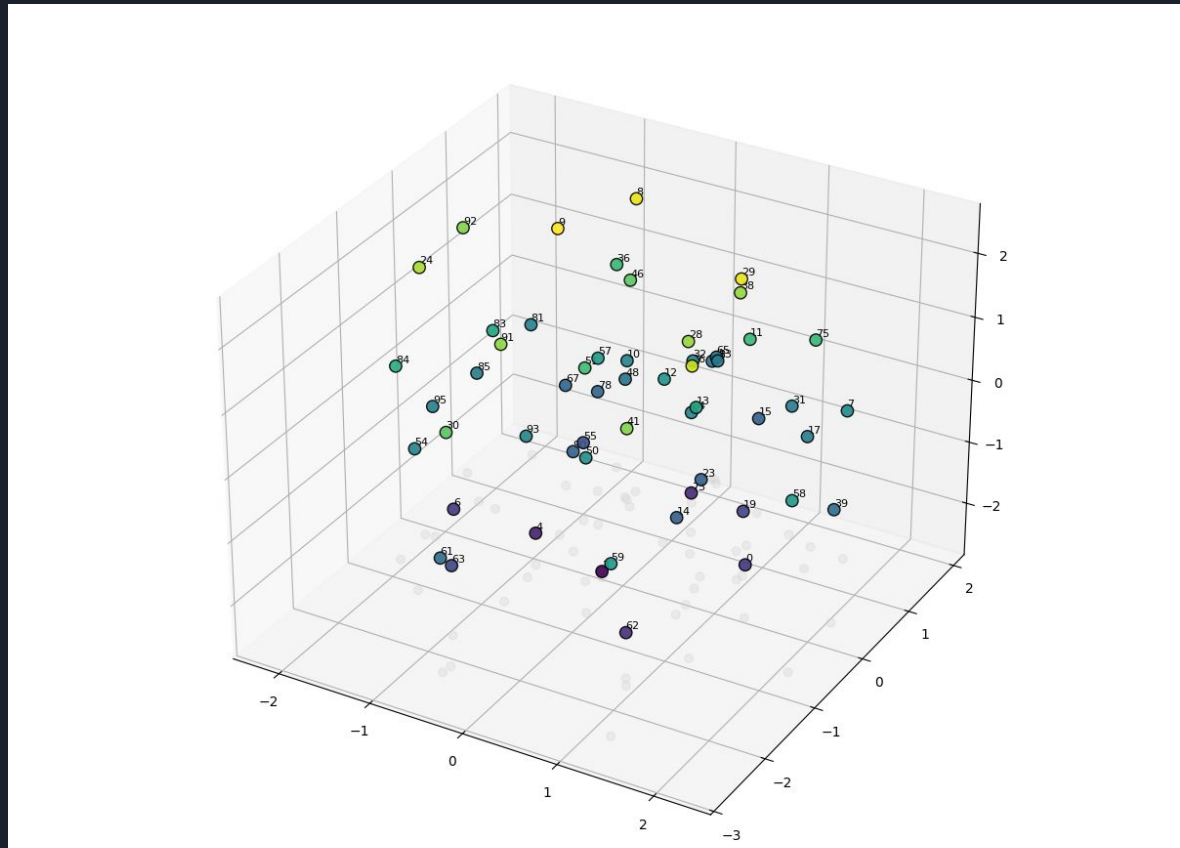
Learning rate: 0.0001

Epochs: 1000

Funciones de activación:

- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Errores de reconstrucción

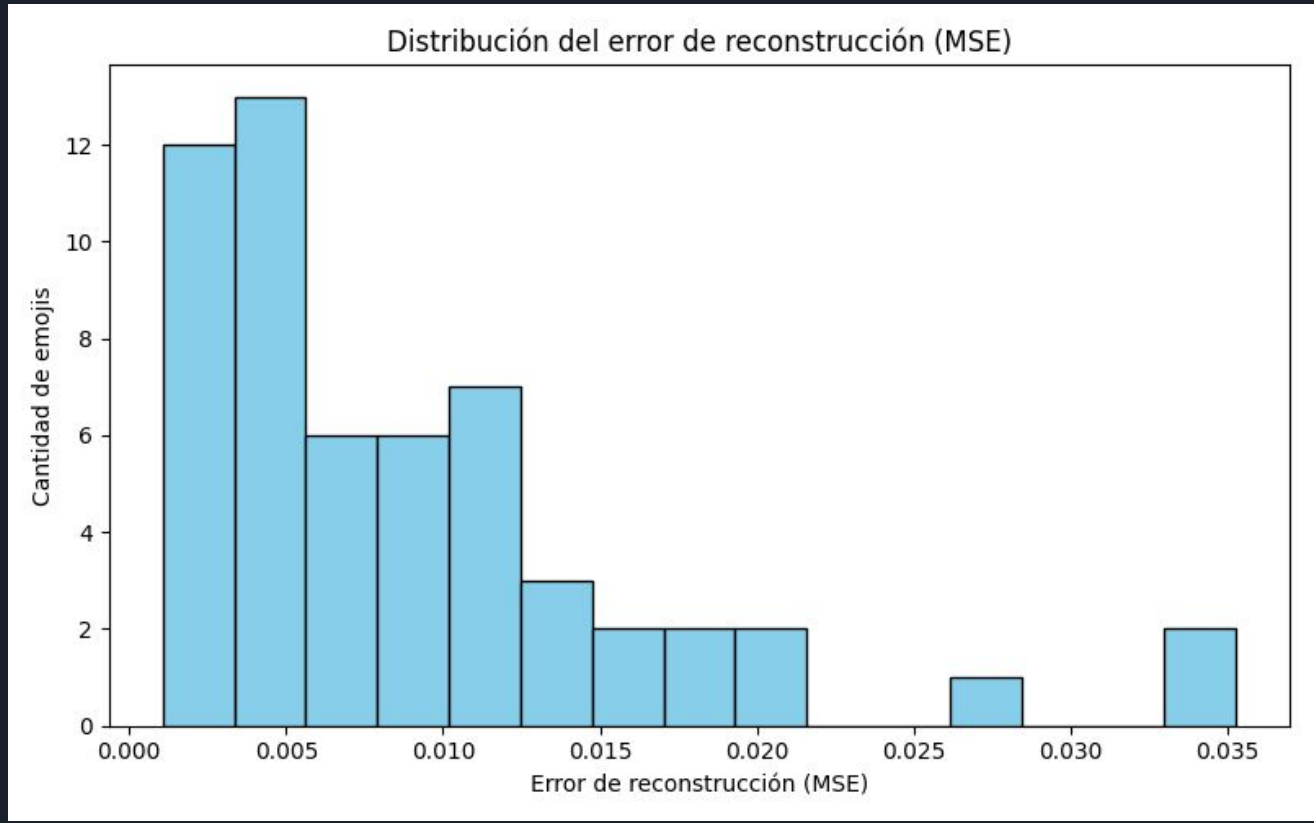
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 3
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

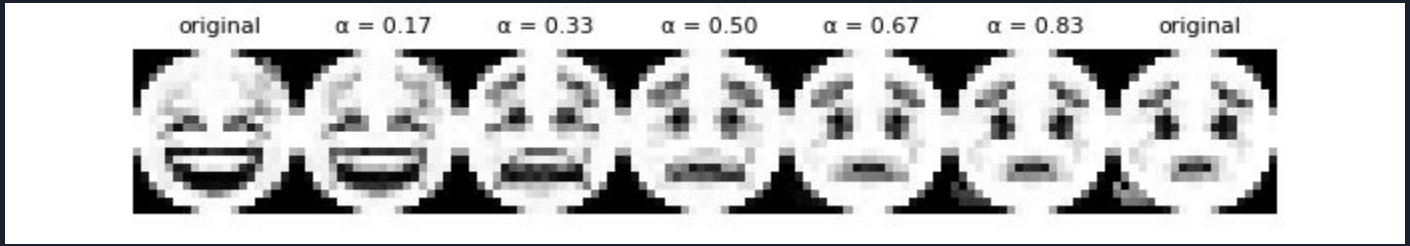
Epochs: 1000

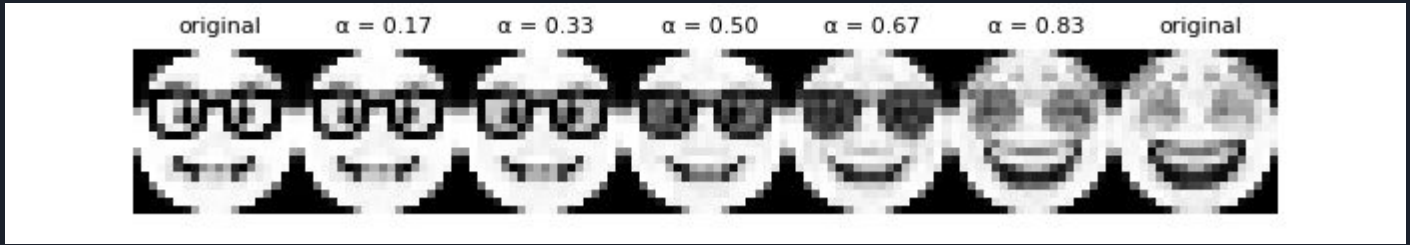
Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

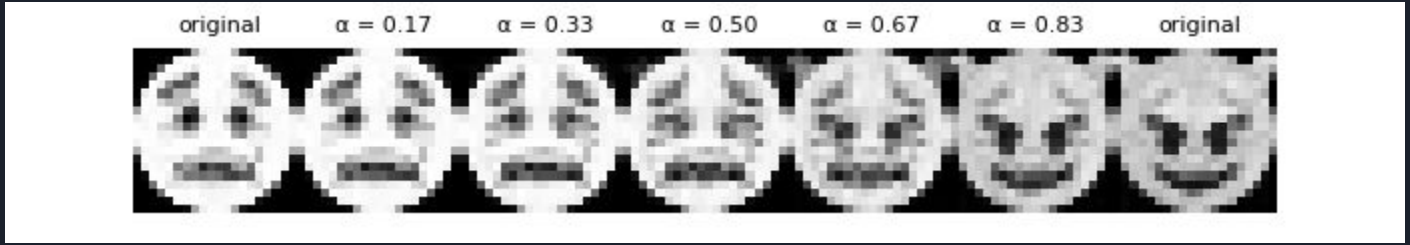
Optimizador: Adam



Generando elementos nuevos







Sexto intento

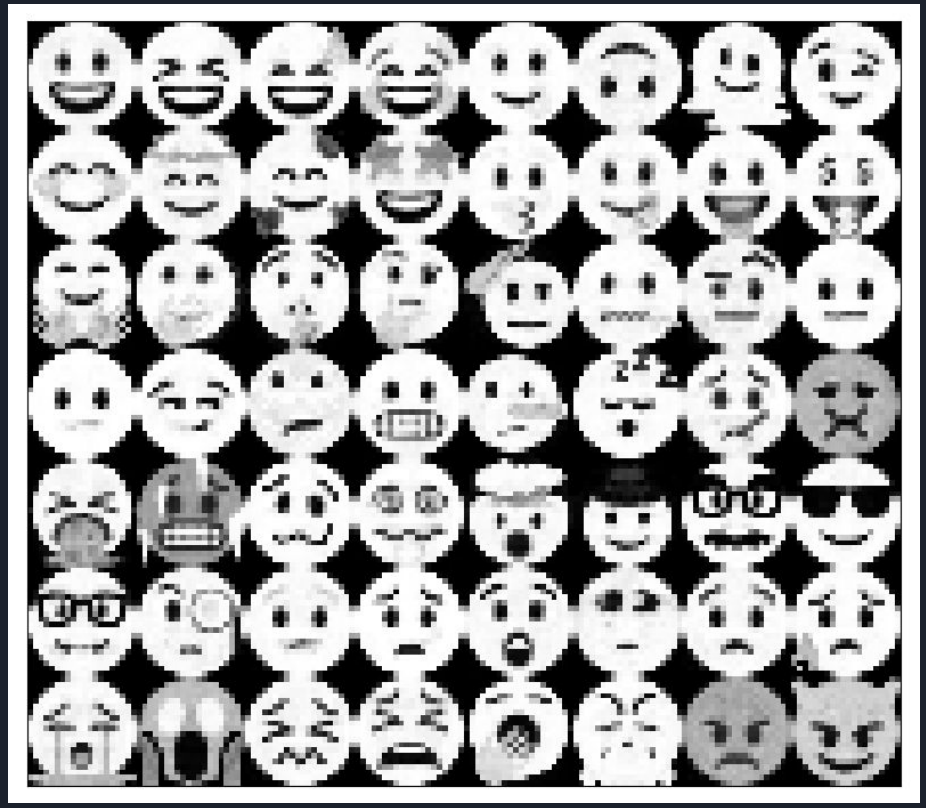
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 6
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

Epochs: 1000

Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Errores de reconstrucción

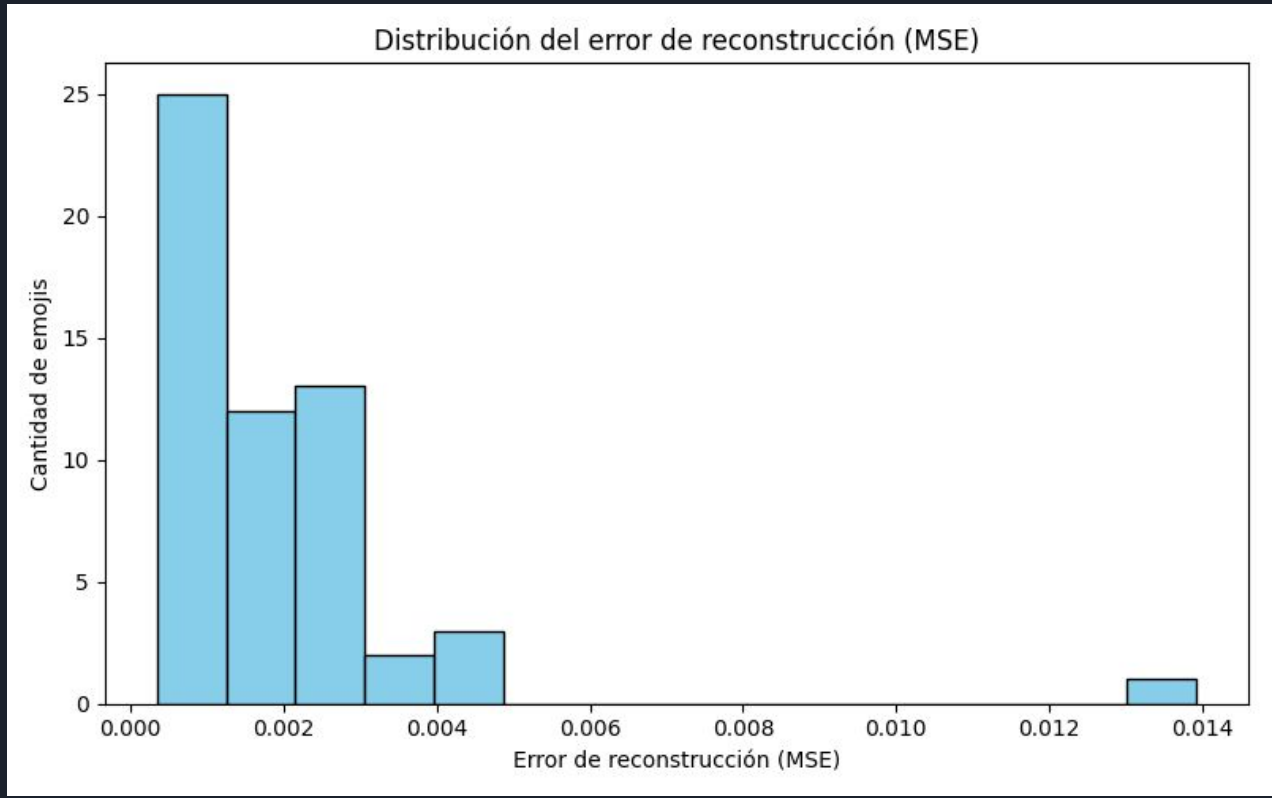
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 6
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

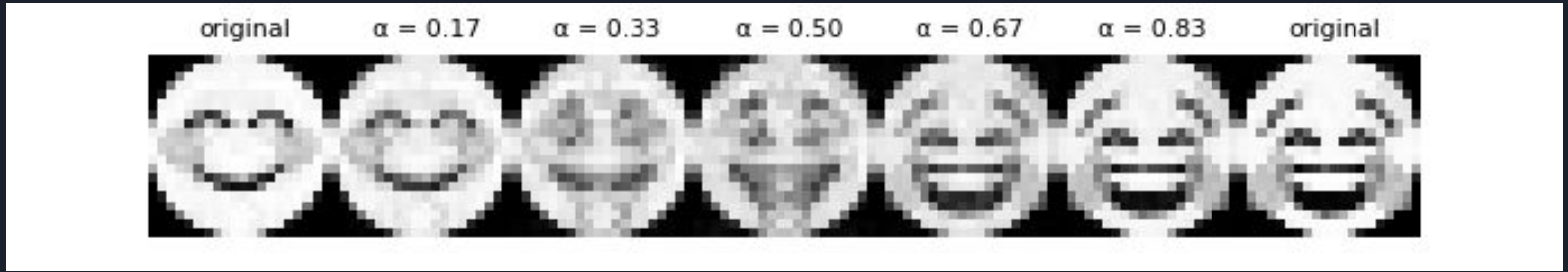
Epochs: 1000

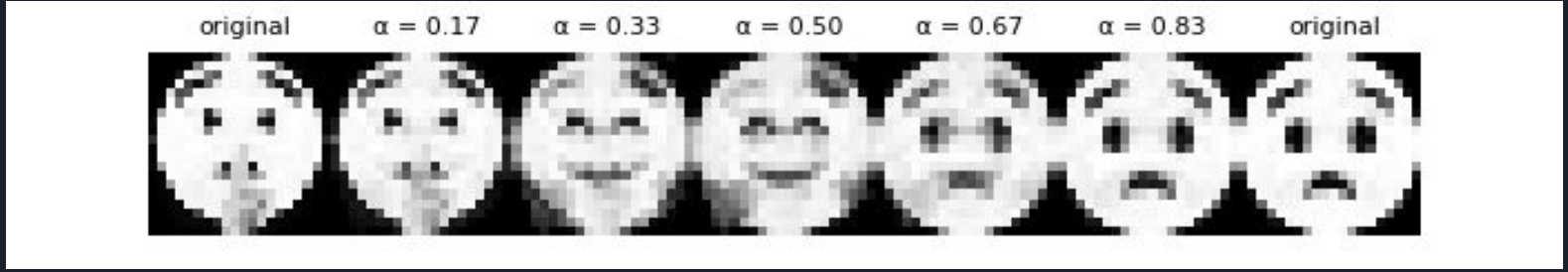
Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Generando elementos nuevos





Generando nuevos elementos

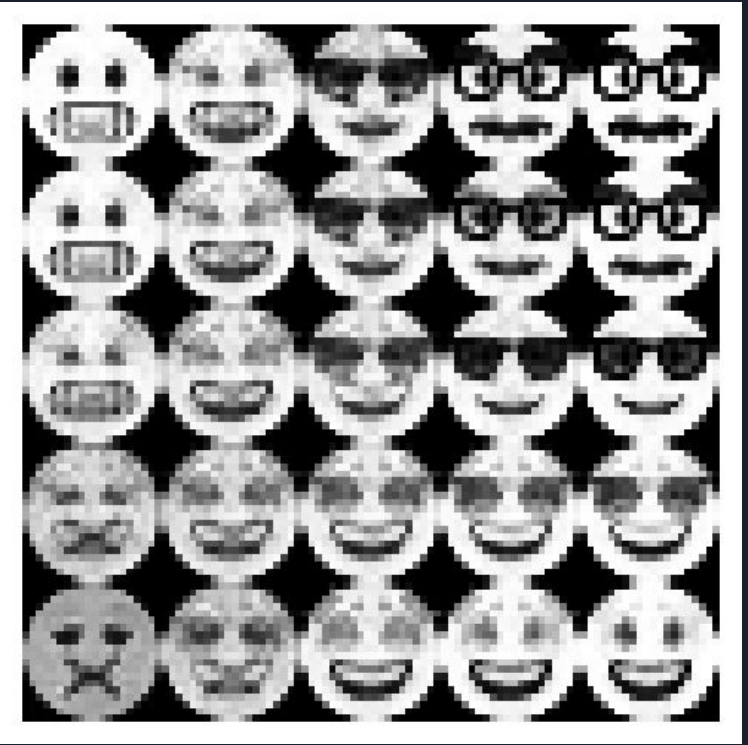
Capas:
- Encoder: 300 -> 200 -> 100
- Espacio latente: 3
- Decoder: 100 -> 200 -> 300

Learning rate: 0.0001

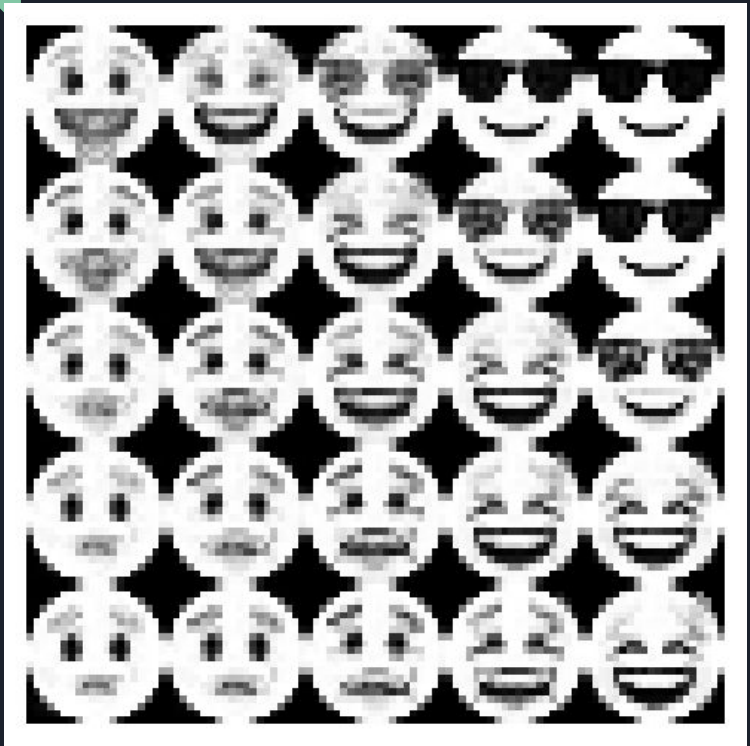
Epochs: 1000

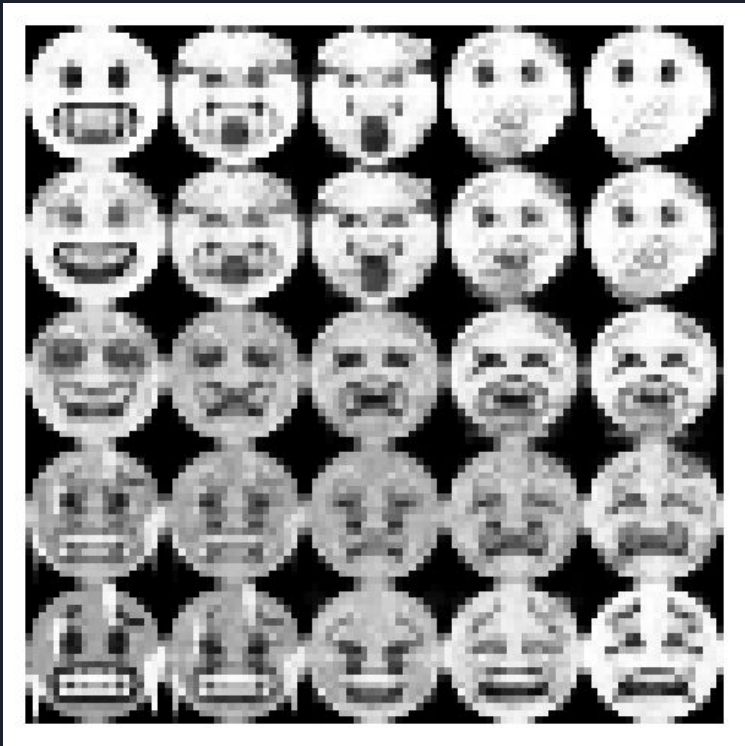
Funciones de activación:
- ReLU
- Sigmoidea (sólo ultima capa)

Optimizador: Adam



Generando nuevos elementos





Conclusiones

- El VAE genera mejores muestras nuevas que un Autoencoder normal, gracias a la estructura probabilística del espacio latente.
- Los emojis generados con el VAE tienen más coherencia con el dataset que las letras generadas con el Autoencoder normal.
- Usar una dimensión de espacio latente adecuada es muy importante para que los datos se puedan reconstruir e interpolar bien.
    - Si la dimensión del espacio latente es muy baja -> Cuesta más reconstruir
    - Si es muy alta -> Cuesta más interpolar