



# ***Deep Learning***

Grupo 4 – Generative autoencoder, DAE & VAE





# ***Correcciones***

# ***Denoising***

# ***Autoencoder***

# Análisis

La idea de este análisis es observar el comportamiento del denoising autoencoder frente al ruido Salt & Pepper

Entrada con ruido



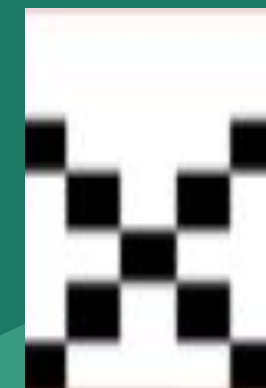
Salida esperada



Salt & Pepper (0.1)



Salt & Pepper (0.2)



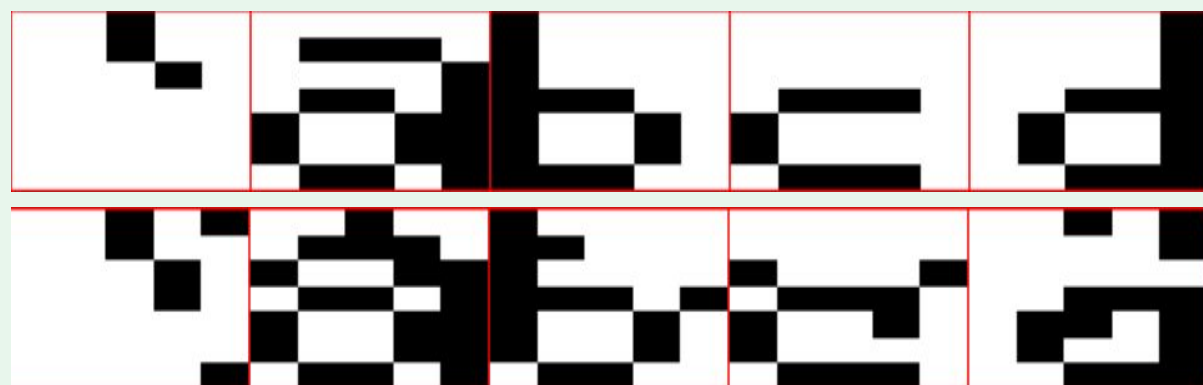
# *Error y entrenamiento*

Para medir el error observamos el Mean Square Error (MSE) utilizando un set de testing con ruido

Para el entrenamiento se le pasa al denoising autoencoder el dataset original y en cada época de entrenamiento se le aplica ruido a los inputs de manera aleatoria ya que buscamos que el perceptrón multicapa realice predicciones sin importar cuales sean los píxeles afectados

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

# Testing Dataset (Salt & Pepper – 0.1)



original

testing

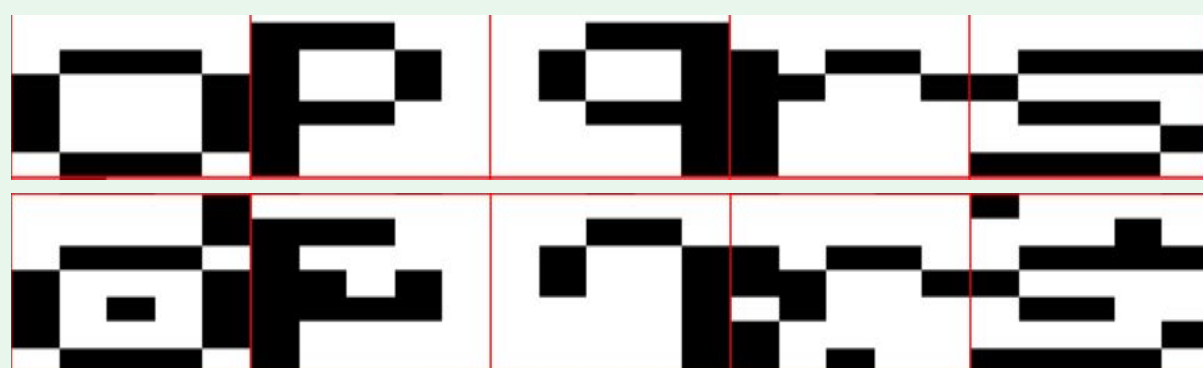
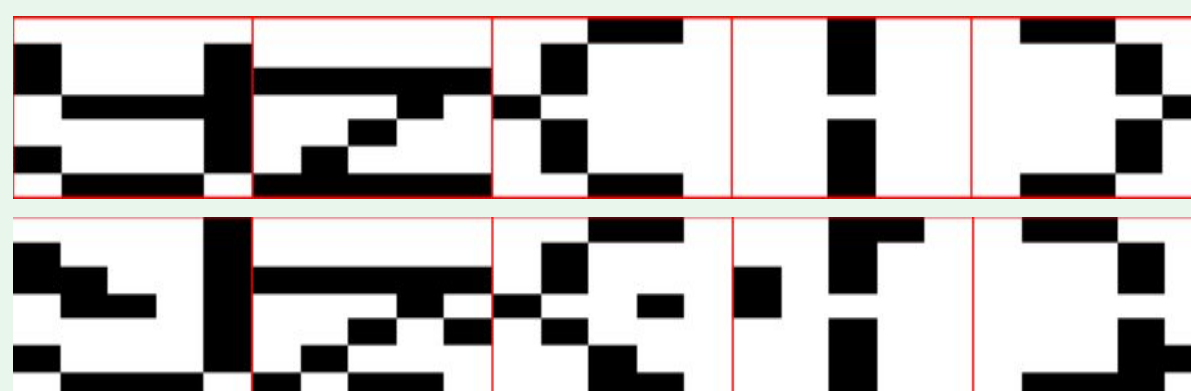
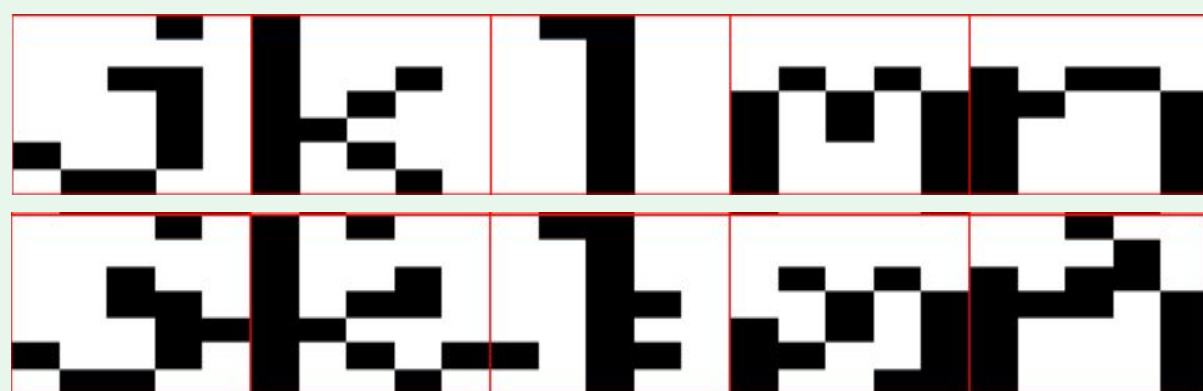
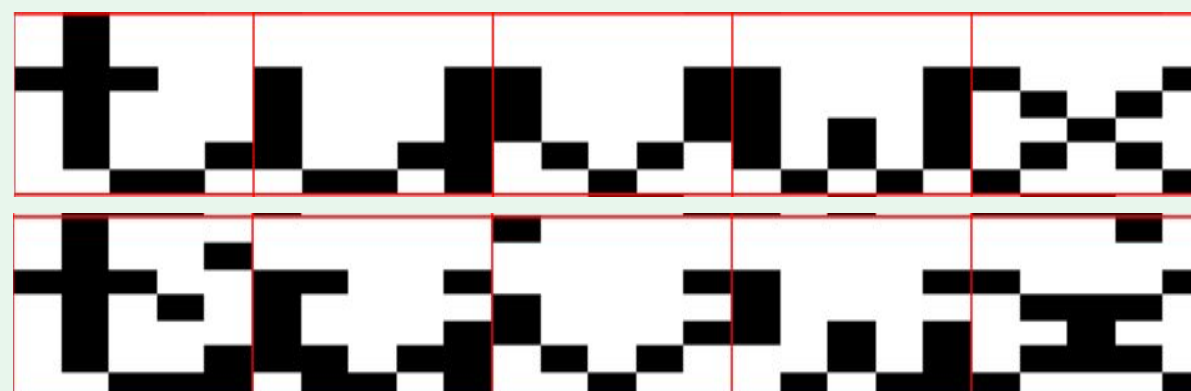
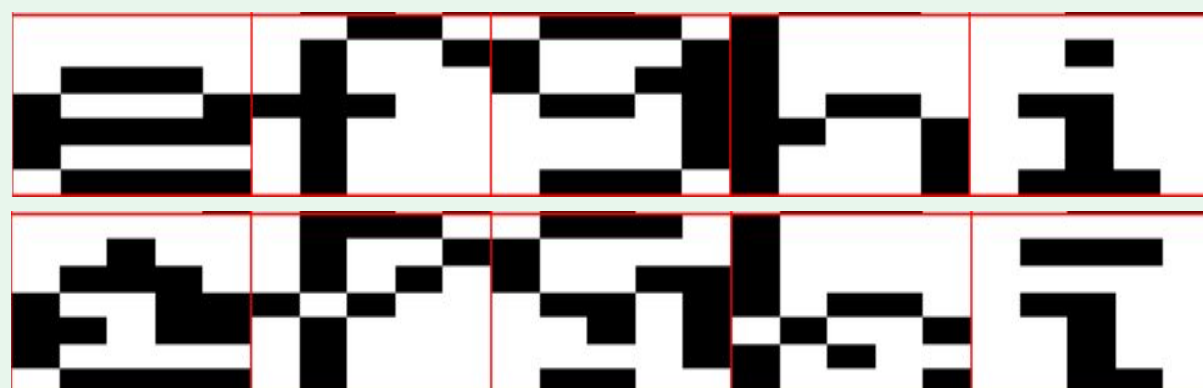
capas ocultas = [32, 8]

$n = 0.001$

epochs = 11500

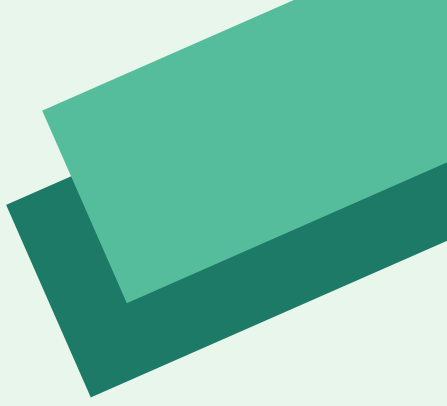
activation = tanh

optimizer = Adam

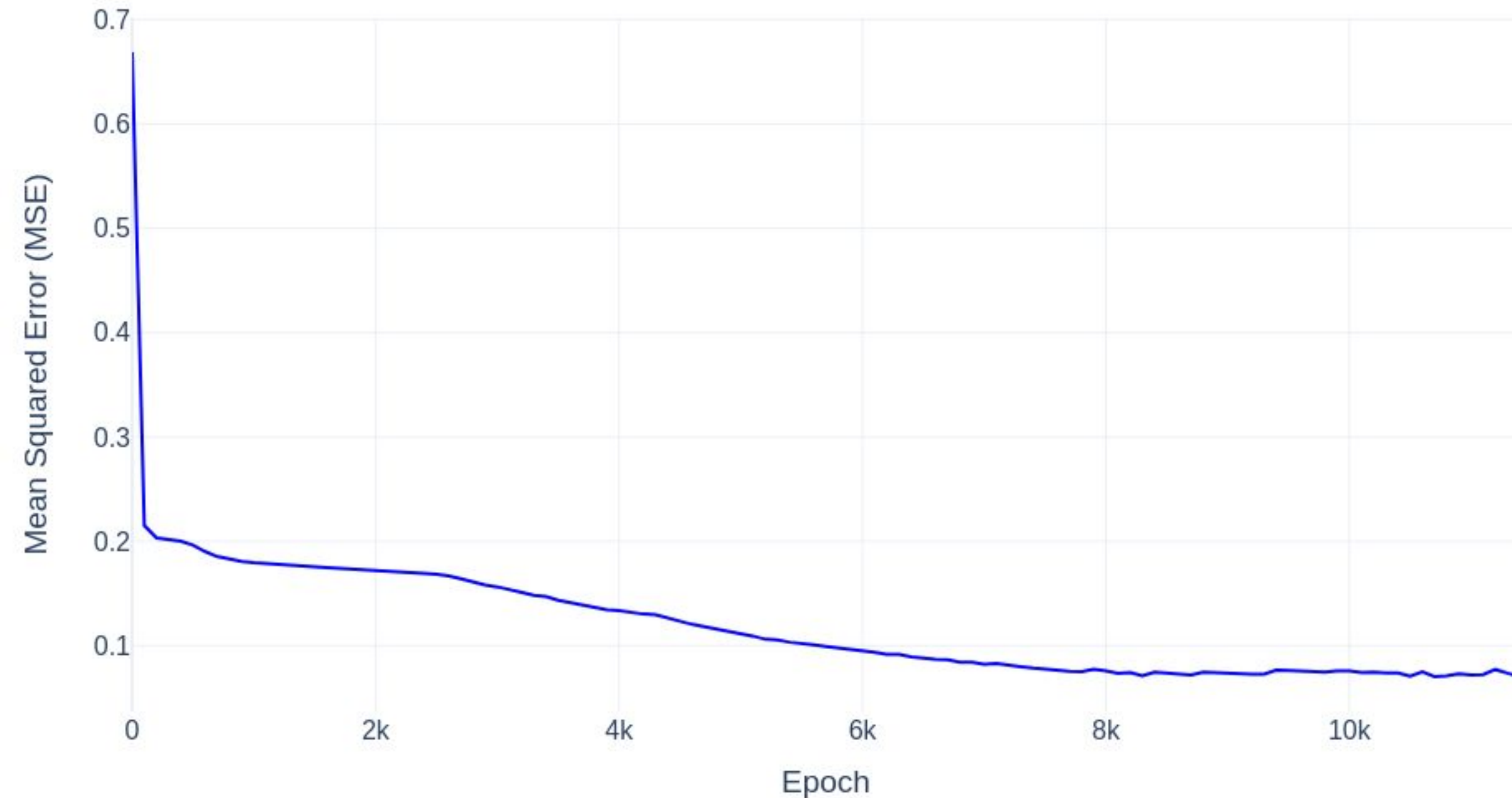


El factor 0.1 implica invertir el 10% de los píxeles, en este caso 3

# Mejor configuración



Epoch vs MSE



*\*La arquitectura se eligió por fuerza bruta*

capas ocultas = [32, 8]

n = 0.001

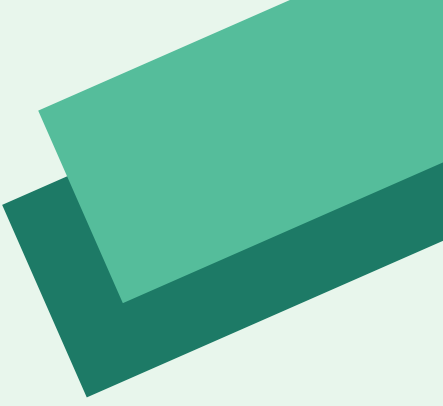
epochs = 11500

activation = tanh

optimizer = Adam

Mín. Avg. Loss=0.06865

# Resultados



capas ocultas = [32, 8]

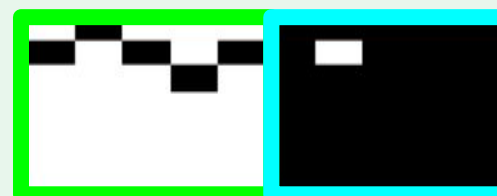
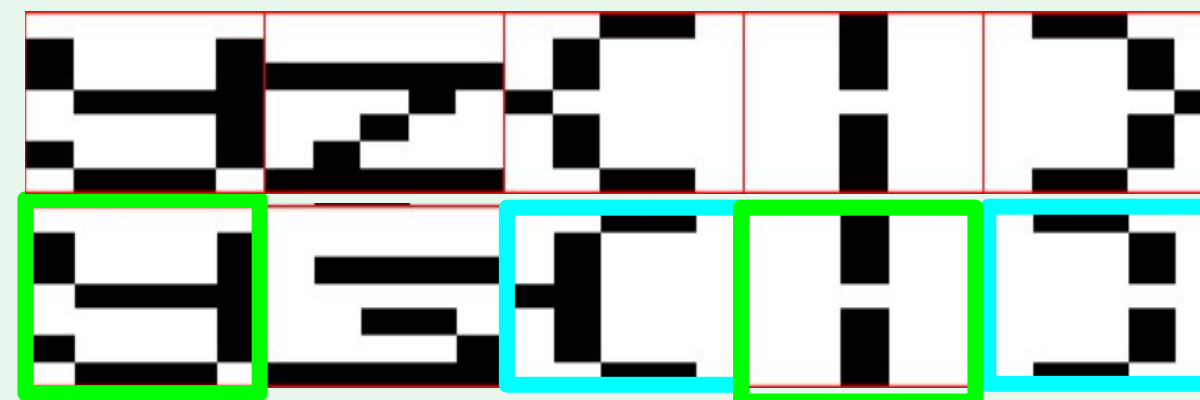
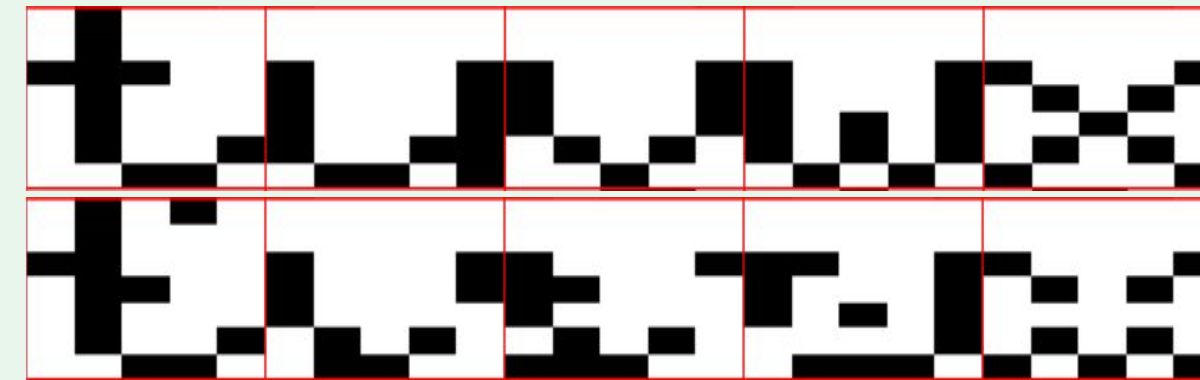
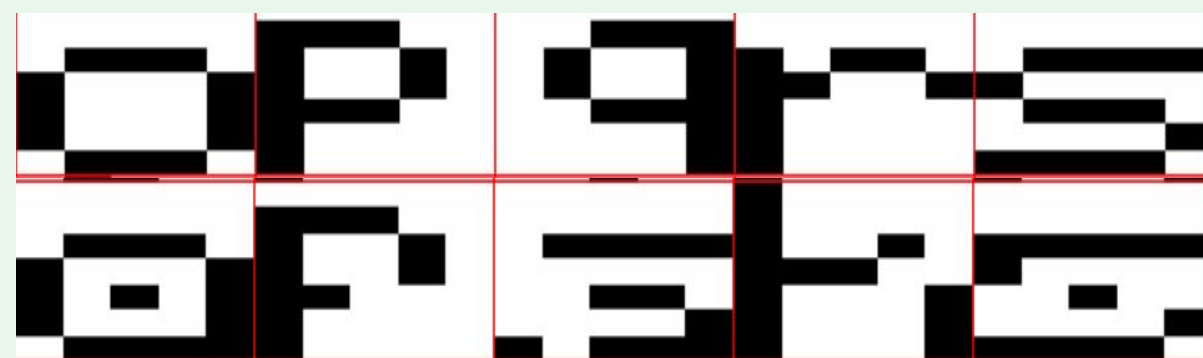
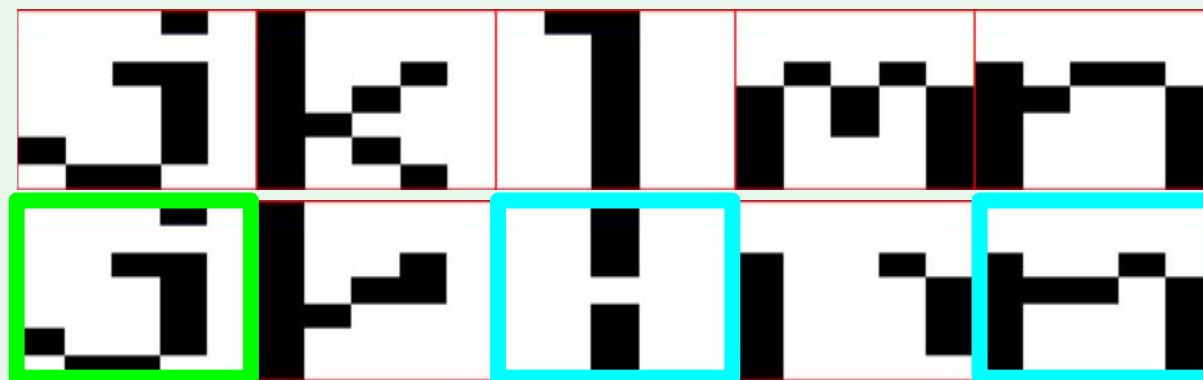
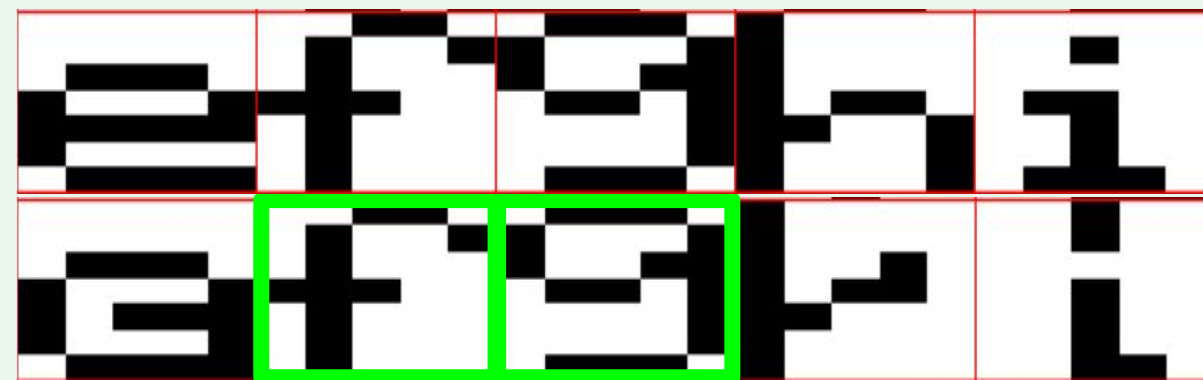
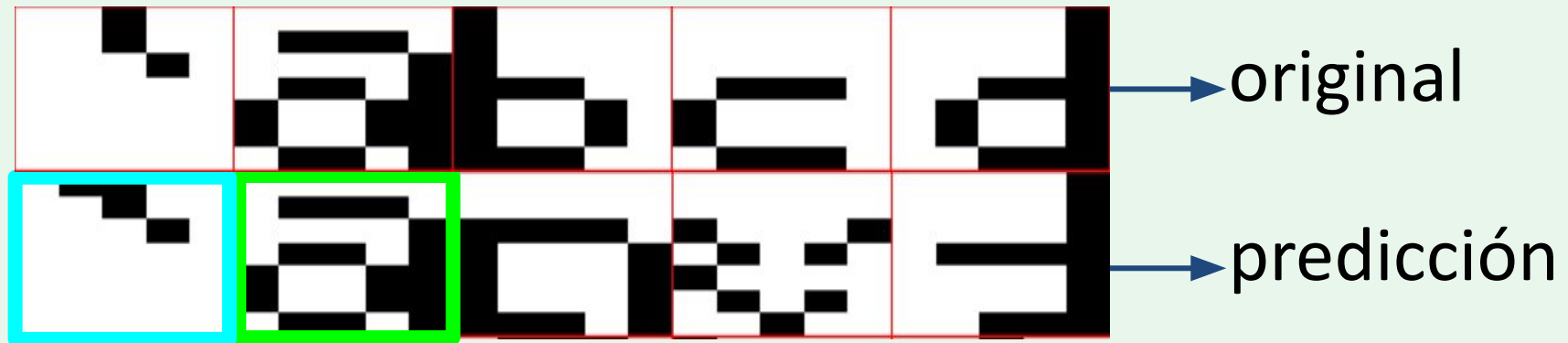
$n = 0.001$



epochs = 11500

activation = tanh

optimizer = Adam

Entrenamiento  
realizado con un  
factor de ruido Salt &  
Pepper de 0.1



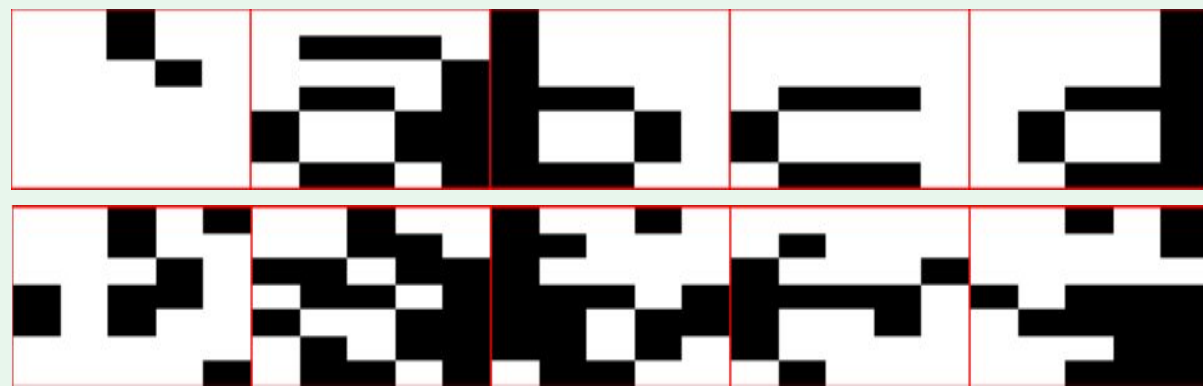
-  Predicción perfecta
-  Predicción con 1 píxel de error

# ***Resultados (Salt & Pepper – 0.1)***

Como podemos apreciar en nuestra mejor configuración obtuvimos 98 píxeles incorrectos dentro de los 1120 píxeles totales del dataset. Lo que da una tasa de error del 8.75%



# Testing Dataset (Salt & Pepper – 0.2)



original

testing

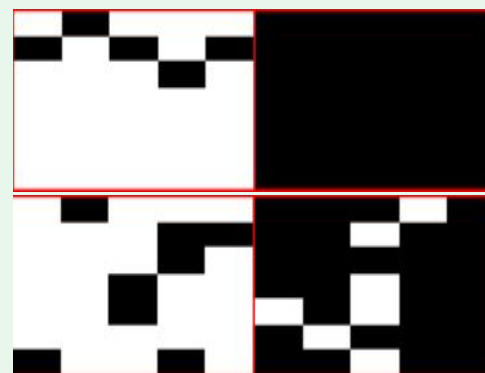
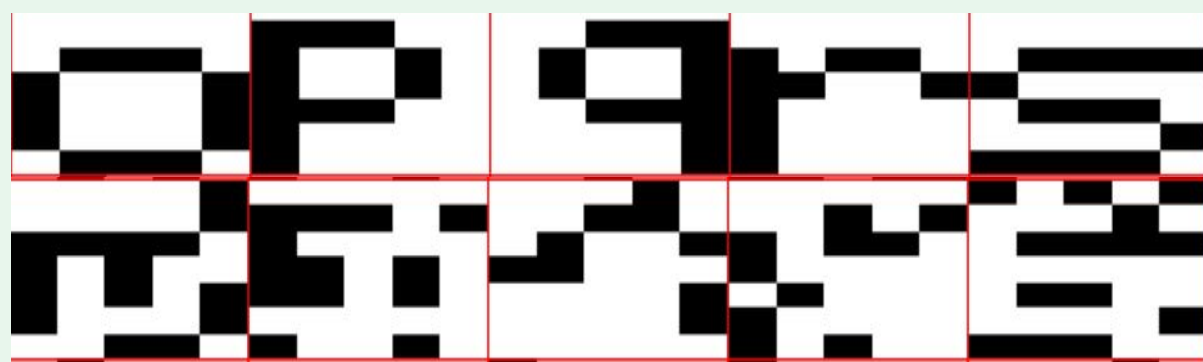
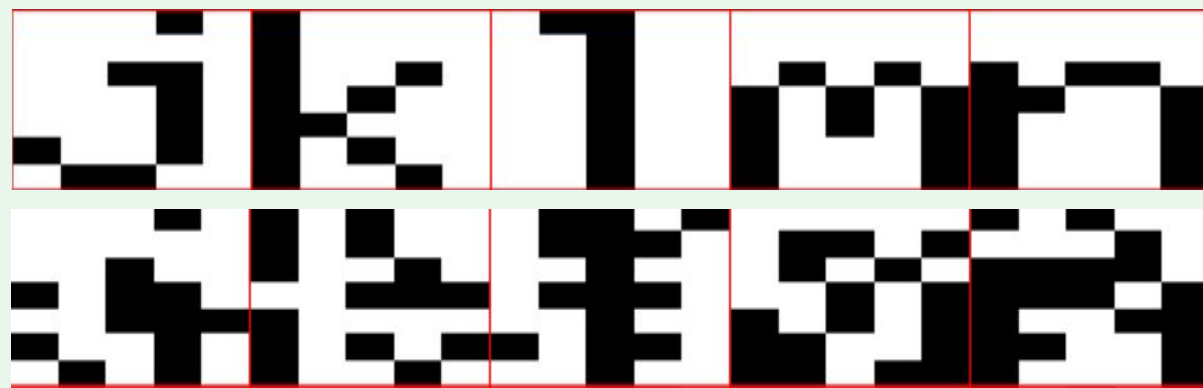
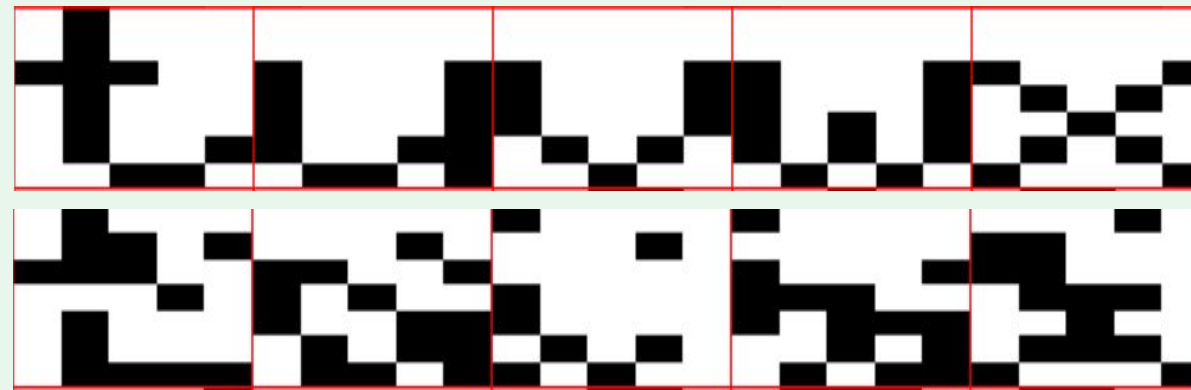
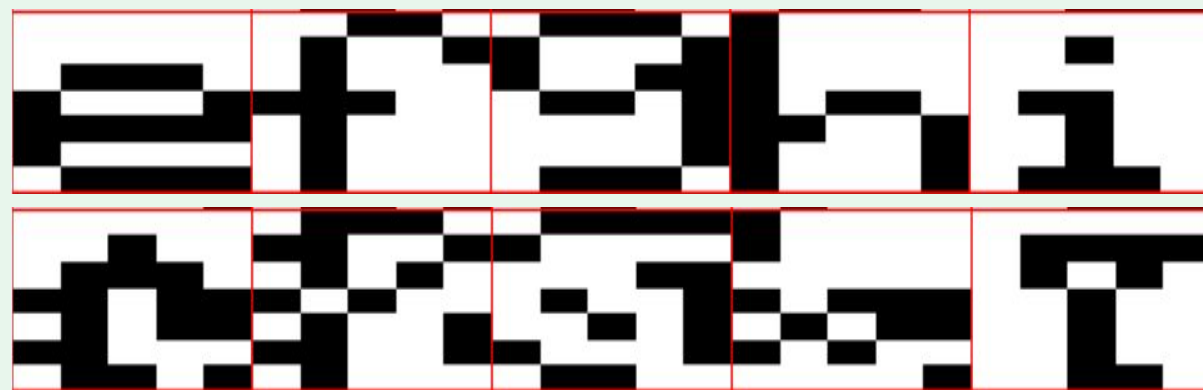
capas ocultas = [32, 8]

$n = 0.001$

epochs = 13000

activation = tanh

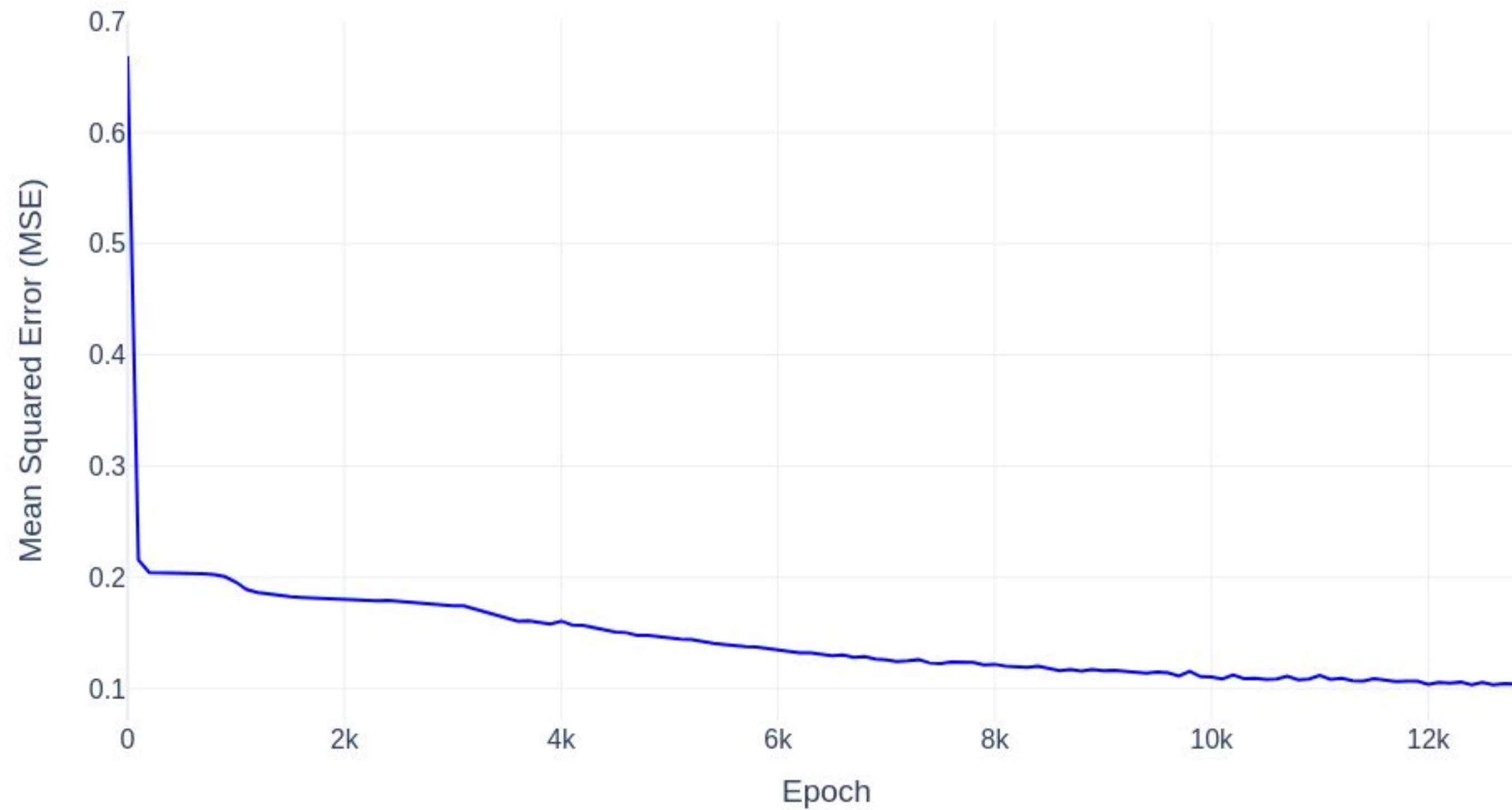
optimizer = Adam



Entrenamiento  
realizado con un  
factor de ruido Salt &  
Pepper de 0.2

El factor 0.2 implica invertir el 20% de los  
píxeles, en este caso 7

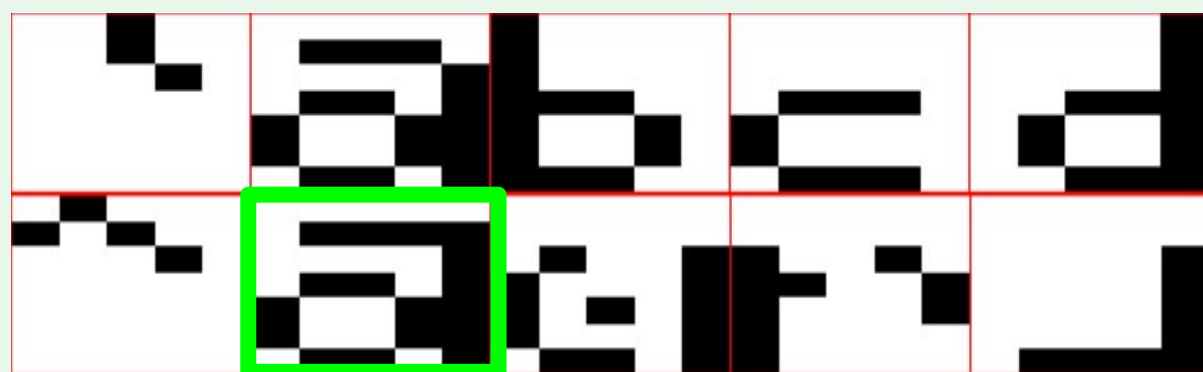
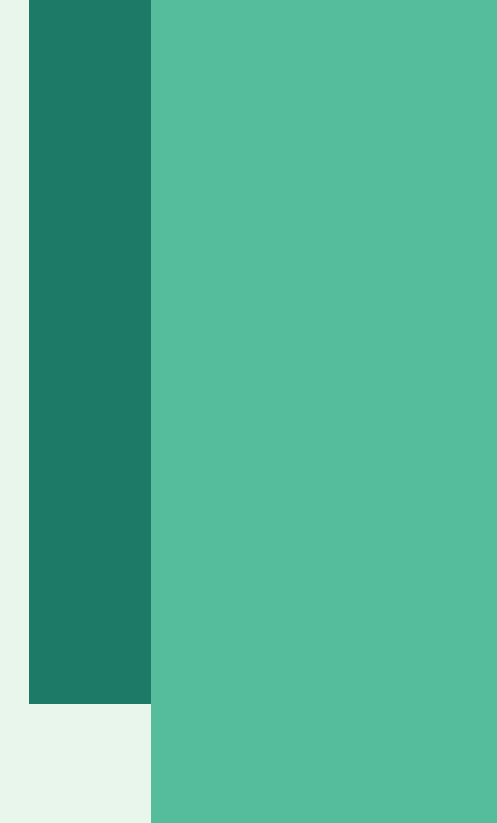
Epoch vs MSE



capas ocultas = [32, 8]  
n = 0.001  
epochs = 13000  
activation = tanh  
optimizer = Adam

Mín. Avg. Loss=0.10176

# Testing Dataset (Salt & Pepper - 0.2)



original

Predicción

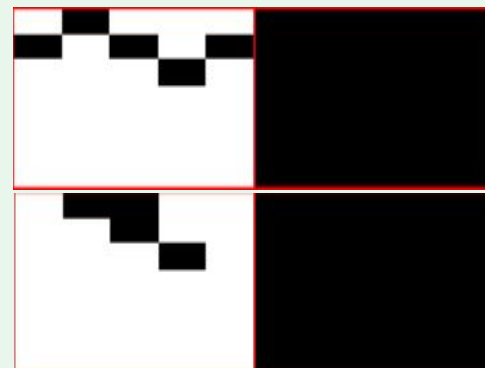
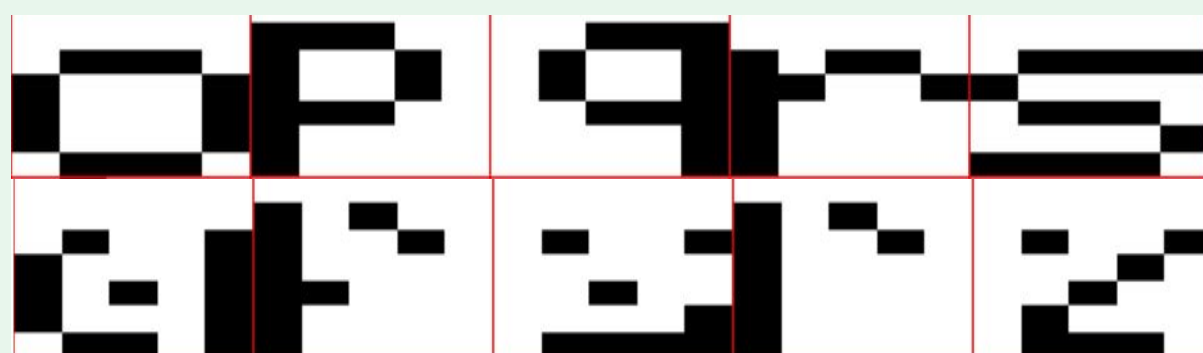
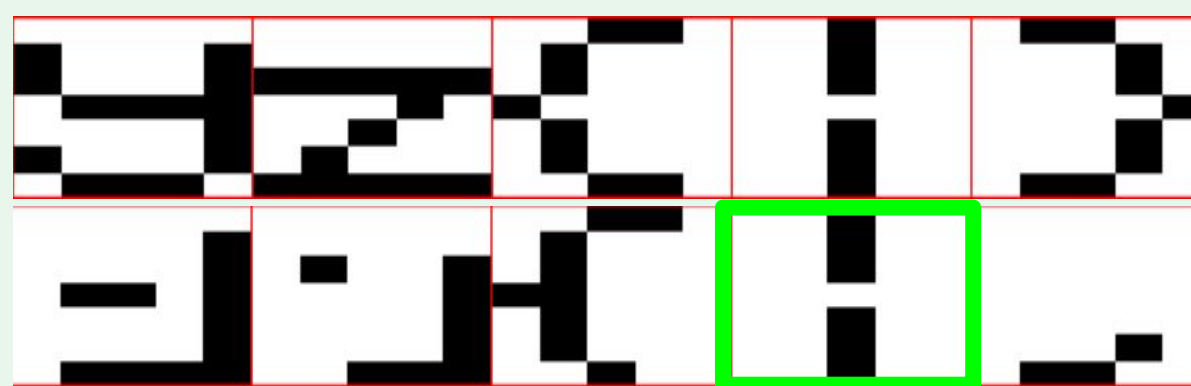
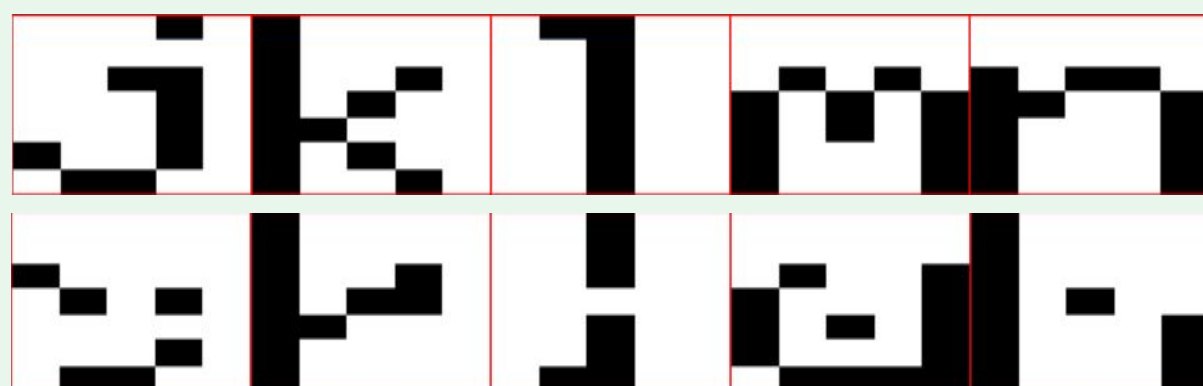
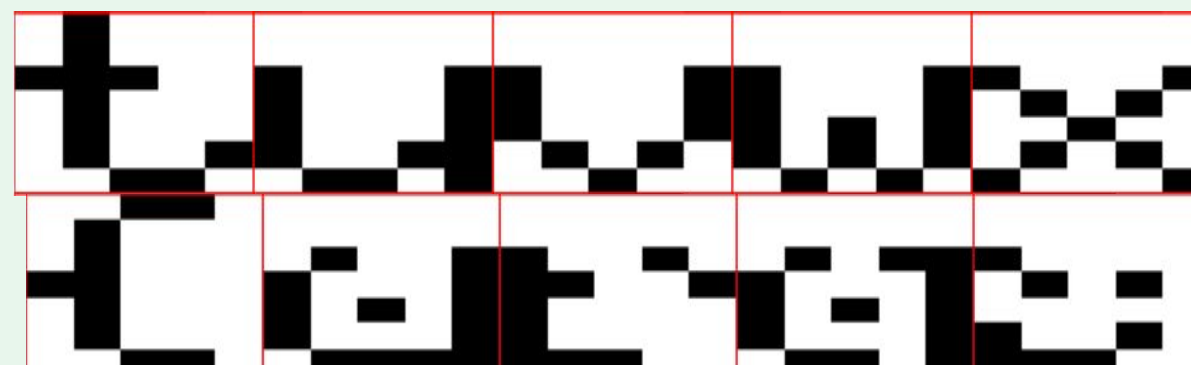
capas ocultas = [32, 8]

$n = 0.001$

epochs = 13000

activation = tanh

optimizer = Adam



■ Predicción perfecta

■ Predicción con 1 píxel de error

# ***Resultados (Salt & Pepper – 0.2)***

Como podemos apreciar en nuestra mejor configuración obtuvimos 172 píxeles incorrectos dentro de los 1120 píxeles totales del dataset. Lo que da una tasa de error del 15.35%

# *Investigación*

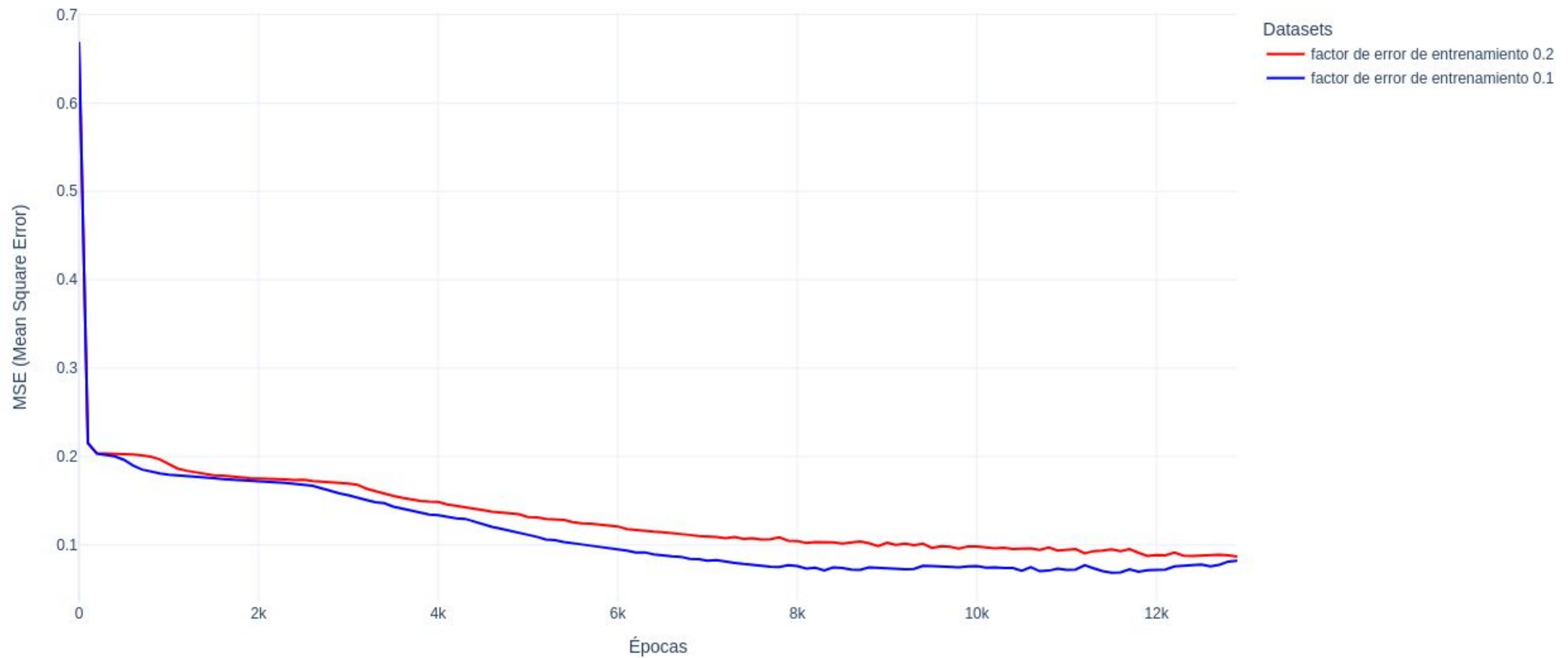
A raíz de los resultados obtenidos surge la siguiente interrogante:

¿Si entrenamos el denoising autoencoder con inputs con un factor de ruido de 0.2 pero lo evaluamos con inputs de ruido 0.1 deberíamos obtener mejores resultados que en el caso donde se lo entrenó con ruido 0.1?

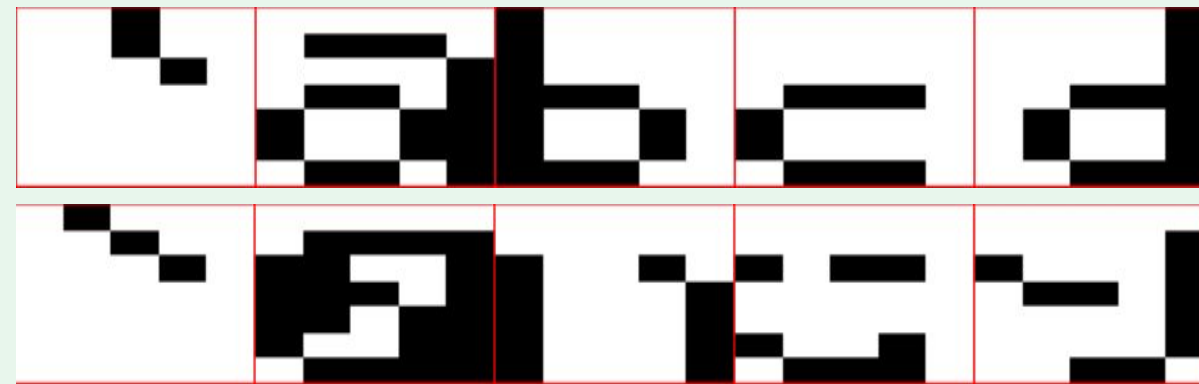
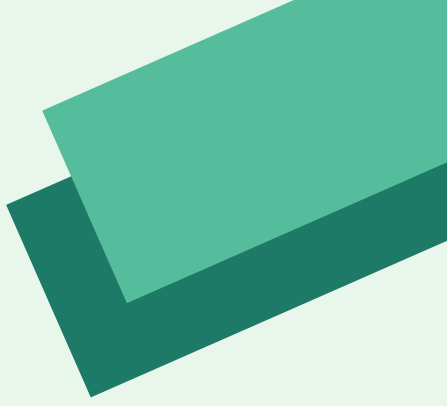
Nuestra hipótesis es que en esta evaluación debería reducirse la tasa de error ya que estamos exigiendo al autoencoder en su entrenamiento para evaluarlo en predicciones más sencillas

capas ocultas = [32, 8]  
n = 0.001  
epochs = 13000  
activation = tanh  
optimizer = Adam

Comparación de MSE vs Épocas para distinta testing set con factor de error de 0.1



# Resultados



original

predicción

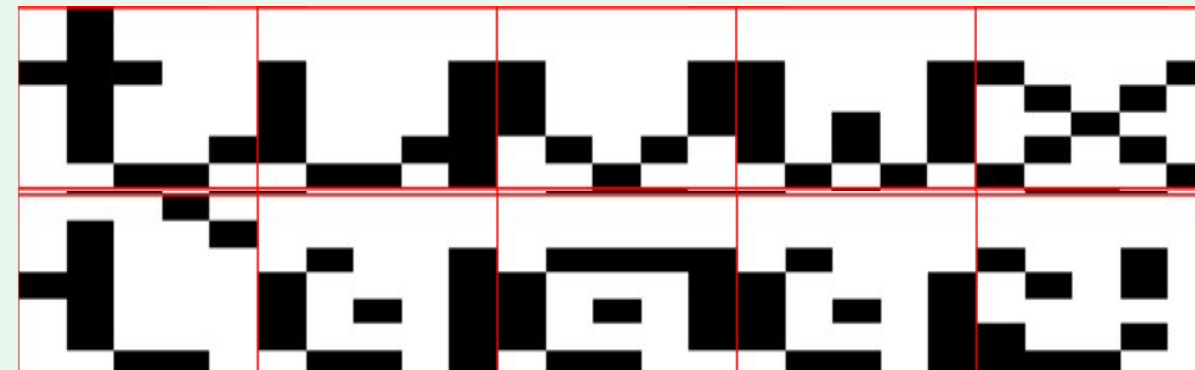
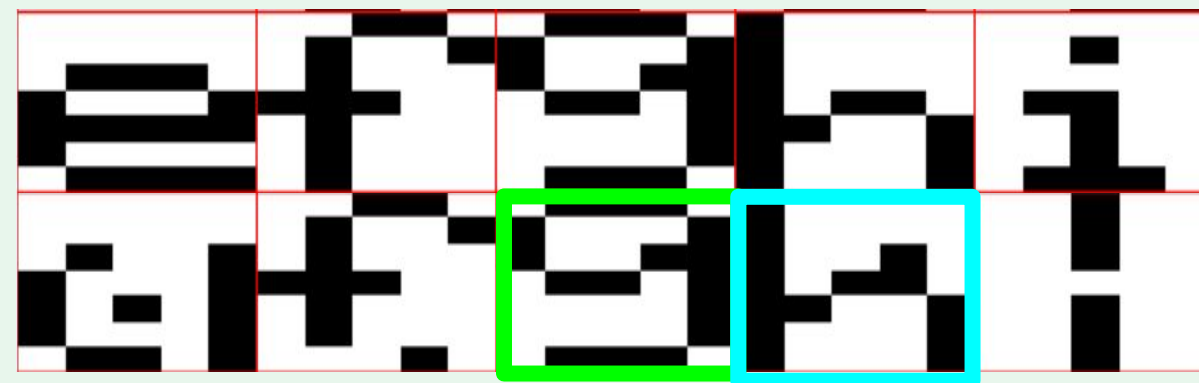
capas ocultas = [32, 8]

$\eta = 0.001$

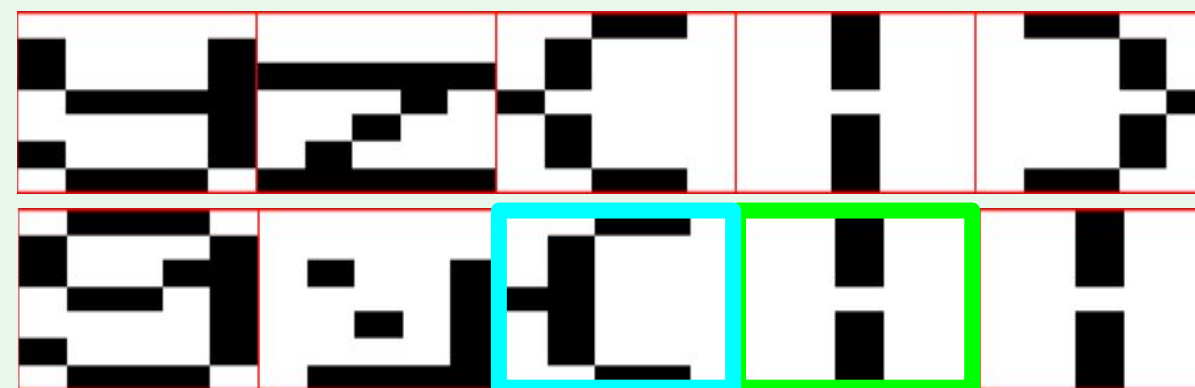
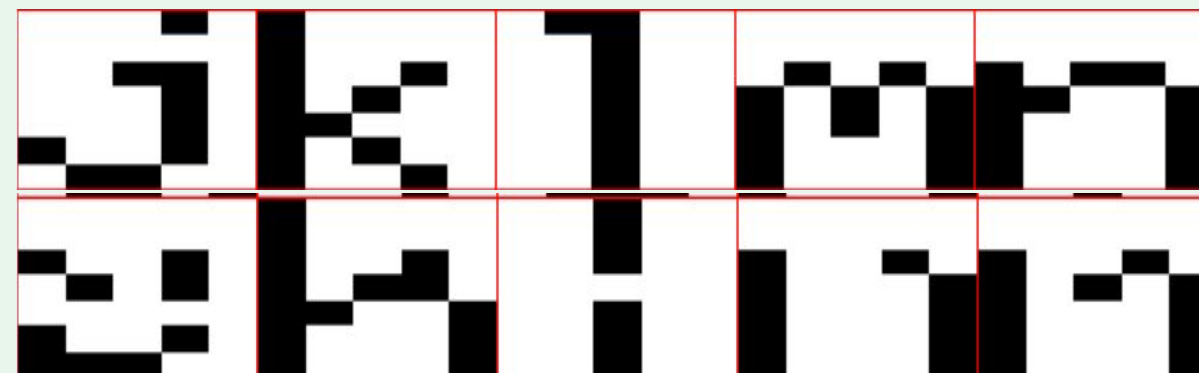
epochs = 11500

activation = tanh

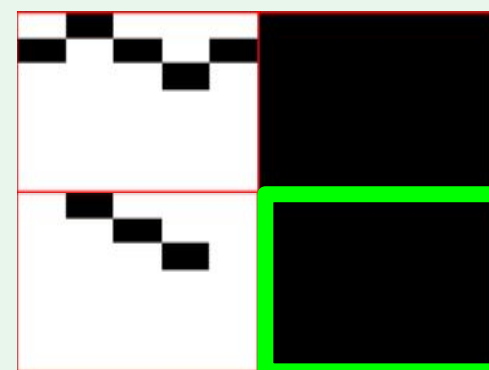
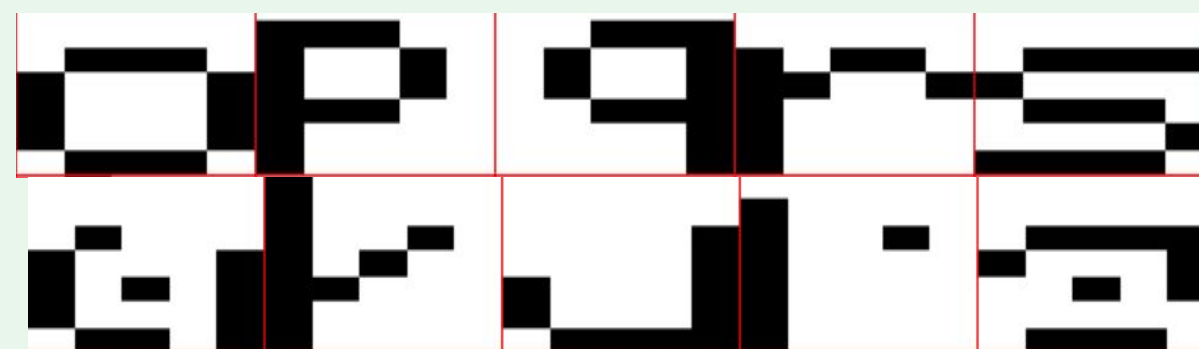
optimizer = Adam



Entrenamiento  
realizado con un  
factor de ruido Salt &  
Pepper de 0.2



Predicción realizada  
con un factor de ruido  
Salt & Pepper de 0.1



■ Predicción perfecta

■ Predicción con 1 píxel  
de error



# ***Resultados investigación***

Como podemos apreciar en nuestra mejor configuración obtuvimos 165 píxeles incorrectos dentro de los 1120 píxeles totales del dataset. Lo que da una tasa de error del 14.73%



# Conclusión

Nuestra hipótesis es incorrecta, de hecho aumentó el mse respecto al entrenamiento con el conjunto 0.1 así como también la cantidad de caracteres correctos y con 1 píxel de error

Esto posiblemente se deba a que el conjunto aprende mejor los patrones de los caracteres al tener una menor cantidad de ruido en el entrenamiento, es decir, no por agregarle cada vez más ruido aprenderá mejor, sino todo lo contrario.



***Entrega Original***  
***26-11-24***



# ***Autoencoder Clásico***

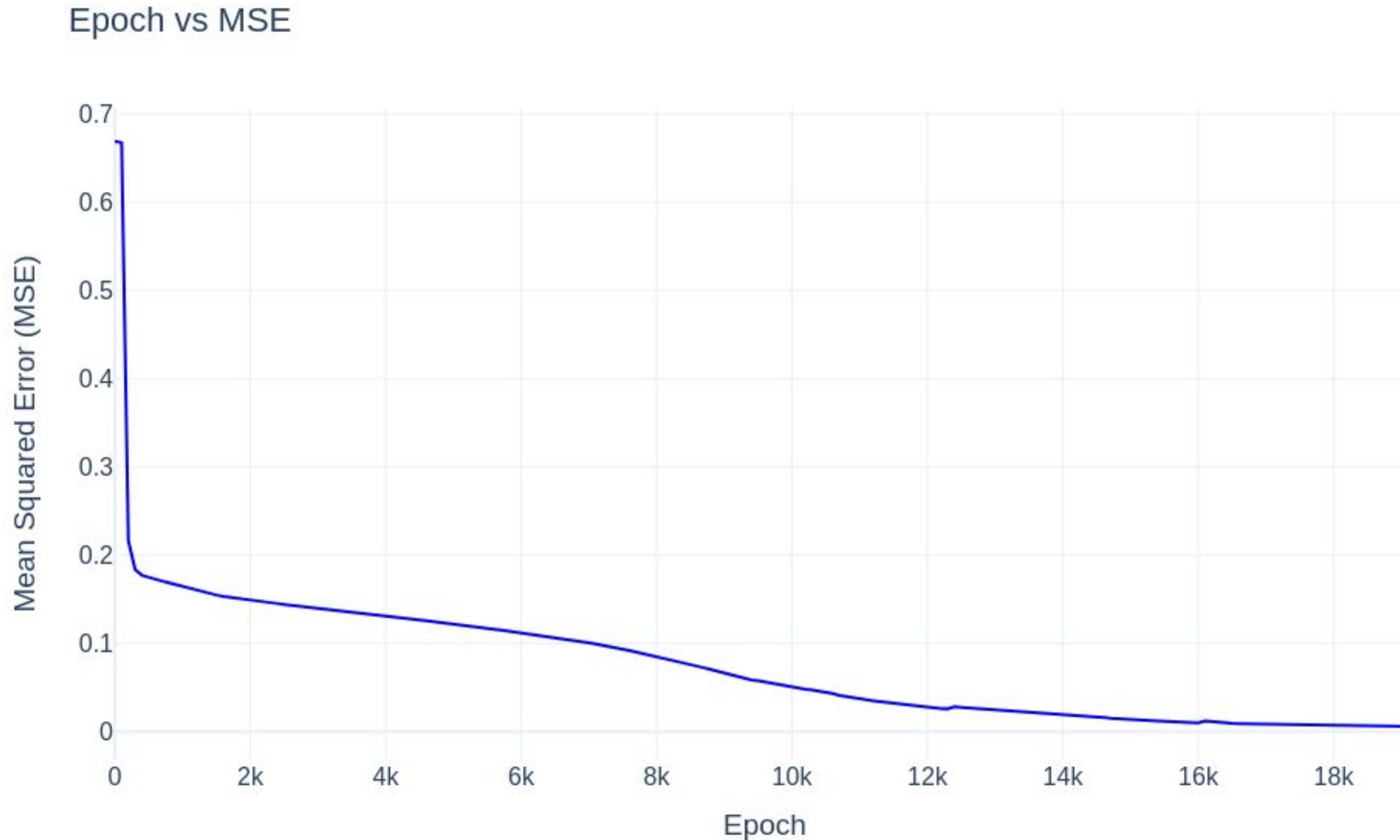
# ***Elección de arquitectura***

Para decidir cuál era la mejor arquitectura observamos el Mean Square Error (MSE) en el perceptrón multicapa

Notamos que es más sencillo dar con arquitecturas que converjan a un MSE bajo durante el entrenamiento cuando cuentan con no más de 2 capas ocultas tanto en el encoder como en el decoder

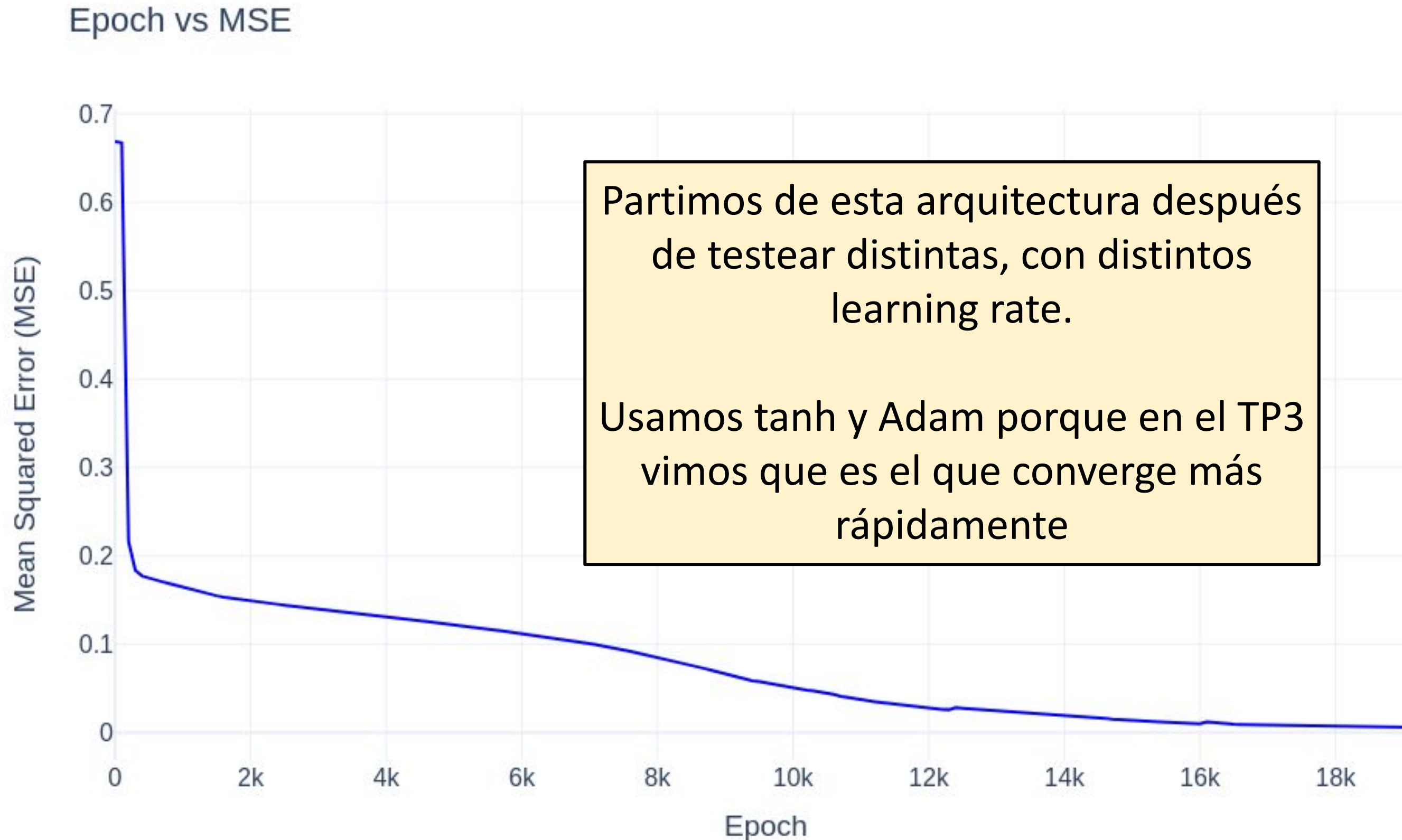
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

# Mejor configuración



capas ocultas = [20]  
n = 0.0005  
epochs = 19300  
activation = tanh  
optimizer = Adam

# Mejor configuración



Partimos de esta arquitectura después de testear distintas, con distintos learning rate.

Usamos tanh y Adam porque en el TP3 vimos que es el que converge más rápidamente

capas ocultas = [20]  
n = 0.0005  
epochs = 19300  
activation = tanh  
optimizer = Adam

# ***Interrogantes***

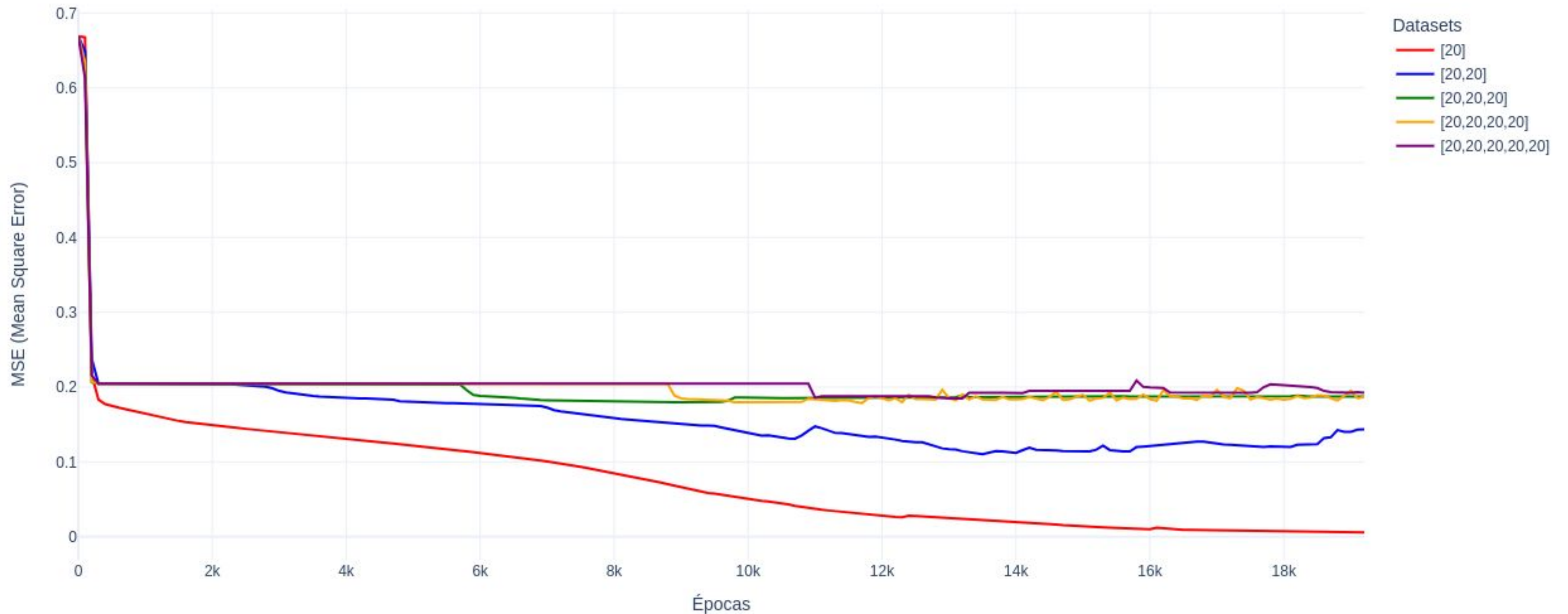
Luego de encontrar la mejor arquitectura que pudimos nos surgieron las siguientes preguntas:

- ¿Realmente son mejores las arquitecturas con menos capas en comparación con las que tienen más?
- Dado que el mse es más chico con un learning rate más chico ¿Será mejor la optimización con Momentum dado que genera una convergencia más lenta que Adam?

# Comparación de arquitectura por cantidad de layers

capas ocultas = [20]  
n = 0.0005  
epochs = 19300  
activation = tanh  
optimizer = Adam

Comparación de MSE vs Épocas para distinta cantidad de layers

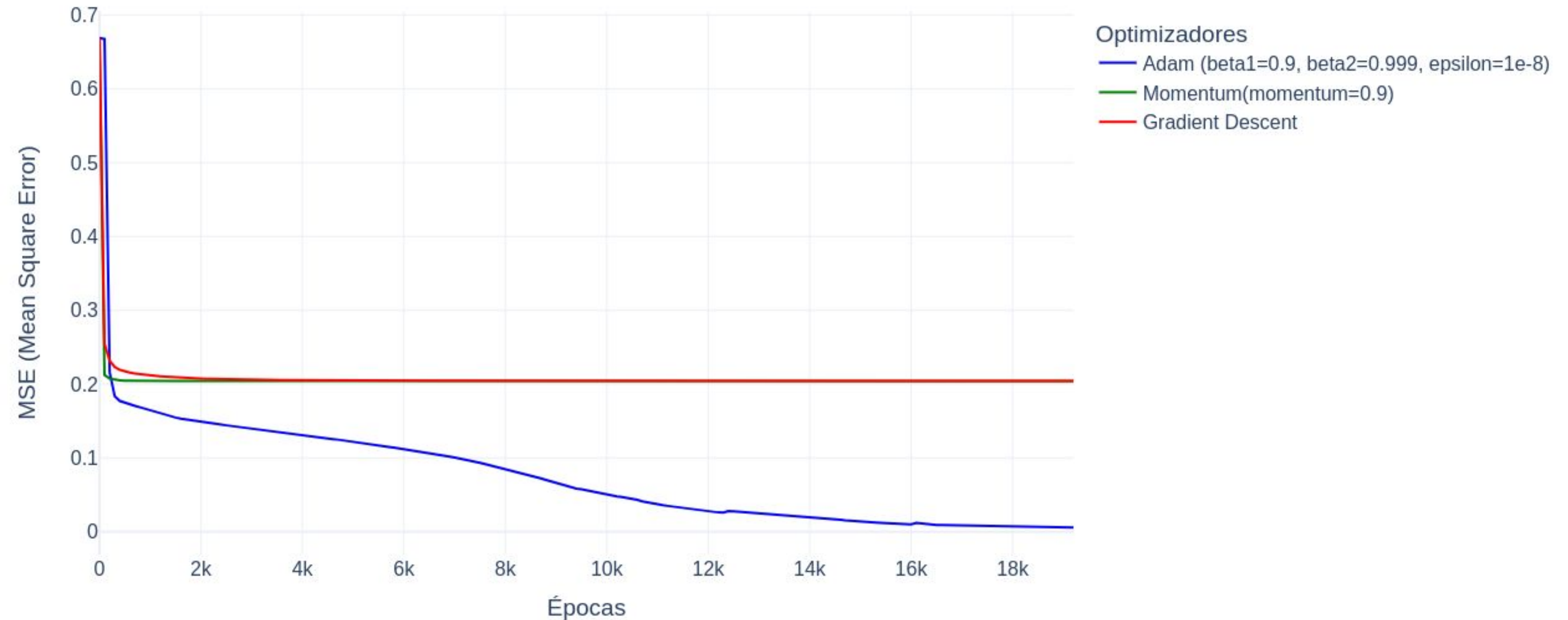




# Comparación de optimizers para la mejor configuración

capas ocultas = [20]  
n = 0.0005  
epochs = 19300  
activation = tanh

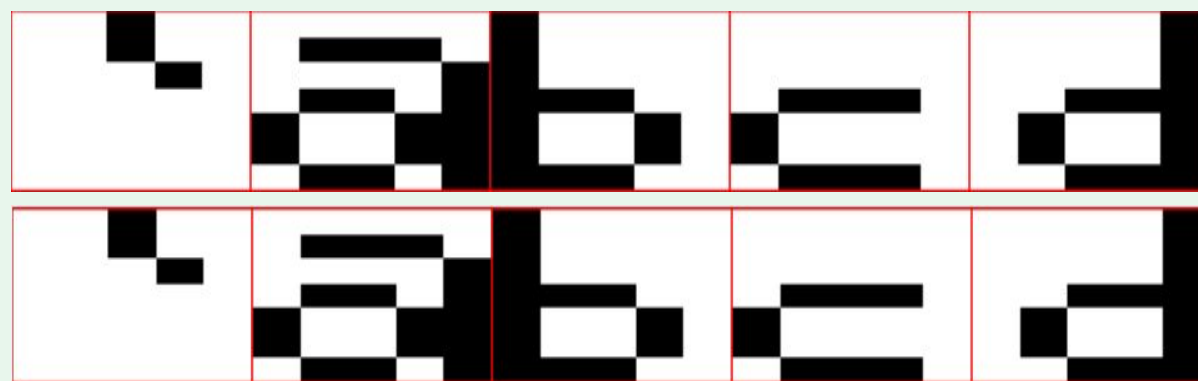
Comparación de MSE vs Épocas para Distintos Optimizadores



# *Resultados y observaciones*

- Es complicado encontrar una arquitectura
- Por lo experimentado, la convergencia del error no es correlativa a los parámetros de la arquitectura: Comparado a los trabajos anteriores, en este no se pueden hacer pruebas donde vamos incrementando de a poco los parámetros hasta alcanzar la solución. Existen soluciones “aisladas”. Una determinada arquitectura puede no tener una convergencia del mse pero al aumentar en 1 sola neurona una de las capas podemos tener una convergencia a un mse chico.

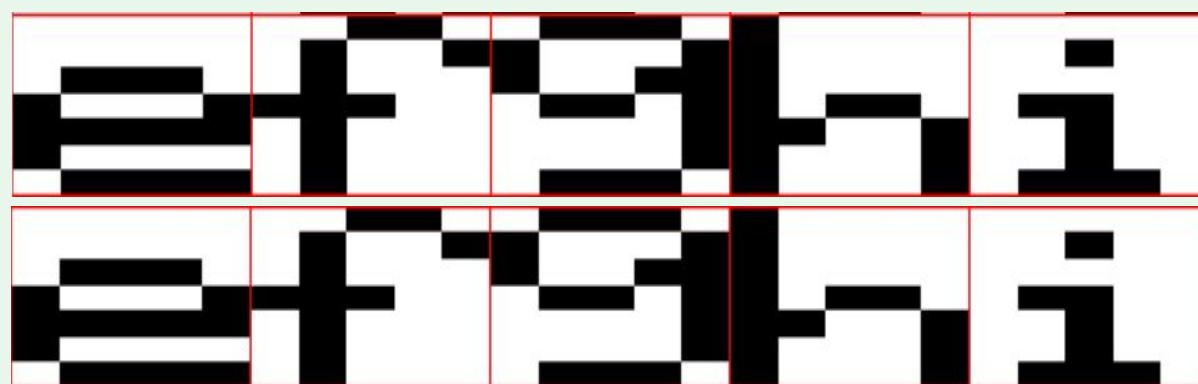
# Predicciones



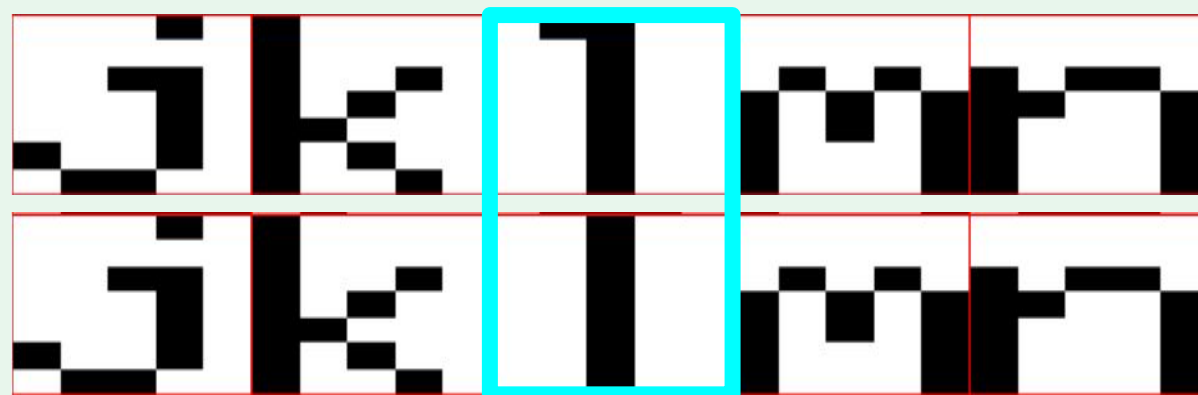
original

predicción

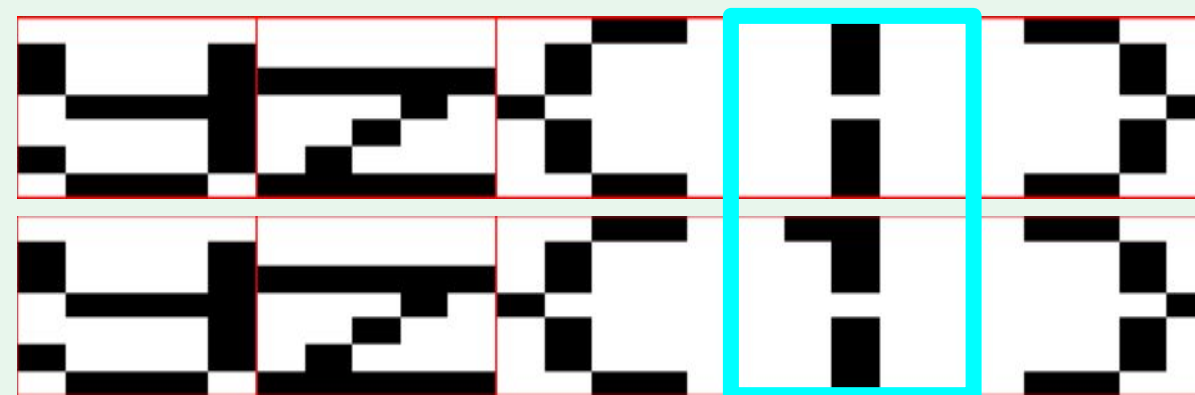
- capas ocultas = [20]
- $n = 0.0005$
- epochs = 19300
- activation = tanh
- optimizer = Adam



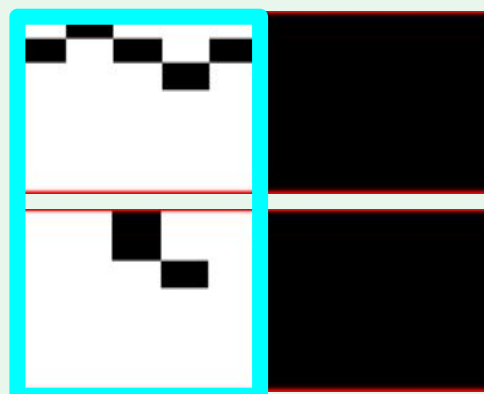
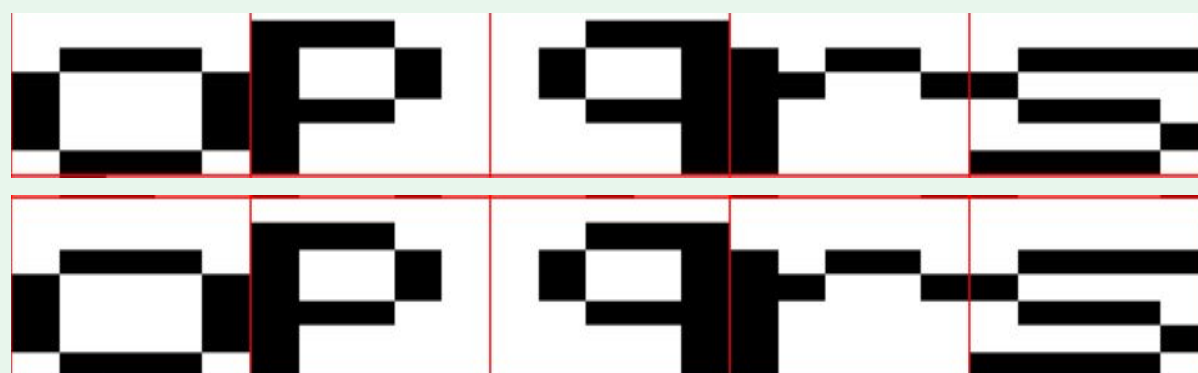
1 Error



1 Error

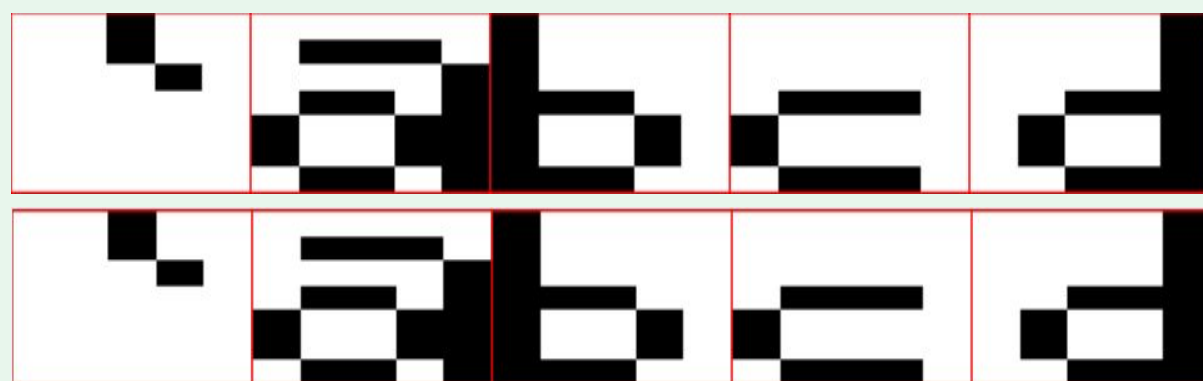


4 Errores: No aprendió el carácter



Lo asoció al  
acento grave

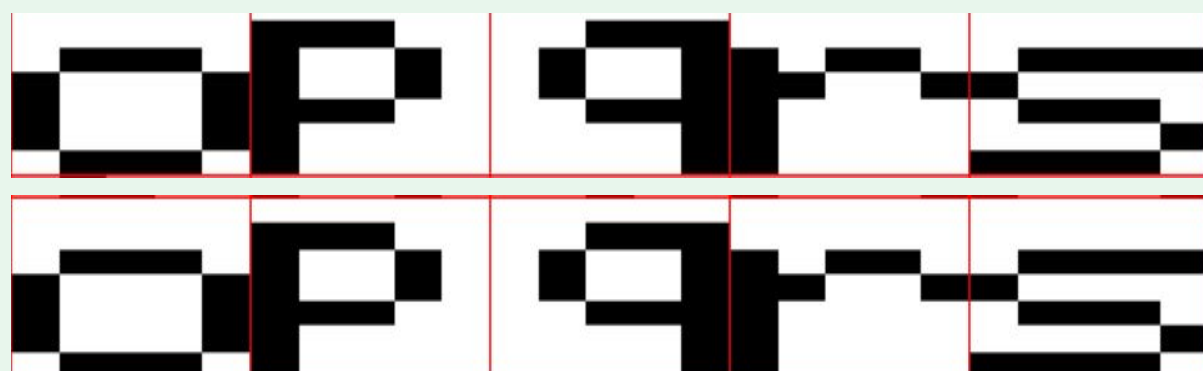
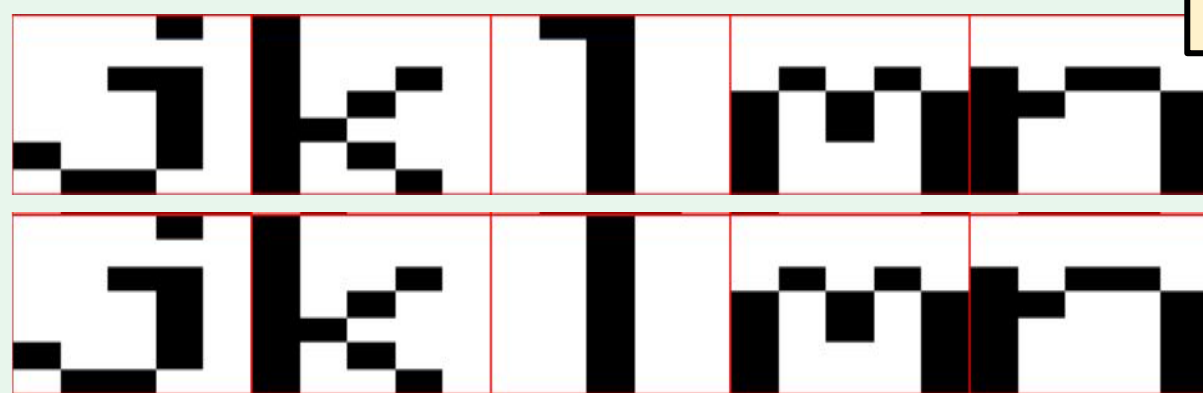
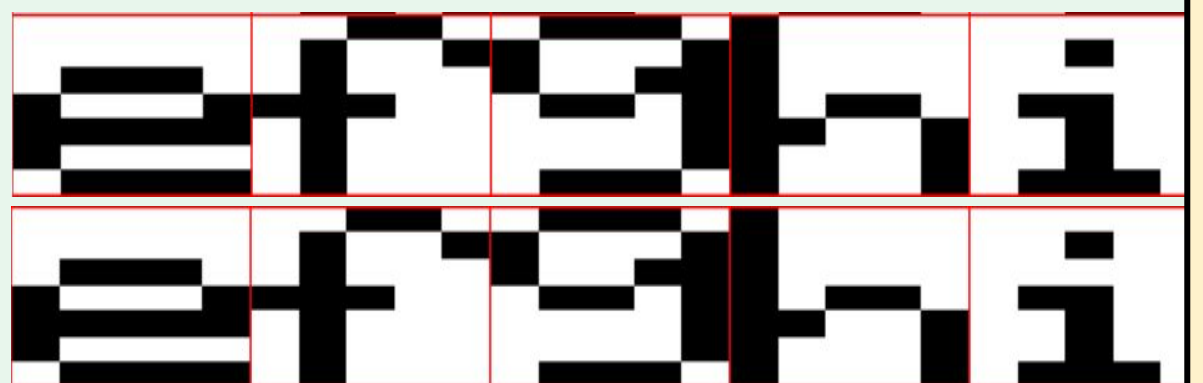
# Predicciones



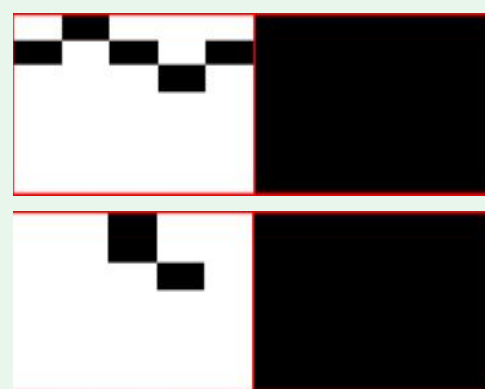
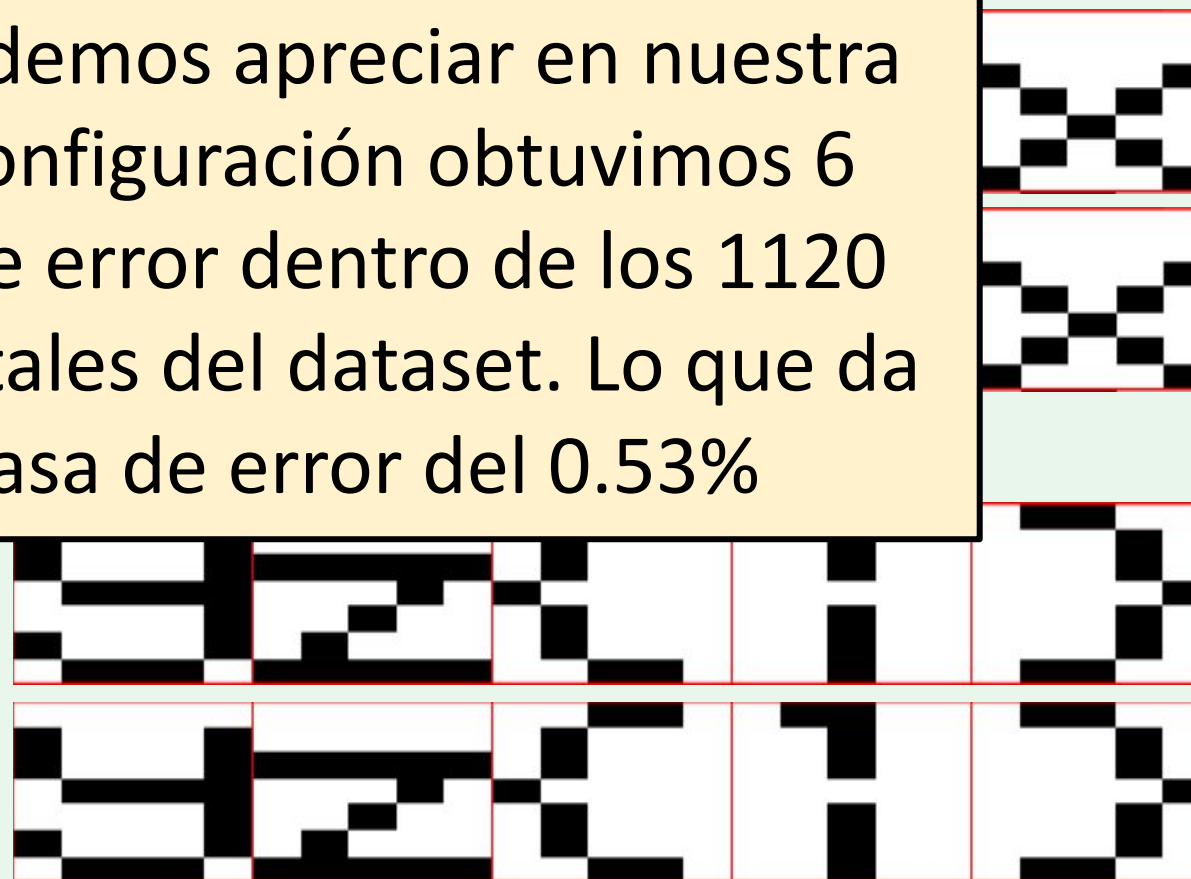
→ original

→ predicción

- capas ocultas = [20]
- $n = 0.0005$
- epochs = 19300
- activation = tanh
- optimizer = Adam



Como podemos apreciar en nuestra mejor configuración obtuvimos 6 píxeles de error dentro de los 1120 píxeles totales del dataset. Lo que da una tasa de error del 0.53%



# ***Distribución del dataset en el espacio latente***

Predicciones

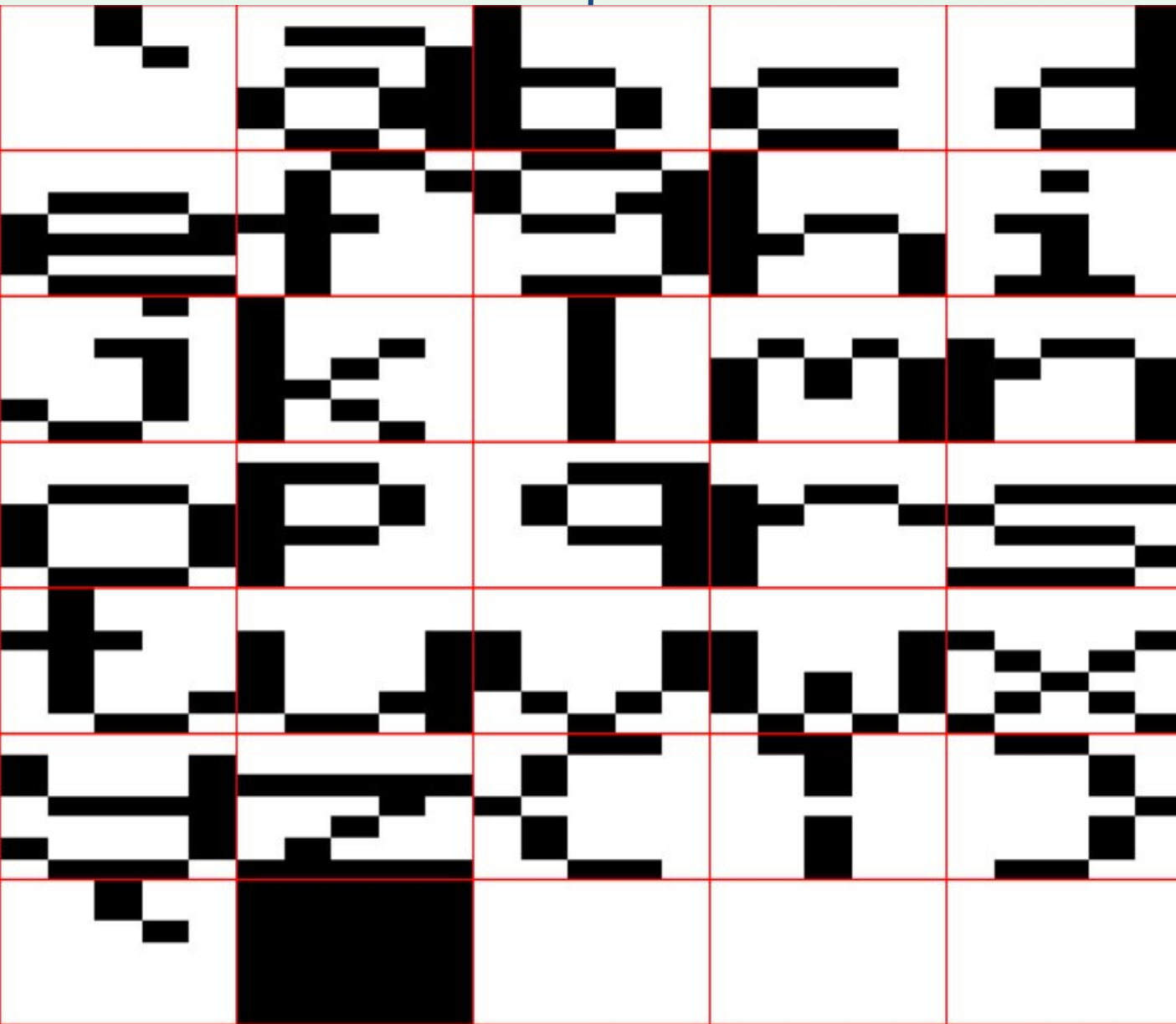
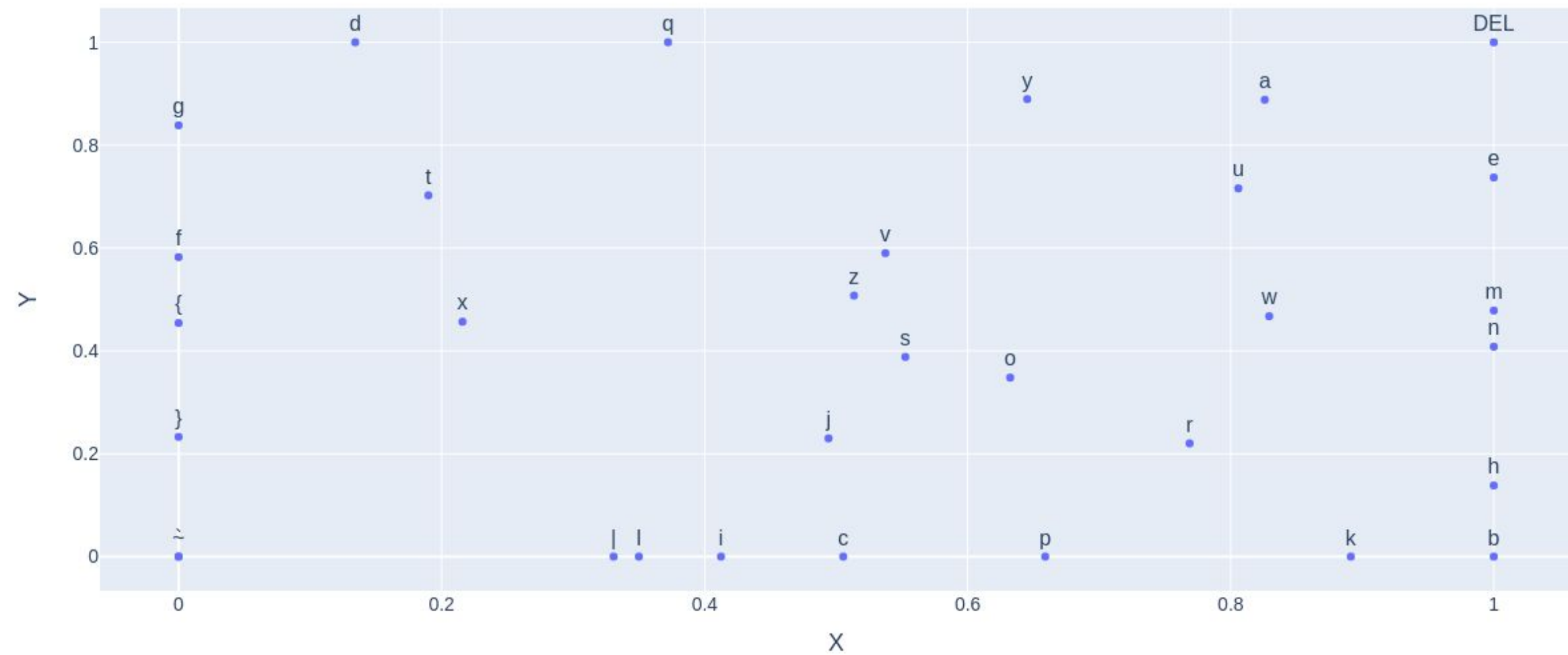


Gráfico de dispersión de las predicciones en el espacio latente

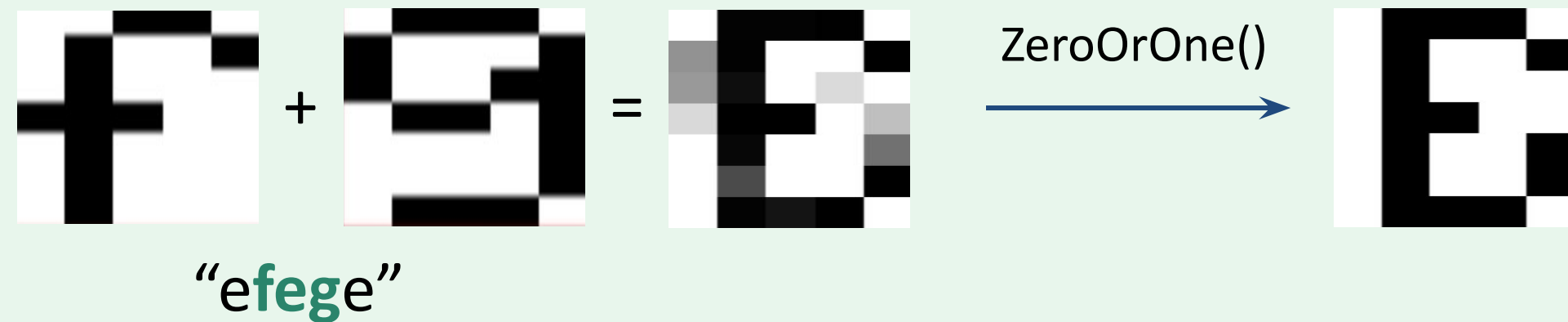




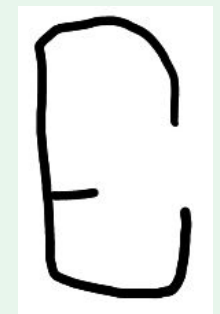


# Caracteres encontrados

Se tomaron dos caracteres cercanos en el espacio latente y se trazó una línea por la cual nos fuimos moviendo para elegir distintos puntos que fuimos representando utilizando el decoder del autoencoder entrando. Luego “limpiamos” las representaciones con la función ZeroOrOne.



La letra “feg” es más bien parecida a una ‘E’ con la diferencia de que no es simétrica



La letra “ble” es como un caracter omega con la parte de arriba “cerrada”





# ***Denoising Autoencoder***

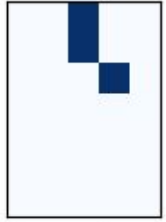


# ***Preparación***

- Se optó por usar Salt and Pepper (con 0.1 de noise\_level) como ruido para la matriz de caracteres.
- Se aplicó dicho ruido para cada uno de los caracteres.

# Caracteres sin ruido

Character 1



Character 2



Character 3



Character 4



Character 5



Character 6



Character 7



Character 8



Character 9



Character 10



Character 11



Character 12



Character 13



Character 14



Character 15



Character 16



Character 17



Character 18



Character 19



Character 20



Character 21



Character 22



Character 23



Character 24



Character 25



Character 26



Character 27



Character 28



Character 29



Character 30



Character 31

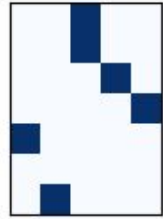


Character 32



# Caracteres con ruido

Character 1



Character 2



Character 3



Character 4



Character 5



Character 6



Character 7



Character 8



Character 9



Character 10



Character 11



Character 12



Character 13



Character 14



Character 15



Character 16



Character 17



Character 18



Character 19



Character 20



Character 21



Character 22



Character 23



Character 24



Character 25



Character 26



Character 27



Character 28



Character 29



Character 30



Character 31



Character 32



# ***Idea del Denoising***

Entrada con ruido



Salida esperada



# ***Elección de arquitectura***

- Similar a lo que se realizó con el autoencoder normal.
- Se comparó el MSE (Mean Squared Error) en el perceptrón multicapa.
- Se notó que con un número adecuado de capas ocultas se obtiene la mejor performance.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

# Mse vs Épocas



$n = 0.001$   
epochs = 9100  
activation = tanh  
optimizer = Adam

# Comparación de optimizadores



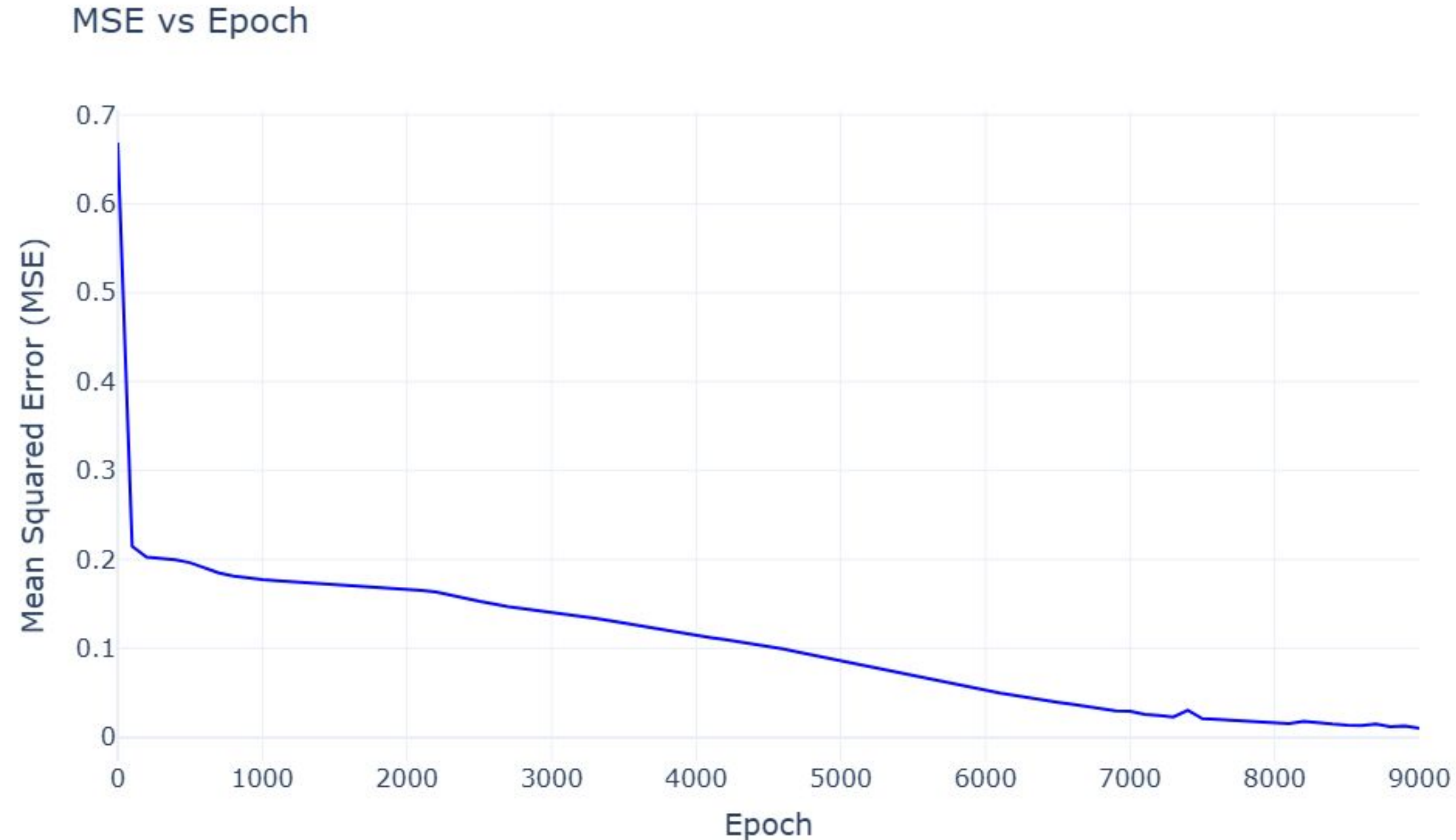


# Comparación de learning rates



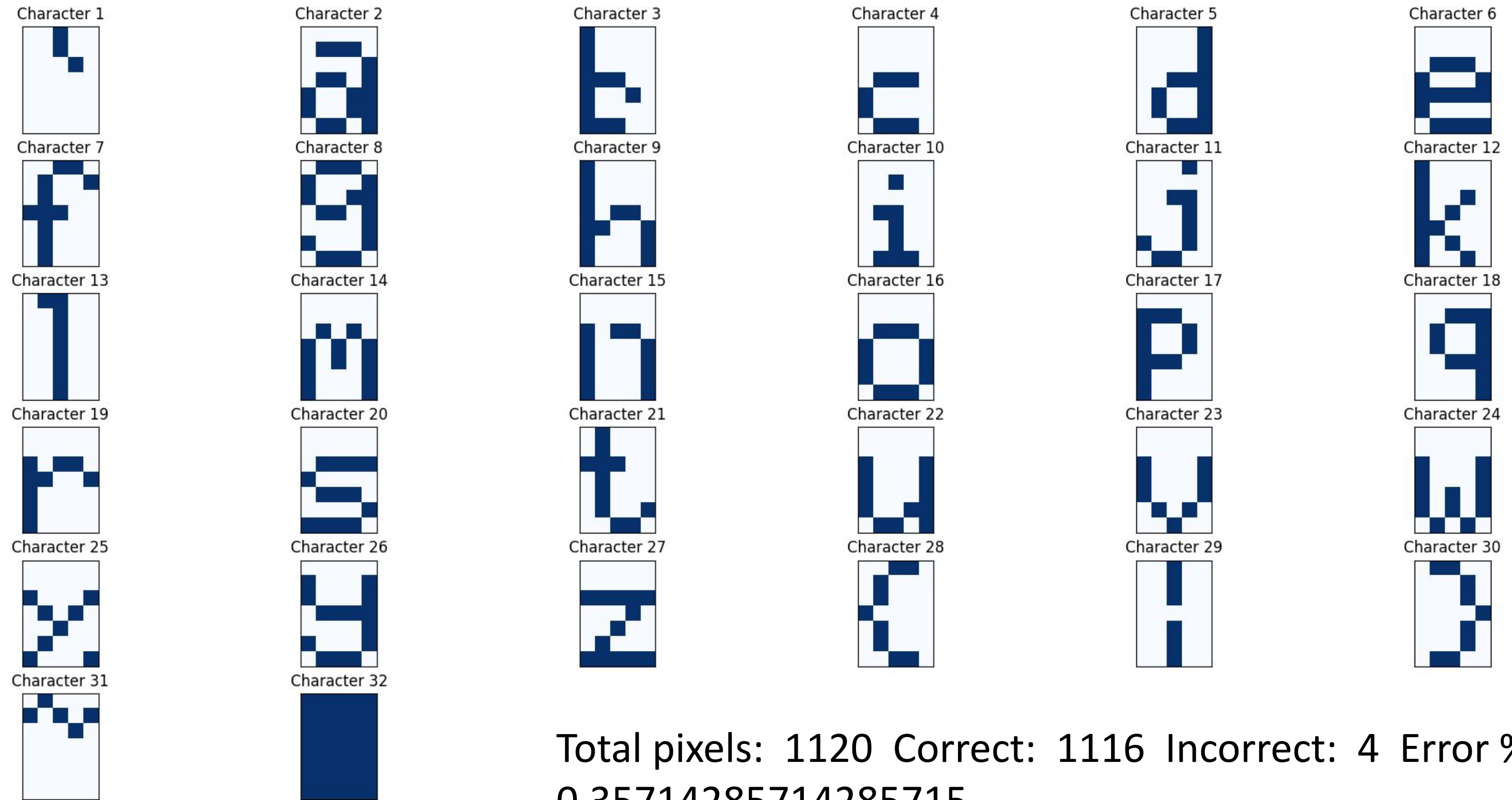


# Mejor configuración



arquitectura =  
[35,32,8,2,8,32,35]  
capas ocultas = [32,8]  
n = 0.001  
epochs = 9100  
activation = tanh  
optimizer = Adam

# Predicción del autoencoder



Total pixels: 1120 Correct: 1116 Incorrect: 4 Error %:  
0.35714285714285715



# ***Predicción del autoencoder***

Original

Character 3



Predicción

Character 3



# ***Predicción del autoencoder***

Original

Character 8



Predicción

Character 8



# ***Predicción del autoencoder***

Original

Character 15



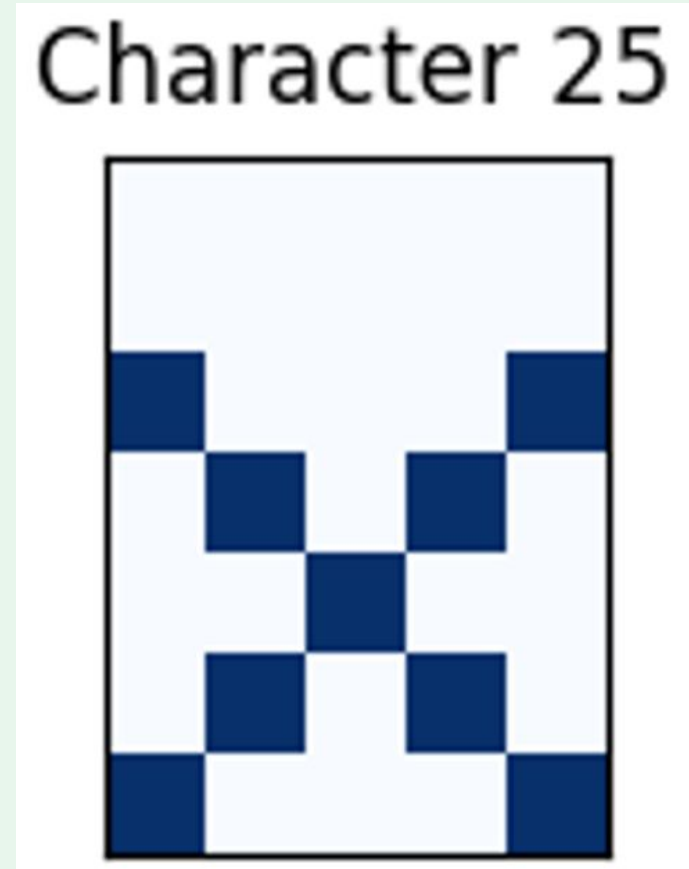
Predicción

Character 15

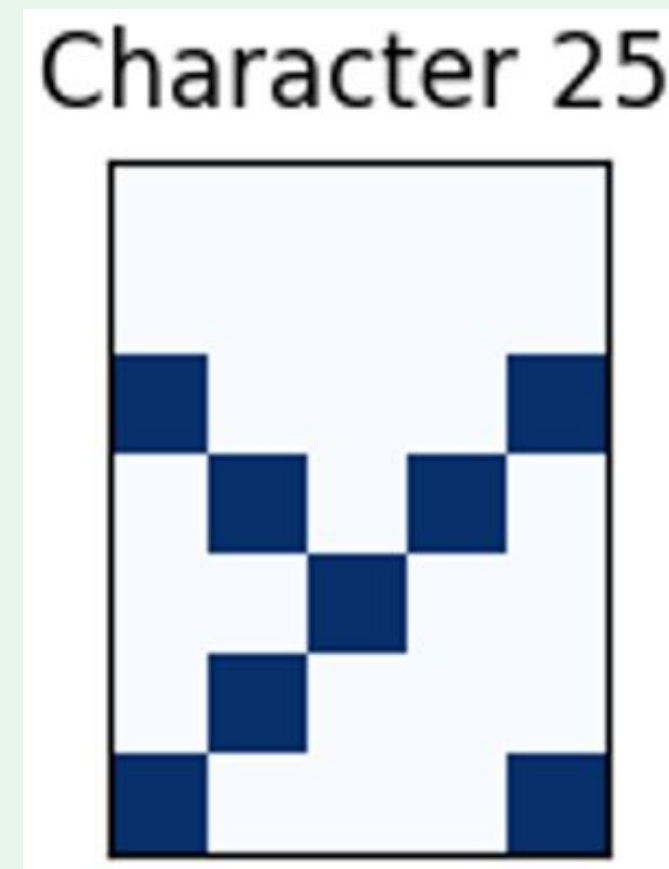


# ***Predicción del autoencoder***

Original



Predicción



# ***Conclusiones***

- Viendo los resultados se puede concluir que realiza una gran aproximación ya que solo 4 letras tuvieron un píxel de error.
- Al igual que en el ítem anterior, no se puede que haya una mejor arquitectura que la propuesta.
- Incluso con caracteres que son similares aplicándoles ruido (como pueden ser la h y la n) los distingue correctamente.





# ***Autoencoder Variacional***

# *Dataset 1*

8 Emojis:



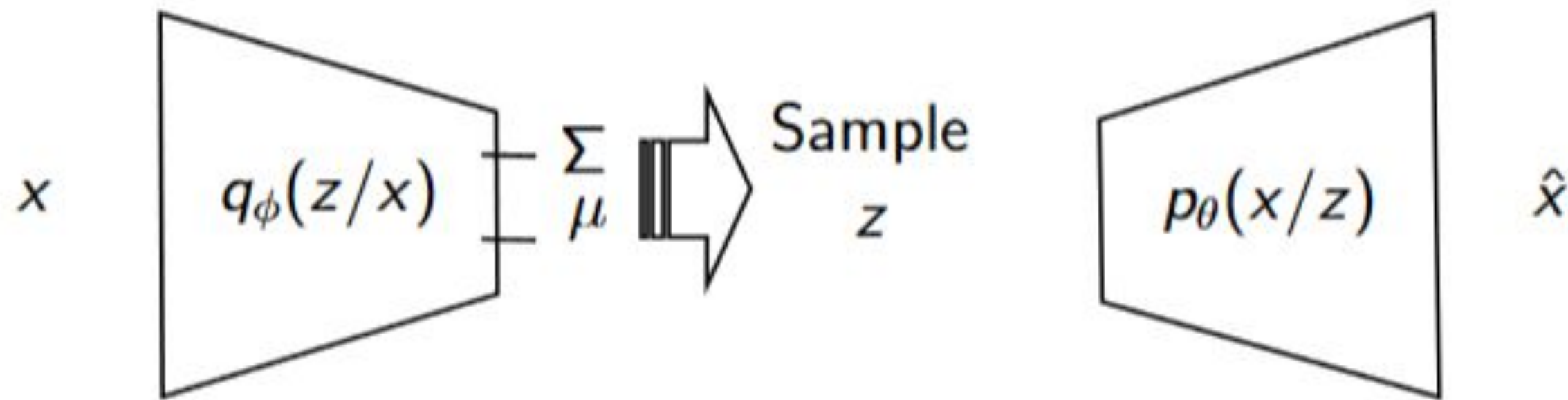
# ***Dataset 1***

20x20 pixels c/u  
(grayscale)

Se aplanan los 400  
píxeles en un vector  
unidimensional.



# Arquitectura



Para elegir la arquitectura se comparó el total loss.  
Se busco la arquitectura que **MINIMICE**:

$$-\mathcal{L} = \underbrace{-\mathbb{E}_{q(z)} \log p(x/z)}_{\text{Error de reconstrucción}} + \underbrace{KL(q(z)||p(z))}_{\text{Término regularizador}}$$

# ***Hiperparametros***

Se testean las siguientes arquitecturas:

[400,200,2,200,400]

[400,200,100,2,100,200,400]

[400,50,30,2,30,50,400]

[400,150,50,15,2,15,50,150,400]

Se usara ADAM con  $b1=0.9$  y  $b2=0.999$ . Se testeara su learning rate:

[0.1,0.01,0.001,0.0001]

Full Batch

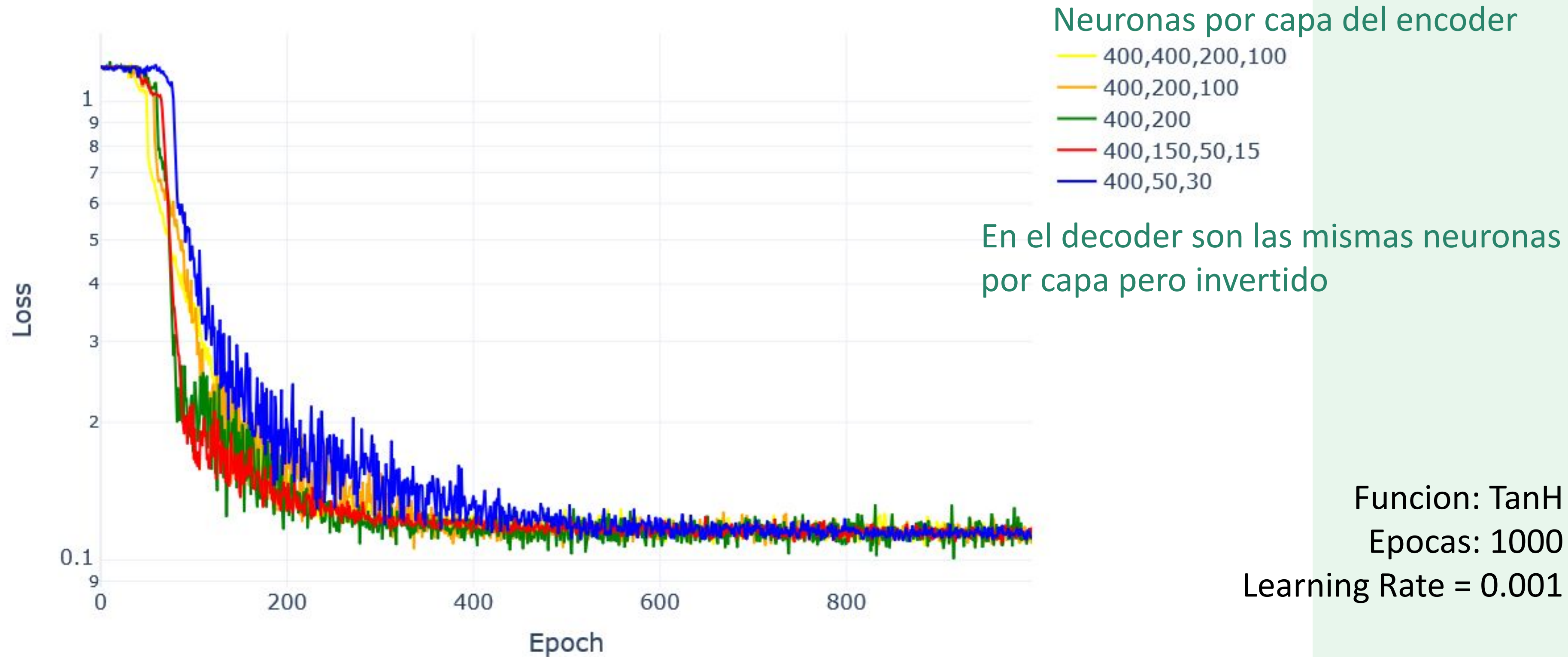
Funcion de activacion: TanH

Epocas: 1000

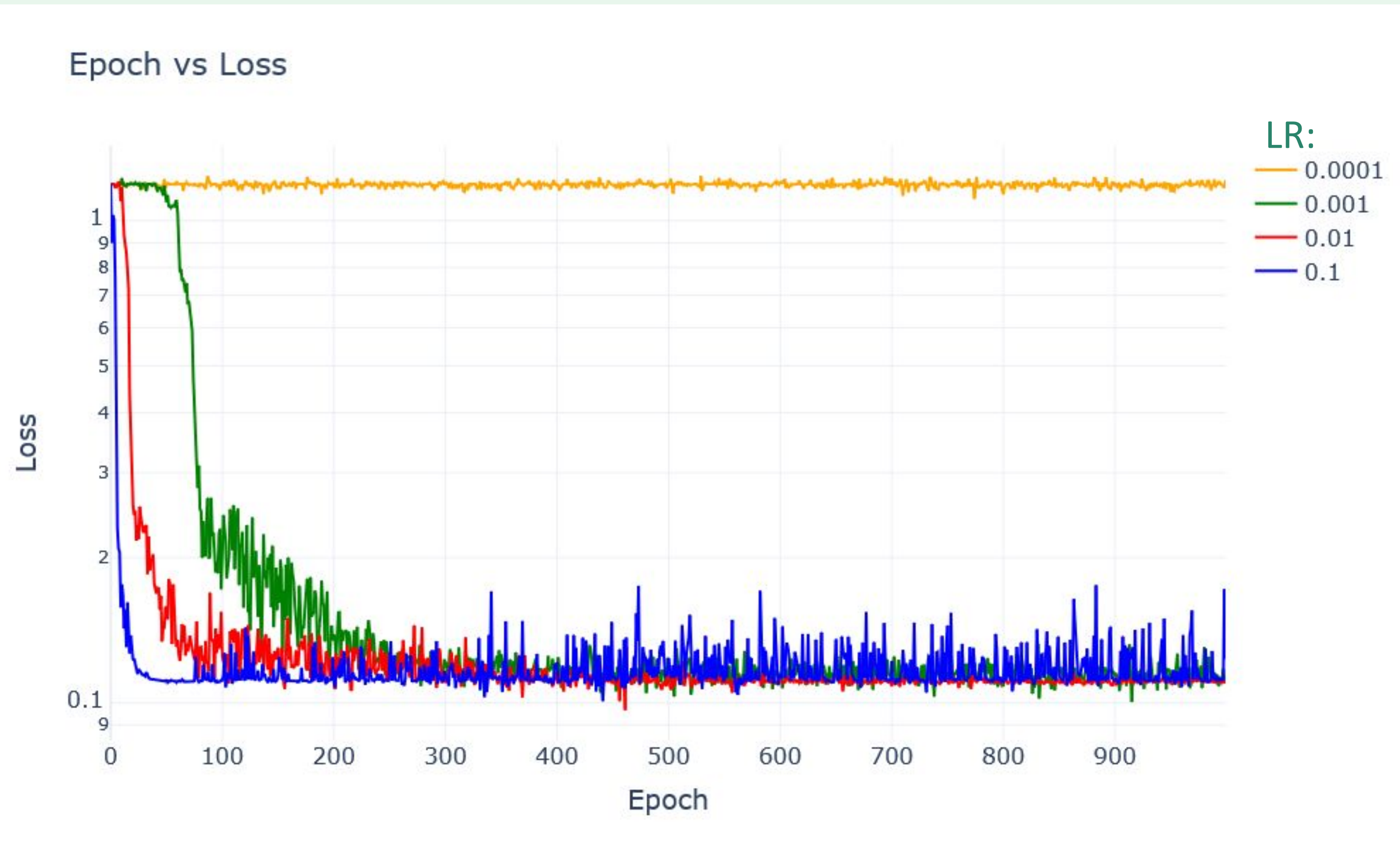


# *Loss para distintas arquitecturas*

Epoch vs Loss



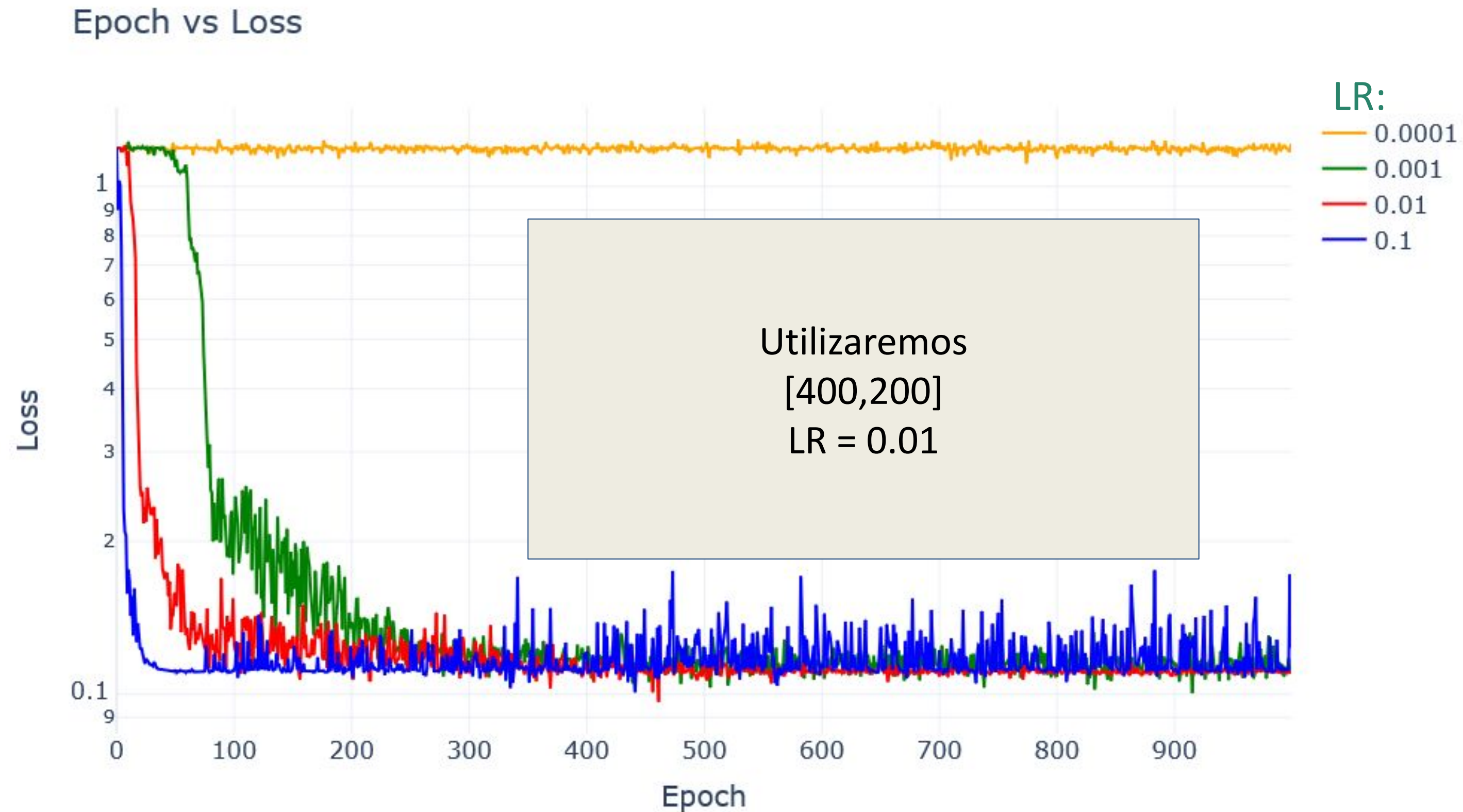
# *Loss para distintos LR*



Funcion: TanH  
Epocas: 1000  
Arquitectura =  
[400,200,2,200,400]



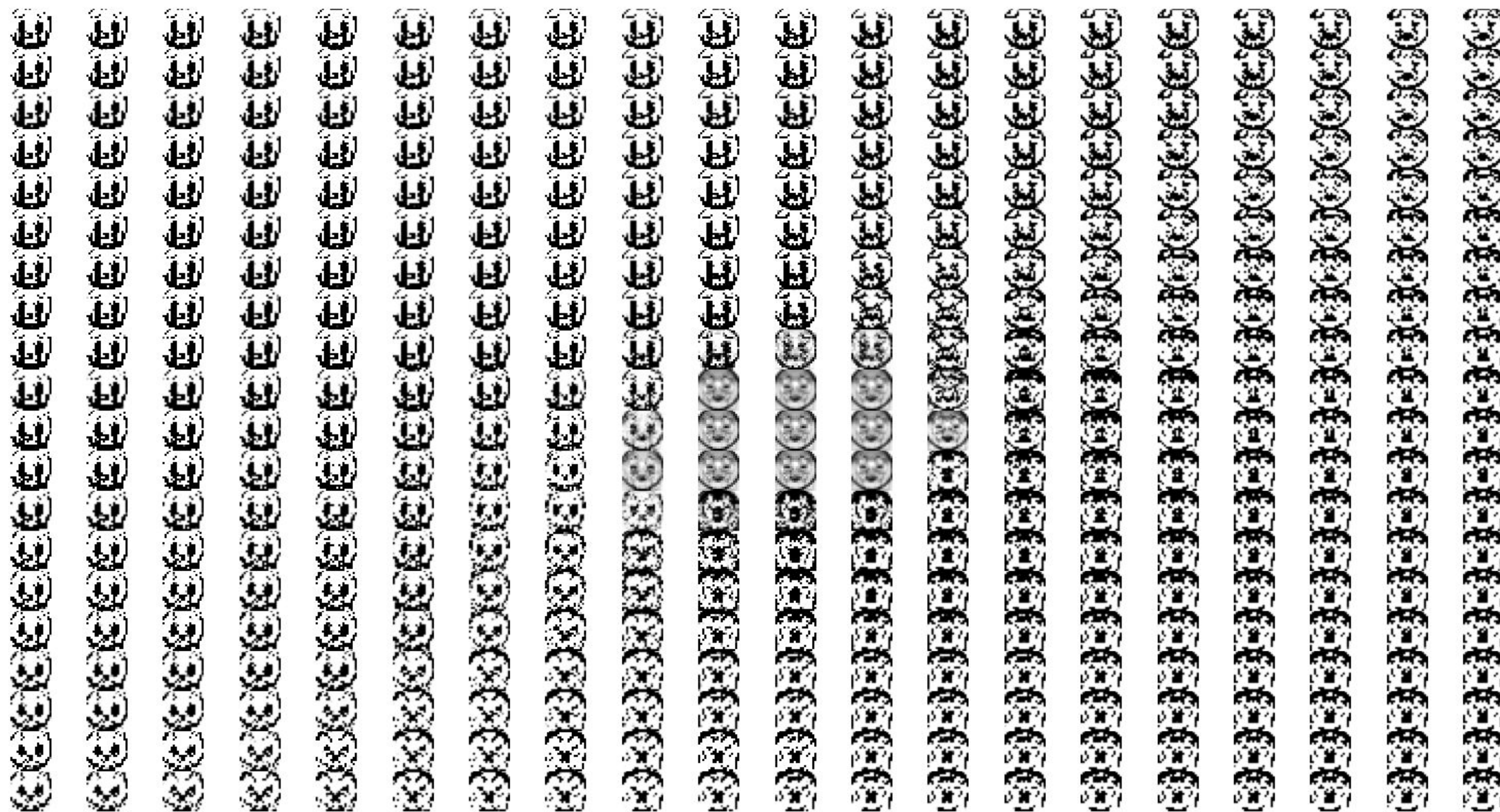
# *Loss para distintos LR*



Funcion: TanH  
Epocas: 1000  
Arquitectura =  
[400,200,2,200,400]



# Emojis generados



Funcion: TanH

Epocas: 1000

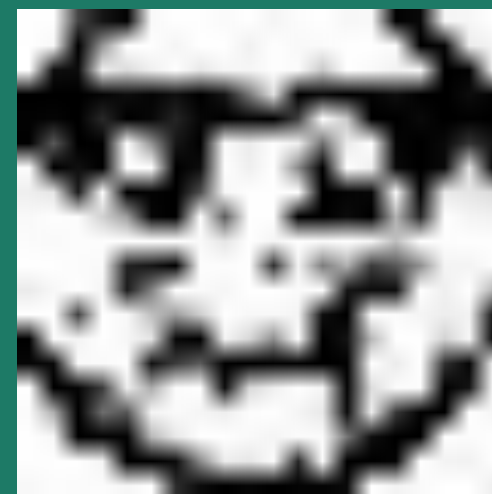
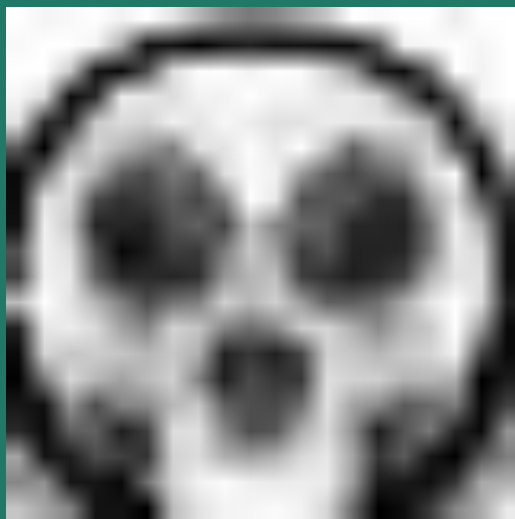
Arquitectura =

[400,200,2,200,400]

Learning Rate = 0.01



# ***Emojis interesantes***



## ***Dataset 2***

4 Emojis, muy diferentes entre si:



# ***Dataset 2***

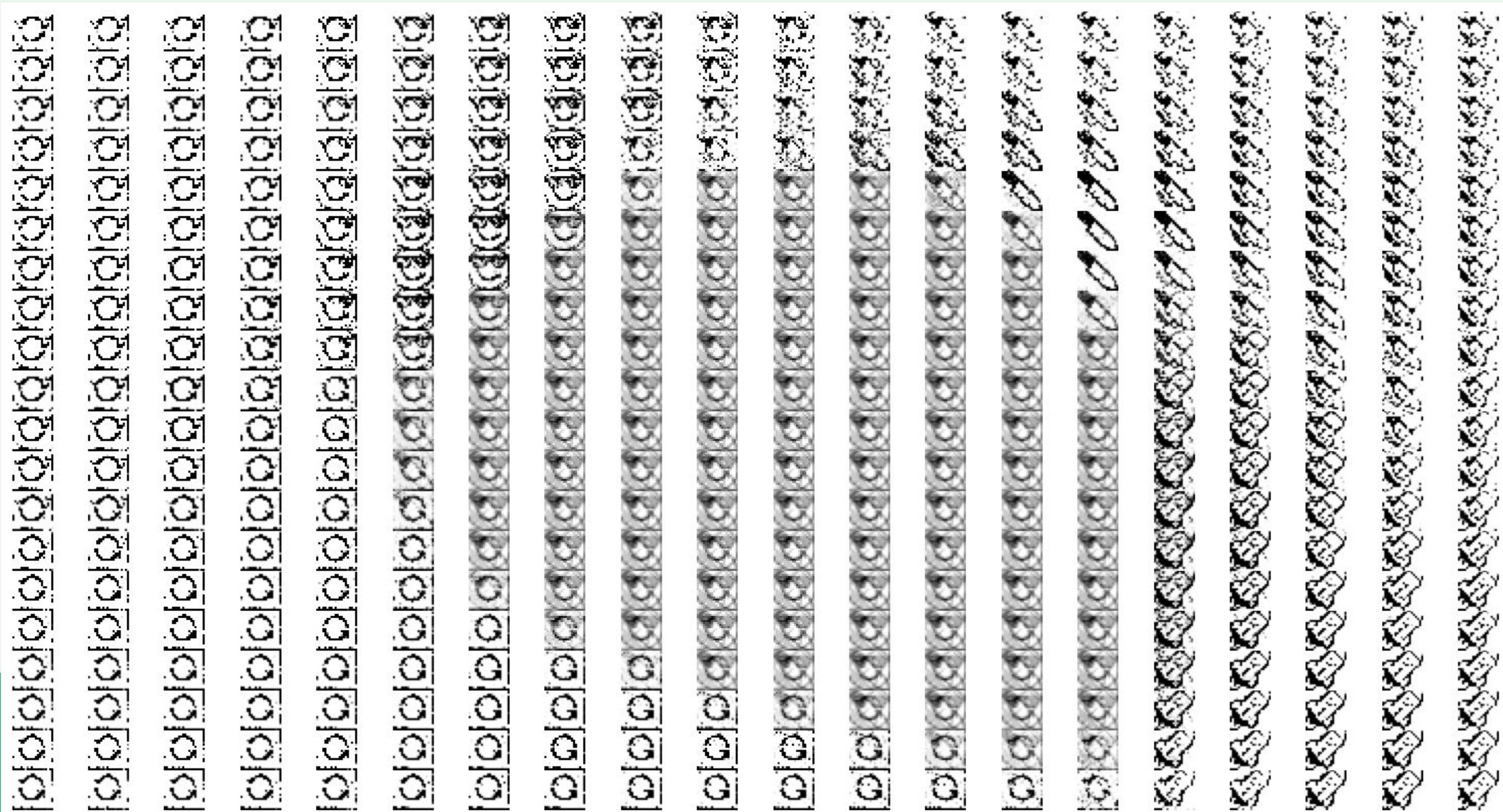
20x20 pixels c/u  
(grayscale)

Se aplanan los 400  
píxeles en un vector  
unidimensional.





# Emojis generados



Funcion: TanH

Epocas: 1000

Arquitectura =

[400,200,2,200,400]

Learning Rate = 0.01

# ***Emojis interesantes***





***Muchas gracias!***

