(72.27) Sistemas de Inteligencia Artificial

Trabajo Práctico Nº5

Deep Learning

Alejo Flores Lucey 62622
Andrés Carro Wetzel 61655
Ian Franco Tognetti 61215
Matías Daniel Della Torre 61016

01 Introducción

Presentación de los problemas a resolver.

02 Conventional Autoencoder

Reducción de dimensionalidad de letras minúscula binarias.

03 Denoising Autoencoder

Filtrado de ruido de letras minúsculas binarias.

04 Variational Autoencoder

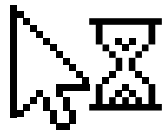
Generación de nuevos elementos a partir de otros elementos.

05 Conclusiones

Limitaciones y mejoras futuras.

Introducción

Hemos implementado una serie de autoencoders para resolver problemas utilizando distintos conjuntos de entrada.
El objetivo es proporcionar un modelo que aprenda reduciendo la dimensionalidad de los datos y además pueda generar elementos nuevos, similares al conjunto de datos proporcionado.



Conventional Autoencoder

Red neuronal que aprende representaciones comprimidas de los datos al codificarlos en un espacio latente más pequeño y luego reconstruirlos a su forma original.

Denoising Autoencoder

Red neuronal que aprende a reconstruir datos limpios a partir de su versión ruidosa, para recuperar su forma original sin ruido.

Variational Autoencoder

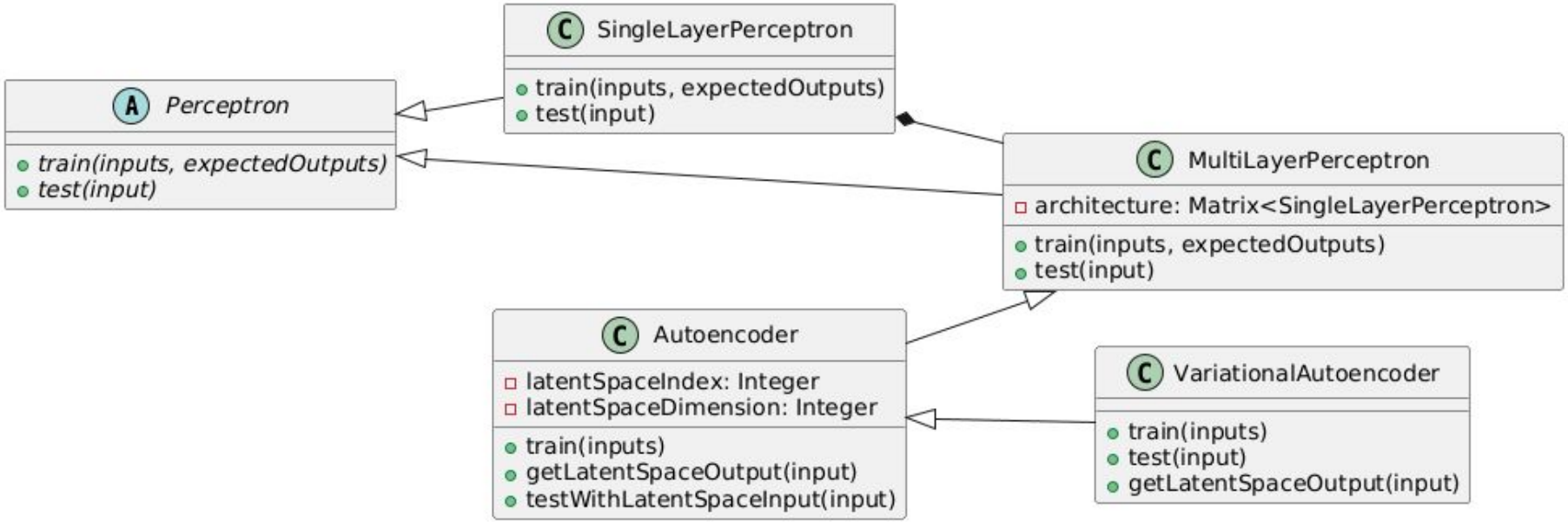
A diferencia de un autoencoder estándar, el VAE impone una estructura probabilística en el espacio latente, para generar nuevos datos similares al conjunto original.

Archivo de configuración

- Toda la configuración se maneja desde un archivo JSON.
- Se configura una semilla para obtener reproducibilidad en el caso de usar valores pseudo-aleatorios.
- Se elige el problema a resolver de las opciones disponibles.
- Se especifican los hiperparámetros de la red neuronal y las opciones específicas de cada problema.
- La arquitectura es completamente parametrizables.

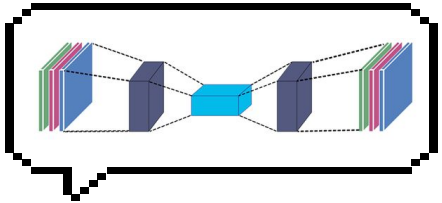
```
{
  "problem": "CONVENTIONAL | DENOISING | VARIATIONAL",
  "problemOptions": {
    "noiseType": "GAUSSIAN | SALT_AND_PEPPER", // Only needed for DENOISING problem
    "noiseThreshold": 0.1, // Only needed for SALT_AND_PEPPER noise type
    "noiseLevel": 0.1 // Only needed for GAUSSIAN noise type
  },
  "hyperparameters": {
    "epsilon": 3e-2,
    "maxEpochs": 10000,
    "optimizer": {
      "type": "GRADIENT_DESCENT | MOMENTUM | ADAM",
      "options": {
        "rate": 1e-2,
        "alpha": 0.9, // Only needed for MOMENTUM type
        "beta1": 0.9, // Only needed for ADAM type
        "beta2": 0.999 // Only needed for ADAM type
      }
    },
    "updater": {
      "type": "ONLINE | BATCH | MINI-BATCH",
      "options": {
        "batchSize": 15 // Only needed for MINI-BATCH type
      }
    }
  },
  "architecture": [
    {
      "neuronQty": 35,
      "activationFunction": {
        "type": "LINEAR | LOGISTIC | TANH | RELU",
        "options": {
          "beta": 0.5 // Only needed for LOGISTIC or TANH type
        }
      }
    }
  ],
  "seed": 732
}
```

Diagrama UML



Detalles de implementación

- Se hace uso del patrón **Singleton** para servir las constantes definidas en el archivo de configuración.
- Se hace uso de **Numpy** para realizar operaciones matriciales y vectoriales con facilidad.
- Los **hiperparámetros** de los métodos de optimización utilizados son los **especificados en el paper** correspiente (η,α,β_1,β_2)
- Se utiliza **Pickle** para almacenar representaciones binarias de las **redes neuronales entrenadas**.

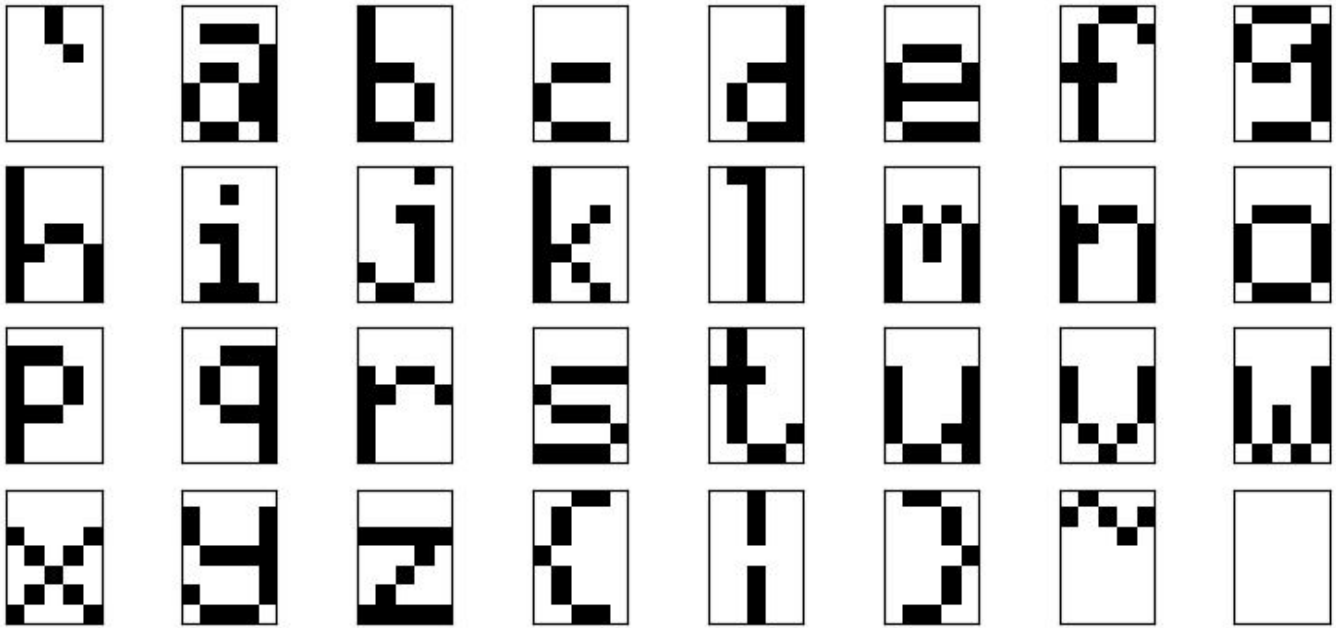


02

Conventional Autoencoder

Diseñaremos un autoencoder básico para representar patrones binarios de letras de 5×7 en un espacio latente de dos dimensiones, buscando un error máximo de un píxel incorrecto.

Definición del alfabeto



Ajuste de hiperparámetros

Se describe el método para seleccionar la mejor arquitectura:

1. Sólo con 5 letras:
    a. Se prueban distintas arquitecturas.
    b. Se varía el optimizador.
2. Se aumenta el subset de letras.
    a. 8 letras.
    b. 16 letras.
    c. 20 letras.
    d. 24 letras.
    e. Full dataset (32 letras).
3. Se varía el método de actualización de pesos.

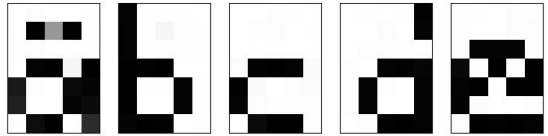
Objetivo: error de 1 píxel $\Longrightarrow$ MSE $\leqslant$ 0.025

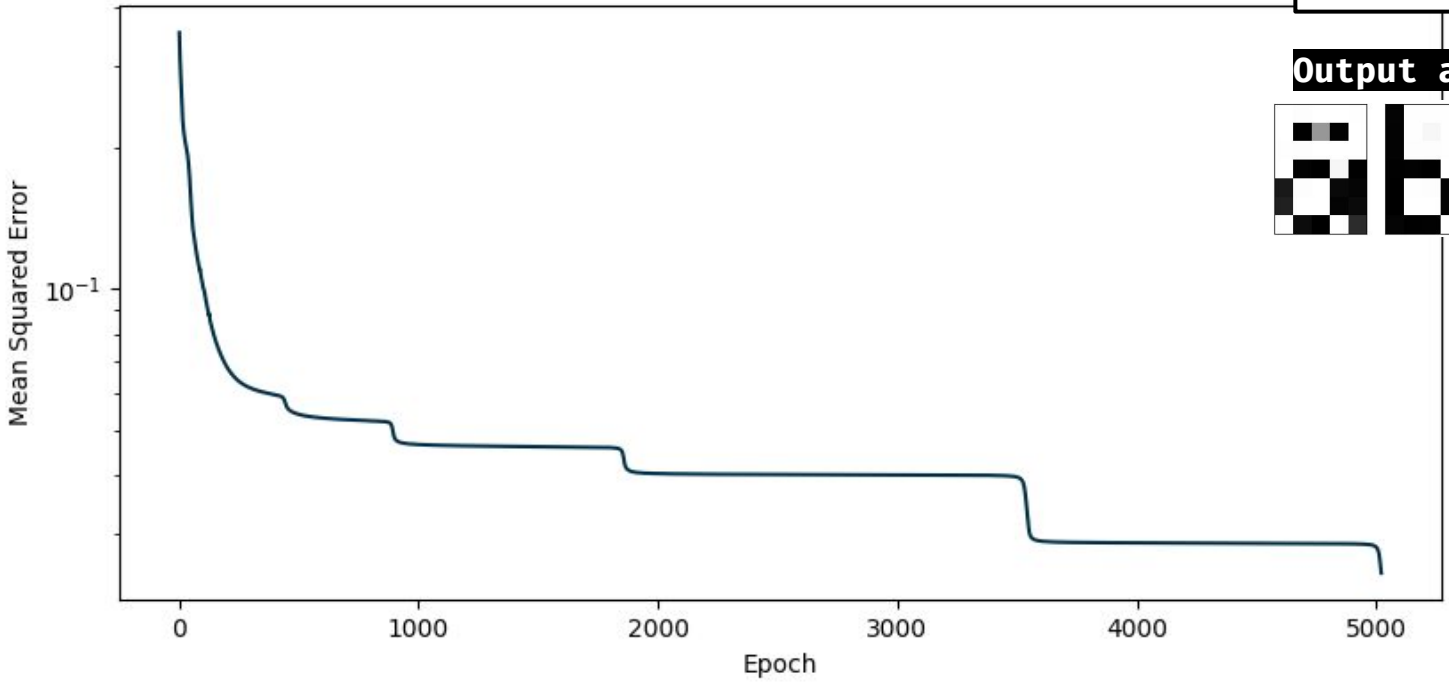
Ajuste de hiperparámetros

¿Cuál es la mejor arquitectura? Probamos sólo con 5 letras en el dataset.

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **24-16-8-2-8-16-24**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 5022 epochs:

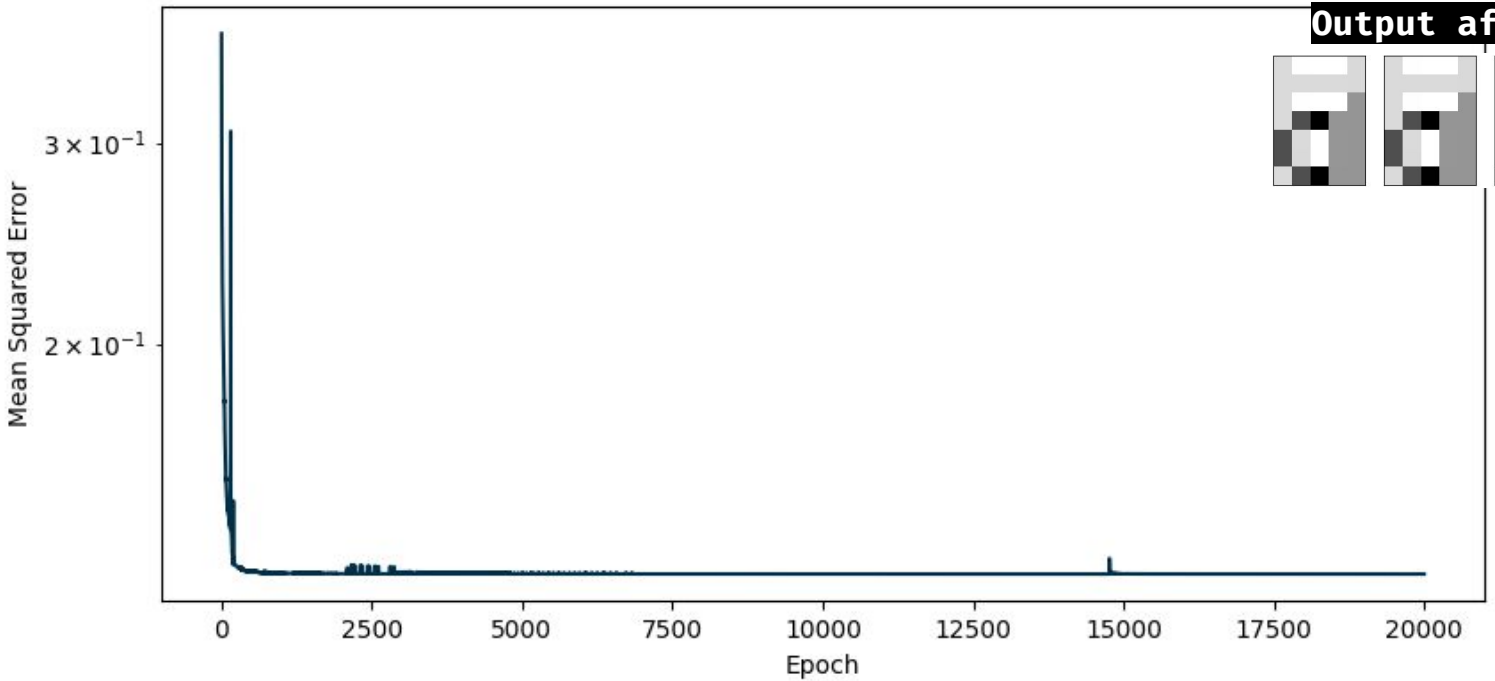




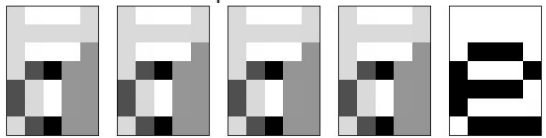
Ajuste de hiperparámetros

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **24-16-8-2-8-16-24**
- Act. Fun. Hidden Layers: **ReLU**
- Act. Fun. Latent Space: **TANH(β=0.5)**

¿Cuál es la mejor arquitectura? Probamos sólo con 5 letras en el dataset.



Output after 20k epochs:

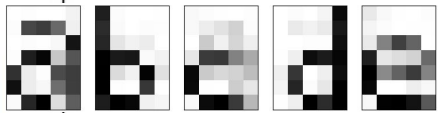


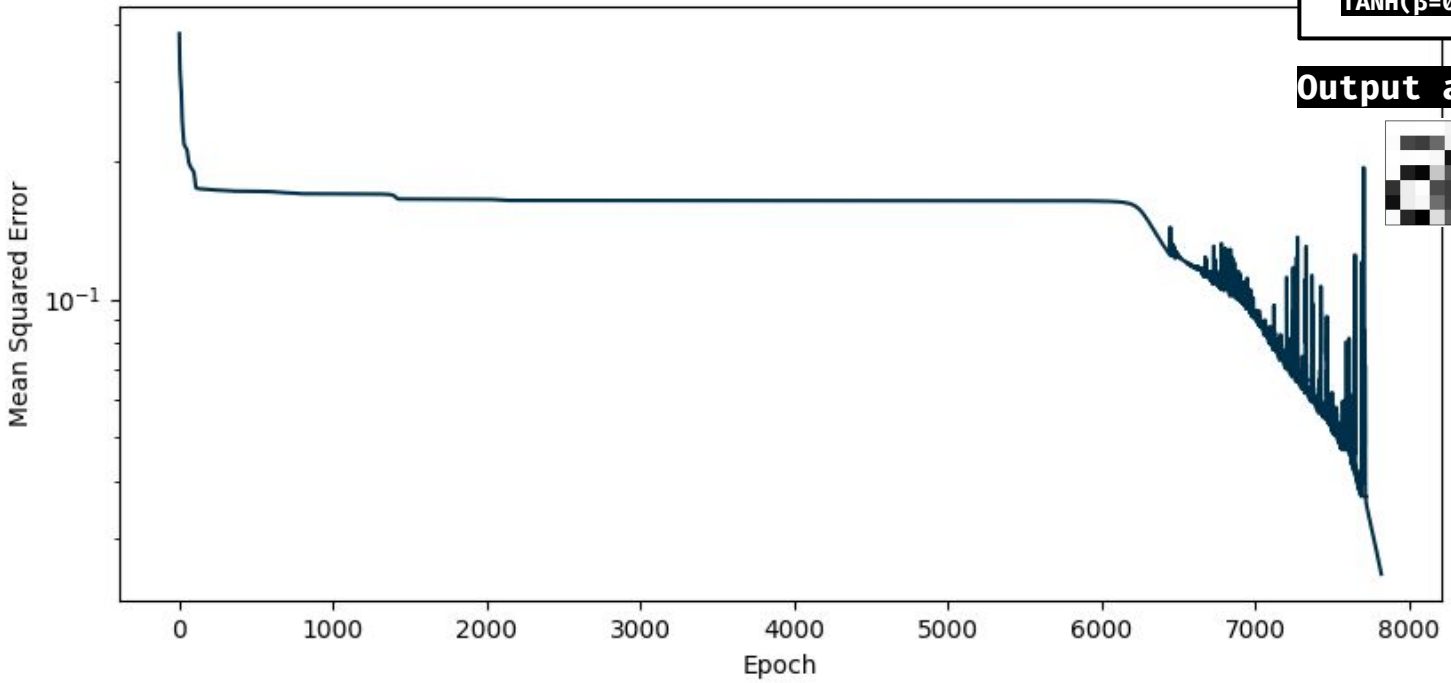
Ajuste de hiperparámetros

¿Cuál es la mejor arquitectura? Probamos sólo con 5 letras en el dataset.

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **30-25-20-15-10-5-2-5-10-15-20-25-30**
- Act. Fun. Hidden Layers: **LOGISTIC(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 7820 epochs:



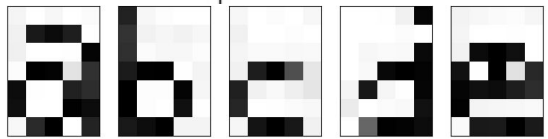


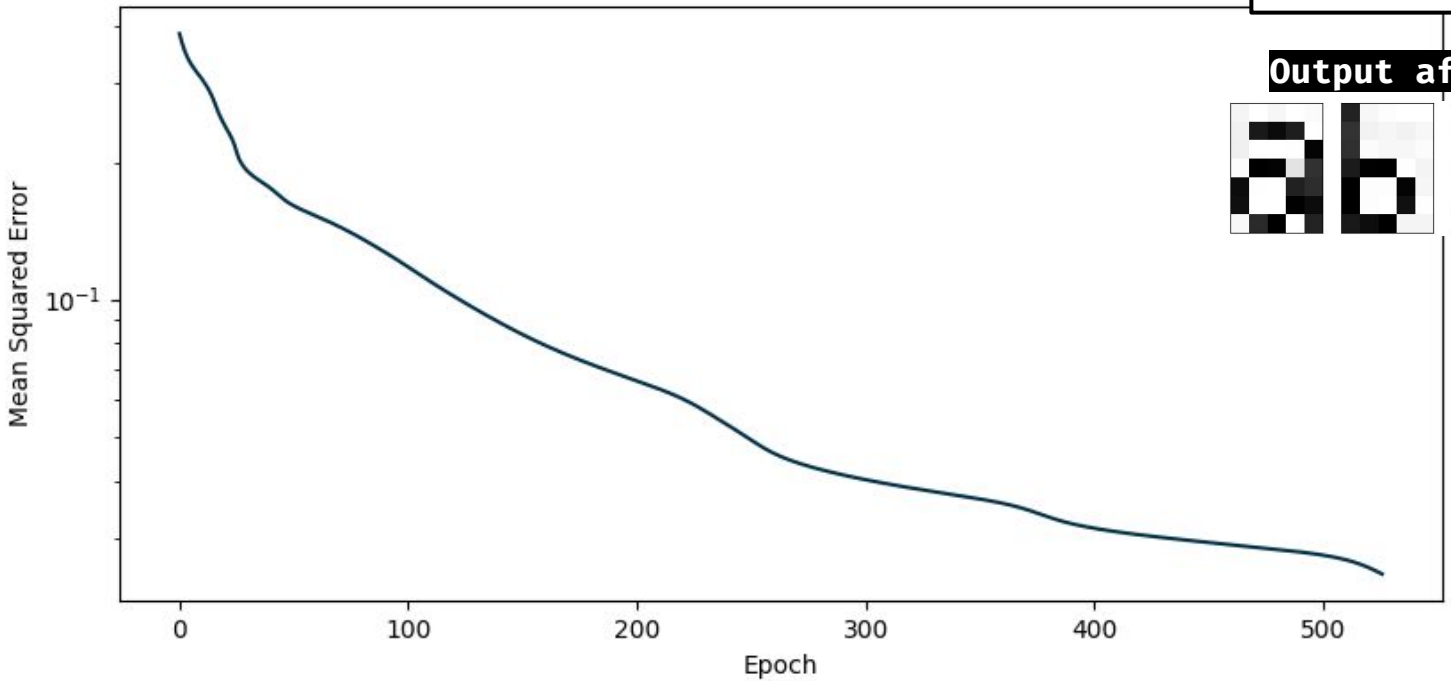
Ajuste de hiperparámetros

¿Cuál es la mejor arquitectura? Probamos sólo con 5 letras en el dataset.

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 526 epochs:



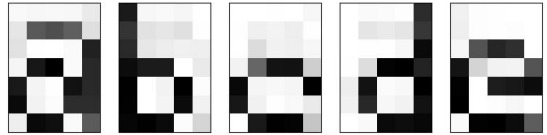


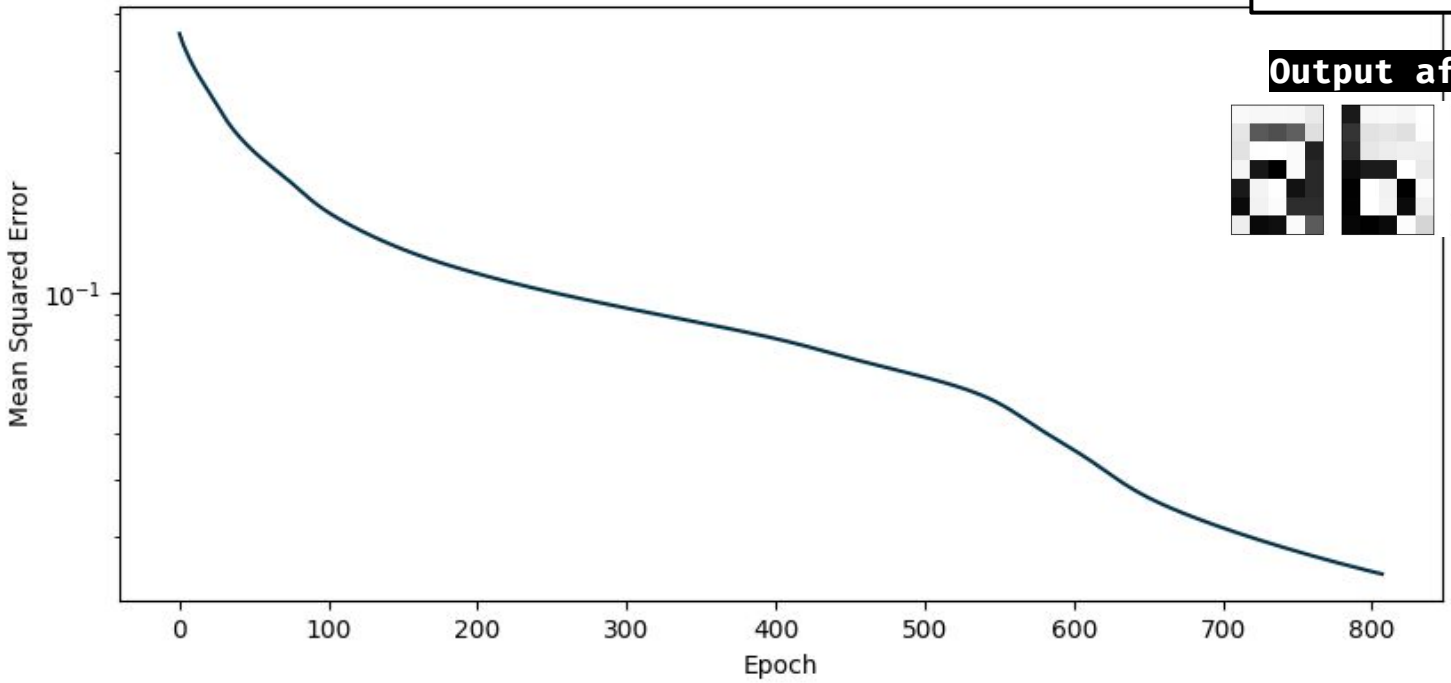
Ajuste de hiperparámetros

¿Cuál es la mejor arquitectura? Probamos sólo con 5 letras en el dataset.

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **10-2-10**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 807 epochs:



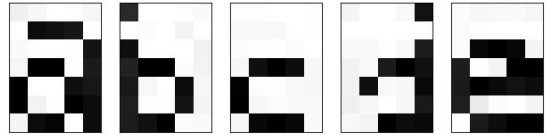


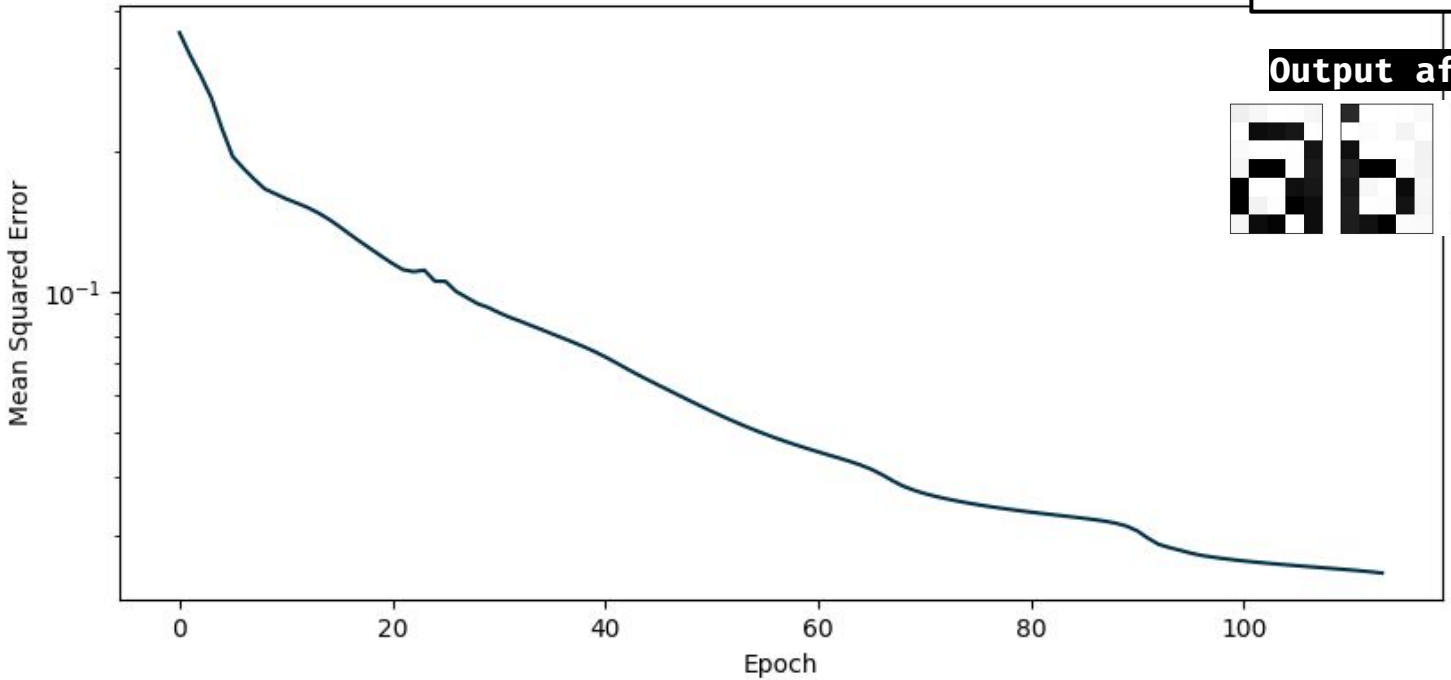
Ajuste de hiperparámetros

¿Cuál es el mejor método de optimización? Con la mejor arquitectura conseguida, variamos el método de optimización.

- Optimizer: **MOMENTUM**
- Updater = **ONLINE**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 113 epochs:



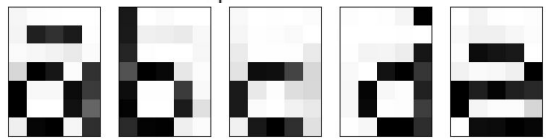


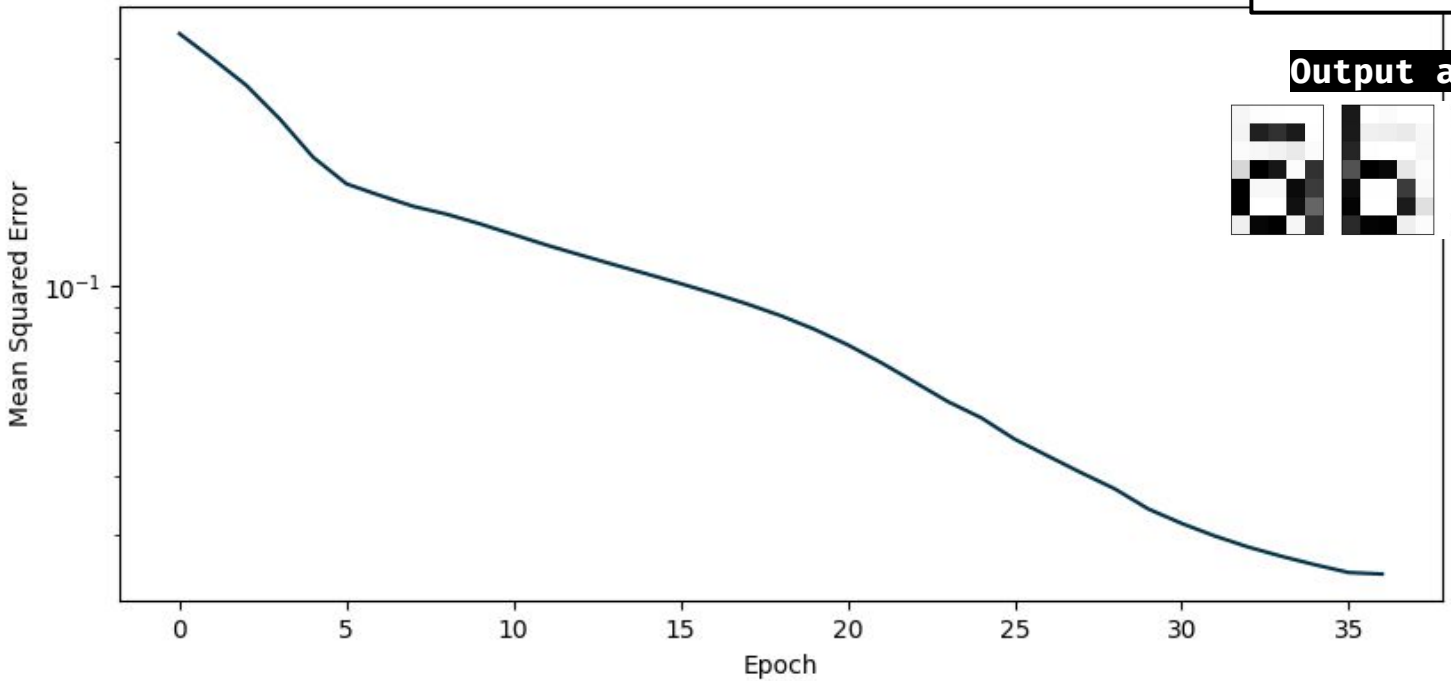
Ajuste de hiperparámetros

¿Cuál es el mejor método de optimización? Con la mejor arquitectura conseguida, variamos el método de optimización.

- Optimizer: **ADAM**
- Updater = **ONLINE**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Output after 36 epochs:

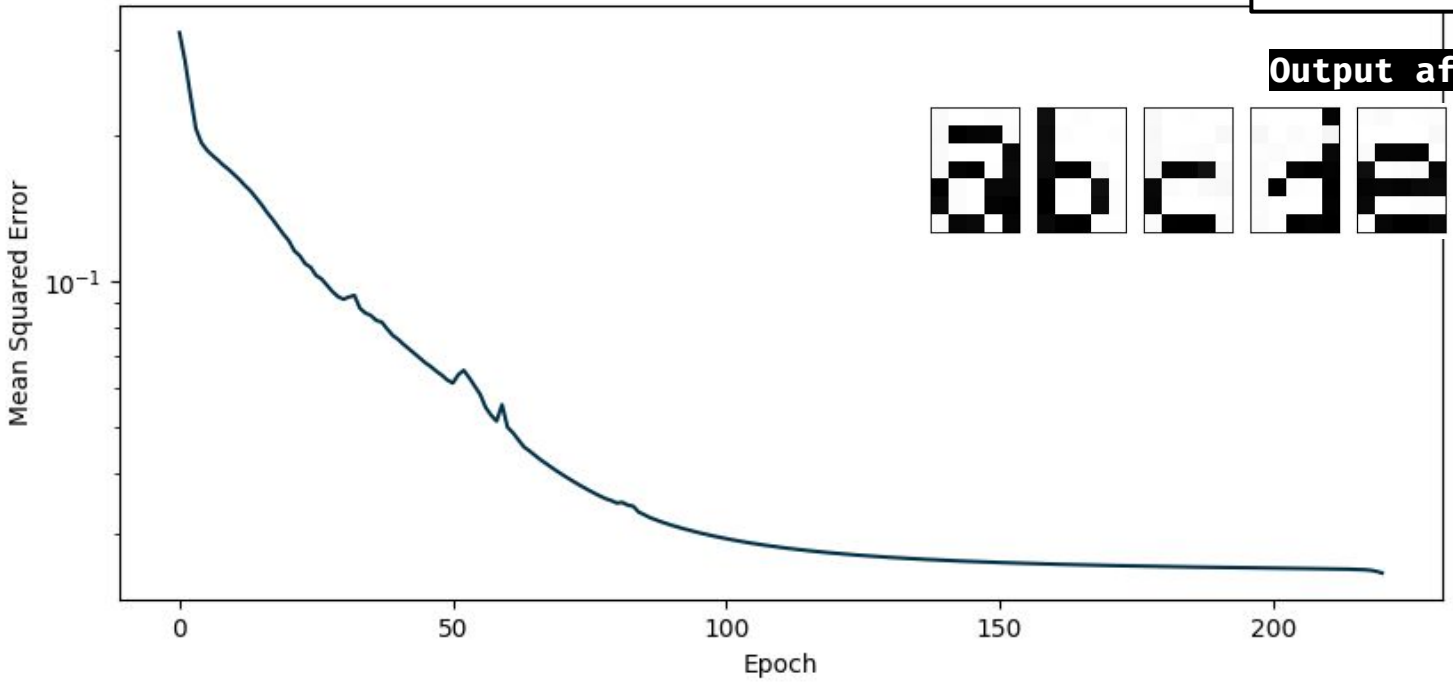




Ajuste de hiperparámetros

- Optimizer: **ADAM**
- Updater = **ONLINE**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

¿Qué tamaño de dataset se puede entrenar? El objetivo es entrenar con todas las letras. **Probamos con 8 letras.**

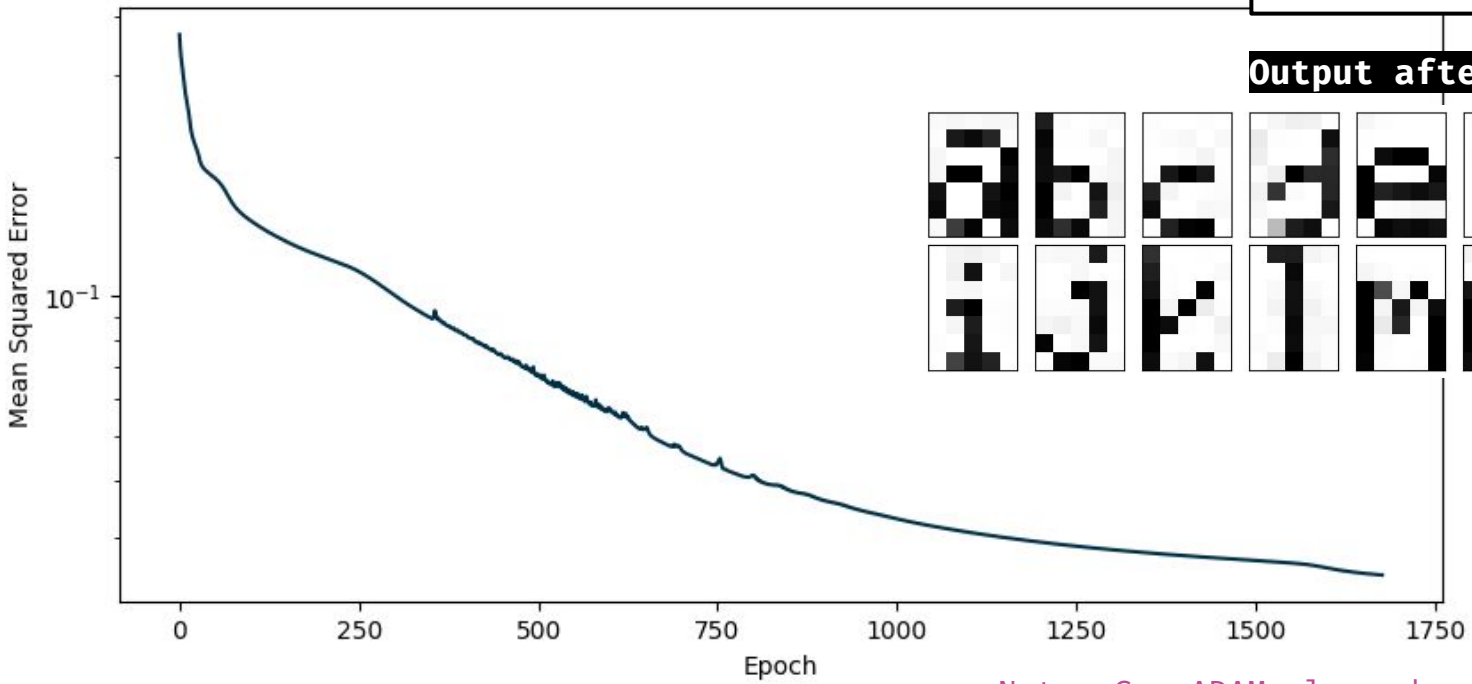


Output after 220 epochs:

Ajuste de hiperparámetros

- Optimizer: **GRADIENT_DESCENT**
- Updater = **ONLINE**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

¿Qué tamaño de dataset se puede entrenar? El objetivo es entrenar con todas las letras. **Probamos con 16 letras.**



Output after 1676 epochs:

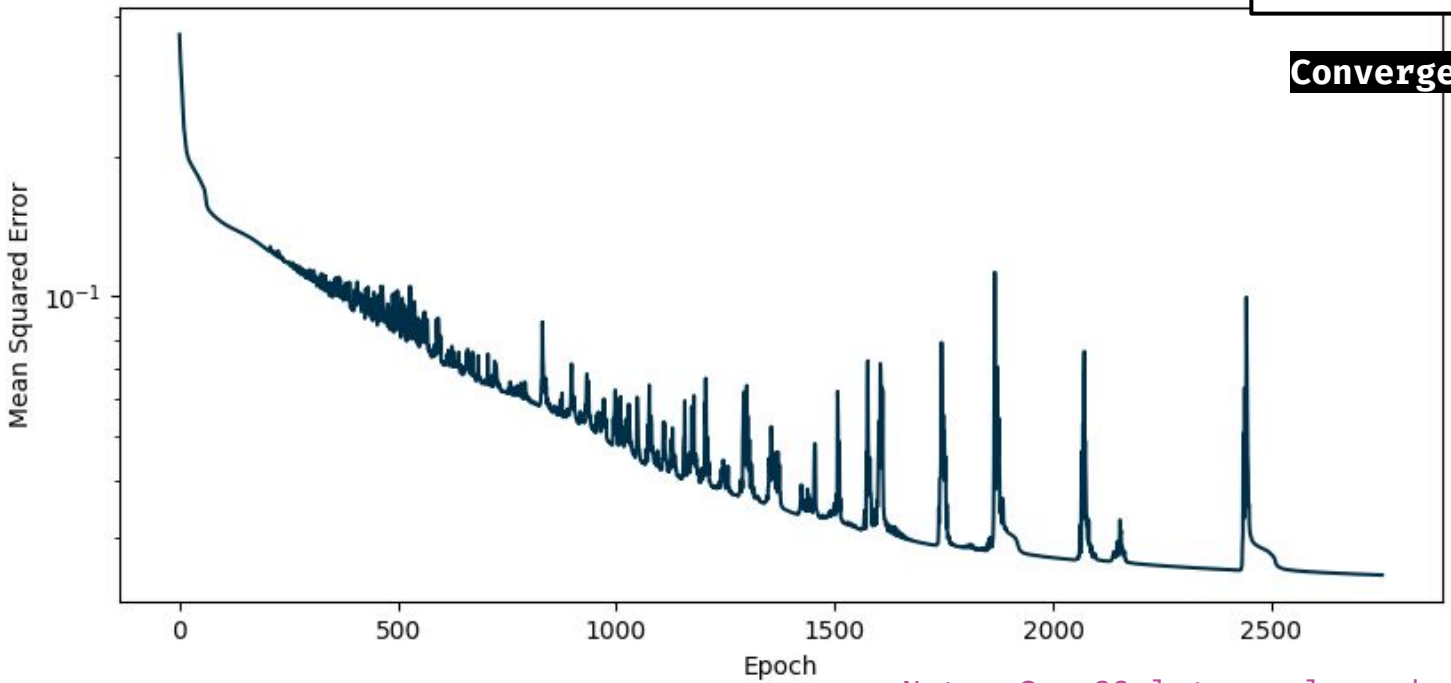
Nota: Con ADAM, la red no logró converger

Ajuste de hiperparámetros

- Optimizer: GRADIENT_DESCENT
- Updater = ONLINE
- Architecture: 16-8-2-8-16
- Act. Fun. Hidden Layers: TANH(β=1)
- Act. Fun. Latent Space: TANH(β=0.5)

¿Qué tamaño de dataset se puede entrenar? El objetivo es entrenar con todas las letras. **Probamos con 24 letras.**

Converge en 2754 épocas



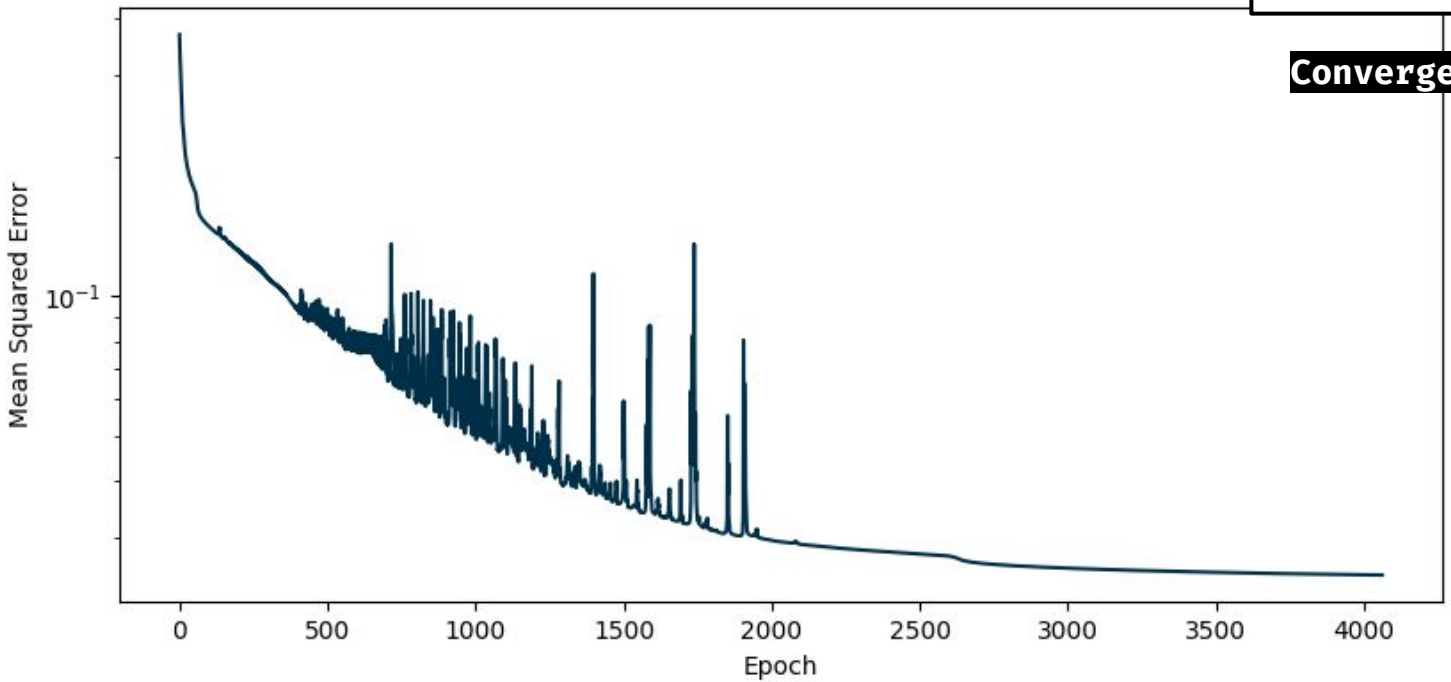
Nota: Con 32 letras, la red no logró converger

Ajuste de hiperparámetros

Habiendo encontrado el tamaño máximo del dataset,
¿qué método de update de pesos es mejor?

- Optimizer: **GRADIENT_DESCENT**
- Updater = **MINI-BATCH(n=8)**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

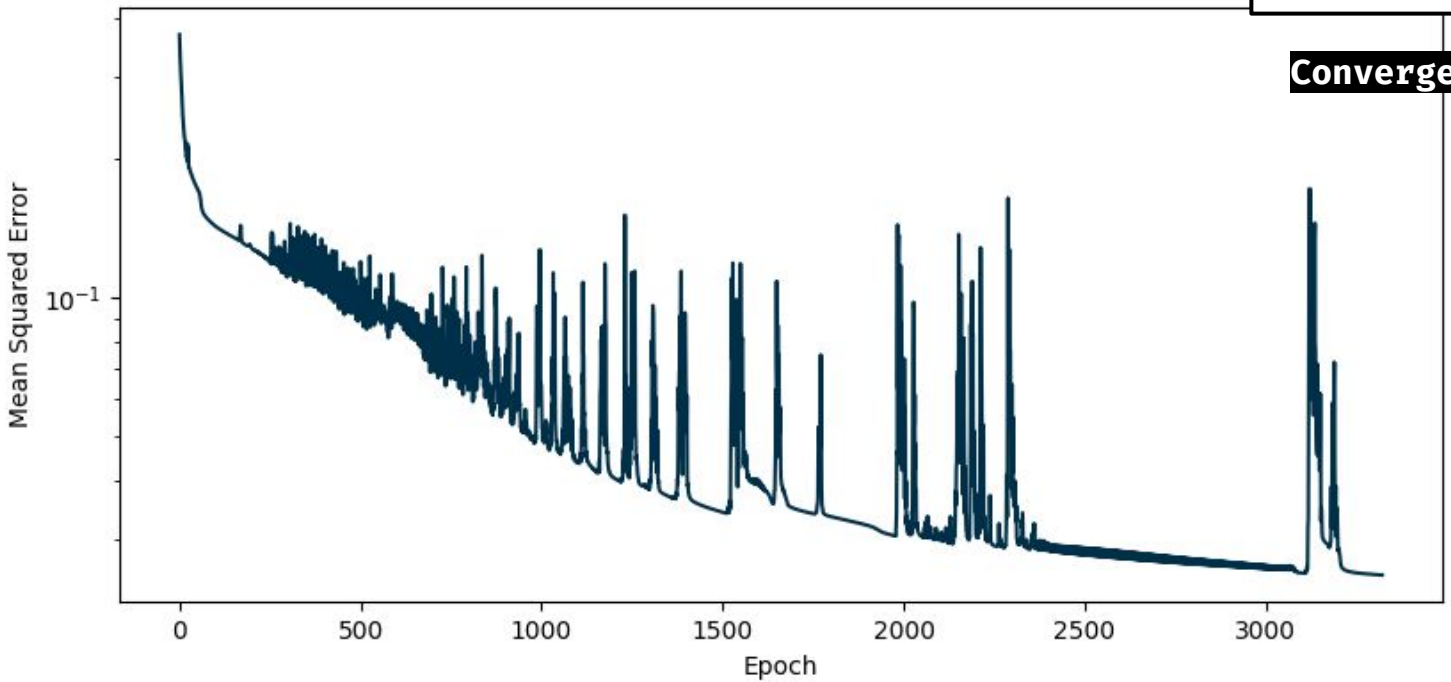
Converge en 4062 épocas



Ajuste de hiperparámetros

- Optimizer: **GRADIENT_DESCENT**
- Updater = **MINI-BATCH(n=12)**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

Habiendo encontrado el tamaño máximo del dataset,
¿qué método de update de pesos es mejor?



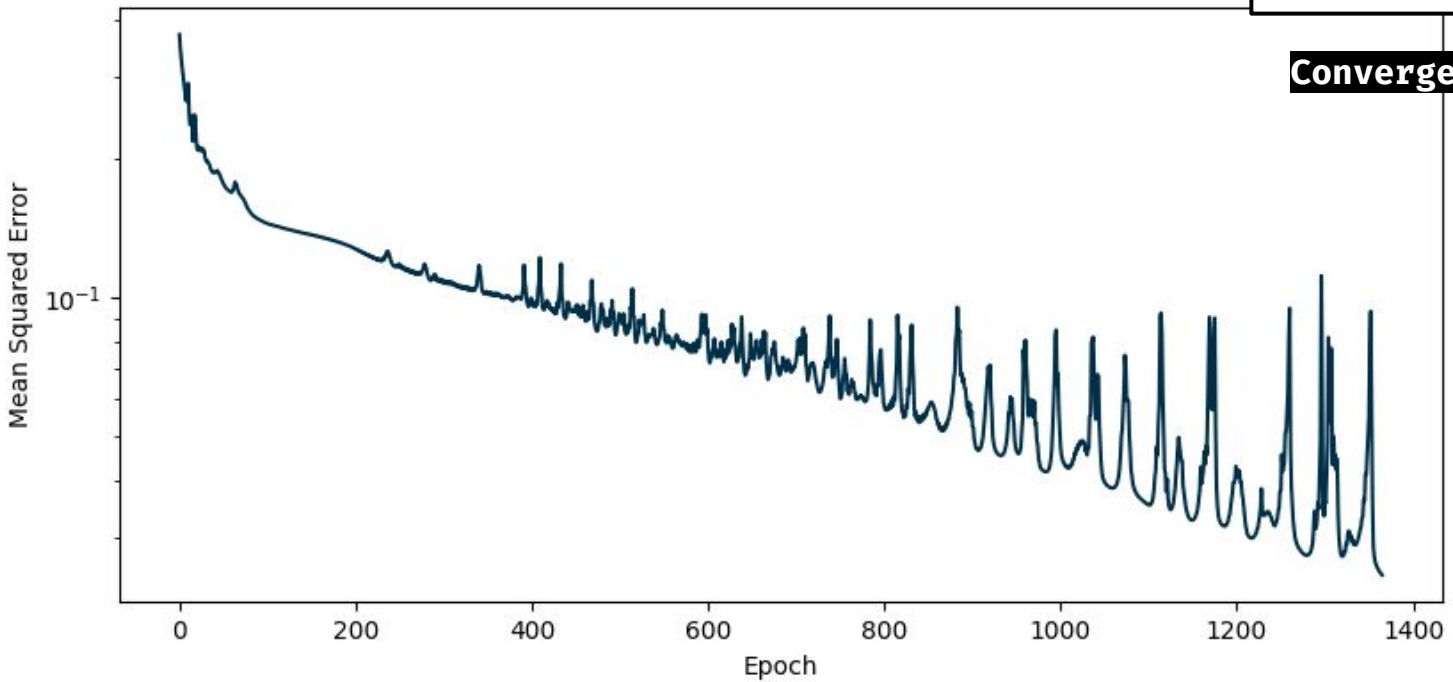
Converge en 3321 épocas

Ajuste de hiperparámetros

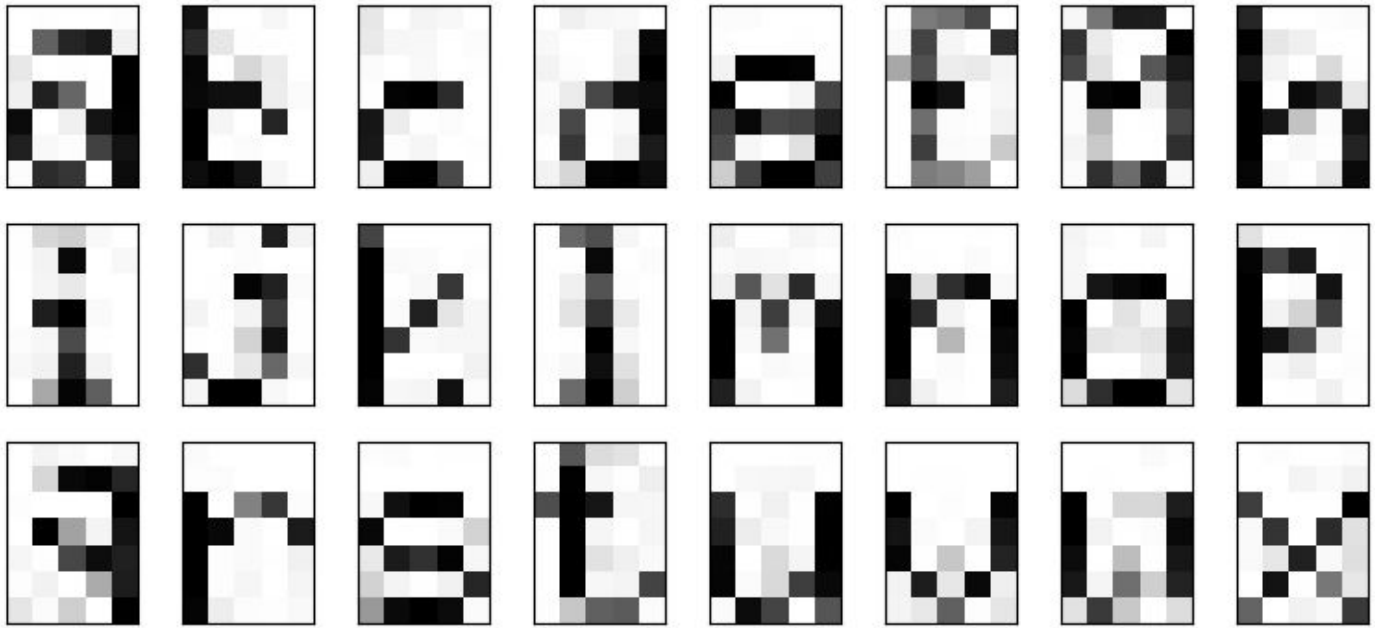
Habiendo encontrado el tamaño máximo del dataset,
¿qué método de update de pesos es mejor?

- Optimizer: **GRADIENT_DESCENT**
- Updater = **BATCH**
- Architecture: **16-8-2-8-16**
- Act. Fun. Hidden Layers: **TANH(β=1)**
- Act. Fun. Latent Space: **TANH(β=0.5)**

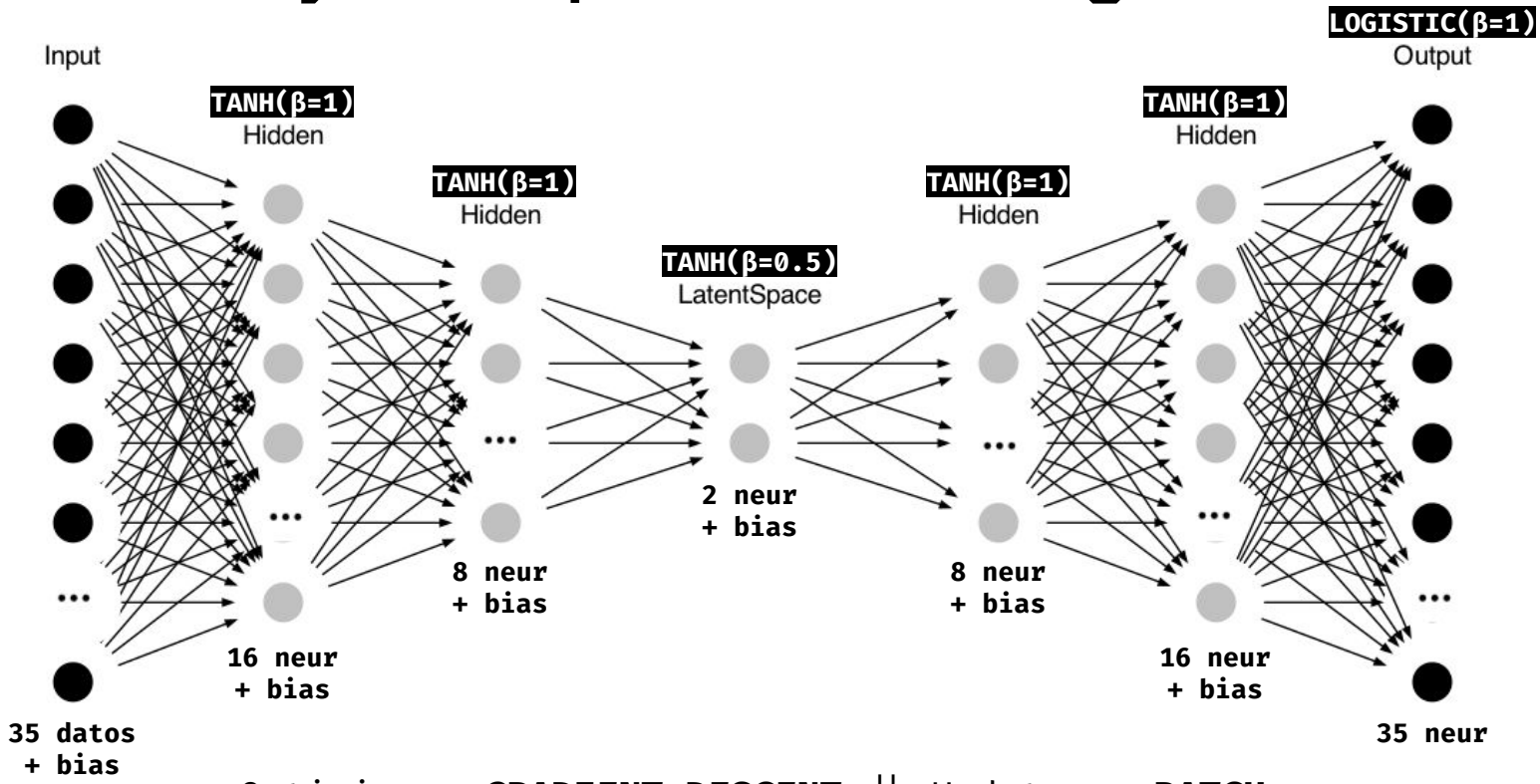
Converge en 1365 épocas



Salida lograda

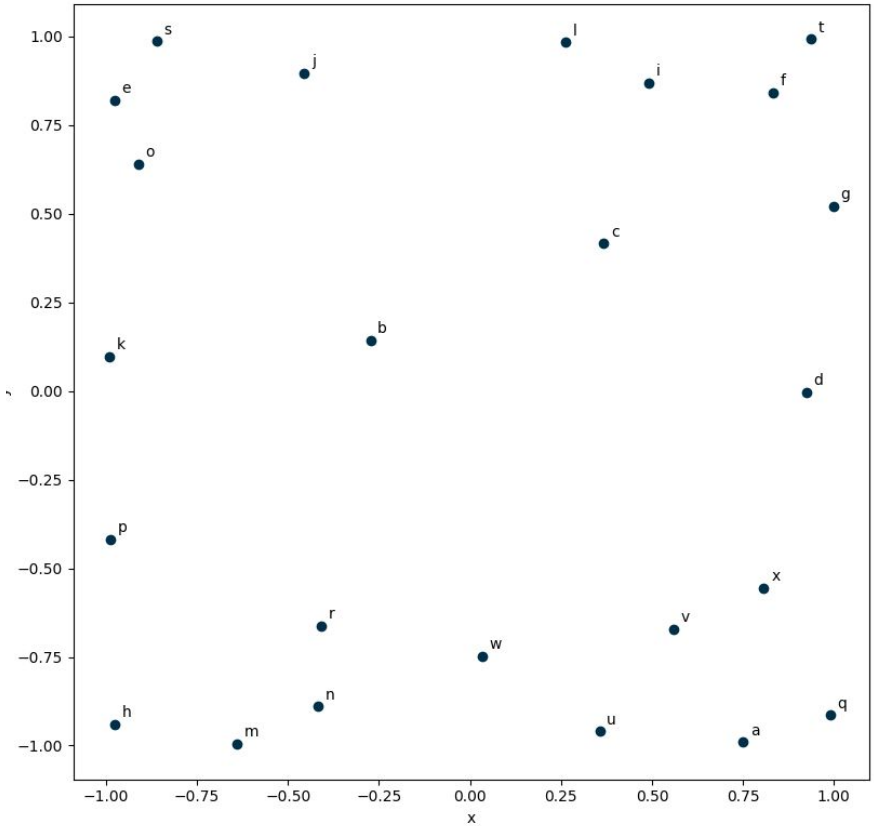


Mejor arquitectura lograda

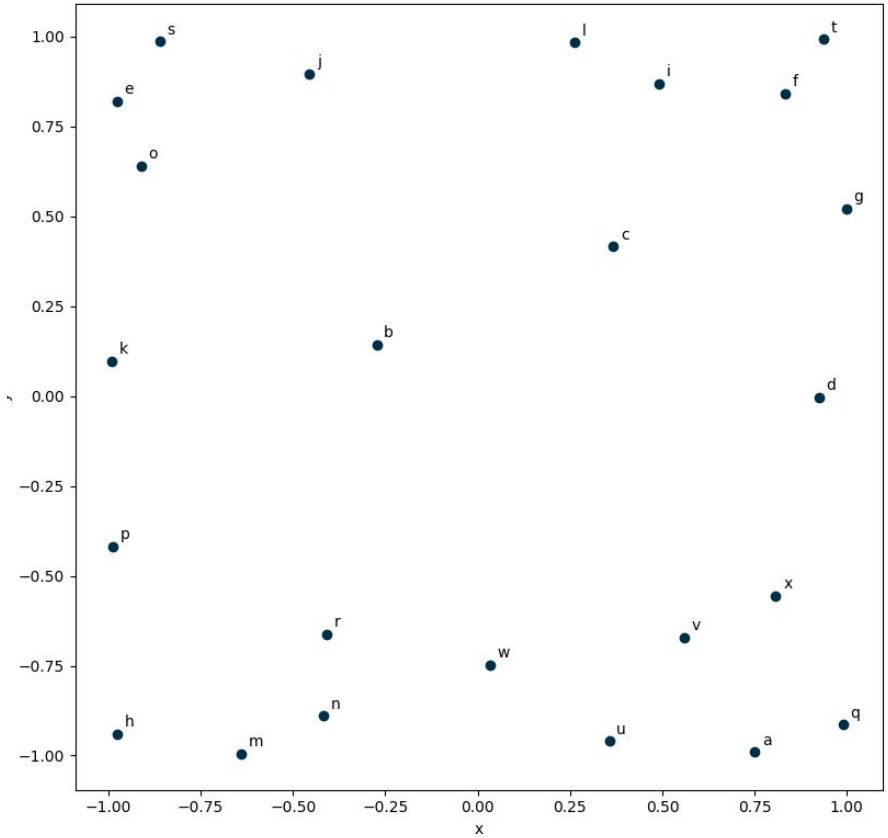


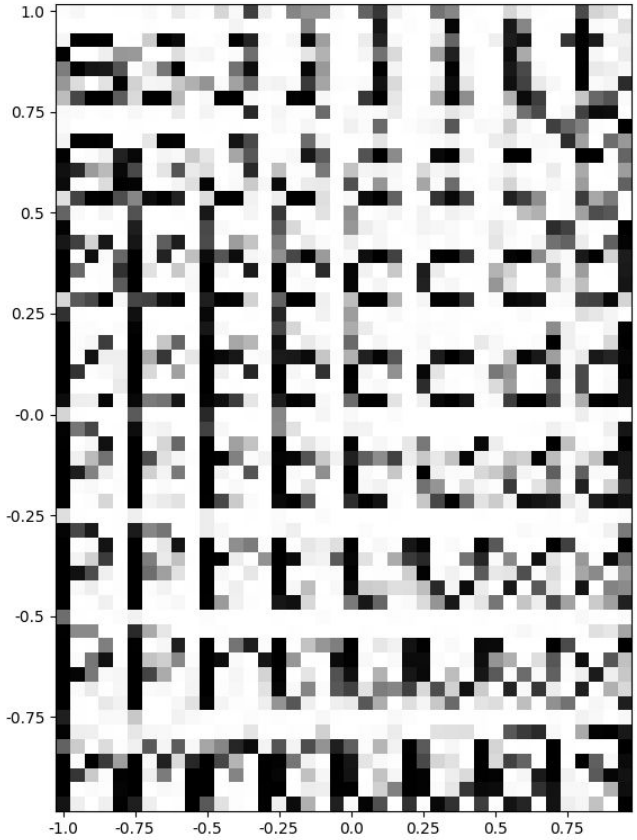
Optimizer: **GRADIENT_DESCENT** || Updater = **BATCH**

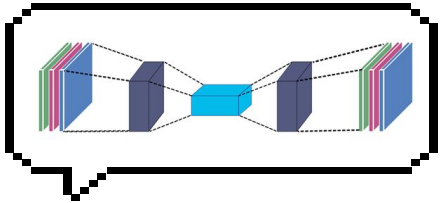
Representación en el espacio latente



Nuevas letras por fuera del dataset





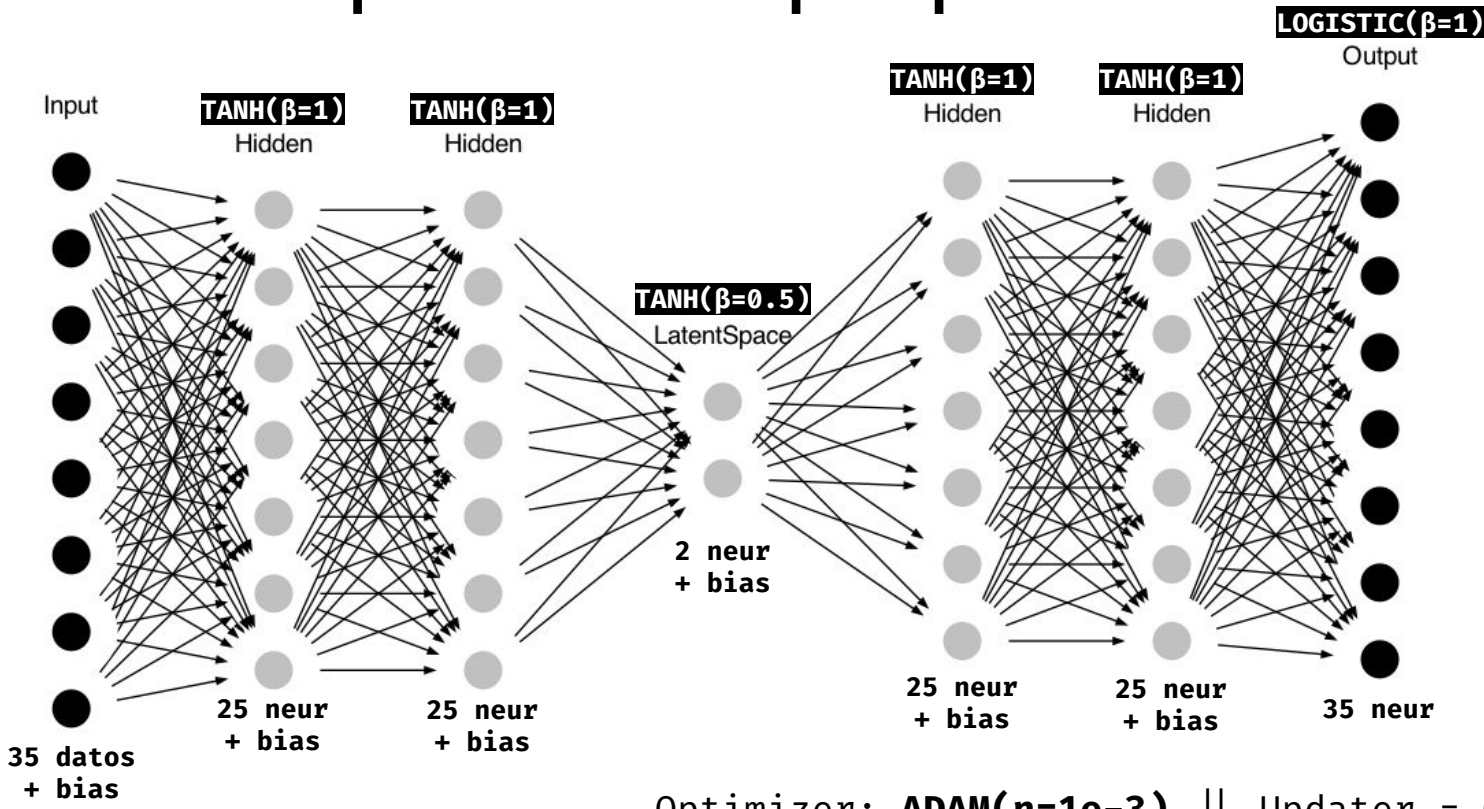


03

Denoising Autoencoder

Implementaremos un denoising autoencoder que, tras distorsionar las entradas a distintos niveles, logre reconstruir los patrones binarios originales eliminando el ruido.

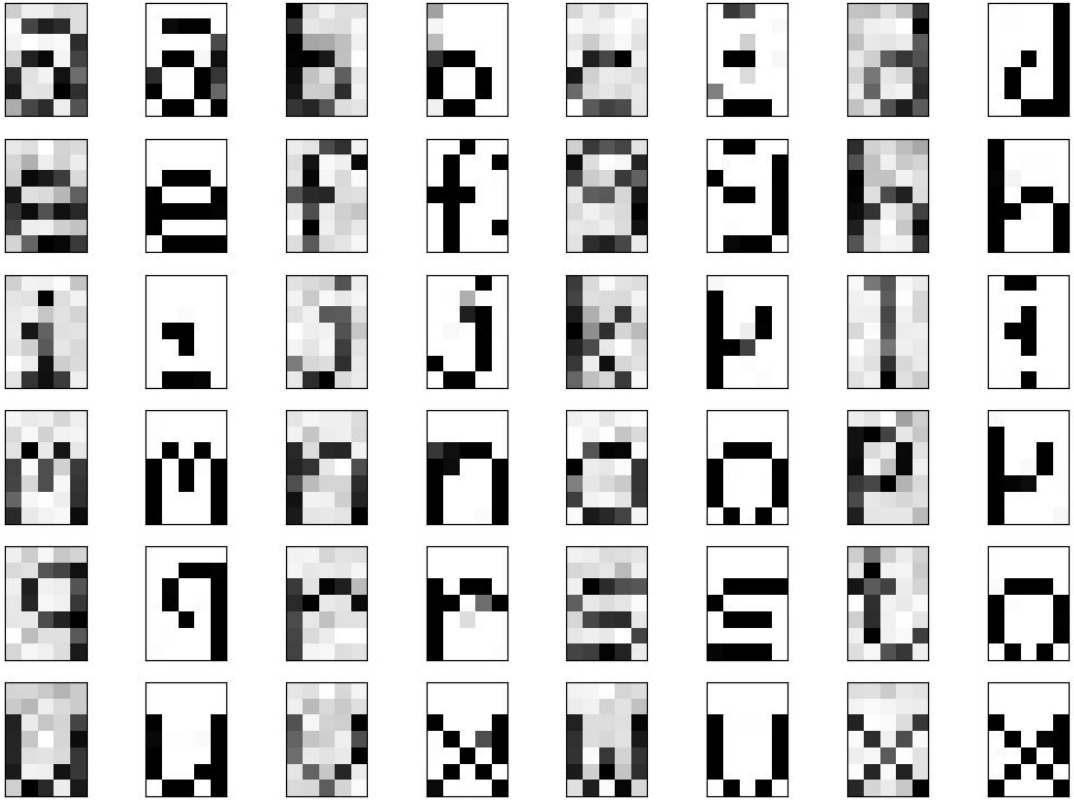
Arquitectura propuesta



Optimizer: **ADAM(η=1e-3)** || Updater = **ONLINE**

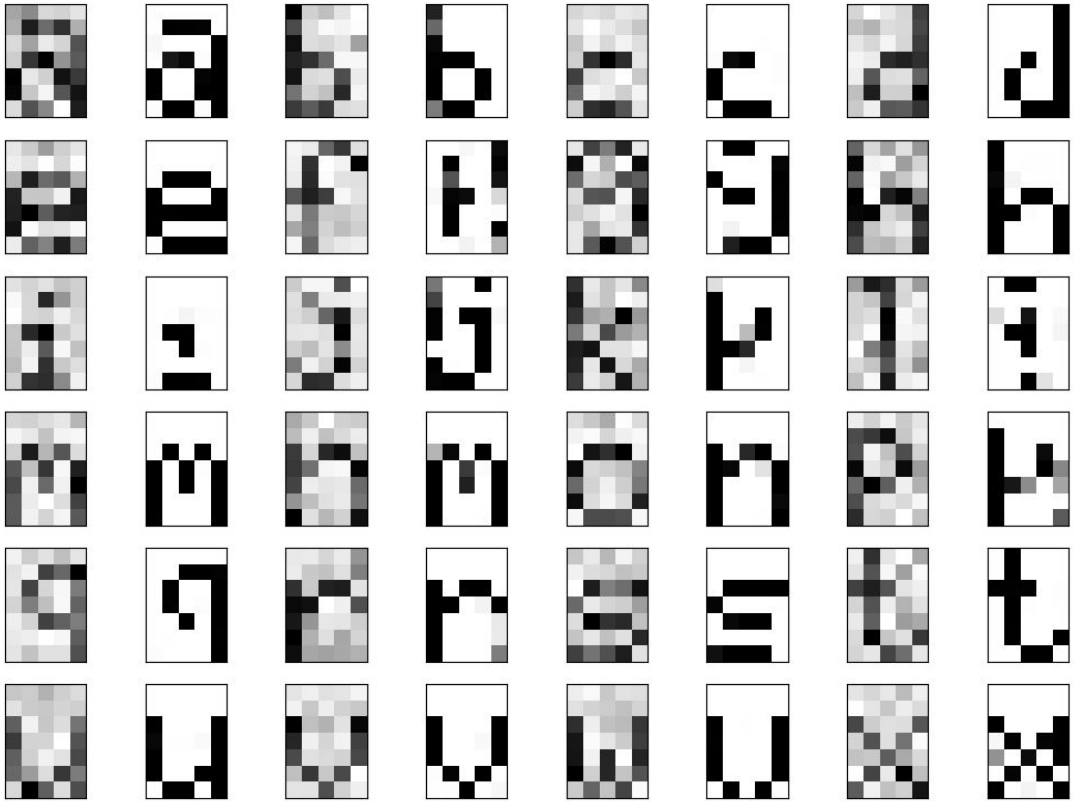
Agregado de ruido

- Noise Function: **GAUSSIAN**
- Noise Level: **0.15**



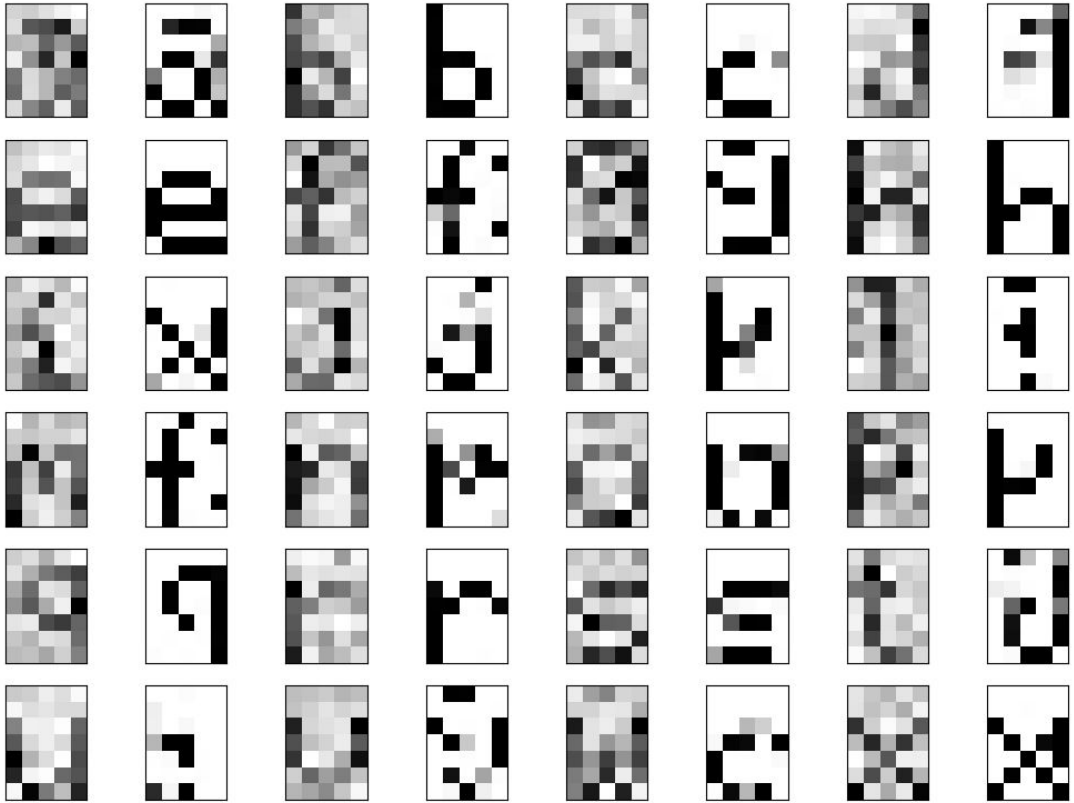
Agregado de ruido

- Noise Function: **GAUSSIAN**
- Noise Level: **0.20**



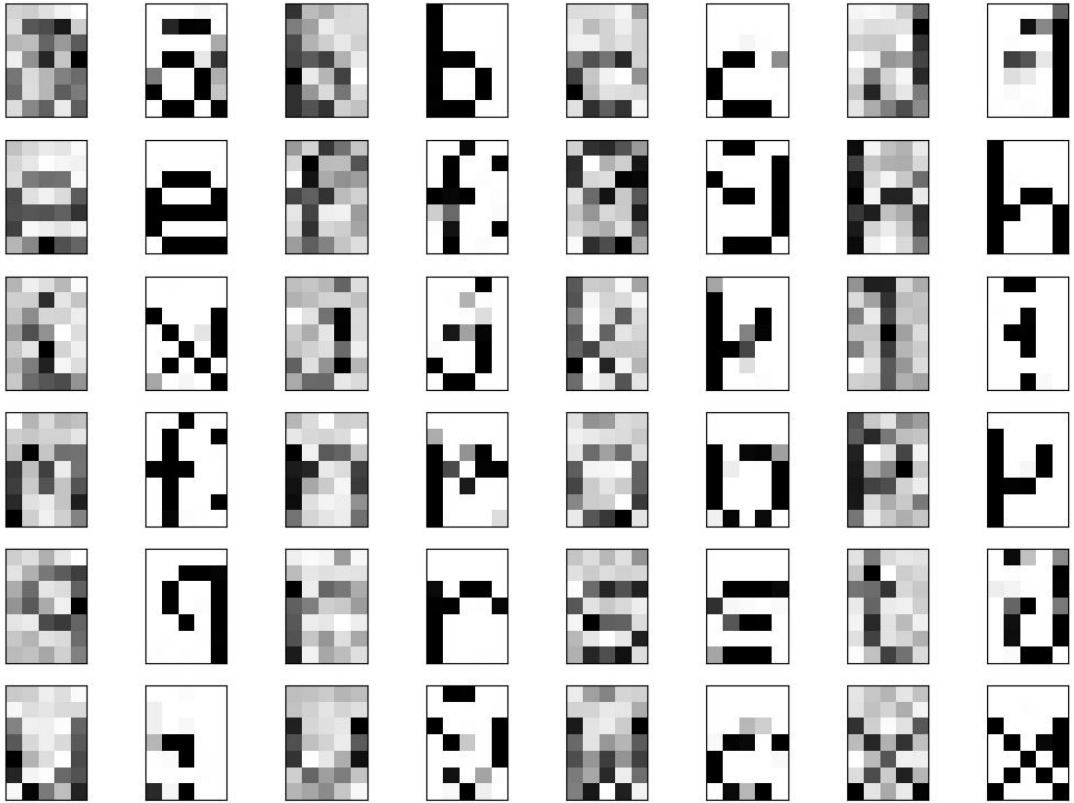
Agregado de ruido

- Noise Function: **GAUSSIAN**
- Noise Level: **0.25**



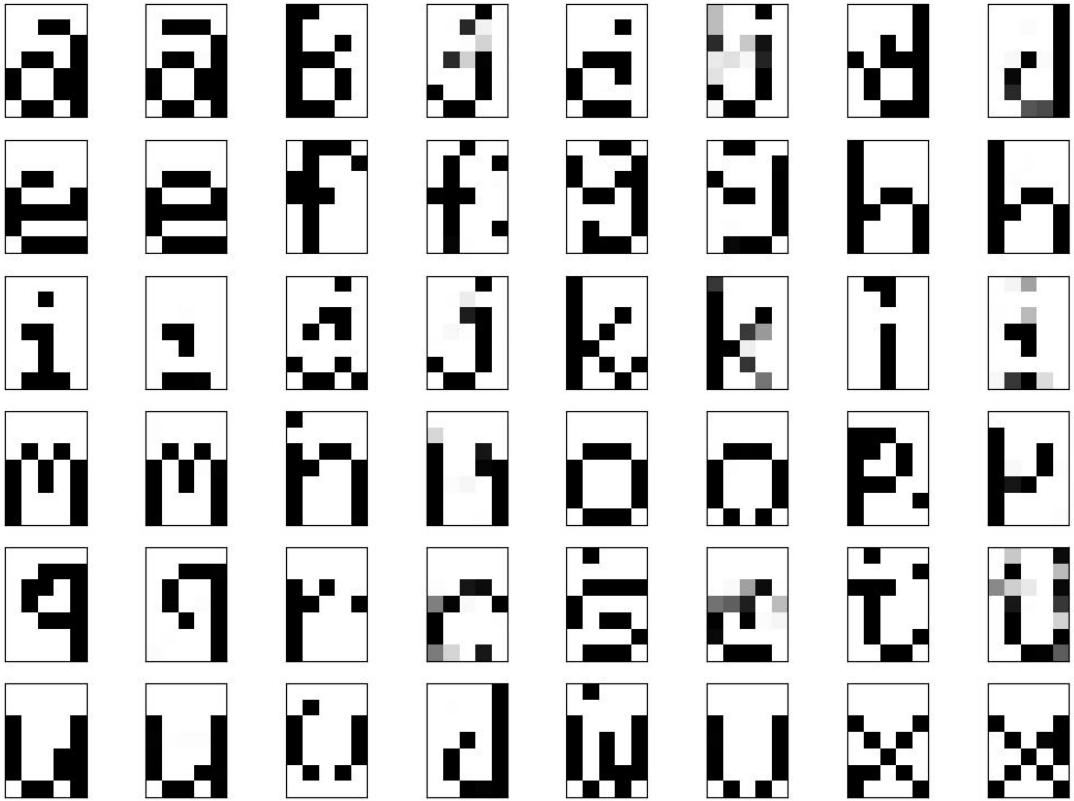
Agregado de ruido

- Noise Function: **GAUSSIAN**
- Noise Level: **0.30**



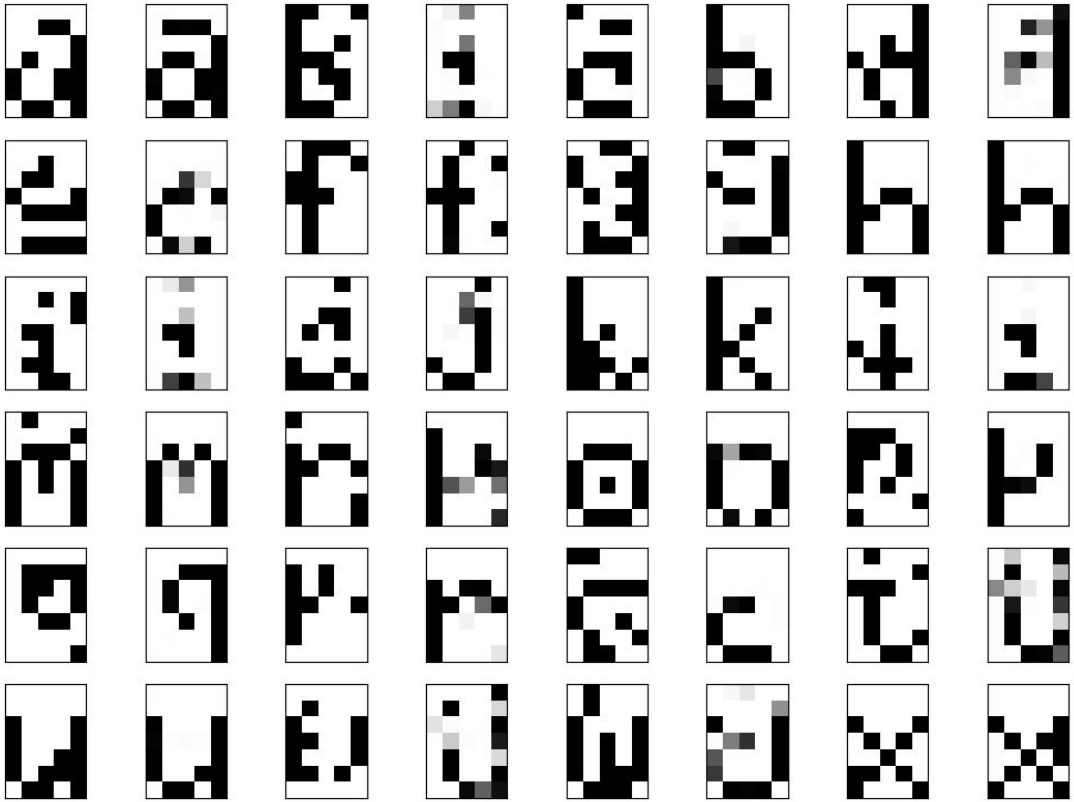
Agregado de ruido

- Noise Function: **SALT AND PEPPER**
- Noise Threshold: **0.05**



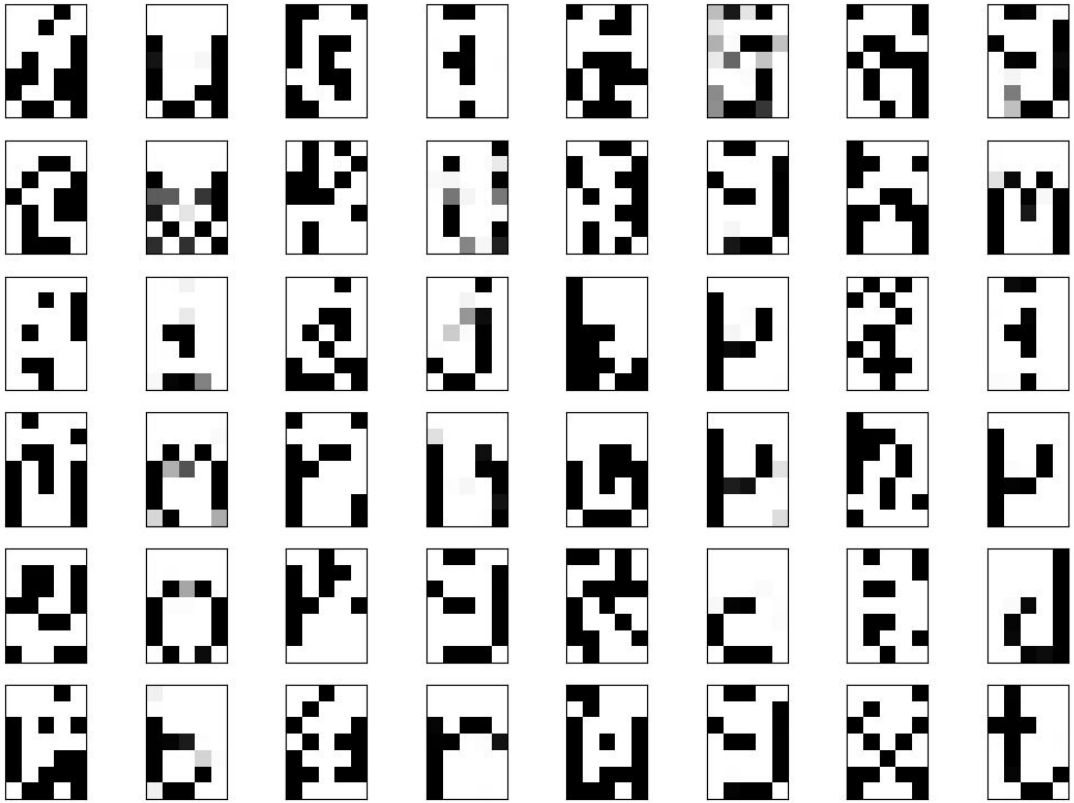
Agregado de ruido

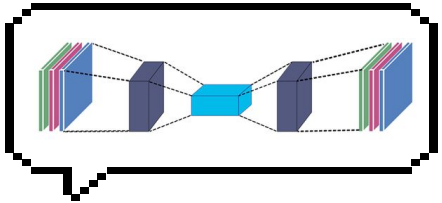
- Noise Function: **SALT AND PEPPER**
- Noise Threshold: **0.10**



Agregado de ruido

- Noise Function: **SALT AND PEPPER**
- Noise Threshold: **0.20**





04

Variational Autoencoder

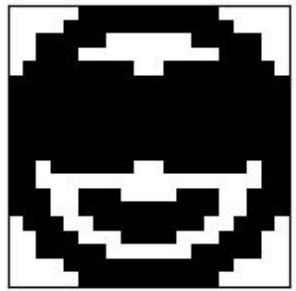
Extenderemos el autoencoder a un esquema variacional para representar los datos en un espacio latente continuo y generar nuevas muestras, utilizando un conjunto de emojis como input.

Definición del dataset

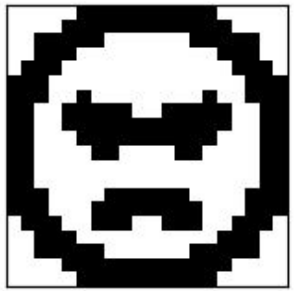
smile

surprise

sunglasses

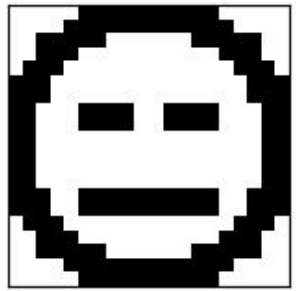


sexy



angry

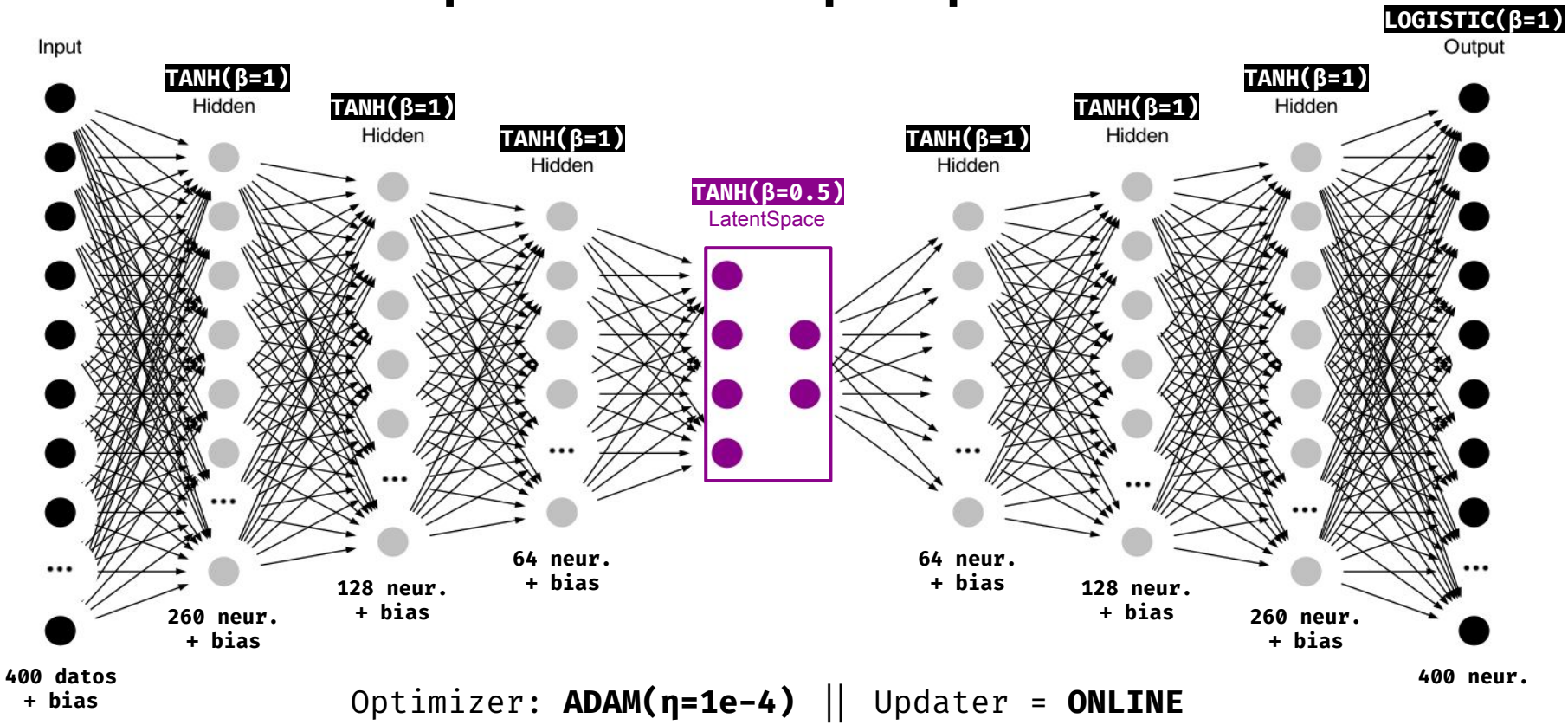
happy



emotionless

dead

Arquitectura propuesta

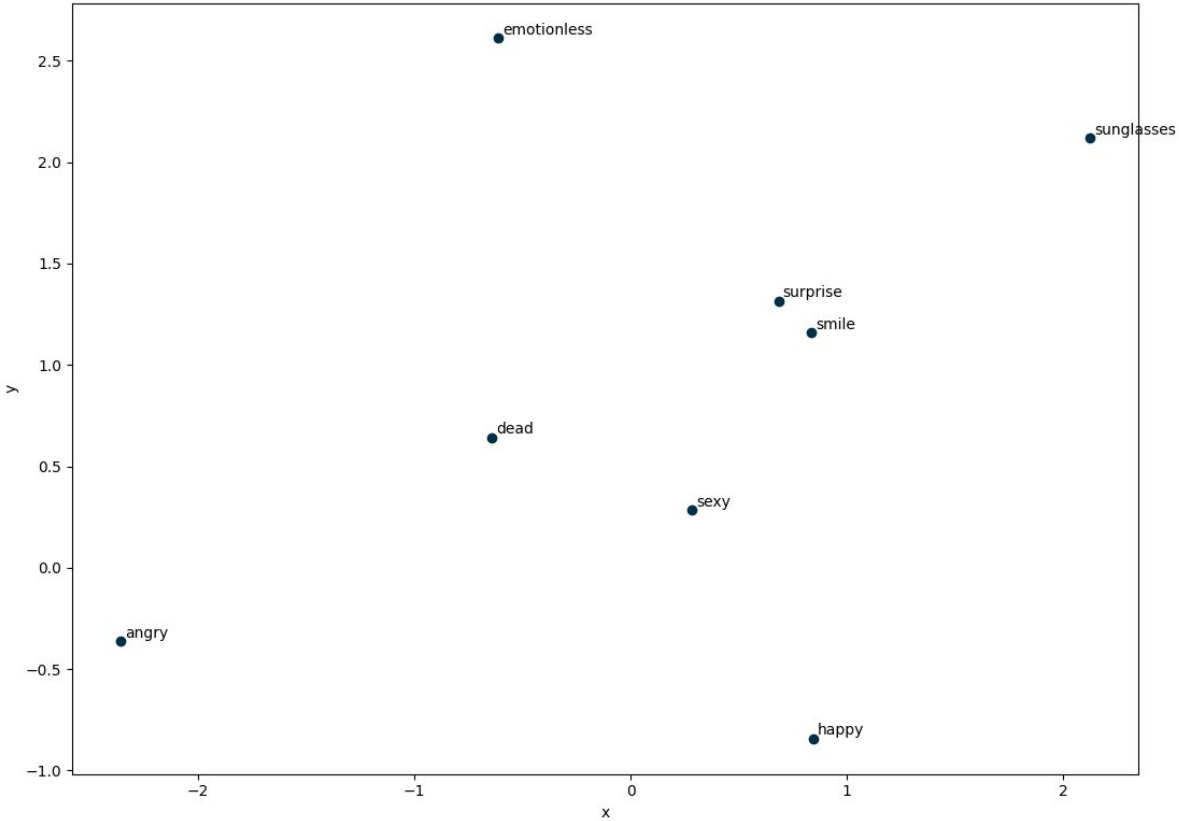


Optimizer: **ADAM(η=1e-4)** || Updater = **ONLINE**

Explicación

- Arquitectura con **más *hidden layers*** porque ayudan a capturar mejor las *features* de los emojis (20×20).
- ***Hidden layers*** con función de activación **TANH(β=1)** que tienen salida (-1, 1), lo que ayuda a preservar la variación de los datos y la *feature extraction*.
- Capa de salida del **espacio latente** con función de activación **TANH(β=0.5)** para generar salidas con valores más probabilísticos y que estos sirvan así tener un espacio latente más "continuo".

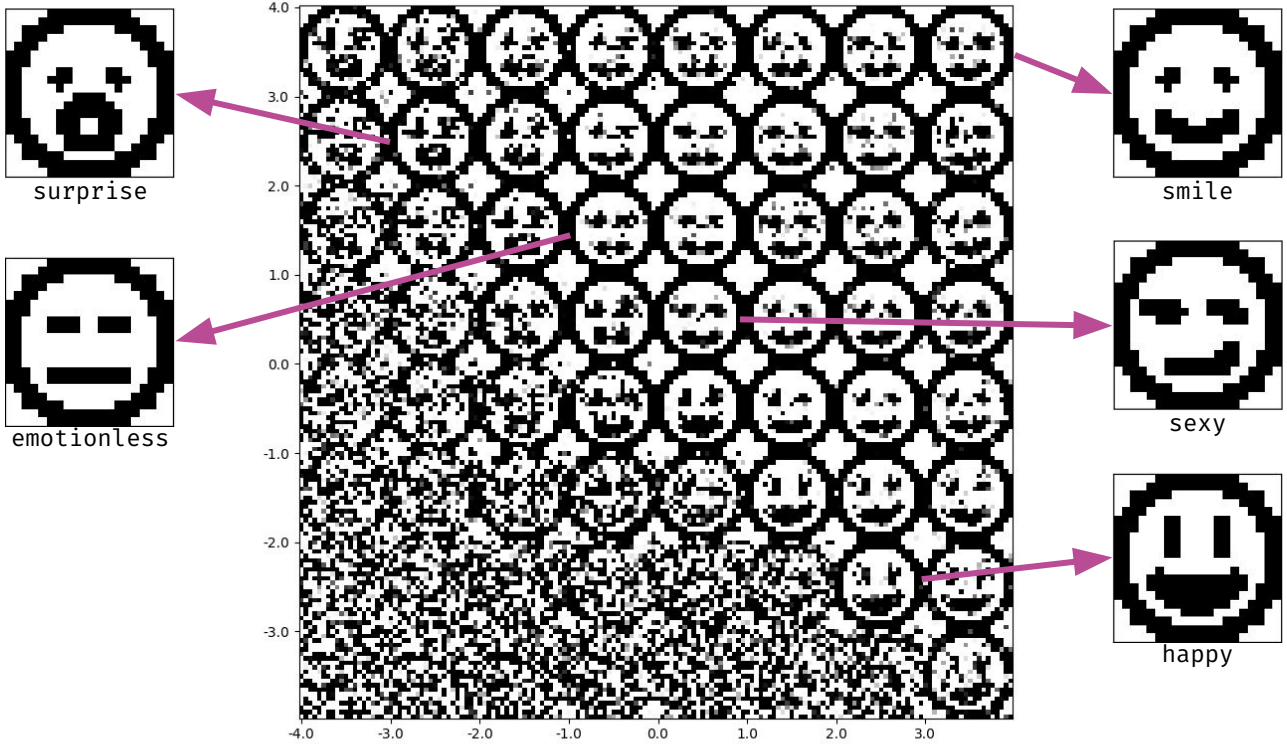
Representación en el espacio latente



Resultados generados

- Partitions = 8
- X Range: [-4, 4]
- Y Range: [-4, 4]

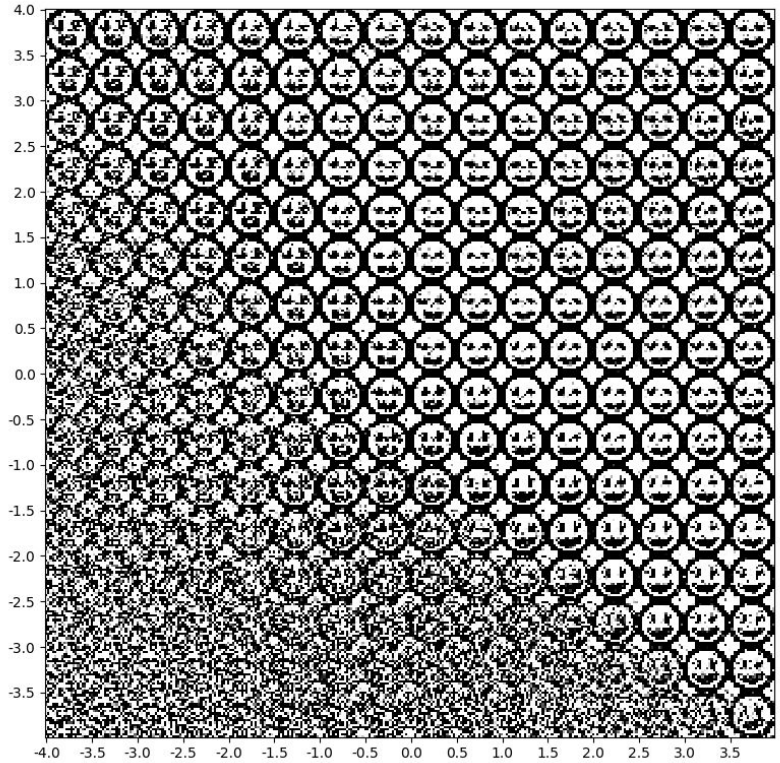
¿Vemos elementos nuevos, relacionados al dataset?



Resultados generados

- Partitions = 16
- X Range: [-4, 4]
- Y Range: [-4, 4]

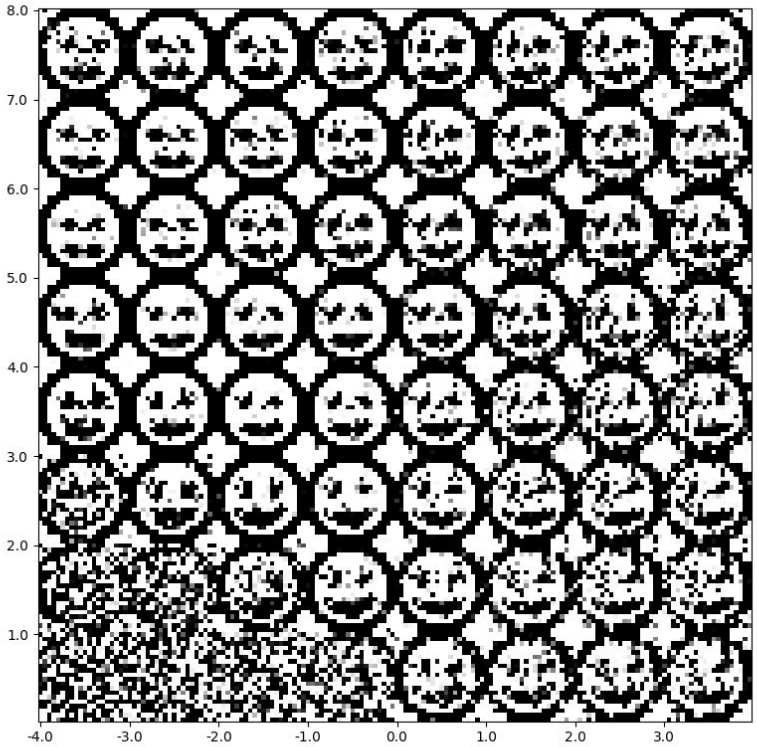
¿Qué pasa si dividimos el espacio latente en **más particiones**?



Resultados generados

- Partitions = 8
- X Range: [-4, 4]
- Y Range: [0, 8]

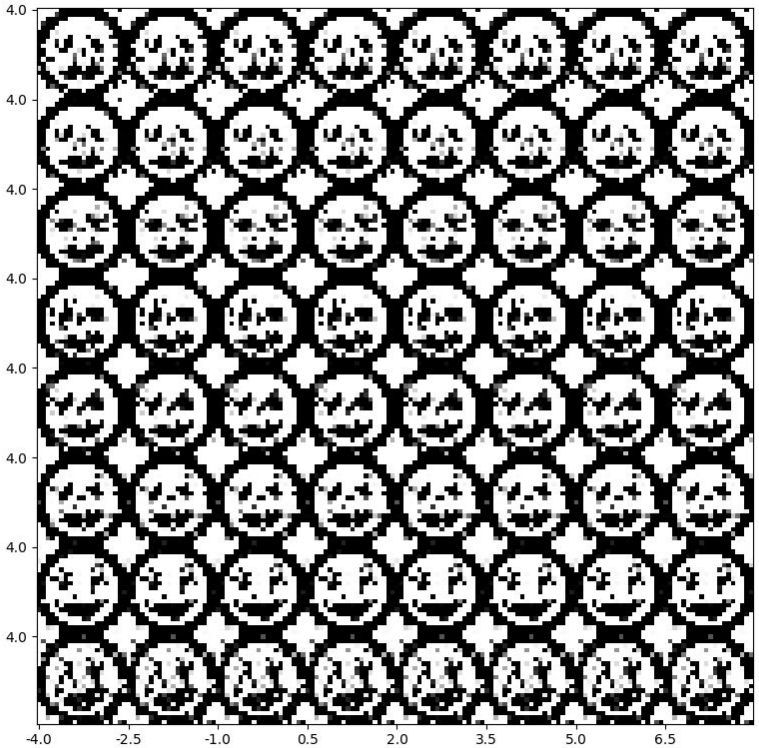
¿Qué pasa si exploramos más por el **eje Y**?

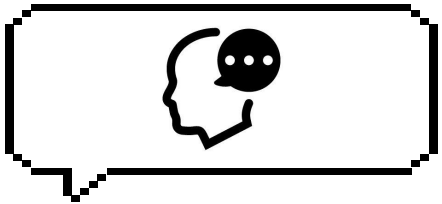


Resultados generados

- Partitions = 8
- X Range: [-4, 8]
- Y Range: [4, 4]

¿Qué pasa si exploramos más por el **eje X**?





05

Conclusiones

Evaluaremos el desempeño de los diferentes autoencoders implementados, analizando su capacidad para resolver los problemas pedidos.

Conclusiones

- (1) Con la implementación lograda, no se logra entrenar el autoencoder con todo el conjunto de datos.
- (1) La mejor arquitectura obtenida utiliza TANH() como función de activación y GRADIENTE DESCENDIENTE como optimizador.
- (1) Se puede reducir la dimensión de las letras (35→2) y representarlas en un plano.
- (1) Se logran crear nuevas variantes de letras sampleando puntos del espacio latente.
- (2) El autoencoder resulta más eficiente para eliminar ruido de tipo GAUSSIANO.
- (2) El autoencoder logra identificar letras ofuscadas con un ruido máximo de 0.2.
- (3) El VAE logra generar nuevas variantes de emojis
- (3) Mover la ventana de exploración del espacio latente permite ver los límites de lo generado

¡Muchas gracias por su atención!

Enlaces Útiles

- https://github.com/alejofl/sia
  Repositorio del proyecto.
- https://blog.keras.io/building-autoencoders-in-keras.html
  Implementación de VAE con Keras
- https://medium.com/@sofeikov/implementing-variational-autoencoders-from-scratch-533782d8eb95
  Implementación de VAE con PyTorch
- https://github.com/martisak/dotnets
  Generador de imágenes de una red neuronal feed-forward.