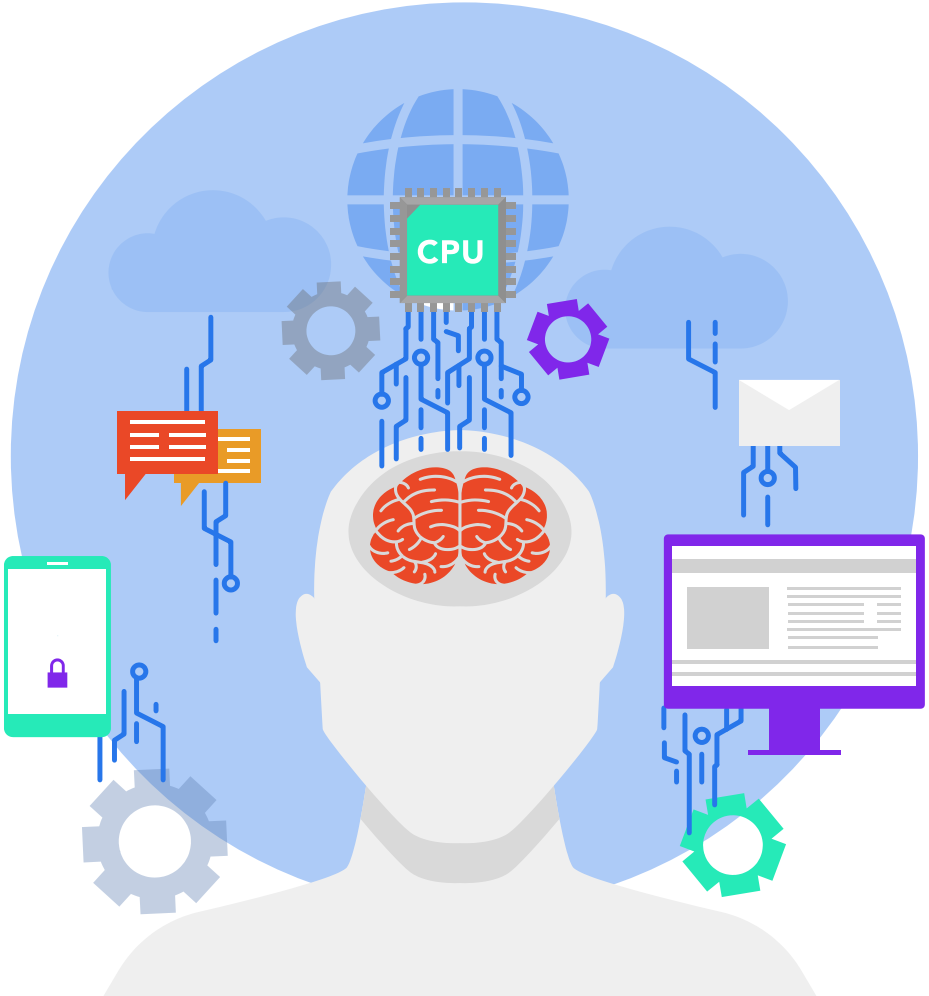



# Deep Learning

# Integrantes

Azul Kim

Felipe Cupitó

Juan Manuel De Luca

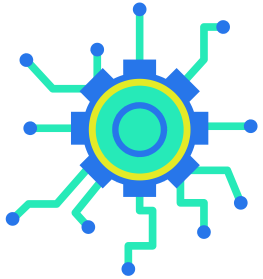
Hernán Finucci

Sol Konfederak

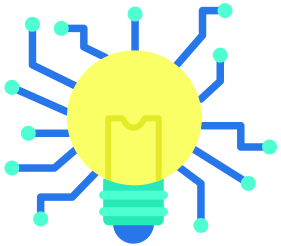
# Set de datos de entrada

```
    [0x04, 0x04, 0x02, 0x00, 0x00, 0x00, 0x00],   # 0x60, `
    [0x00, 0x0e, 0x01, 0x0d, 0x13, 0x13, 0x0d],   # 0x61, a
    [0x10, 0x10, 0x10, 0x1c, 0x12, 0x12, 0x1c],   # 0x62, b
    [0x00, 0x00, 0x00, 0x0e, 0x10, 0x10, 0x0e],   # 0x63, c
    [0x01, 0x01, 0x01, 0x07, 0x09, 0x09, 0x07],   # 0x64, d
    [0x00, 0x00, 0x0e, 0x11, 0x1f, 0x10, 0x0f],   # 0x65, e
    [0x06, 0x09, 0x08, 0x1c, 0x08, 0x08, 0x08],   # 0x66, f
    [0x0e, 0x11, 0x13, 0x0d, 0x01, 0x01, 0x0e],   # 0x67, g
    [0x10, 0x10, 0x10, 0x16, 0x19, 0x11, 0x11],   # 0x68, h
    [0x00, 0x04, 0x00, 0x0c, 0x04, 0x04, 0x0e],   # 0x69, i
    [0x02, 0x00, 0x06, 0x02, 0x02, 0x12, 0x0c],   # 0x6a, j
    [0x10, 0x10, 0x12, 0x14, 0x18, 0x14, 0x12],   # 0x6b, k
    [0x0c, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04],   # 0x6c, l
    [0x00, 0x00, 0x0a, 0x15, 0x15, 0x11, 0x11],   # 0x6d, m
    [0x00, 0x00, 0x16, 0x19, 0x11, 0x11, 0x11],   # 0x6e, n
    [0x00, 0x00, 0x0e, 0x11, 0x11, 0x11, 0x0e],   # 0x6f, o
```

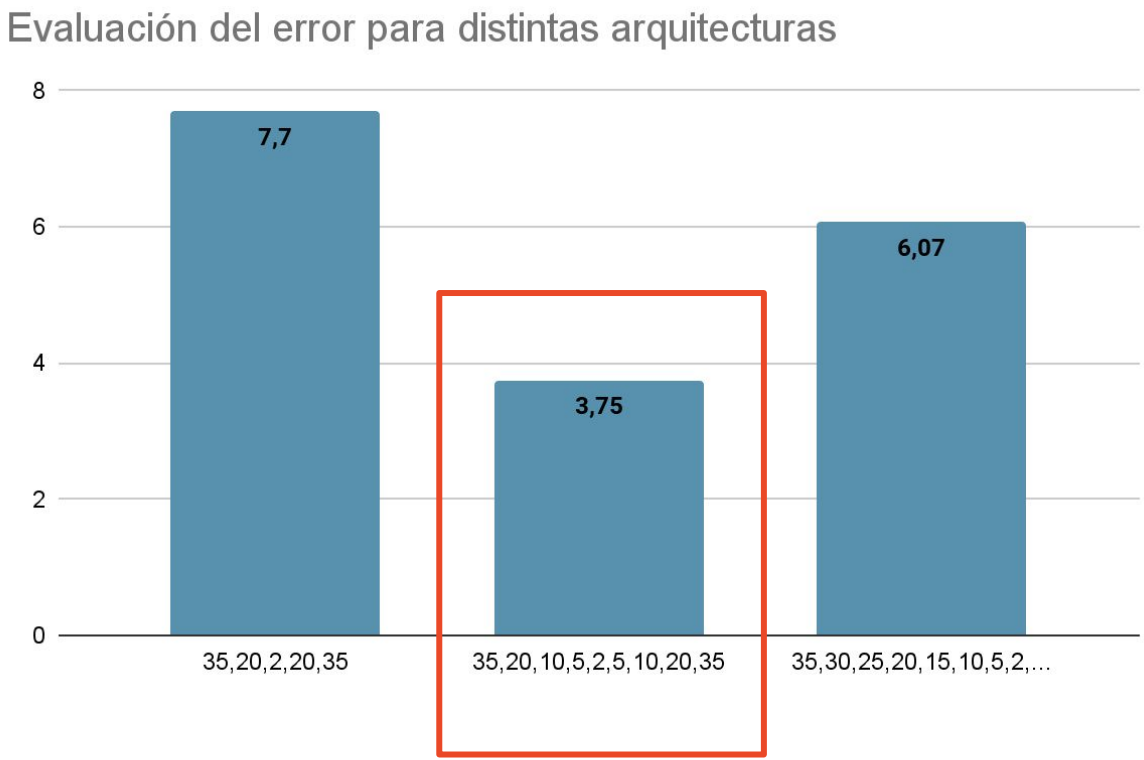
```
    [0x00, 0x1c, 0x12, 0x12, 0x1c, 0x10, 0x10],   # 0x70, p
    [0x00, 0x07, 0x09, 0x09, 0x07, 0x01, 0x01],   # 0x71, q
    [0x00, 0x00, 0x16, 0x19, 0x10, 0x10, 0x10],   # 0x72, r
    [0x00, 0x00, 0x0f, 0x10, 0x0e, 0x01, 0x1e],   # 0x73, s
    [0x08, 0x08, 0x1c, 0x08, 0x08, 0x09, 0x06],   # 0x74, t
    [0x00, 0x00, 0x11, 0x11, 0x11, 0x13, 0x0d],   # 0x75, u
    [0x00, 0x00, 0x11, 0x11, 0x11, 0x0a, 0x04],   # 0x76, v
    [0x00, 0x00, 0x11, 0x11, 0x15, 0x15, 0x0a],   # 0x77, w
    [0x00, 0x00, 0x11, 0x0a, 0x04, 0x0a, 0x11],   # 0x78, x
    [0x00, 0x11, 0x11, 0x0f, 0x01, 0x11, 0x0e],   # 0x79, y
    [0x00, 0x00, 0x1f, 0x02, 0x04, 0x08, 0x1f],   # 0x7a, z
    [0x06, 0x08, 0x08, 0x10, 0x08, 0x08, 0x06],   # 0x7b, [
    [0x04, 0x04, 0x04, 0x00, 0x04, 0x04, 0x04],   # 0x7c, |
    [0x0c, 0x02, 0x02, 0x01, 0x02, 0x02, 0x0c],   # 0x7d, ]
    [0x08, 0x15, 0x02, 0x00, 0x00, 0x00, 0x00],   # 0x7e, ~
    [0x1f, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f]   # 0x7f, DEL
```



# Ej 1.a.1 - Arquitecturas



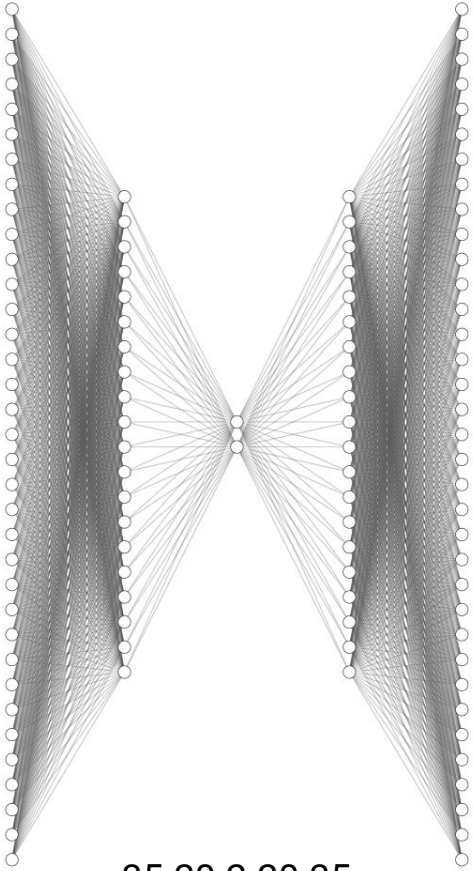
# Evaluación de arquitectura



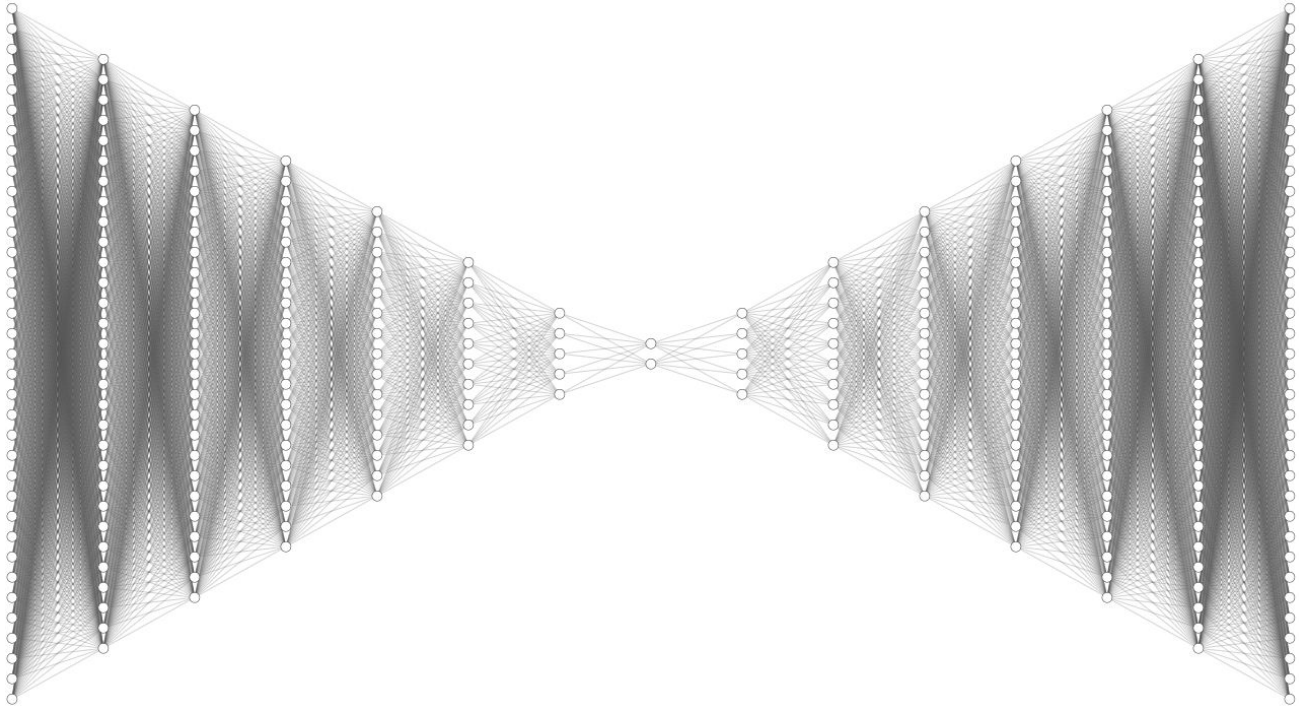

Iteraciones: 10000

Learning rate: 0,0005

# Arquitectura de neuronas

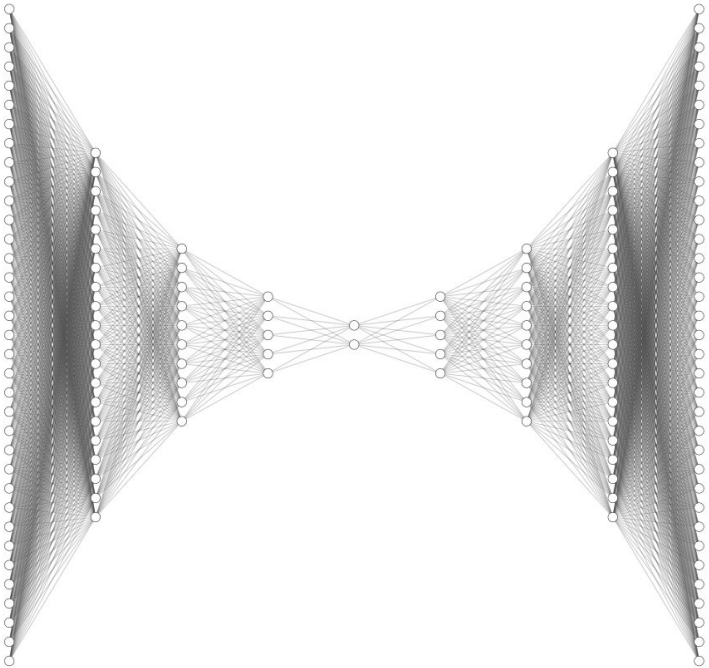


35-20-2-20-35



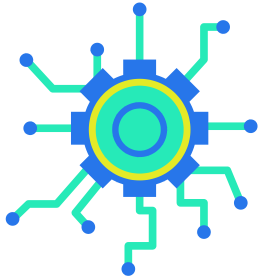
35-30-25-20-15-10-5-2-5-10-15-20-25-30-35

# Arquitectura de neuronas

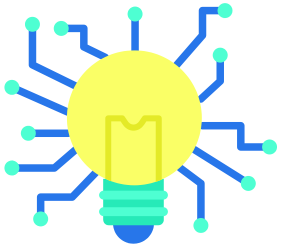


35-20-10-5-2-5-10-20-35

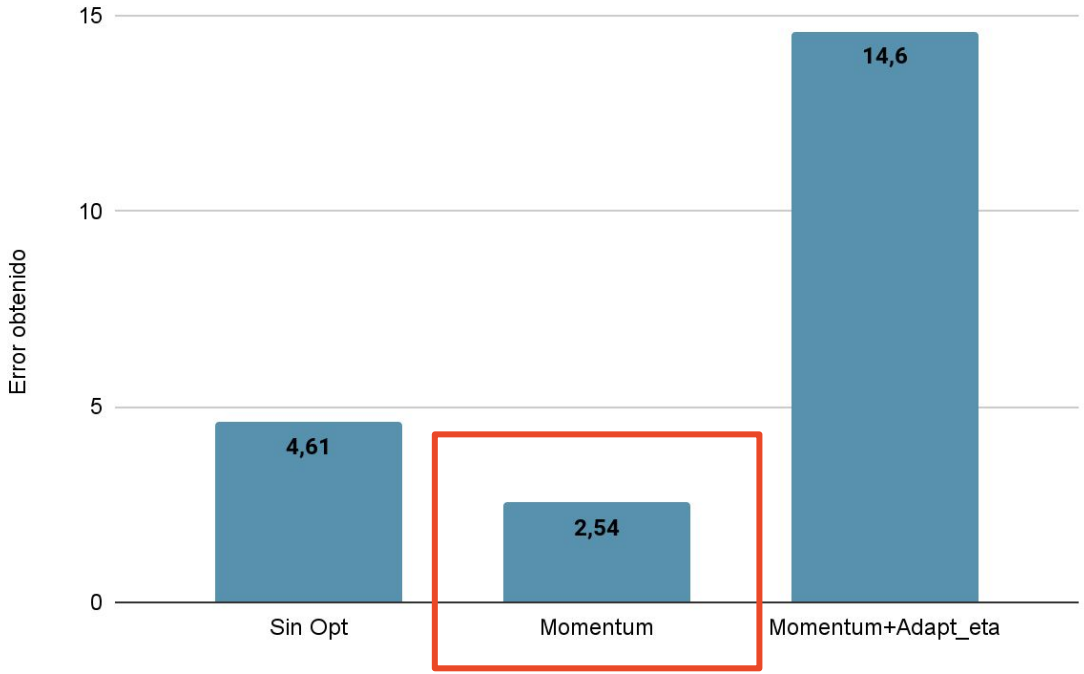
- Estructura simetrica
- Capa latente de 2 neuronas
- 35 neuronas iniciales y finales debido a la cantidad de pixeles



# Ej 1.a.2- Optimizaciones



# Evaluación de arquitectura: Error obtenido

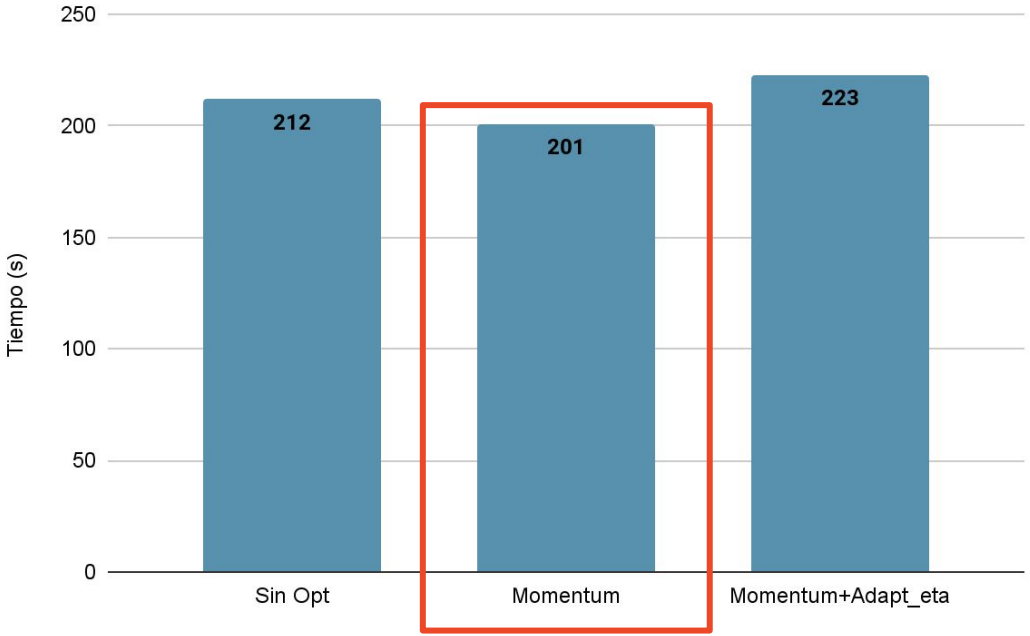



Utilizando la arquitectura:
35-20-10-5-2-5-10-20-35

Iteraciones: 10000

Learning rate: 0,0005

# Evaluación de arquitectura: Tiempo de ejecución

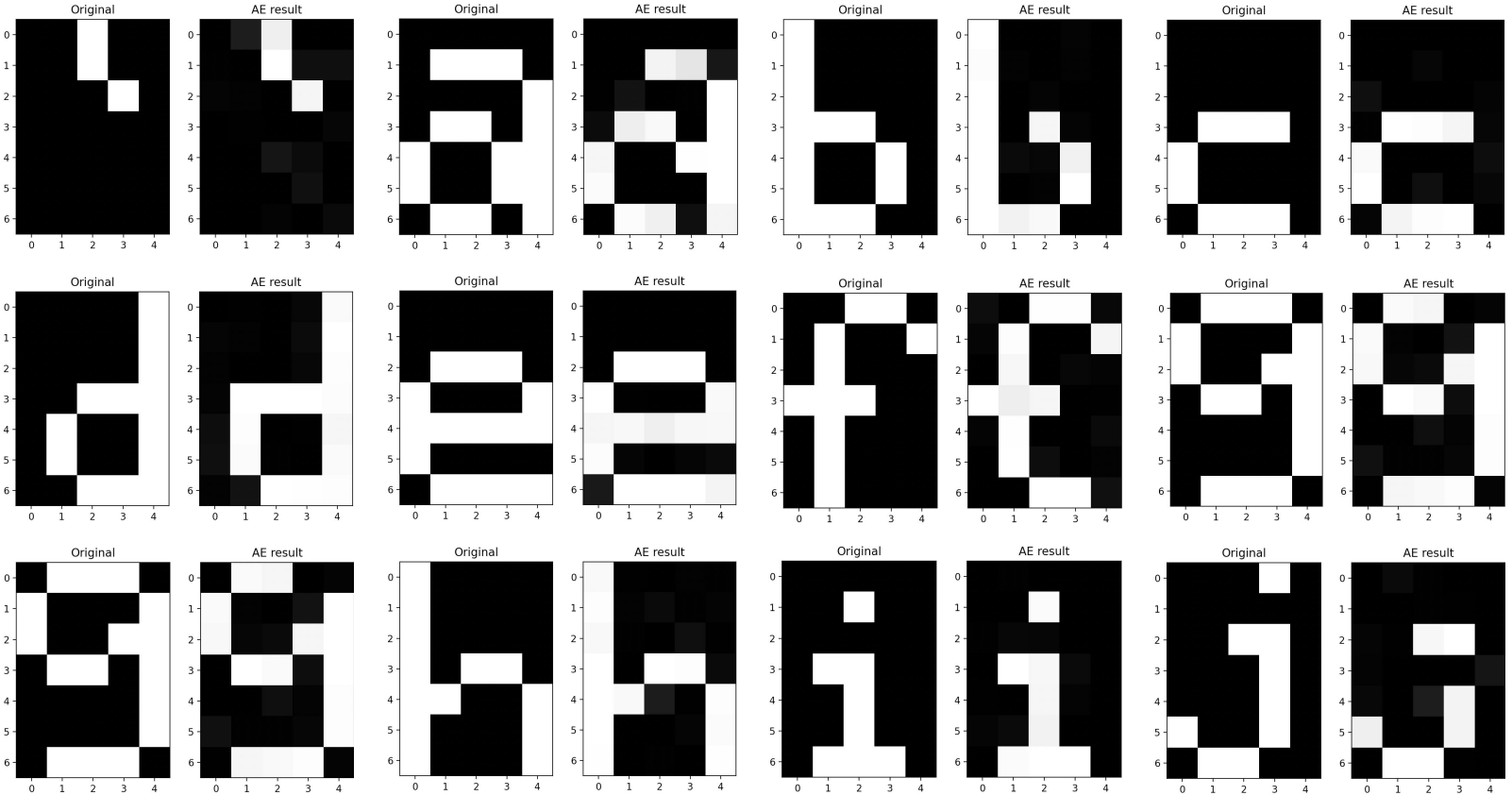



Utilizando la arquitectura:
35-20-10-5-2-5-10-20-35

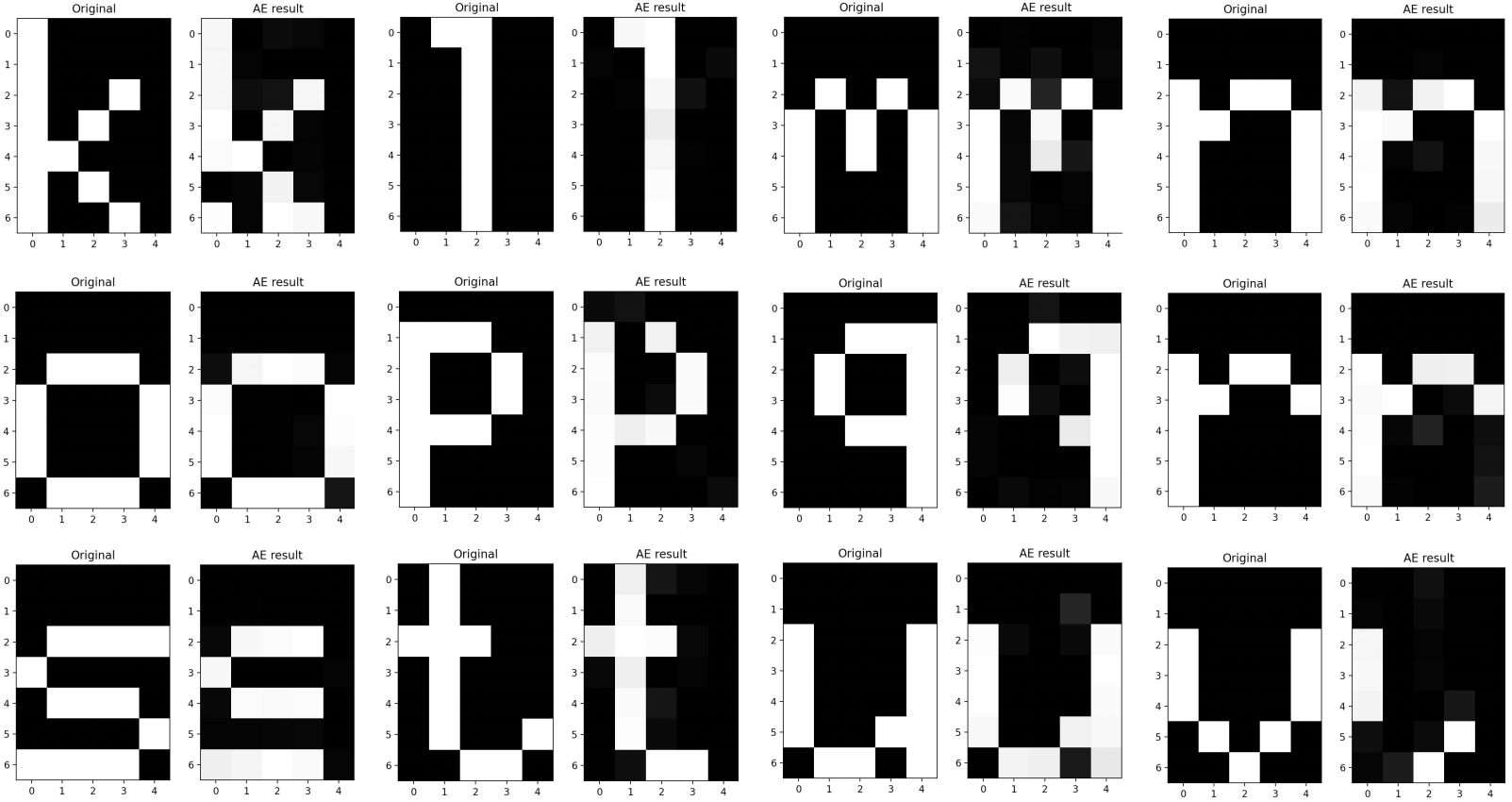
Iteraciones: 10000

Learning rate: 0,0005

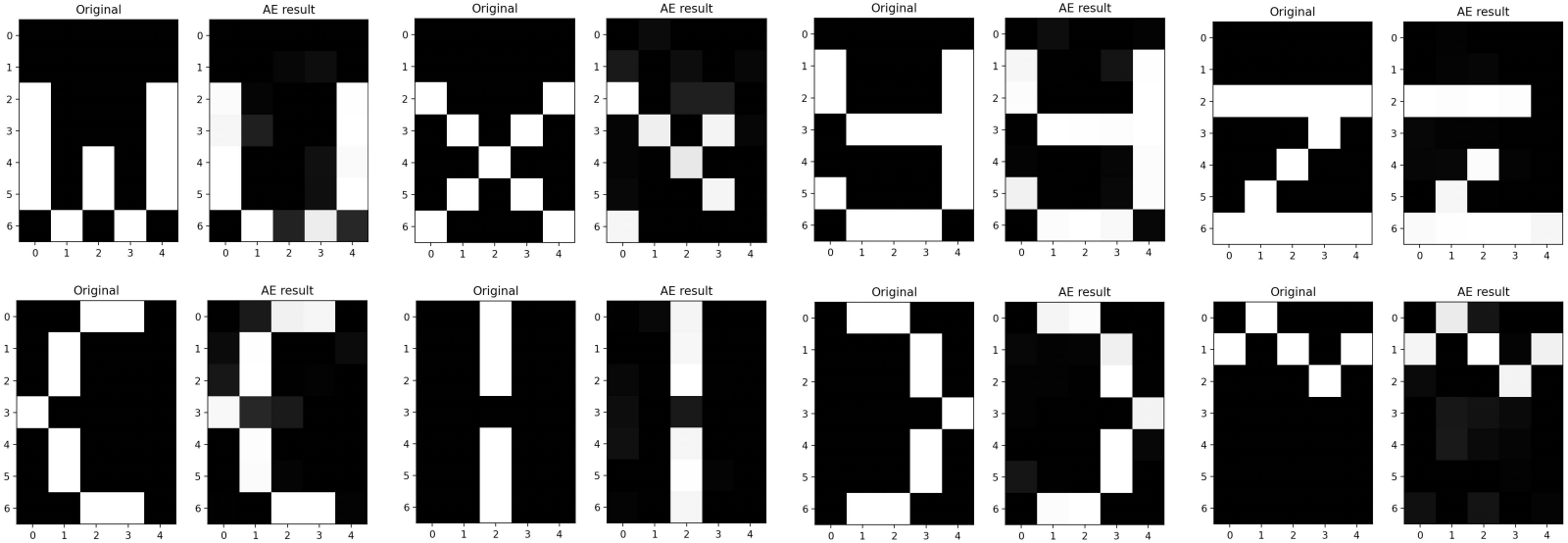
# Resultados


Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result

# Resultados

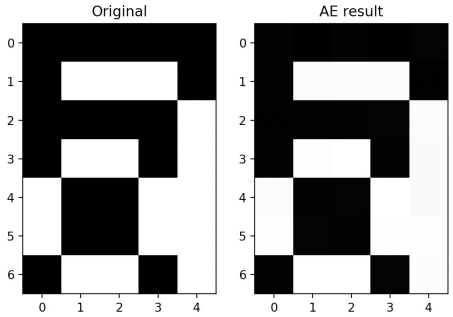

Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result

# Resultados

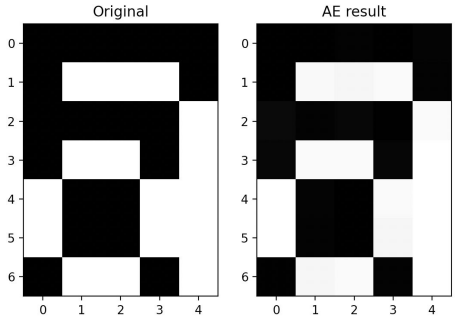

Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result
Original
AE result

# Probando distintos subconjuntos

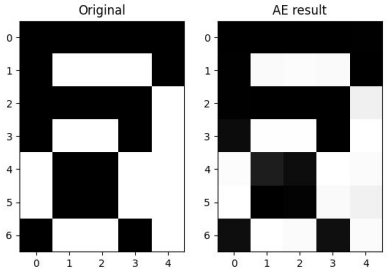
Train set: 4
Error: 0.03



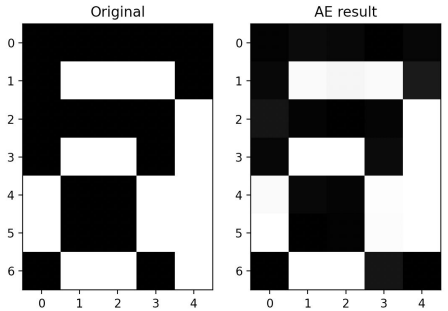

Train set: 16
Error= 1.25




Train set: 8
Error: 2.44



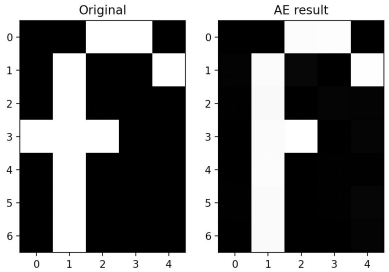

Train set: 32
Error: 3.67

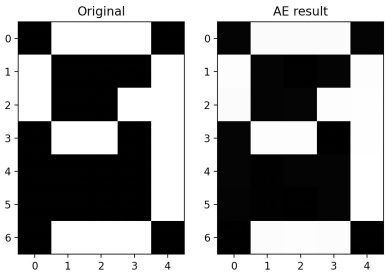


# Capacidad de generalización

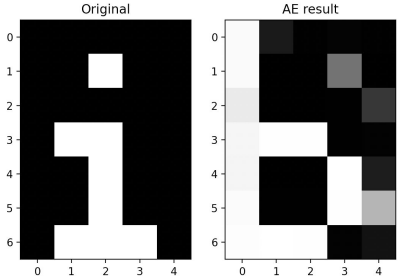
Tamaño del conjunto train: 8

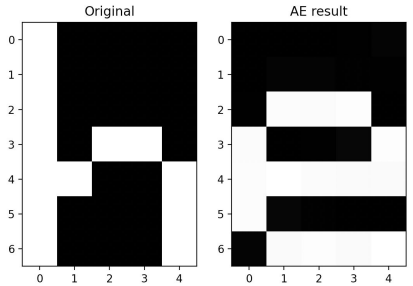
Train

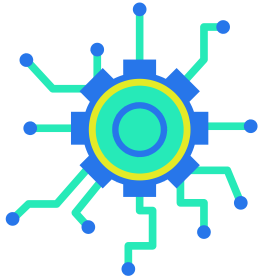


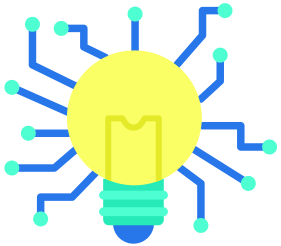

Test



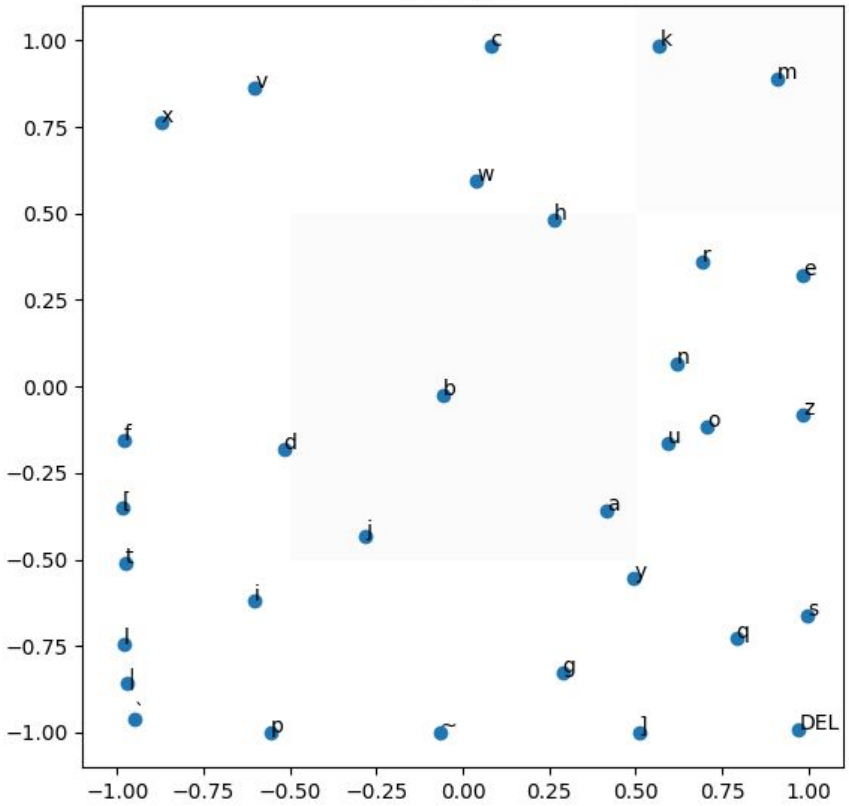



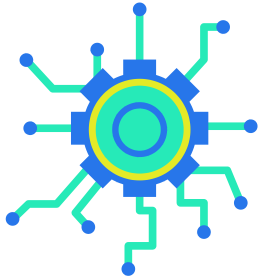


# Ej 1.a.3 - Espacio latente

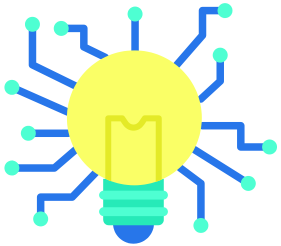


# Espacio latente: 2 dimensiones

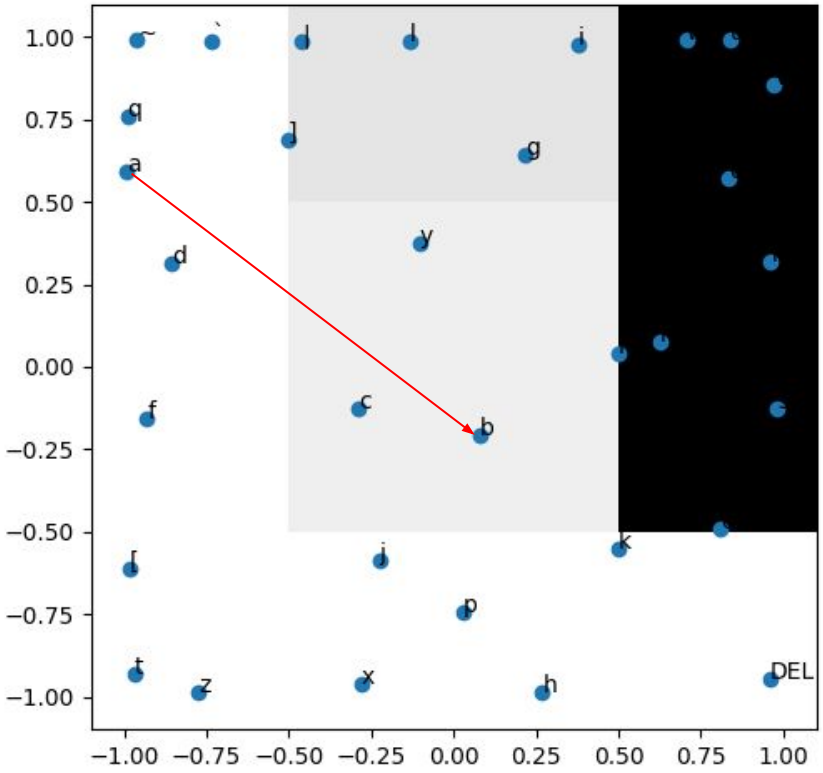




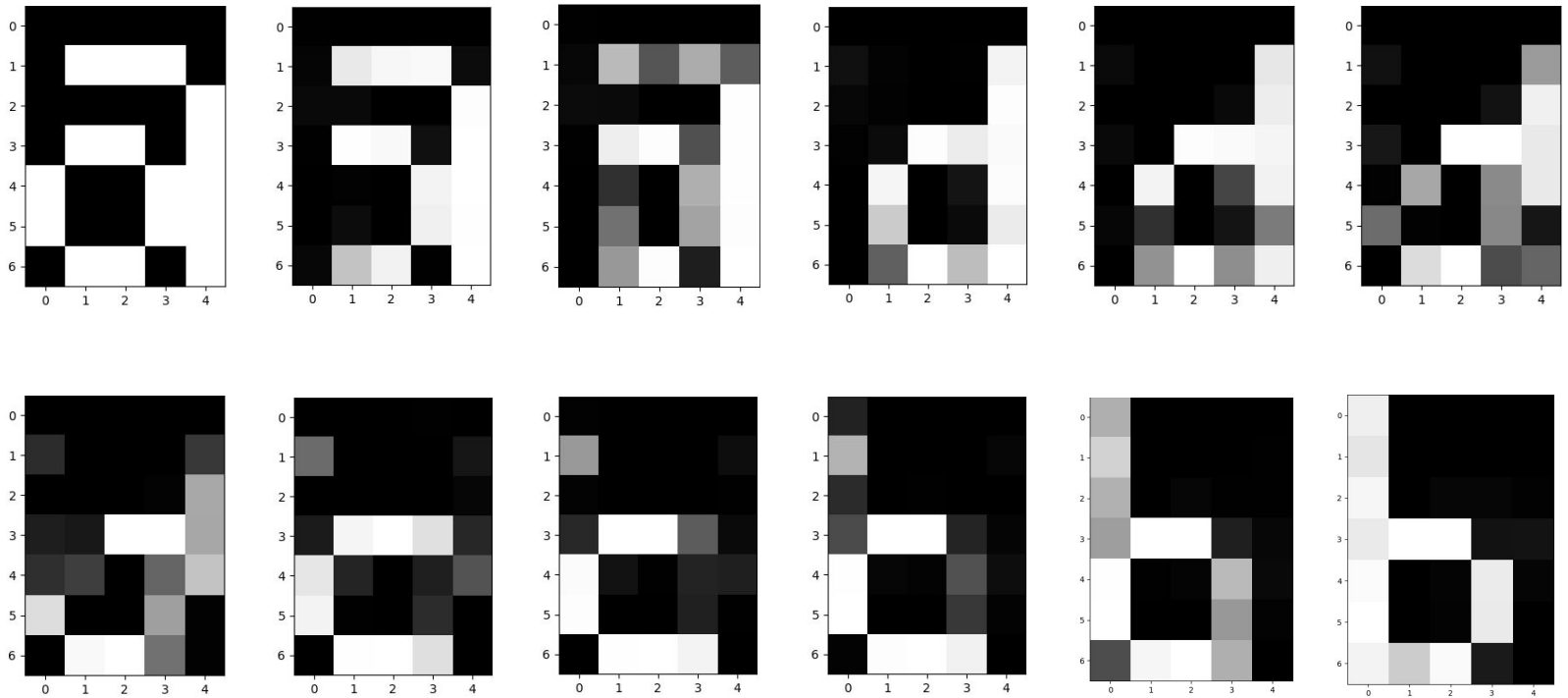
# Ej 1.a.4 - Generación de letras

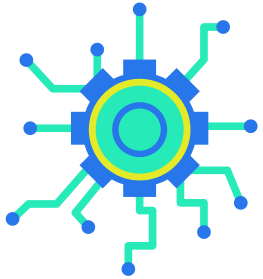


# Exploración del espacio latente

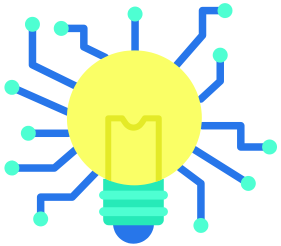


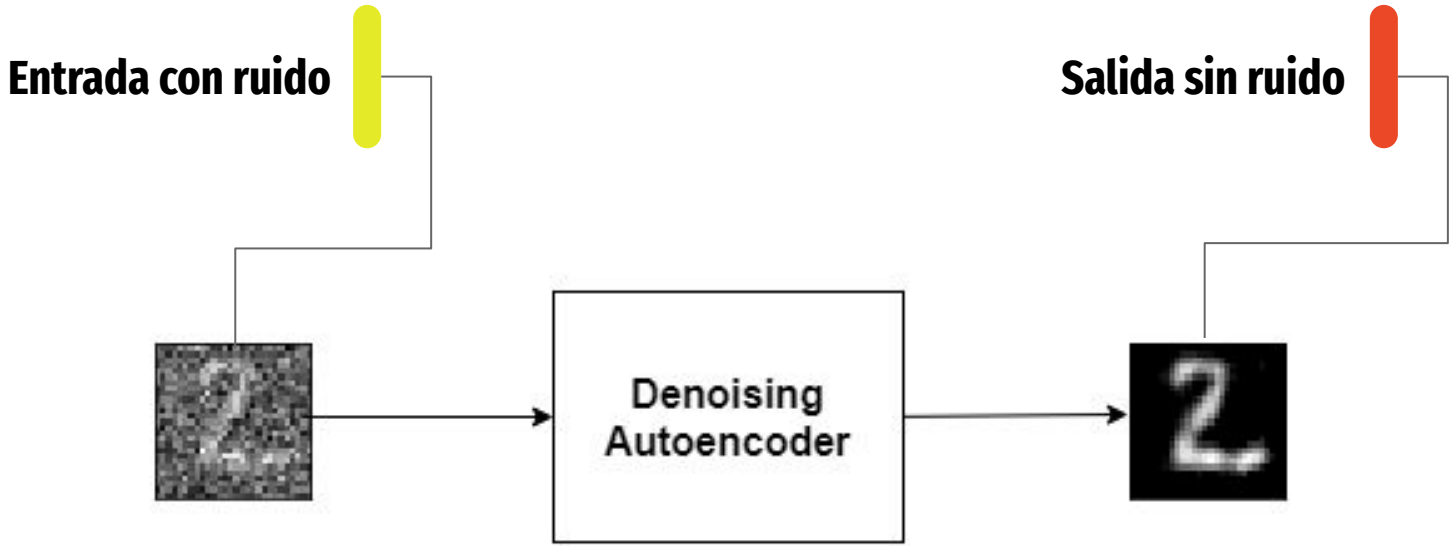
# Exploración del espacio latente





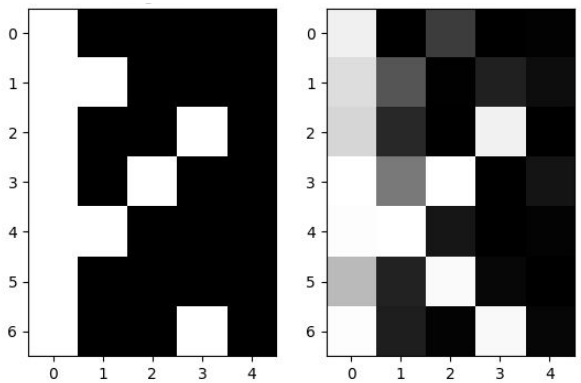
# Ejercicio 1.b: Denoising Autoencoder






La idea es eliminar el ruido de la entrada para obtener una salida “más clara”

# Capacidad de eliminar ruido

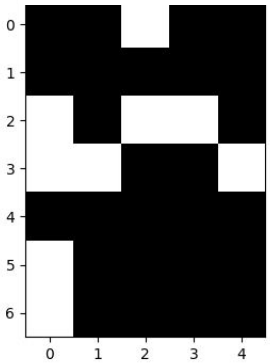



Entrenamos al autoencoder a partir de imágenes con ruido y su versión sin ruido, con el fin de que eventualmente pueda reconocer patrones que están fuera de lugar y corregirlos
Utilizamos una distribución uniforme

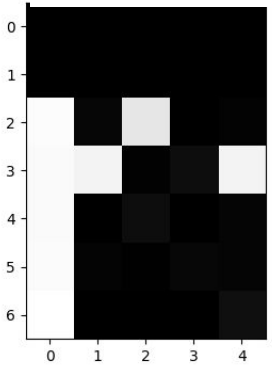
**Observación**: Utilizamos una distribución uniforme para ingresar ruido en los tests de prueba.

# Denoising autoencoders

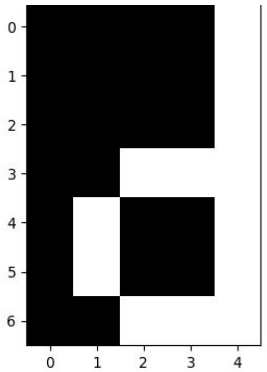
Character with noise



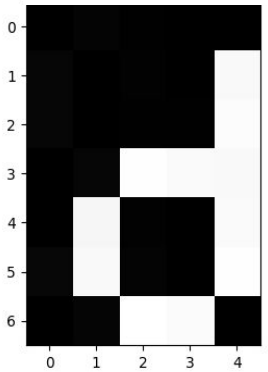
DAE result



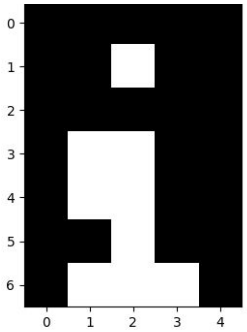
Character with noise



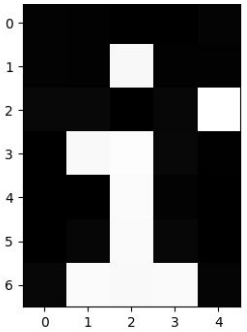
DAE result



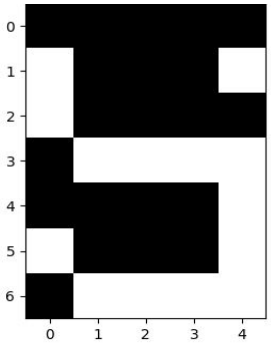
Character with noise



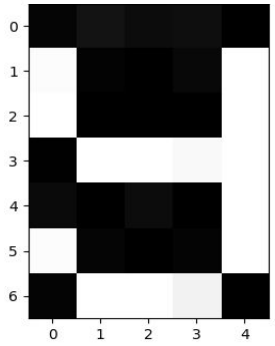
DAE result



Character with noise

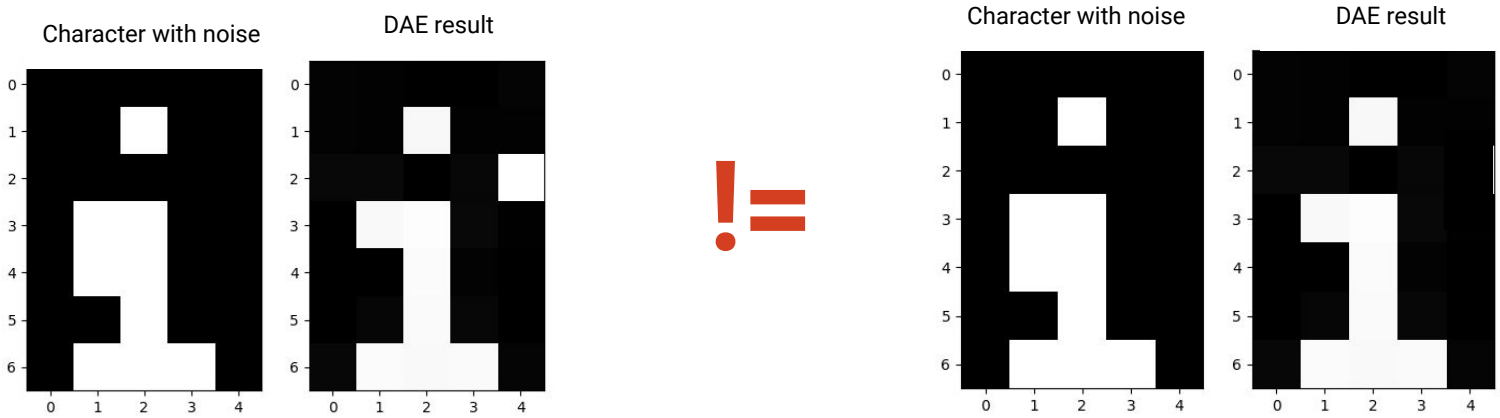


DAE result



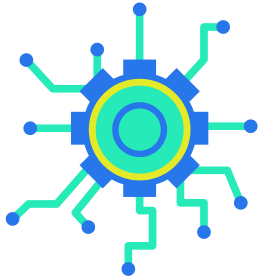
**eta=0,005**
**10000 iteraciones**

# Denoising autoencoders

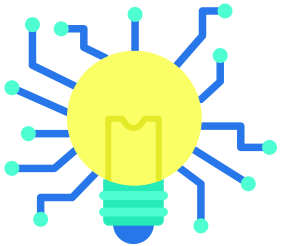


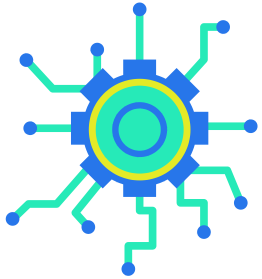

**Observación**: Los autoencoders, como el resto de resto de las redes vistas anteriormente, no garantizan devolver el mismo resultado siempre

# Conclusiones

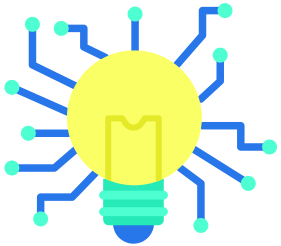


- Se puede observar cómo se agrupan los datos con propiedades similares en la representación del espacio latente.
- Cuanto mayor ruido, mayor error va a tener el denoising
- Generamos ruido utilizando una distribución uniforme. Es posible que utilizando una distribución distinta (como una Gaussiana) se pueda llegar a conclusiones más interesantes.
- Muy computacionalmente demandante, incluso para datasets con poca información.
- Una buena distribución en el espacio latente indica mejor aprendizaje.
- Utilizamos la optimización Momentum. Si hubiésemos utilizado una optimización más compleja como Adam el error sería menor y por lo tanto los outputs serían más similares al input

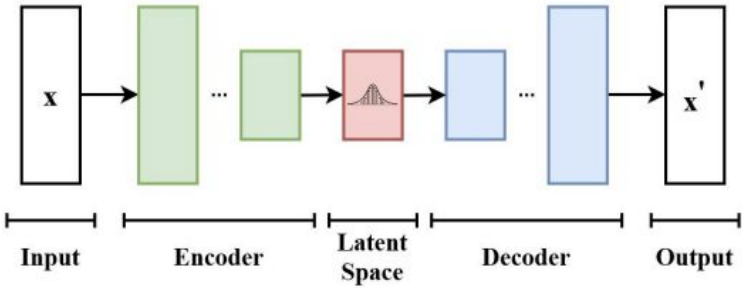




# Ej 2 - Autoencoder variacional



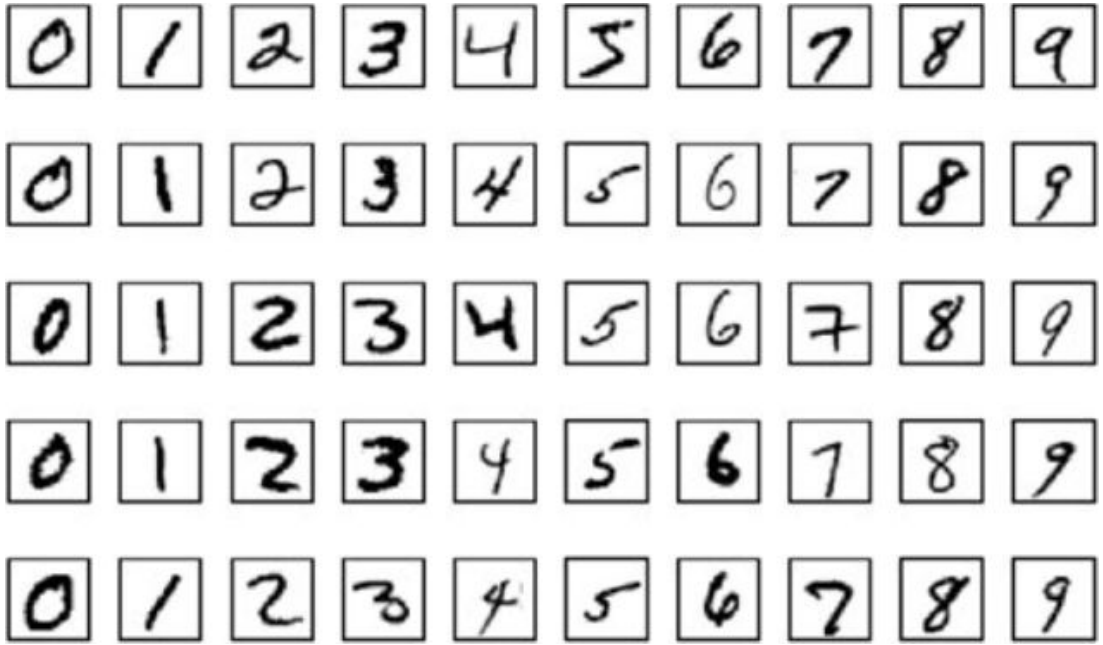
# ¿Por qué un autoencoder variacional?



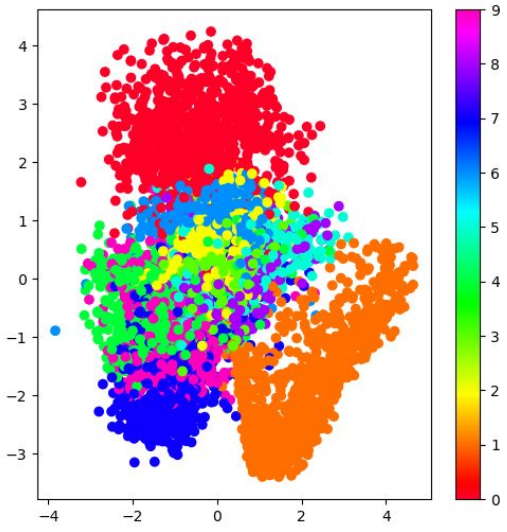

Un autoencoder variacional permite generar objetos en una ubicación arbitraria del espacio latente continuo, generando representaciones interesantes

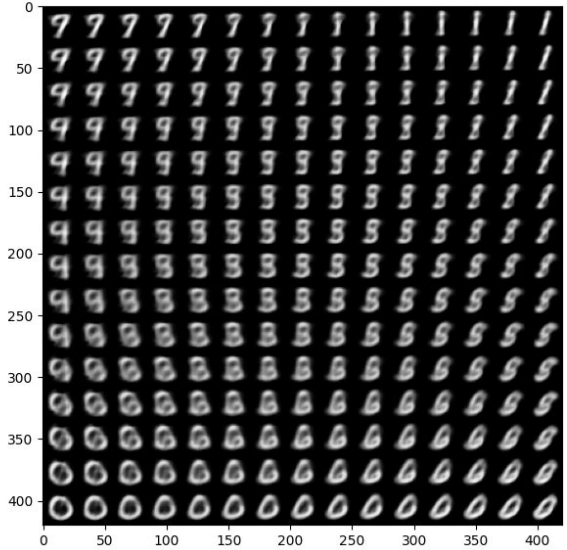
Utilizamos un dataset de 60000 imágenes compuestas de los números del 0 al 9 en formato bitmap de 28x28.

Estas imágenes se obtuvieron a partir de la database MNIST:



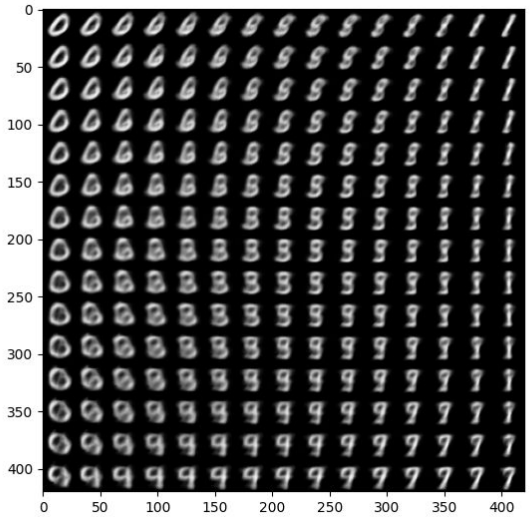
# Representación del espacio latente



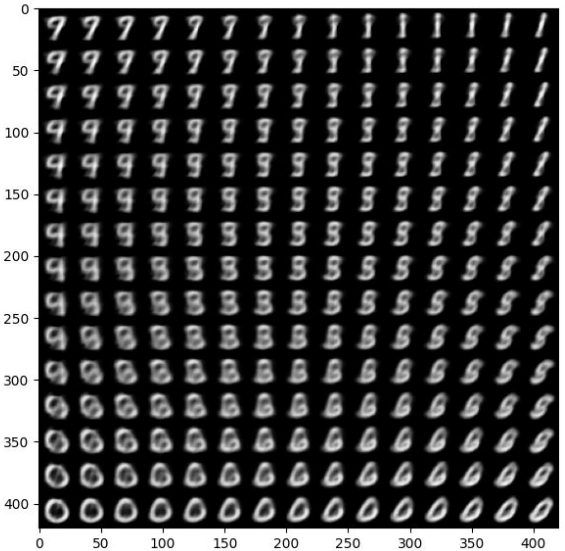




**Observación**: No se observan características de todos los números.

# Representación del espacio latente

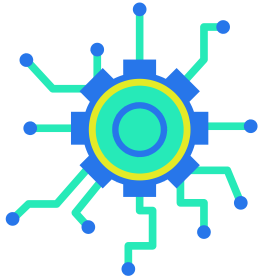



!=




**Observación**: La representaciones generadas de los espacios latentes no siempre son iguales

# Conclusiones



- Un autoencoder variacional puede crear representaciones con características de varios datos distintos.
- Las imágenes representadas en el espacio latente generalmente son muy borrosas
- Muy computacionalmente demandante, incluso para datasets con poca información.
- El espacio latente no necesariamente va a tener características de todos los datos de entrada
- La representaciones generadas de los espacios latentes no siempre son iguales