Deep Learning

Felipe Oliver (58439)
Juan Bensadon (57193)

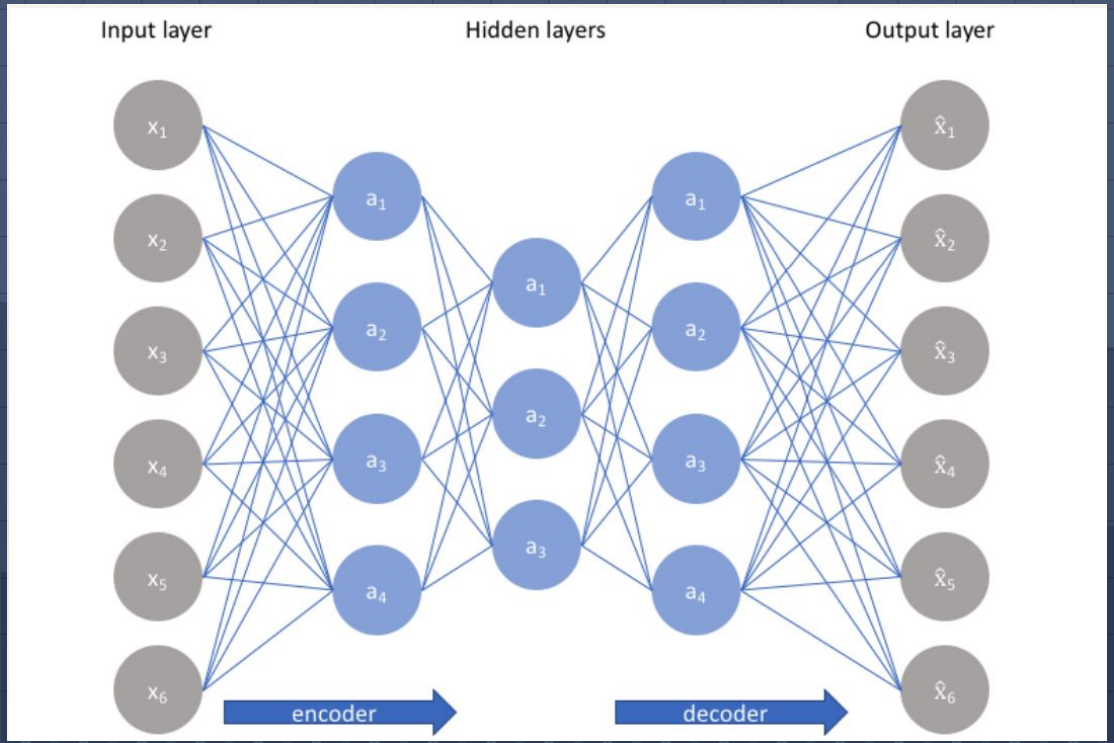
Sistemas de Inteligencia Artificial: TP5

Autoencoding de fuentes

1a

Autoencoders

El autoencoder se puede separar en 3 zonas. Encoder, espacio latente y decoder. La arquitectura seleccionada depende del ejercicio ya que según lo requerido la arquitectura varia. La red se entrena con conjunto de entrenamiento, para luego poder ser testeada.



¿Podemos representar todo el dataset en dos dimensiones?

Hipótesis: Si, porque 2 números de coma flotante = 2 * 4 bytes = 2 * 2^32 bits y tenemos solo 32 caracteres.

Pero qué pasaría si hiciéramos todas las letras posibles a partir de una matriz de 5x7, entonces tendríamos una cantidad igual a 2^35. Con esa cantidad de letras, los 2 números de coma flotante no serian capaz de representar todo ese dataset. Ya que 2^35 > 2 ^ 33 = 2 * 2 ^ 32!

Ahora vamos a ver que sucede utilizando la red con 32 caracteres y sus distintos tipos de optimizaciones

Arquitecturas probadas

[35, 20, 10, 6, 2, 6, 10, 20, 35] → (Ganadora)

- 10 letras aprendidas con éxito
- 11 letras aprendidas con algunos errores
- Mayor tiempo para entrenar
- Error siempre < 1

[35, 5, 2, 5, 35]

- 9 letras aprendidas con éxito
- 10 letras aprendidas con algunos errores
- Menor tiempo para entrenar
- Error > 1

Tipos de Optimizaciones

- Sin optimización, learning rate con un delta fijo
- Momentum
- Learning Rate adaptable (Aumento o decremento de error)
- Needle learning rate

Sin optimización

- Mayor “desorden” en capa latente
- Menor espacio espacio entre números
- Puesto N°3 en desempeño del autoencoder, se confunde a veces entre “(“ y “)”
- Error alrededor de 1.3
- Alrededor de 115 segundos de entrenamiento

Momentum(0.8)

- Menor “desorden” en capa latente que sin optimización
- Mayor espacio entre números (ligeramente)
- Puesto N°1 en desempeño del autoencoder
- Error entre 0.9 y 0.45
- Alrededor de 90 segundos de entrenamiento

Learning Rate adaptable

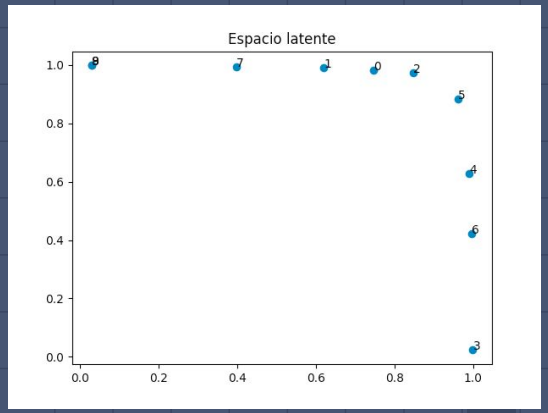
- Menor “desorden” en capa latente
- Mayor espacio espacio entre números
- Puesto N°2 en desempeño del autoencoder, se confunde a veces entre “(“ y “)”
- Error alrededor de 0.9
- Alrededor de 110 segundos de entrenamiento

Mala Corrida

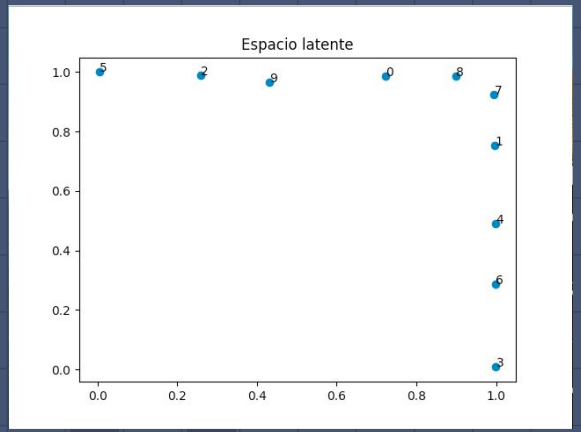
- Mucho desorden en la capa latente
- El espacio entre números es aleatorio
- Puesto N°4 en desempeño del autoencoder, muchos errores
- Error > 5
- Llega al final de las épocas

Gráfico en dos dimensiones (capa latente)

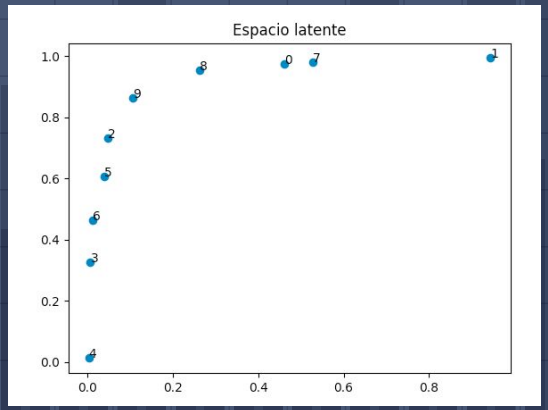
Sin optimización



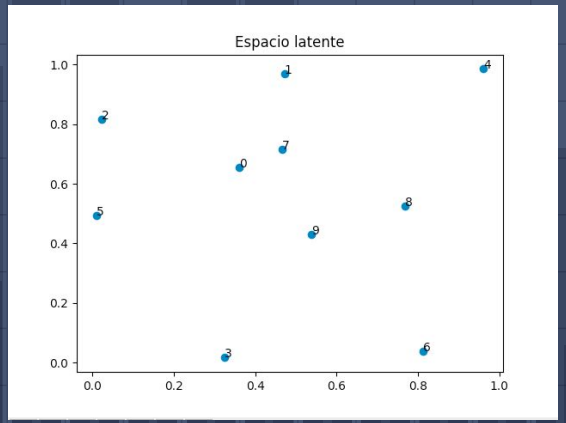
Momentum



Learning Rate

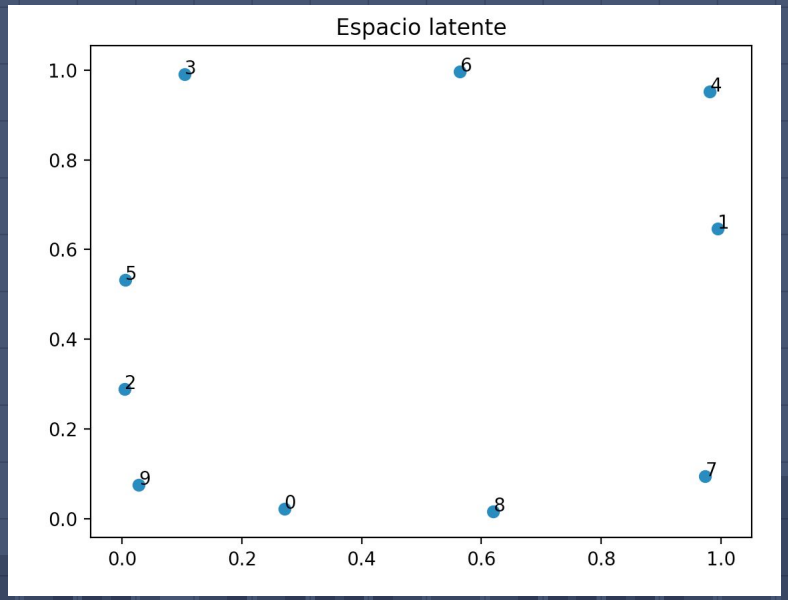


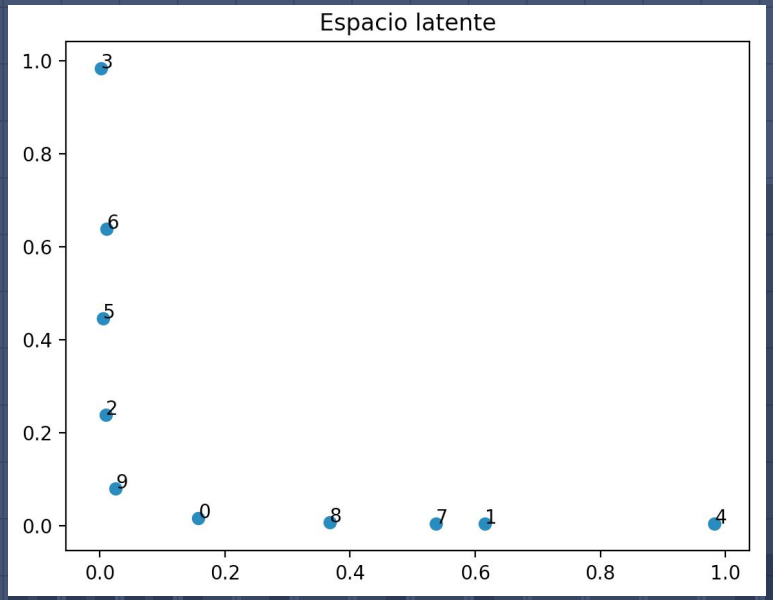
Mala Corrida



0 -> Space
1 -> !
2 -> "
3 -> #
4 -> $
5 -> %
6 -> &
7 -> '
8 -> (
9->)

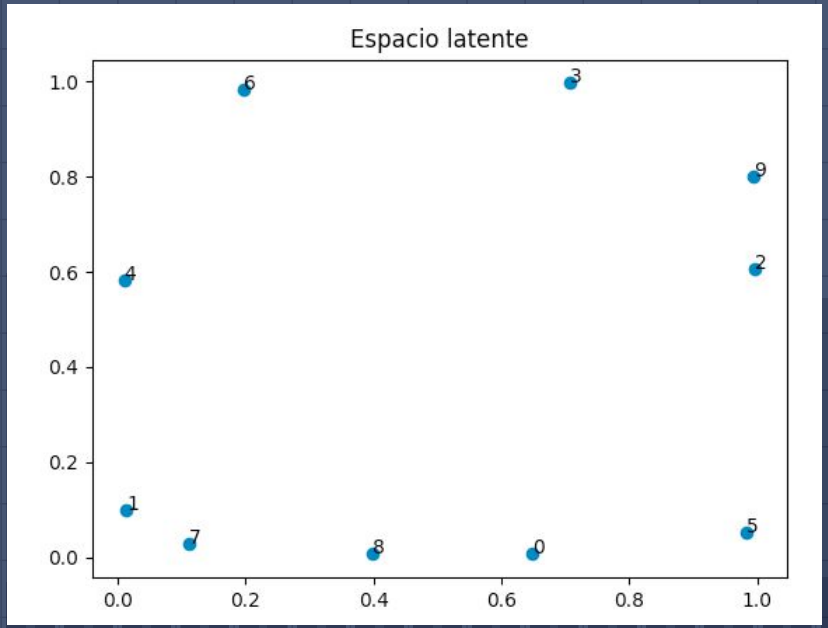
Corridas Interesantes





Mejor Combinación

Se realizó el entrenamiento con 10 letras, momentum en 0.8 y lr en 0.8



- Error en 0.5
- Tiempo 31 segundos
- Sin errores o como mucho un bit
- Capa latente con números bien separados

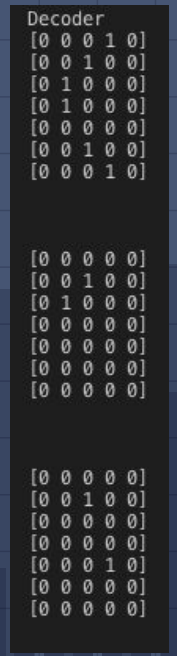
Generar una nueva letra

Decidimos generar 3 muestras distintas aleatorias de números flotantes para ver como el decoder del autoencoder genera nuevas letras.

Muestra N1:
[0.1; 0.1]

Muestra N2:
[0.6; 0.6]

Muestra N3:
[0.9; 0.9]



Generación con números al azar

```
Decoder
Numero generado al azar:

[0.2322644948078939, 0.6695890450563083]
[0 0 1 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]


Numero generado al azar:

[0.4266174460200909, 0.419389315277937]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 1 0]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]


Numero generado al azar:

[0.17005613321154567, 0.3760677934749319]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 1 0 1 0]
[0 0 0 0 0]
[0 0 1 0 0]
[0 1 0 1 0]
```

```
Decoder
Numero generado al azar:

[0.5821365289782626, 0.16884497794088138]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 1 0]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]


Numero generado al azar:

[0.5025500797498806, 0.7904161398508585]
[0 0 1 0 0]
[0 1 0 1 0]
[1 1 1 1 0]
[0 1 0 1 0]
[1 0 1 0 1]
[1 1 0 1 0]
[0 1 1 0 0]


Numero generado al azar:

[0.33503686219425, 0.11025769101775607]
[0 1 0 0 1]
[0 1 0 0 1]
[1 0 0 1 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```

```
Decoder
Numero generado al azar:

[0.4442317701673226, 0.620091382482282]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 1 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]


Numero generado al azar:

[0.8882117220292151, 0.594587449424878]
[1 1 0 0 1]
[1 1 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
[1 0 0 1 1]
[1 0 0 1 1]


Numero generado al azar:

[0.11001955288238163, 0.8177410713480965]
[0 0 1 0 0]
[0 1 1 1 1]
[1 0 1 0 0]
[0 1 1 1 0]
[0 0 1 0 1]
[1 1 1 0 0]
[0 0 1 0 0]
```

Obs: Se obtuvieron caracteres ya existentes!

Denoising Autoencoder

1b

Arquitectura denoising autoencoder

[35, 20, 10, 6, 2, 6, 10, 20, 35] → (Ganadora de nuevo)

- 5 letras intentadas
- 2 letras adivinadas
- Mayor tiempo para entrenar
- Error < 0.1

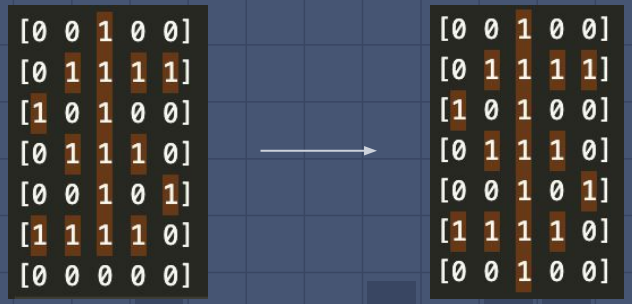
[35, 5, 2, 5, 35]

- 5 letras intentadas
- 0 letras adivinadas
- Menor tiempo para entrenar
- Error > 0.1

Maneras de hacer ruido

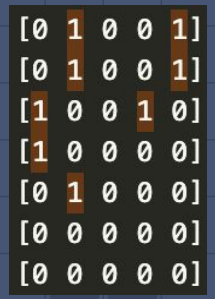
- Salt and Pepper
- Noisy function lighter, transforma cada número hexadecimal a binario primero, luego cada 0 o 1 tiene la probabilidad de ser cambiado con una probabilidad X.
- Noisy function light, no cambia los números de hexadecimal a binario, sino que cada número hexadecimal tiene una probabilidad X de cambiarse a otro número hexadecimal entre [0x00 y 0x1f]. La consideramos heavier ya que estas cambiando toda una columna entera de caracteres.

Resultados del denoising (probabilidad ~0.05)



No logramos quitar ruido extremo con éxito, pero sí en ciertos casos pudimos hacerlo.

Extremely Noisy (probabilidad ~0.5)



=

!

=

#

Si bien no se llegaron a símbolos, sí se notó una tendencia a eliminar “ruido” (unos)

Nuevo set de datos

2

Alfabeto Cirílico

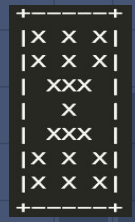
Д

Б

Щ

Т

Ж

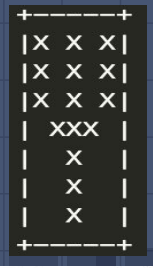


П

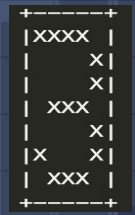
И

Ф

Ψ



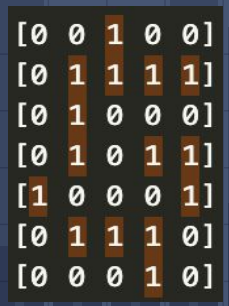
З

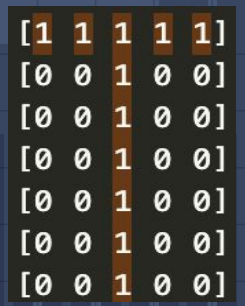


Arquitectura para el nuevo set

[35, 20, 10, 6, 2, 6, 10, 20, 35]

Letras generadas:





Gracias!