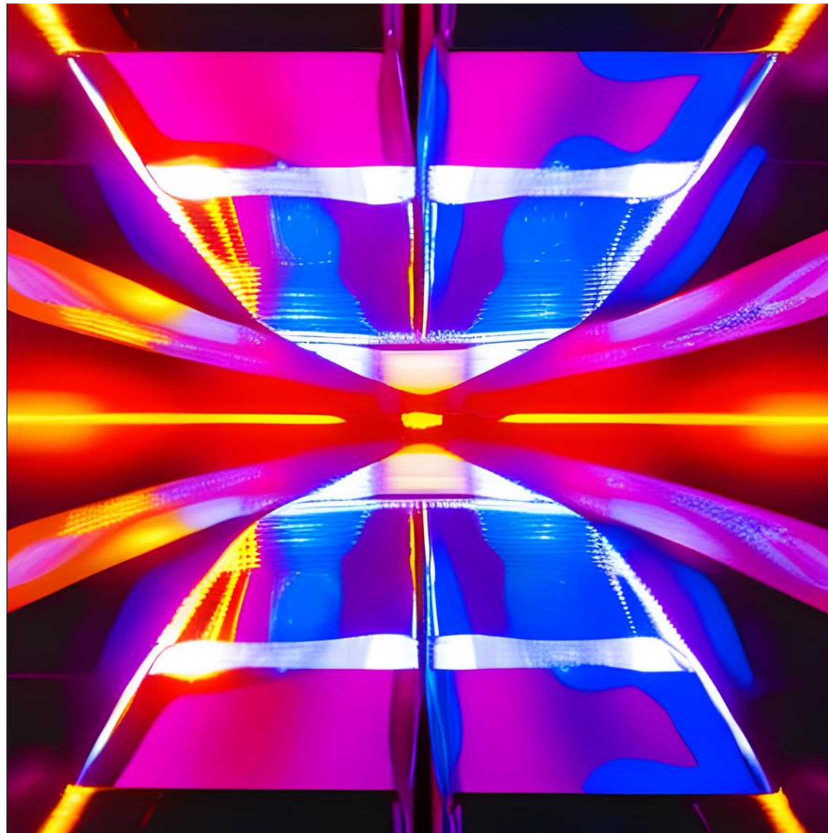
INTRODUCCIÓN A LOS AUTOCODIFICADORES (AUTOENCODERS)



"Two parallel mirrors, different reflections of the same object" (www.canva.com)

José Ángel Martínez-Huertas (jamartinez@psi.uned.es)

Modelos de redes neuronales

Máster en Metodología de las Ciencias del Comportamiento y de la Salud

Universidad Nacional de Educación a Distancia

ÍNDICE

1. ¿Qué es un autocodificador (*autoencoder*)?	**3**
2. Breve repaso histórico	**6**
3. Una analogía psicológica de los procesos de codificación y decodificación	**7**
4. Arquitectura y complementos añadidos	**9**
5. La función de pérdida en el contexto de los autocodificadores	**10**
6. Tipos de autocodificadores (entre otros)	**11**
7. Ejemplo de aplicación en R	**14**

Este documento pretende ser una introducción conceptual (que no analítica) a las redes neuronales llamadas autocodificadores (*autoencoders*) en el contexto del Máster en Metodología de las Ciencias del Comportamiento y de la Salud en la Universidad Nacional de Educación a Distancia. Así, este documento asume que la/el estudiante que se enfrente a los autocodificadores (*autoencoders*) tiene una base conceptual y analítica suficiente como para entender algunos conceptos fundamentales de las redes neuronales clásicas como "retropropagación", "nodos" o "capas ocultas". En cualquier caso, los contenidos presentados en este documento pretenden explicar conceptualmente todos los pasos que comúnmente se utilizan para estimar autocodificadores (*autoencoders*).

1. ¿Qué es un autocodificador (*autoencoder*)?

Aunque la fundamentación matemática de los autocodificadores (autoencoders) es relativamente compleja, su definición conceptual es bastante simple. Son una arquitectura de red neuronal moderna de aprendizaje no supervisado y, en su parametrización más frecuente, consiste en la combinación de dos funciones diferentes: la función codificadora, encargada de codificar o transformar la información del input (normalmente, reduciendo su dimensionalidad), y la función decodificadora, encargada de transformar de nuevo la información codificada a su estado original. Por tanto, el input y el output de esta arquitectura de red neuronal son iguales, véase: **X** y **X**'.

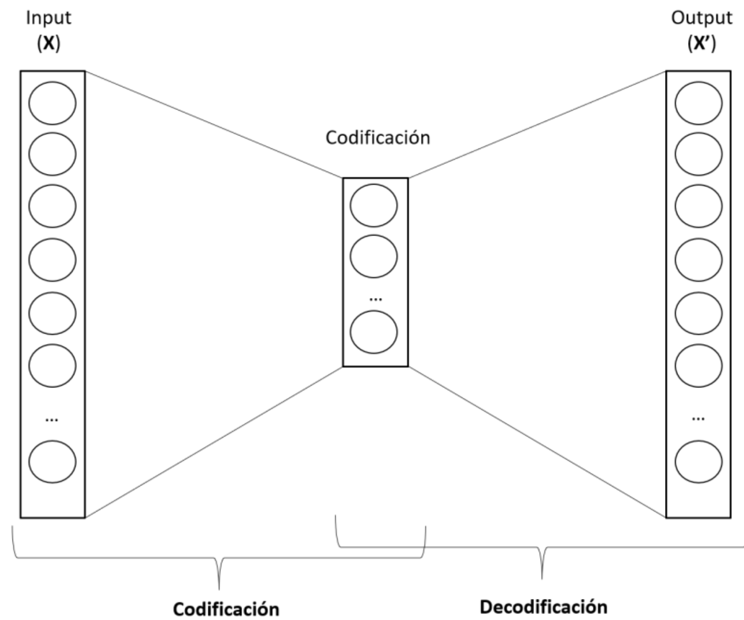


Figura 1. Esquema básico de un autocodificador (*autoencoder*) donde un input, **X**, de dimensionalidad *m* tiene como output a sí mismo, **X'**, también de dimensionalidad *m*. Este input es codificado por *n* capas en la función de codificación, donde se suele reducir considerablemente la dimensionalidad del input. Después, *p* capas de la función de decodificación reconstruyen la información del input en **X'** a partir de la información codificada.

Los autocodificadores (*autoencoders*) son muy utilizados en el área de ciencias computacionales para el análisis y procesamiento de datos debido a que son capaces de capturar patrones complejos y extraer características relevantes de los conjuntos de las bases de datos. Uno de los principales beneficios de los estos modelos de redes neuronales es su capacidad para reducir la dimensionalidad de los datos. Como bien es sabido, las bases de datos en psicología suelen ser multidimensionales y pueden contener una gran cantidad de variables. Esto puede extenderse significativamente si consideramos algunas aplicaciones de la metodología donde se pueden manejar cientos

de indicadores como, por ejemplo, los indicadores computacionales y/u otras medidas. Al aplicar un autocodificador (*autoencoder*) se pretende comprimir y representar estos datos en un espacio de menor dimensionalidad. Esta reducción de dimensionalidad permite una visualización más clara de los patrones y estructuras subyacentes presentes en los datos, lo que facilita su interpretación y su análisis. En cierto modo, estas representaciones más simples podrían verse como un reflejo de las técnicas de análisis factorial, pero desde un punto de vista no lineal y mucho más complejo.

Además, los autocodificadores (*autoencoders*) son especialmente útiles en la identificación de características relevantes en los datos porque aprenden automáticamente las características más distintivas y significativas durante el proceso de entrenamiento. Aunque es poco común ver el uso de estos modelos de redes neuronales en estudios aplicados en psicología, esta capacidad de identificar características relevantes es especialmente valiosa ya que podría permitir comprender diferentes aspectos de los datos analizados de manera automática. De manera similar, con este tipo de modelos también se pueden generar datos artificiales o simulados que mantengan los patrones y características del set de entrenamiento. Este tipo de aplicaciones podría ayudar a aquellas investigaciones donde la recogida de datos es muy costosa o limitada, o se pretende autogenerar un output para una tarea concreta.

A modo de ejemplo, los autocodificadores (*autoencoders*) se han utilizado en ámbito de la neurociencia cognitiva para analizar y procesar datos de actividad cerebral[1]. En teoría, esta arquitectura de red neuronal nos permitiría utilizar distintos datos cerebrales, como la señal electroencefalográfica, por ejemplo, para extraer

[1] Marino, J. (2022). Predictive coding, variational autoencoders, and biological connections. *Neural Computation, 34*(1), 1-44. https://doi.org/10.1162/neco_a_01458.

características latentes o patrones de activación asociados a ciertos procesos cognitivos. Así, por ejemplo, estos modelos permitirían detectar patrones anormales de la señal recibida o hacer predicciones en base a parte de la señal recibida.

2. Breve repaso histórico

El origen de los autocodificadores (*autoencoders*) se encuentra en las memorias asociativas[2], que consisten en el almacenamiento y recuperación de información por asociación aprendida con otras informaciones. El origen de estos modelos se encuentra en los inicios de los modelos conexionistas de la cognición. Por ejemplo, aunque en ese momento no se les conocía con ese nombre, las bases teóricas de los autocodificadores fueron descritas por Hinton y Anderson en 1981 al describiendo un modelo de red neuronal que podía aprender a codificar y decodificar patrones de entrada, permitiendo la reconstrucción de los mismos. Así se asentó la idea fundamental que supone la base para el desarrollo posterior de los autocodificadores es la codificación de información relevante en una representación latente de menor dimensionalidad que luego se utiliza para la decodificación para reconstruir los patrones originales.

---

[2] Las memorias autoasociativas forman parte de los primeros temas de la asignatura y se recomienda leer antes dichos temas.

3. Una analogía psicológica de los procesos de codificación y decodificación

Los autocodificadores (*autoencoders*) se nutren de dos procesos distintos: el proceso de codificación y el proceso de decodificación. Así, estos procesos guardan una estrecha relación con cómo se representa la información porque, básicamente, esta arquitectura de redes neuronales tiene el objetivo de aprender de manera eficiente los datos que recibe en el input a través de una codificación comprimida o reducida, y luego reconstruir o recodificar la información inicial a partir de esa información reducida.

Como se ha comentado, el proceso de codificación se encarga de comprimir la información del input en una representación latente[3] de dimensionalidad reducida. Esto implica que la información es comprimida y destilada[4] en un espacio de características más compacto (es decir, de menor dimensionalidad). Durante el proceso de entrenamiento, el autocodificador (*autoencoder*) aprende a extraer automáticamente las características más relevantes y distintivas de la información del input, lo que permite obtener una representación comprimida pero informativa. Así, en teoría, la codificación captura las principales características y patrones presentes en los datos, lo que facilita su interpretación y análisis. En términos psicológicos, podríamos relacionar los procesos de codificación de esta arquitectura de red neuronal con el procesamiento de la información que luego es almacenada en nuestra memoria. Por ejemplo, cuando guardamos información o recuerdos de nuestras vivencias, no solemos guardar

---

[3] El concepto de “latente” tiene algunas connotaciones distintas a las del análisis factorial y los modelos de ecuaciones estructurales, pero la lógica es la misma: reducir la dimensionalidad de las variables observadas en una estructura o componente más abstracto.

[4] El concepto de “destilación” no deja de ser una metáfora del refinamiento de las representaciones por parte del modelo en cuestión. En otras áreas más relacionadas con la evaluación en psicología, este concepto de “destilación” podría entenderse como la depuración de las puntuaciones computacionales para la resolución de una tarea concreta.

información con un alto grado de detalle y totalmente fidedigna con la realidad, sino que almacenamos una versión reducida de la representación de nuestro aprendizaje.

Por su lado, el proceso de decodificación se encarga de reconstruir (recodificar) la información original del input originales a partir de la representación latente (de dimensionalidad recudida) obtenida en la etapa de codificación. El objetivo es que la reconstrucción sea lo más precisa posible, de modo que la información recuperada se asemeje lo máximo posible a la información del input original. Durante el proceso de entrenamiento, el autocodificador (*autoencoder*) aprende a ajustar sus parámetros para minimizar la diferencia entre la información reconstruida y la información del input original. En términos psicológicos, podríamos relacionar los procesos de decodificación de esta arquitectura de red neuronal con la recuperación de la información que tenemos almacenada en nuestra memoria. Como hemos comentado antes, no solemos recuperar la información almacenada de manera totalmente fidedigna y, además, a veces cambiamos parte de la información almacenada en el proceso de recuperación.

La idea básica de los autocodificadores (*autoencoders*) reside en que la interacción entre el proceso de codificación y decodificación durante la fase de entrenamiento es crucial para generar representaciones latentes eficientes de baja dimensionalidad de la información del input. A modo de resumen, podemos afirmar que el proceso de codificación resume la información del input para generar una representación más simple, y que el proceso de decodificación pretende reconstruir la información original a partir de esas representaciones simples.

4. Arquitectura y complementos añadidos

La arquitectura básica de un autocodificador consta de dos partes principales: el codificador (encoder) y el decodificador (decoder). Estas dos partes trabajan en conjunto para comprimir y luego reconstruir los datos de entrada, permitiendo que el autocodificador aprenda una representación eficiente de los datos.

- El codificador toma los datos de entrada y los procesa a través de una serie de capas de neuronas. Cada capa reduce la dimensionalidad de la representación, extrayendo características más abstractas y de mayor nivel a medida que avanzamos en la red neuronal. El codificador finaliza con una capa de salida que produce la representación latente comprimida del conjunto de datos original. Esta representación latente es una codificación eficiente de los datos y contiene la información esencial necesaria para reconstruir los datos originales.
- El decodificador, por su parte, toma la representación latente del codificador y la procesa a través de una estructura de capas que busca reconstruir los datos originales. Al igual que el codificador, el decodificador consta de múltiples capas que aumentan gradualmente la dimensionalidad de la representación. La capa de salida del decodificador intenta recrear los datos de entrada originales a partir de la representación latente.

Entre otras ventajas, cabe destacar que los autocodificadores pueden adaptar los componentes de su estructura para procesar la información de la manera más conveniente:

- Por ejemplo, si se quiere procesar imágenes, se pueden utilizar capas convolucionales en el codificador y decodificador. Si se quiere procesar

texto, se pueden emplear capas recurrentes o modelos de atención para capturar la estructura secuencial.

- Además, se pueden añadir capas adicionales al codificador que permitan resolver tareas concretas como, por ejemplo, para clasificar, predecir o generar información a partir del input y/o de las representaciones latentes.

5. La función de pérdida en el contexto de los autocodificadores

En estos modelos de redes neuronales, la función de pérdida es esencial para evaluar la calidad de la reconstrucción de los datos y guiar el proceso de entrenamiento. Esta función de pérdida compara los datos reconstruidos por el decodificador con los datos de entrada originales, calculando la discrepancia o error entre ellos. Esta medida de pérdida permite al autocodificador ajustar sus parámetros para minimizar la diferencia y lograr una reconstrucción lo más precisa posible. Entre otras cosas, esta medida también permite detectar anomalías o patrones atípicos en los datos que se están procesando, ya que, al aprender una representación de los datos normales durante el entrenamiento, el autocodificador podría identificar desviaciones significativas en la reconstrucción cuando se le presentan datos anómalos.

En cualquier escenario, es relevante tener en cuenta que se utilizan diferentes funciones de pérdida según la naturaleza de los datos. Por ejemplo, para datos continuos, se puede utilizar la función de error cuadrático medio (MSE), que compara la diferencia cuadrática entre los valores originales y los valores reconstruidos. Esta función de pérdida penaliza los errores más grandes, lo que puede ser adecuado para

variables psicológicas que tienen una escala continua y una interpretación cuantitativa. En el caso de datos categóricos o discretos, se puede utilizar la función de entropía cruzada, que mide la discrepancia entre la distribución de probabilidad original y la distribución de probabilidad reconstruida. Esta función de pérdida es útil cuando los datos psicológicos se presentan en forma de categorías o clasificaciones, como diagnósticos o etiquetas de clasificación.

En cierto modo, la interpretación de la función de pérdida en términos psicológicos puede ser entendida como una medida de la fidelidad de la reconstrucción. Cuanto menor sea la pérdida, más precisa será la reconstrucción y mayor será la similitud entre los datos originales y los datos reconstruidos. Así, esta información puede utilizarse para comparar diferentes arquitecturas para seleccionar el modelo óptimo (seleccionando el modelo que genera una representación latente más precisa de la información inicial). Dicha comparación de modelos también puede realizar en base al rendimiento en tareas concretas si el modelo de red neuronal tiene complementos (por ejemplo, seleccionando el modelo autocodificador que genere un output que obtenga una mejor clasificación o predicción).

6. Tipos de autocodificadores (entre otros)

Esta sección recoge algunos autocodificadores para mostrar la enorme variedad de tareas que se pueden plantear y modelar, aunque es reseñable afirmar que el campo avanza a gran velocidad y surgen múltiples variaciones y/o combinaciones. Para ello, se

va a presentar un resumen de una publicación de deeplearningofpython.blogspot.com[5] (que también incluye código para su implementación en *keras*). En esta sección, como no existe mucha cultura sobre modelos de redes neuronales autocodificadores en castellano, se ha decidido mantener los nombres originales en inglés para evitar traducciones inexactas o idiosincráticas de este documento.

- ***(Vanilla) Autoencoder***. Este es el modelo autocodificador clásico y estándar que hemos comentado hasta el momento. Ya se ha comentado que la función de pérdida que se utiliza es el error cuadrático medio (MSE) para variables continuas o la función de entropía cruzada para datos categóricos o discretos.
- ***Denoising Autoencoder***. Este tipo de modelo tiene la misma arquitectura que el autocodificador clásico y estándar ya descrito, pero en este caso se utiliza información incompleta o con ruido de input para que el modelo aprenda a eliminar ruido del input procesado.
- ***Convolutional Autoencoder***. Siguiendo con la misma estructura que el autocodificador clásico y estándar, este modelo incorpora capas convolucionales para procesar imágenes e información espacial.
- ***Recurrent Autoencoder***. Es igual que los codificadores convolucionales (*convolutional Autoencoders*), aunque tiene capas de redes neuronales recurrentes como LSTM o GRU como codificadores y decodificadores.
- ***Variational Autoencoder***. Este autocodificador es un modelo generativo que produce información nueva a partir de las representaciones latentes que se han aprendido durante la fase de aprendizaje. Básicamente, la arquitectura de este tipo de autocodificador consiste en un codificador y un decodificador que

---

[5] https://deeplearningofpython.blogspot.com/2023/05/Typesofautoencoders-implementation-keras.html?source=post_page-383cfecc4d0e

transmiten la información del input a la distribución de probabilidad en el espacio latente. Al contrario de otros autocodificadores, este modelo aprende una distribución de probabilidad asociada al espacio latente aprendido con el objetivo de reducir la divergencia entre la distribución aprendida y la distribución a priori del espacio latente, así como del error de reconstrucción entre el input y el output.

- ***Sparse Autoencoder***. Este tipo de modelo tiene la misma arquitectura que el autocodificador clásico y estándar, pero con un término de regularización extra que promueve la activación de unidades ocultas aisladas. Es decir, aparte del aprendizaje normal y estándar del modelo de red neuronal, podríamos decir que se prioriza la activación o detección de unidades que no tienen activación alrededor (mientras que se reduce la probabilidad de activación de unidades que ya tienen unidades adyacentes activas).
- ***Contractive Autoencoder***. Este tipo de modelo tiene la misma arquitectura que el autocodificador clásico y estándar, pero con un término de regularización extra que reduce la sensibilidad a cambios pequeños en el input. Esto se consigue a partir del aprendizaje de una versión comprimida del input para que el modelo aprenda únicamente de las características más llamativas o importantes del input.

7. Ejemplo de aplicación en R

Para llevar a cabo este ejemplo de aplicación de autocofidicador en `R`, vamos a utilizar la API de `keras` y `Rstudio`[6]. Para ello, una vez que estamos en nuestro directorio de trabajo habitual, debemos cargar el paquete `keras`:

```
library(keras)
```

Para este ejemplo concreto, vamos a utilizar la base de datos MNIST (i.e., *MNIST handwritten digit dataset*). Esta famosa base de datos consiste en un total de 70000 números que han sido escritos a mano, tal y como se muestra en la Figura 2. Concretamente, esta base de datos consta de 60000 estímulos clasificados como de entrenamiento, y de 10000 estímulos clasificados como test. Concretamente, en esta base de datos se presenta una variable "y" que indica el número que se ha escrito (e.g., 0, 1, 2… 9), y una matriz de dimensionalidad 28x28 para cada número que está formada por 255 píxeles en cada elemento que determinar la información escrita a mano (e.g., una matriz de 28 filas y 28 columnas definiría la presencia o ausencia de información a través de números de 0 a 255 que indicarían la presencia o ausencia de información). La Figura 3 muestra un ejemplo de dicha matriz para el primer estímulo, resultado de procesar la información en la que se ha escrito manualmente un número 5.

[6] https://www.datatechnotes.com/2020/02/how-to-build-simple-autoencoder-with-keras-in-r.html

Figura 2. Ejemplo de distintos estímulos de la base de datos MNIST (i.e., *MNIST handwritten digit dataset*) disponible en el paquete `keras` de `R`.

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]	[,13]	[,14]	[,15]	[,16]	[,17]	[,18]	[,19]	[,20]	[,21]	[,22]	[,23]	[,24]	[,25]	[,26]	[,27]	[,28]
[1,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[2,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[3,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[4,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[5,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[6,]	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
[7,]	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
[8,]	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
[9,]	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
[10,]	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
[11,]	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[12,]	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
[13,]	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
[14,]	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
[15,]	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
[16,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
[17,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
[18,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
[19,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
[20,]	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
[21,]	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
[22,]	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
[23,]	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
[24,]	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[25,]	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[26,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[27,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[28,]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3. Ejemplo de un estímulo de la base de datos MNIST (i.e., *MNIST handwritten digit dataset*) que indica la escritura del número 5 disponible en el paquete `keras` de `R`.

Como es habitual en modelos de redes neuronales, vamos a generar una base de datos para entrenamiento y otra para testar. Para ello, utilizaremos el siguiente código:

```
c(c(xtrain, ytrain), c(xtest, ytest)) %<-% dataset_mnist()
```

Siguiendo con el preprocesamiento de la información, transformaremos la métrica original que va de 0 a 255 en cada píxel de la imagen a una escala de 0 a 1:

```
xtrain = xtrain/255

xtest = xtest/255
```

Lo siguiente que haríamos es especificar la dimensionalidad de la información que queremos procesar y el tamaño del vector latente que queremos utilizar:

```
input_size = dim(xtrain)[2]*dim(xtrain)[3]

latent_size = 10
```

Así, podemos amoldar también el tamaño de la información del input para procesarlo con nuestro futuro modelo:

```
x_train = array_reshape(xtrain, dim=c(dim(xtrain)[1],
input_size))

x_test = array_reshape(xtest, dim=c(dim(xtest)[1],
input_size))
```

Ahora sí, podríamos comenzar a construir nuestro autocodificador. Como ya hemos comentado, los autocodificadores tienen dos componentes: el codificador y el decodificador. Comenzaremos especificando cómo queremos que sea nuestro codificador (donde la última capa define el vector latente que hemos definido previamente y, por tanto, deberá tener la misma dimensionalidad):

```
enc_input = layer_input(shape = input_size)

enc_output = enc_input %>%
```

```
  layer_dense(units=256, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units=latent_size) %>%
  layer_activation_leaky_relu()
encoder = keras_model(enc_input, enc_output)
```

Como no podría ser de otra forma, el siguiente paso será especificar las características del decodificador (donde la última capa deberá tener la misma dimensionalidad que el input y, en este caso, activación sigmoide):

```
dec_input = layer_input(shape = latent_size)
dec_output = dec_input %>%
  layer_dense(units=256, activation = "relu") %>%
  layer_activation_leaky_relu() %>%
  layer_dense(units = input_size, activation = "sigmoid")
%>%
  layer_activation_leaky_relu()
decoder = keras_model(dec_input, dec_output)
```

En el código presentado, es fácil observar que se está utilizado la función de activación ReLU en las capas ocultas tanto del codificador como del decodificador. Para generar nuestro autocodificador, solamente tenemos que combinar ambos a través de una capa de entrada adicional:

```
aen_input = layer_input(shape = input_size)
aen_output = aen_input %>%
```

```
  encoder() %>%
  decoder()
aen = keras_model(aen_input, aen_output)
```

Si ejecutamos la función `summary()` sobre el objeto que define nuestro modelo, podemos observar cómo el tamaño original del input (que queda transformado en 784 elementos) se reduce a 10 elementos (dimensionalidad del vector latente) y luego se vuelve a transformar a su tamaño original (i.e., 784 elementos).

```
> summary(aen)
Model: "model_2"
____________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #
===========================================================
 input_3 (InputLayer)          [(None, 784)]            0
 model (Functional)           (None, 10)              203530
 model_1 (Functional)         (None, 784)             204304
===========================================================
Total params: 407,834
Trainable params: 407,834
Non-trainable params: 0
___________________________________________________________
```

Una vez definido nuestro modelo, podemos entrenarlo. Para ello, utilizaremos la siguiente sintaxis donde especificamos que queremos entrenar nuestro modelo con la base de datos de entrenamiento (`x_train`) utilizando el optimizador *Rmsprop* (aunque podríamos elegir cualquier otro) y que utilizaremos la función de pérdida de entropía cruzada (`binary_crossentropy`).

```
aen %>% compile(optimizer="rmsprop",
loss="binary_crossentropy")

aen %>% fit(x_train,x_train, epochs=50, batch_size=256)
```

Tras ejecutar la sintaxis previa, comenzaremos el entrenamiento de nuestro autocodificador. Dado que cada ordenador tiene especificaciones distintas y que la calidad de los datos recogidos puede variar respecto a la tarea que se pretende resolver, el modelo nos presenta un resumen del proceso de estimación en tiempo real. El resultado de dicha ejecución se presenta en la Figura 4. Esta figura resume todo el proceso de optimización del modelo a través de las 50 etapas o épocas (*epochs*) que hemos establecido. Cada una de estas etapas o épocas supone un proceso de optimización interno que también nos muestra esta función por defecto. Dicho output de la función se presenta en la Tabla 1, donde se puede ver de manera más pormenorizada el proceso de optimización de cada una de las etapas o épocas (*epochs*).

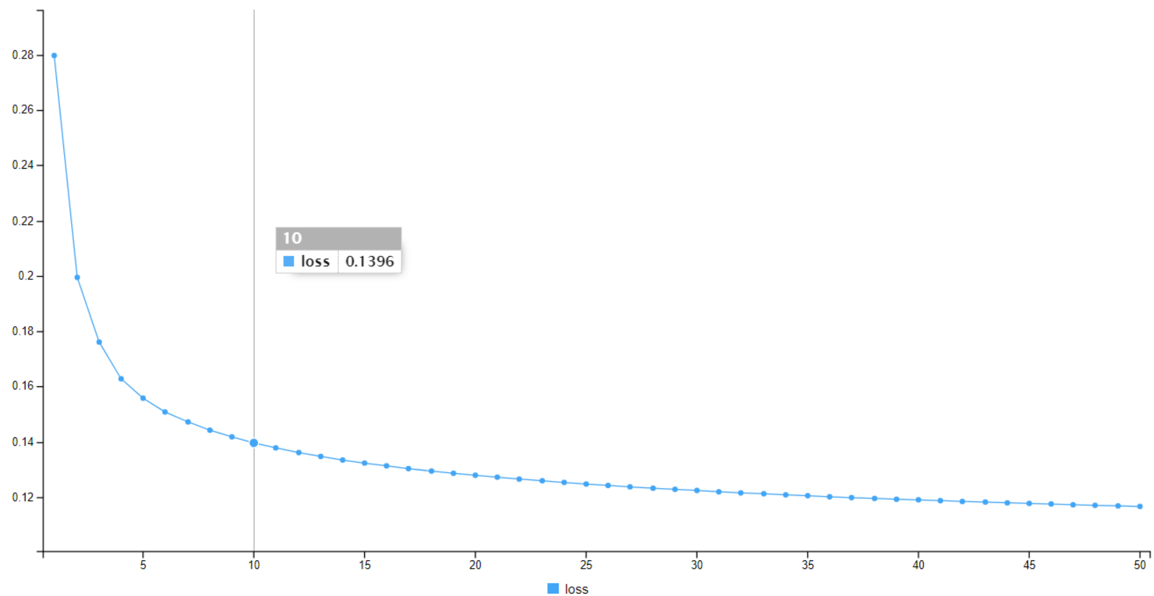


Figura 4. Función de pérdida durante la fase de entrenamiento del autocodificador en el paquete `keras` de `R`. Se resalta la estimación para la iteración número 10.

Tabla 1. Ejemplo de proceso de optimización de las estimaciones del modelo en la etapa o época (*epoch*) 50 (paquete `keras` de `R`).

```
Epoch 50/50
  1/235 [..............................] - ETA: 1s - loss: 0.1225
  7/235 [..............................] - ETA: 2s - loss: 0.1175
 13/235 [>.............................] - ETA: 1s - loss: 0.1177
 20/235 [=>............................] - ETA: 1s - loss: 0.1173
 27/235 [==>...........................] - ETA: 1s - loss: 0.1171
 34/235 [===>..........................] - ETA: 1s - loss: 0.1173
 41/235 [====>.........................] - ETA: 1s - loss: 0.1174
 48/235 [=====>........................] - ETA: 1s - loss: 0.1172
 55/235 [======>.......................] - ETA: 1s - loss: 0.1170
 62/235 [======>.......................] - ETA: 1s - loss: 0.1169
 69/235 [=======>......................] - ETA: 1s - loss: 0.1168
 76/235 [========>.....................] - ETA: 1s - loss: 0.1167
 83/235 [=========>....................] - ETA: 1s - loss: 0.1167
 90/235 [==========>...................] - ETA: 1s - loss: 0.1168
 97/235 [===========>..................] - ETA: 1s - loss: 0.1168
104/235 [============>.................] - ETA: 1s - loss: 0.1169
111/235 [=============>................] - ETA: 1s - loss: 0.1167
118/235 [==============>...............] - ETA: 0s - loss: 0.1167
125/235 [==============>...............] - ETA: 0s - loss: 0.1168
131/235 [===============>..............] - ETA: 0s - loss: 0.1168
137/235 [================>.............] - ETA: 0s - loss: 0.1169
143/235 [=================>............] - ETA: 0s - loss: 0.1169
```

```
150/235 [==================>...........] - ETA: 0s - loss: 0.1170
157/235 [===================>..........] - ETA: 0s - loss: 0.1169
164/235 [===================>..........] - ETA: 0s - loss: 0.1167
171/235 [====================>.........] - ETA: 0s - loss: 0.1167
177/235 [=====================>........] - ETA: 0s - loss: 0.1167
184/235 [======================>.......] - ETA: 0s - loss: 0.1167
191/235 [=======================>......] - ETA: 0s - loss: 0.1166
198/235 [========================>.....] - ETA: 0s - loss: 0.1166
205/235 [=========================>....] - ETA: 0s - loss: 0.1166
212/235 [==========================>...] - ETA: 0s - loss: 0.1166
219/235 [==========================>...] - ETA: 0s - loss: 0.1166
226/235 [===========================>..] - ETA: 0s - loss: 0.1166
233/235 [============================>.] - ETA: 0s - loss: 0.1166
235/235 [==============================] - 2s 9ms/step - loss: 0.1166
```

Una vez entrenado nuestro modelo, podemos comenzar a hacer predicciones con nuestra base de datos reservada para el test o validación (llamada, en este ejemplo, `x_test`). Para ello, podemos codificar la información de la información de input o entrada en nuestro vector latente a través de este código:

```
encoded_imgs = encoder %>% predict(x_test)
```

Que realizará un proceso de codificación similar a este para transformar cada uno de los números recibidos (con 784 elementos cada uno) en un vector latente de solo 10 elementos por estímulo procesado:

```
  1/313 [..............................] - ETA: 6s
 43/313 [===>..........................] - ETA: 0s
 84/313 [=======>......................] - ETA: 0s
127/313 [===========>..................] - ETA: 0s
171/313 [===============>..............] - ETA: 0s
216/313 [===================>..........] - ETA: 0s
261/313 [========================>.....] - ETA: 0s
306/313 [============================>.] - ETA: 0s
313/313 [==============================] - 0s 1ms/step
```

Este proceso de codificación dará lugar a coordenadas de vectores latentes de una dimensionalidad determinada (en nuestro ejemplo, 10 elementos por cada vector) para cada uno de los estímulos incluidos en la función `predict()`, que en nuestro caso son 10000 números escritos manualmente. La Figura 5 presenta un ejemplo de dicho vector latente.

```
> head(encoded_imgs )
          [,1]      [,2]      [,3]      [,4]        [,5]      [,6]      [,7]      [,8]      [,9]     [,10]
[1,] 22.327887 13.596170 15.943604  4.526051 19.96569633  6.191276  9.931662 17.487740  6.725374 19.353439
[2,]  4.470120  3.849552  9.606205 22.646566  6.68575335 12.828539 14.139002 19.192104 15.104708  8.138153
[3,]  7.453702 13.898234 18.946232  4.347703 -0.05187978  2.462875 18.814928 14.992856 12.163232 10.409455
[4,] 24.580345 15.328952  8.947348 26.012701 26.74885750 24.225225 17.612366 16.576389 14.863931 12.076040
[5,] 10.613400 15.744991  4.854032 14.980162 12.53227711 11.064473 11.739936  3.683559 10.447183 25.836311
[6,]  6.372479 14.319201 22.168005  3.534900  3.59927583  2.535952 22.935701 15.709912 10.239327 10.617098
```

Figura 5. Ejemplo de cinco números que han sido codificados en un vector latente de 10 elementos por el autocodificador estimado en el paquete `keras` de `R`.

Como ya sabrá el lector de este documento, el autocodificador también pretende restaurar el estado original del input o entrada a través de la información resumida en este vector latente. Para ello, podremos utilizar la siguiente función:

```
decoded_imgs = decoder %>% predict(encoded_imgs)
```

Que, igualmente, llevará a cabo un proceso de decodificación (para reconstruir el input o entrada original a partir del vector latente de dimensionalidad igual a 10) similar a este par:

```
  1/313 [..............................] - ETA: 7s
 46/313 [===>..........................] - ETA: 0s
 98/313 [========>.....................] - ETA: 0s
148/313 [=============>................] - ETA: 0s
197/313 [=================>............] - ETA: 0s
247/313 [======================>.......] - ETA: 0s
```

```
295/313 [===========================>..] - ETA: 0s
313/313 [==============================] - 0s 1ms/step
```

A modo de ilustración, podemos analizar la recuperación de algunos de los estímulos procesados por nuestro autocodificador. Para ello, podemos utilizar la sintaxis que se presenta a continuación, donde se especifica que queremos recuperar las 28 filas y las 28 columnas del primer elemento que hemos procesado en la base de datos de test con su formato original.

```
pred_images = array_reshape(decoded_imgs,
dim=c(dim(decoded_imgs)[1], 28, 28))
```

La Figura 6 presenta, a modo de ilustración, 15 números escritos manualmente y la recuperación del autocodificador a partir del vector latente de 10 elementos que se ha generado automáticamente. Este tipo de gráfico podría conseguirse a través de la siguiente sintaxis en R base:

```
n = 15

op = par(mfrow=c(15,2), mar=c(1,0,0,0))

for (i in 1:n){

  plot(as.raster(pred_images[i,,]))

  plot(as.raster(xtest[i,,]))

}
```

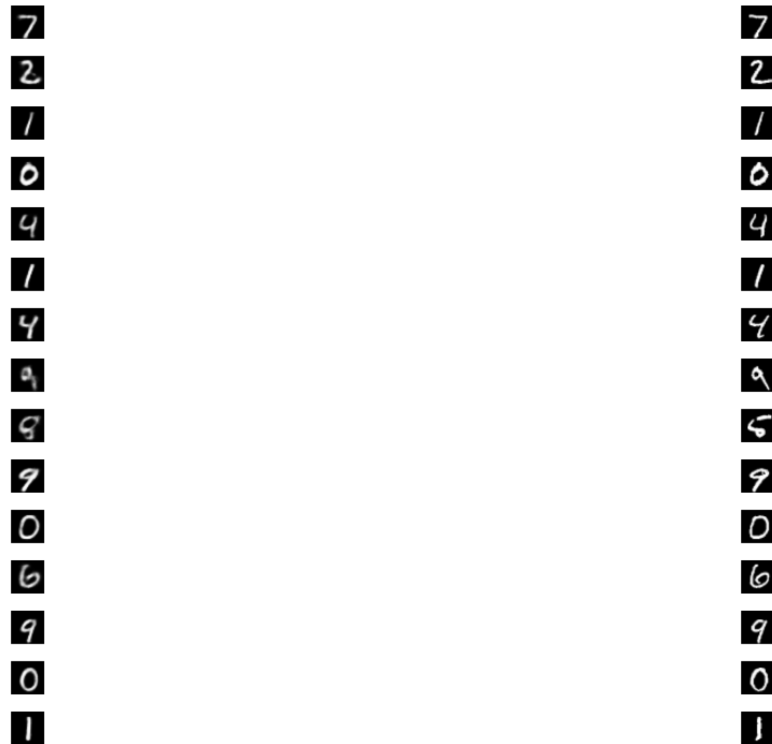


Figura 6. Ejemplo de 15 números escritos manualmente (columna izquierda) y la recuperación del autocodificador a partir del vector latente de 10 elementos que se ha generado automáticamente (columna derecha). Se puede observar el alto grado de precisión, aunque se debe tener claro que la tarea realizada es relativamente sencilla.

Dicho esto, se debe tener en cuenta que en este código se pueden manipular múltiples hiperparámetros (entre los que se incluyen, por ejemplo, el número de épocas o etapas, el tamaño de los lotes o *batch*) y que el paquete `keras` de `R` muchas otras funcionalidades y opciones de configuración. Dada la enorme complejidad de estos modelos (e.g., el número de capas y de parámetros a estimar puede crecer considerablemente en tareas sencillas), siempre se recomienda la generación de hipótesis de trabajo para el modelado de procesos psicológicos con estos modelos.