Capítulo 17
REDES NEURONALES ARTIFICIALES

Este capítulo tiene como objetivo introducir los conceptos básicos de las redes neuronales artificiales y el aprendizaje automático denominado ***deep learning***. Los temas abarcan desde la definición y aplicación del aprendizaje automático hasta la estructura y entrenamiento de las redes neuronales. También se exploran diversas arquitecturas de redes neuronales y sus aplicaciones en diferentes áreas.

17.1 Introducción a las Redes Neuronales Artificiales

Cuando hablamos de la historia de la inteligencia artificial [apartado 1.4], hicimos una breve mención a modelo neuronal creado por Warren McCulloch y Walter Pitts. Vamos a dedicar este capítulo a todo lo que se ha producido desde entonces, que podamos plasmar en un sólo capítulo.

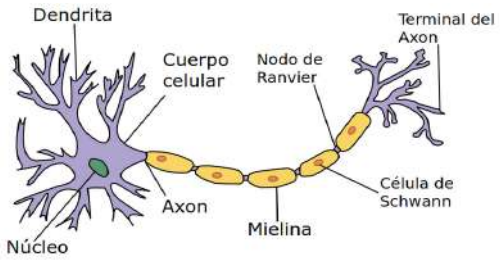


Figura 88: Ilustración de una neurona biológica

Fuente: Wikipedia Commons

Una neurona biológica típica [Figura 88] consta de tres partes principales: [1] El **soma o cuerpo celular**, es el centro de la neurona y contiene el núcleo, que alberga la información genética de la célula; [2] **dendritas** que son extensiones ramificadas que salen del soma, funcionan como antenas electro-químicas, recibiendo señales eléctricas de otras neuronas; y [3] **axón**, es una extensión larga y delgada que transmite señales eléctricas desde el soma a otras células o neuronas y que ter-

mina en múltiples terminales axónicas o botones sinápticos.

El funcionamiento de una de estas neuronas comienza por las dendritas, las cuales reciben señales, generalmente en forma de neurotransmisores liberados en una sinapsis por una neurona vecina. Estas señales pueden ser excitatorias (que tienden a despertar la neurona) o inhibitorias (que tienden a inhibir la actividad neuronal). Si la suma de las señales excitatorias e inhibitorias que una neurona recibe alcanza un cierto umbral, la neurona se *activa* y genera un impulso eléctrico llamado potencial de acción. Este se propaga a lo largo del axón hasta sus terminales, donde provoca la liberación de neurotransmisores en la sinapsis. Estos neurotransmisores luego afectan a otras neuronas o células a las que está conectada la neurona.

Estas neuronas forman tupidas redes con arquitecturas, formas y configuraciones muy diferentes. Los puntos de contacto entre una neurona y otra[256] se denomina **sinapsis**. No es un contacto directo, sino un pequeño espacio llamado hendidura sináptica. Cuando el potencial de acción llega a un botón sináptico, provoca la liberación de neurotransmisores en esta hendidura. Los neurotransmisores luego se unen a receptores específicos en la membrana de la célula receptora, transmitiendo así la señal.

> ***Mi IA favorita*** *lo explica con más sencillez: Las neuronas son como pequeños mensajeros en nuestro cerebro. Tienen ramas llamadas dendritas que "escuchan" mensajes de otras neuronas, y una "cola" llamada axón que envía mensajes. Cuando una neurona recibe suficientes mensajes excitantes (como un "¡sí, actúa!") en comparación con los mensajes inhibidores (como un "espera un momento"), se activa y envía una señal eléctrica a través de su axón. Al final del axón, libera sustancias llamadas neurotransmisores en un espacio (sinapsis) que luego se unen a la siguiente neurona, pasando el mensaje. Así es como nuestras neuronas se comunican y trabajan juntas para que podamos pensar, sentir y actuar.*

Es difícil entender cómo a partir de un funcionamiento tan simple, nuestro cerebro es capaz de hacer tantas cosas tan complejas[257]. Por ello desde siempre el conocimiento humano vio en el cerebro la frontera, la última aventura, el reto. ¿Y que reto mejor que hacer que una máquina emule el funcionamiento del cerebro?

Como dijimos, en 1943, Warren McCulloch y Walter Pitts introdujeron un revolucionario modelo neuronal[258] conocido como *Threshold Logic Unit* (TLU) o *Linear Threshold Unit*. Este pionero diseño conceptual no solo marcó el nacimiento de los modelos neuronales modernos, sino que también ha servido como fuente inagotable de inspiración para generaciones posteriores de teorías y aplicaciones que estas tienen.

En la actualidad, en donde las implementaciones computacionales de neuronas artificiales se realizan con abrumadora frecuencia en sistemas digitales, el modelo McCulloch-Pitts ofrece ventajas

256 *Puede ser otra neurona, un músculo, o una glándula. Aquí presupongo que es otra neurona.*

257 *El todo es superior a la suma de sus poartes.*

258 *Modelo matemático basado en los trabajos de Santiago Ramón y Cajal.*

prácticas innegables. De esta forma, el modelo McCulloch-Pitts sigue siendo no solo una figura histórica de importancia, sino también un recurso contemporáneo de relevancia creciente en el campo de las neurociencias computacionales.

Metáfora biológica de las neuronas

Las redes neuronales artificiales están inspiradas en el funcionamiento del cerebro humano, en el cual las neuronas biológicas son células especializadas en recibir, procesar y transmitir información mediante señales electro-químicas, como indicamos antes. En el contexto de las redes neuronales artificiales, se utiliza un modelo matemático simplificado de las neuronas para simular el procesamiento de información.

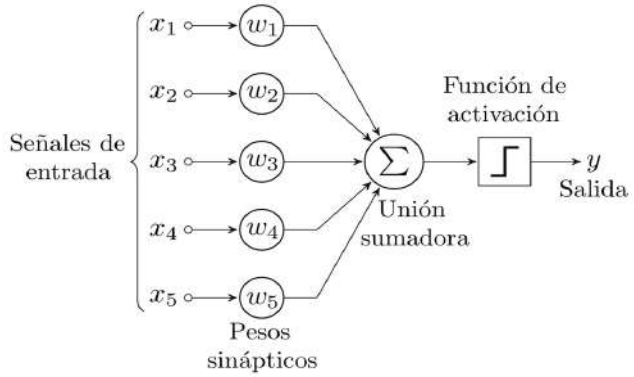


Figura 89: Esquema básico del modelo matemático de una neurona con 5 entradas

Fuente: Wikipedia Commons

Una neurona artificial consta de tres componentes principales: las entradas (x_i), los pesos (w_i) y la función de activación, cuya salida es la salida de la neurona artificial. Las entradas son los datos de entrada que la neurona recibe, cada uno multiplicado por un peso correspondiente. Los pesos representan la importancia relativa de cada entrada en el procesamiento de la información. La función de activación determina si la neurona se activa o no, en función de la suma ponderada de las entradas y los pesos.

$$y = f_{activación}\left(\sum_{i=1}^{n} x_i w_i\right)$$

Calcular la salida (y) de una neurona artificial es sorprendentemente simple: para cada i de 1 a n, siendo n el numero total de entradas, multiplicas x_i con el peso que le corresponde w_i, y los sumas todos. El resultado de esa suma es introducido en una función de activación ($f_{activacion}$), cuya salida será la salida de la neurona artificial. Simple.

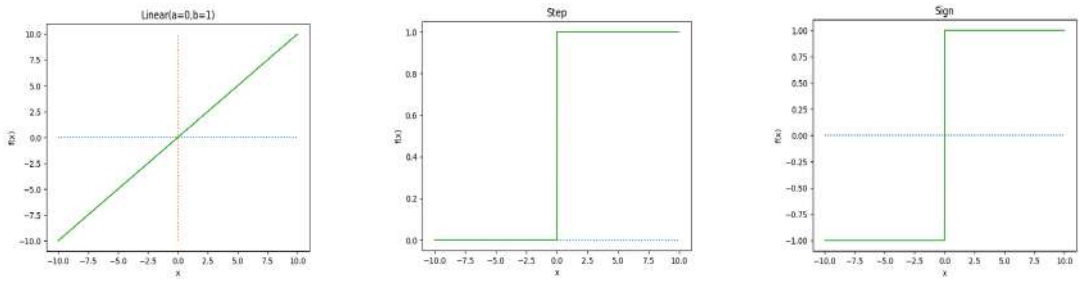
Ahora surgen dos dudas, ¿que es una función de activación? y ¿cómo aprende una neurona artificial? Empecemos por saber y entender que es una función de activación.

El objetivo[259] del modelo de neurona artificial es ser activada o ser inhibida ante determinadas entradas. Esto es, que ante determinadas entradas [x_i] emita un número diferente de cero o un cero [y].

Y esa es la función [original] de la función de activación: a partir de la suma ponderada de las entradas, activarse o desactivarse, esto es, emitir un 0 [inhibida] o un número diferente a 0 [activa].

Vamos a ver unas funciones de activación muy simples para entender mejor esto.

Figura 90: Funciones de activación básicas (linear, step y sign)



La Figura 90 muestra varias funciones de activación[260]. Recordemos que sus x [abscisas] es la suma ponderada de sus entradas, y sus salidas $f_{activación}[x]$ es la salida de la neuronal artificial. Tomando la función *step* como ejemplo, si su entrada es menor o igual que 0, su salida es cero; mientras que si su entrada es mayor que 0 su salida es 1. Si consideramos que 1 es activa y 0 inhibida, tenemos un mecanismo que se activa o no dependiendo de su entrada y que – además – puede aprender.

17.2 El aprendizaje en neuronas artificiales

El proceso de aprendizaje de una neurona artificial está relacionado con la búsqueda de la combinación de pesos [w_i] que hacen que esta emita la salida deseada ante determinadas entradas. En general, todas las arquitecturas de redes neuronales son entrenadas con un algoritmo de aprendizaje, el cual consiste en encontrar la combinación de pesos que, a partir de determinadas entradas se produzcan determinadas salidas.

> ***Nota**: simplificando mucho, el proceso de aprendizaje de una o mas neuronas puede reducirse a un proceso de optimización, en concreto de minimización: buscar el conjunto de pesos (w_i) tal que minimicen el error ($|y-\hat{y}|$) que comete la neurona o red de neuronas ante determinadas entradas (x_i).*

Las primeras propuestas de cómo entrenar una neurona se basaron en **la regla de Hebb** [1949], la

259 *Originalmente las entradas x_i era booleanas, 0 o 1, igual que la salida y. Aquí damos un paso más y hablamos de entradas y salidas numéricas.*

260 *Me adelantaré un poco: en realidad trabajamos con funciones de activación diferentes a estas, pero la mayor parte basadas en ellas.*

cual propugnaba que el aprendizaje ocurre mediante el fortalecimiento de las conexiones entre neuronas que se activan simultáneamente. Este concepto sentó las bases para muchos algoritmos posteriores.

Posteriormente, en 1957, Frank Rosenblatt introdujo el **Perceptrón**, una red neuronal de una sola capa, y su correspondiente regla de aprendizaje. Este algoritmo solo podía resolver problemas linealmente separables[261].

Y, en 1986, Rumelhart, Hinton y Williams introdujeron el **algoritmo de retropropagación** (*backpropagation*), que hizo posible entrenar redes neuronales multicapa y resolver problemas más complejos que los que podían manejar los perceptrones. Para describir este algoritmo necesitamos más de una neurona (artificial).

Desde la década de los 90 se han creado multitud de arquitecturas de redes neuronales artificiales, pero aunque estén potenciadas gracias a técnicas matemáticas[262] y tecnologías de optimización y aceleración, su modelización, la forma de entender matemáticamente como funcionan, sigue siendo el modelo de McCulloch-Pitts.

Vamos a condensar cómo funciona el algoritmo de retropropagación.

El algoritmo de retropropagación es el método utilizado para entrenar redes neuronales mediante la actualización iterativa de sus pesos. Es un algoritmo basado en la regla de la cadena del cálculo diferencial. Recordemos, igual que siempre partimos de un conjunto de entrenamiento, *trainset*, formado por muestras:

1. **Inicialización**: Antes de que comience el proceso de entrenamiento, los pesos de la red neuronal se inicializan con valores aleatorios.
2. **Feedforward**: Se introduce la siguiente muestra del conjunto de entrenamiento en la red, y se propaga a través de sus capas hasta obtener una salida. Esta salida se compara con la salida deseada (etiqueta) para calcular el error.
3. **Cálculo del error**: El error se calcula generalmente usando una función de pérdida (*loss*), como MAE[263] o MSE[264]. Esta función de coste cuantifica la diferencia entre la salida prevista por la red y la salida real o deseada.
4. **Retropropagación del error**: A partir de la capa de salida, se calcula la derivada de la función de pérdida con respecto a cada peso, es decir, el gradiente de la función de pérdida. Esto se hace retropropagando el error a través de cada capa, desde la última hasta la primera, utilizando la regla de la cadena para calcular las derivadas parciales. Esta retropropagación del error da al algoritmo su nombre.

   Este punto es el difícil de entender si no se domina el cálculo diferencial. La idea principal

261 *Es fácil de entender si volvemos al tema de los clasificadores binarios: un problema linealmente separable es aquel en el que puedes poner una línea recta entre dos clases. ¿Recuerdas los hiperplanos de SVM?*

262 *Principalmente, cálculo, álgebra lineal y estadística/probabilidad.*

263 *Recordemos: Median absolute error, error absoluto medio.*

264 *Recordemos: Median squared error, error cuadrático medio.*

es ir desde la salida y hacia la entrada, capa por capa, [por eso se llama retropropagación] modificando los pesos de forma que minimicen ese error [*loss*] si vuelven a ver la misma muestra en la entrada.

5. **Actualización de pesos**: Una vez que se tiene el gradiente, se utilizan algoritmos de optimización, como el descenso del gradiente, para ajustar los pesos en la dirección que reduce el error. La magnitud del ajuste está determinada por la tasa de aprendizaje, un hiperparámetro importante en el entrenamiento de redes neuronales.
6. Y volvemos al punto 2 mientras haya muestras en el conjunto de entrenamiento.

El proceso se repite para cada muestra o ejemplo en el conjunto de entrenamiento, a menudo múltiples veces. Cada paso completo a través del conjunto de entrenamiento se denomina ***epoch***.

Haciendo una analogía, imagina que estás enseñando a una niña los tiros libres de baloncesto. Al principio, esta no tiene idea de cuánta fuerza usar o en qué ángulo lanzar. Cada vez que lanza la pelota, tú observas dónde cae: si fue demasiado corto, demasiado largo, si fue hacia la izquierda o hacia la derecha del aro. El proceso de enseñar a ajustar su tiro es similar al algoritmo de retropropagación:

1. Inicialización: Comienza a lanzar la pelota sin ninguna técnica. Sus movimientos iniciales son aleatorios.
2. *Feedforward*: Ella lanza la pelota y tú observas dónde cae.
3. Cálculo del error: Comparas dónde querías que cayera la pelota [en el aro] y dónde realmente cayó. Si cae corto, sabes que necesita más fuerza. Si va hacia la derecha, necesita ajustar su puntería a la izquierda, etc.
4. Retropropagación del error: Le das retroalimentación sobre lo que hizo mal y cómo puede corregirlo. Por ejemplo, "*lanza con un poco más de fuerza*" o "*apunta más a la izquierda*", etc.

   Este es el punto en que debemos pararnos para explicar lo que es retropropagar el error siguiendo la regla de la cadena usando derivadas parciales, pero aplicado a la analogía. En el lanzamiento de un tiro libre influyen varias "capas", los dedos, la muñeca, el codo, hombros, la posición del resto del cuerpo [especialmente rodillas], etc. Para corregir un tiro no llega sólo con corregir la mano, hay que propagar ese error hacia atrás en la cadena, asumiendo cada "capa" parte del error del resultado anterior, para conseguir que el siguiente tiro consiga un error menor. Esto se hace entre este punto y el siguiente.
5. Actualización de pesos: En el siguiente intento, ella ajusta su tiro[265] basándose en tu retroalimentación. Esto es el equivalente a actualizar los pesos en la red neuronal.
6. Iteración: La niña sigue practicando [vuelve al punto 2] y ajustando su tiro con cada lanzamiento usando tus consejos hasta que se convierte en un experto/a en tiros libres.

Esta analogía es mala para explicar lo que es un *epoch*. Pero, siendo muy aventurados, un *epoch* en este ejemplo sería un día completo de entrenamiento, suponiendo que se va a entrenar día tras días hasta conseguir la perfección [o casi].

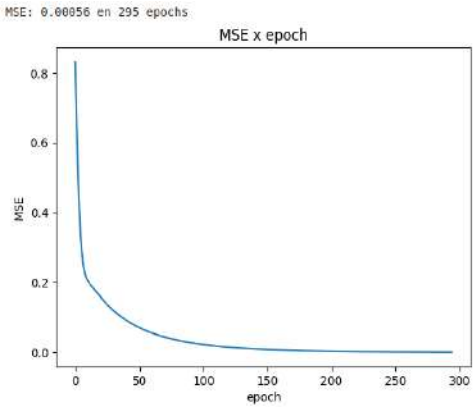
265 *Aquí, ajustar el tiro no es volver a tirar. Eso lo hace en el punto 2 de nuevo. Ajustar el tiro sería cuando un jugador, antes de tirar, ensaya el tiro sin balón.*

Pongamos un ejemplo más cercano a la técnica. Vamos a enseñar a un perceptrón a contar.

```
trainset=[
          ((0,0,0), 0),
          ((0,0,1), 1),
          ((0,1,0), 1),
          ((0,1,1), 2),
          ((1,0,0), 1),
          ((1,0,1), 2),
          ((1,1,0), 2),
          ((1,1,1), 3)
         ];
```

Este es el conjunto de entrenamiento. Por simplicidad en este ejemplo no va a haber conjunto de testeo. El *trainset* está compuesto[266] por una entrada de tres bits [0 o 1] y una salida, indicando cuantos 1s hay en la entrada.

Hemos entrenado un perceptrón de tres entradas, con una función de activación ReLu con a=0,001, un *learning rate*[267] de 0,01 y lo hemos entrenado durante 1000 *epochs*. Como función de coste usé MSE [*median squared error*].



Finalmente el aprendizaje finalizó en el *epoch* 295, cuando el error disminuyó tanto [0,00056] que no merecía la pena seguir entrenándolo[268].

Como resultado podemos ver esta gráfica que muestra en el eje de abscisas los *epochs* y en el eje de coordenadas el error cometido en ese *epoch* [MSE].

Esta gráfica se denomina **curva de aprendizaje**, y es la guía de lo que está pasando[269] o pasó con el entrenamiento de una red neuronal. En nuestro caso, empezamos muy mal, con un gran error, y paulatinamente esta fue disminuyendo, hasta alcanzar un plano pasados los 250 *epochs*. Cuando acabó, el error cometido era muy muy pequeño.

> ***Un apunte**: estamos entrenando y evaluando con el mismo conjunto de datos, lo más probable es que este modelo haya sobreentrenado (overfitting), pero en este caso no nos importa.*

266 *Es una estructura ligeramente compleja de python: una lista de tuplas, cada una de ellas siendo una tupla de 2 elementos, el primero una tupla de tres escalares y el segundo un escalar.*

267 *Velocidad con la que aprende, relacionado con el algoritmo del descenso del gradiente. Ver §Error: no se encontró el origen de la referencia.*

268 *La finalización es automática, al igual que el entrenamiento.*

269 *Si el entrenamiento dura días o meses, ir viendo esta curva (con más elementos) ayuda a saber cómo está yendo el aprendizaje.*

Vamos a probarlo:

```
Introduce la entrada (a b c): 1 1 1
3.0
Introduce la entrada (a b c): 1 0 1
2.0
Introduce la entrada (a b c): 0 0 0
0.0
```

Funciona. Si introduzco 1 1 1, devuelve 3; y así el resto. Hemos creado un perceptrón que es capaz de contar el número de 1s que hay en la entrada. ¿Y que pasa si en vez de 0 y 1, introducimos números mayores?

```
Introduce la entrada (a b c): 1 2 3
6.0
Introduce la entrada (a b c): 2 3 4
9.0
Introduce la entrada (a b c): 1 3 4
8.0
```

¿Qué ha pasado aquí?

¡Le hemos enseñado a contar 1s y este ha aprendido, sin ejemplos, a sumar números naturales! Como veremos las redes neuronales artificiales no sólo aprenden bien, también generalizan muy bien[270].

Si las redes neuronales artificiales funcionan tan bien, ¿por qué tardaron tanto en salir de los laboratorios? Hay dos razones, la primera porque para entrenar una red neuronal hacen falta muchos datos, de muy buena calidad y muchísima potencia de computación. Elementos que no estuvieron a nuestro alcance hasta inicios del siglo XXI.

Para la segunda razón tengo que explayarme más. Marvin Minsky fue un prominente científico, como vimos cuando explicamos la historia de la IA [Figura 2]. En la década de 1960 hubo mucho entusiasmo en torno al perceptrón y, a medida que se publicaban más investigaciones sobre ellos, se generó mucho optimismo sobre sus potenciales capacidades. Como acabamos de ver.

Sin embargo, Minsky y su colega Seymour Papert identificaron y destacaron las limitaciones de los perceptrones en su libro "*Perceptrons*" publicado en 1969. En particular, demostraron que los perceptrones simples no podían realizar ciertas tareas fundamentales, como la función XOR [*exclusive or*, "o exclusivo"[271]]. Aunque esta limitación[272] ya era conocida en ciertos círculos de investigación, la forma en que Minsky y Papert presentaron sus hallazgos, junto con su prominencia en el campo, **tuvo un impacto muy significativo.**

El libro, lamentablemente, tuvo un efecto no deseado en la comunidad de investigación de inteli-

270 *Seamos honestos, según él 9+9+9 es 26. Tiene sentido, cuanto más nos alejemos del trainset más error va a cometer,.*

271 *Or exclusivo: sólo es cierto si "1 xor 0" o "0 xor 1", el resto de casos es falso.*

272 *Esta limitación no es más que la imposibilidad de un perceptrón de establecer un límite entre dos clases que no pueden ser separables mediante una línea.*

gencia artificial. Fue interpretado por muchos (a veces de manera incorrecta) como una crítica a todas las formas de redes neuronales, no sólo a los perceptrones simples. Esto llevó a un descenso en la financiación y la investigación en este área durante una buena parte de la década de 1970 y 1980. ¿Te suena el "invierno de la IA"?

A pesar de esto, la investigación en redes neuronales no se detuvo completamente. Con el tiempo, se descubrió que al añadir más capas a las redes neuronales (**perceptrones multicapa**) y mediante el uso de técnicas como el algoritmo de retropropagación que acabamos de ver, se podían superar las limitaciones señaladas por Minsky y Papert. Esta actualización llevó a un renacimiento de las redes neuronales, principalmente en los años 90, que finalmente culminó en la explosión del aprendizaje profundo desde principios del siglo XXI.

17.3 Estructura básica de una red neuronal

Con una sólo neurona en realidad no podemos hacer mucho más. Para alcanzar su potencial, las neuronas artificiales deben ser **agrupadas en capas** (*layers*). Cada capa consta de k neuronas, cada una de las cuales recibe las mismas entradas (x_1, x_2, .., x_n). Fíjate en la Figura 91 en la capa oculta (*hidden*) de la red de la derecha (*feedforward neuronal network*).

Una **red neuronal** está compuesta por neuronas interconectadas y organizadas en capas. Aunque hay varios tipos, nos centraremos en las redes *feedforward*, aquellas en las que no hay ciclos[273]. La estructura básica de una red neuronal consta de tres tipos de capas (ver Figura 91)[274]:

- **Capa de entrada** (*input layer*): Es la capa inicial de la red neuronal, encargada de recibir, y quizá adaptar, los datos de entrada. Cada neurona de esta capa está conectada a cada neurona de la siguiente capa mediante conexiones ponderadas (con los pesos).
- **Capas ocultas** (*hidden layers*): Son capas intermedias entre la capa de entrada y la capa de salida. Estas capas realizan el procesamiento de la información a medida que se propaga a través de la red neuronal. Pueden existir una o varias capas ocultas, dependiendo de la complejidad del problema que se pretende resolver.
- **Capa de salida** (*output layer*): Es la capa final de la red neuronal y proporciona los resultados o predicciones de la red. Cada neurona de esta capa representa una clase o una variable de salida y se activa en función de la información procesada en las capas anteriores.

Antes de este apartado parecía que la unidad básica de una red neuronal era la neurona artificial, pero no. Al escalar el tamaño de estas redes la unidad que manejamos es la capa. Por simplicidad. Cuando creamos una red neuronal, por ejemplo *feedforward*, lo que realmente hacemos es apilar capas, posiblemente de diferentes tipos. El tipo de capas que estamos viendo se denominan densas (*fully dense*).

273 *Que no haya ciclos significa que la información fluye de un lado a otro, sin volver atrás en ningún momento. De no cumplirse esta condición hablaríamos de redes recurrentes.*

274 *Los sufijos "feedforward" y "deep" no son excluyentes. El primero nos indica que no hay ciclos y el segundo que hay más de una capa oculta. Así es posible decir "deep feedforward neural network". Frecuentemente omitimos ambos sufijos según el contexto.*

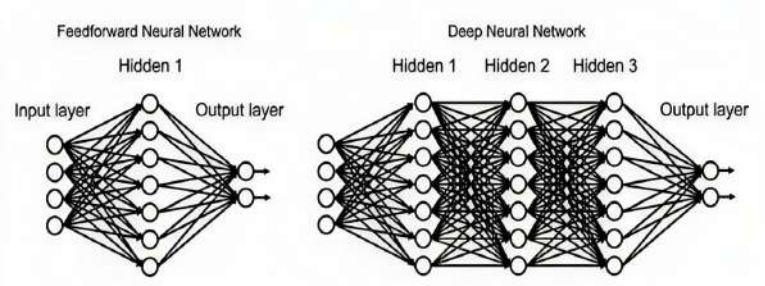


Figura 91: Esquema de redes neuronales artificiales multicapa densas (fully connected)

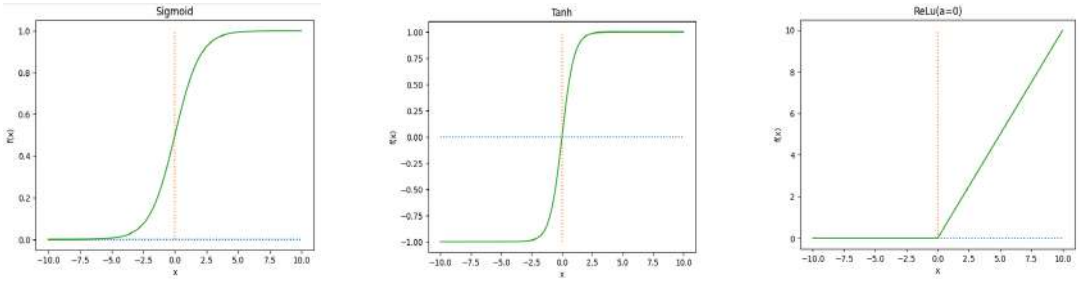
Fuente: "Artificial intelligence, machine learning, and deep learning for clinical outcome prediction". Rowland et al,.2021. Emerging Topics in Life Sciences. 5. 10.1042/ETLS20210246.

Ya tenemos todos los aspectos básicos de una red neuronal artificial, pero debemos volver un momento a las funciones de activación.

Funciones de activación revisadas

Como ya hemos dicho, las funciones de activación juegan un papel fundamental en el procesamiento de la información en una red neuronal[275]. Estas funciones se aplican a la suma ponderada de las entradas y los pesos de una neurona, determinando si se activa o no. Ya hemos nombrado algunas básicas, que apenas se usan, pero hay otras[276].

Figura 92: Funciones de activación modernas



Las funciones de activación que se muestran en la Figura 92, son las versiones continuas de las mostradas anteriormente [*sigmoid* ≈ *step*, *tanh* ≈ *sign* y *relu* es una versión parametrizada[277] de *linear*]. Quizá asuste ver tanta función de activación y no comprender bien que función realiza dentro de una neurona artificial, pero debes saber que la mayor parte de las veces elegir una u otra es cuestión de probar[278], al menos en capas ocultas. Por ejemplo, es habitual ver una *sigmoid* en la

275 *Son las que aportan la no linealidad. Las funciones de activación no lineales permiten que los modelos capten relaciones y patrones complejos en los datos que no serían posibles con solo operaciones lineales.*

276 *Incluso más de las que muestro aquí, pero suelen ser variantes (Softmax, MaxOut, Leaky ReLu, PReLu, ELU, etc.). El arquitecto/a de redes neuronales incluso puede crear la suyas propias.*

277 *La imagen muestra ReLu con a=0, si a fuese 1 entonces sería igual a "linear".*

278 *De hecho se va probando con funciones de activación y comparando los modelos. Con experiencia se suele elegir más eficazmente una función de activación para cada capa oculta.*

capa de salida si lo que quieres es obtener una o más probabilidades, ya que esta función de activación emite en el rango [0, 1].

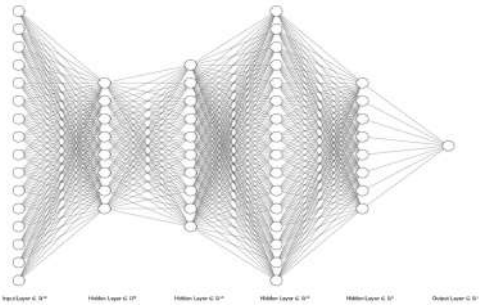
17.4 Arquitecturas de Redes Neuronales

Ya hemos visto las unidades de que están formadas las redes neuronales artificiales, sus unidades individuales, como se agrupan en una capa y como las capas se agrupan para formar redes. Ahora vamos a ver las arquitecturas que estas redes pueden formar:

El **perceptrón** es el modelo más básico de red neuronal. Consiste en una única capa de neuronas que hace las veces de entrada y salida. Cada neurona en esta capa comparte una serie de entradas y genera una salida utilizando una función de activación.

Las **redes neuronales multicapa**, también conocidas como perceptrones multicapa (MLP, por sus siglas en inglés) o *deep feedforward neural network*, son una extensión del perceptrón básico. Estas redes constan de una capa de entrada, una o más capas ocultas y una capa de salida. Cada neurona en una capa está conectada a todas las neuronas de la capa anterior y de la capa siguiente, formando una estructura en forma de grafo sin ciclos (*feedforward*). Las capas ocultas permiten que la red neuronal aprenda características y patrones más complejos en los datos.

Tenemos un ejemplo simple de aplicación de MLP si usamos el *dataset cancer_breast*. Un MLP analiza esta información para determinar la naturaleza del tumor: maligno (1) o benigno (0). El MLP opera de la siguiente manera: la capa de entrada recibe las 30 características numéricas, las capas ocultas (4 de tipo *dense*) procesan y identifican patrones clave en esos números, y la capa de salida genera una predicción, indicando si el tumor es probablemente maligno o benigno. A través del entrenamiento con ejemplos ya etiquetados, la red mejora su capacidad predictiva.



Por ejemplo, a la izquierda podemos ver la estructura del modelo usado, y en la Figura 93 podemos las curvas de precisión y aprendizaje de un MLP aplicada al *dataset cancer_breast*, y como finalmente la precisión (usando el *testset*[279]) fue del 97,37%. Como referencia, el mejor modelo[280] del Capítulo 16, arrojó una precisión de 95,89%.

279 *La gráfica representa un poco menos porque muestra la precisión respecto al conjunto de validación.*

280 *Fue un XGBoost, con los hiperparámetros optimizados.*

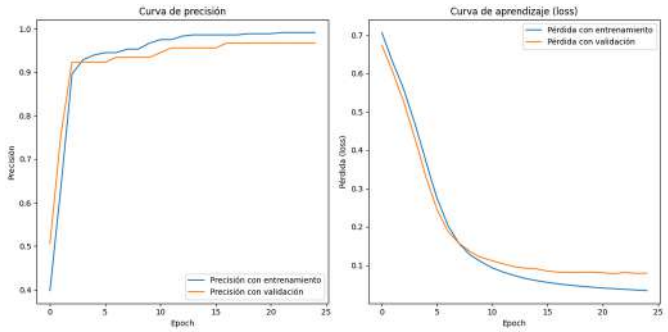


Figura 93: Curvas de aprendizaje para un MLP sobre el dataset "cancer_breast"

Redes neuronales convolucionales

Las redes neuronales convolucionales[281] (CNN, por sus siglas en inglés) son arquitecturas especializadas en el procesamiento de datos bidimensionales, especialmente imágenes. Las CNN utilizan filtros convolucionales para extraer características locales en la imagen y luego combinan estas características en capas posteriores para obtener una representación más global. Las redes convolucionales son ampliamente utilizadas en tareas de reconocimiento visual, detección de objetos y segmentación de imágenes.

> ***Mi IA favorita dice**: Imagina que tienes una gran foto y un pequeño marco o ventana transparente [...]. Este pequeño marco es lo que llamamos "filtro". Al deslizar o mover esta ventana por toda la foto y mirar a través de ella, puedes cambiar o resaltar ciertas partes de la imagen. Así, si quieres encontrar todos los bordes o ciertas texturas en la foto, usarías un tipo específico de ventana (filtro) para hacerlo. Este proceso de mover la ventana y obtener una nueva imagen que muestra lo que estás buscando se llama "convolución", y es la idea detrás de los filtros convolucionales en las imágenes. ¡Es como un juego de buscar detalles con una lupa!*

Un ejemplo de aplicación de redes convolucionales es la clasificación de imágenes. La red neuronal convolucional tomaría una imagen como entrada y aplicaría una serie de filtros convolucionales para detectar características como bordes, texturas y formas en la imagen. Estas características se combinan en capas posteriores para realizar la clasificación final de la imagen en una de las clases predefinidas, como perro, gato, pájaro, etc.

Vamos a entrenar un clasificador multiclase con una red neuronal convolucional para clasificar el *dataset MNIST digits* y comparar los resultados con los del apartado 16.6, el código es el que sigue:

281 *Las hemos usado con profusión en la sección primera: "Visión artificial".*

1. Lleva a cabo las importaciones necesarias en el resto del código.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.keras.datasets import mnist, fashion_mnist
from tensorflow.keras.models   import Sequential
from tensorflow.keras.layers   import Dense, Conv2D, MaxPooling2D, Flatten
```

2. Lee el *dataset* que usarás para entrenar y evaluar el modelo. Si fuese necesario haz la limpieza y las adaptaciones adecuadas de los datos. Posteriormente[282] divide el *dataset* en *trainset* y *testset*.

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalizando258 las imágenes para que los valores estén en el rango [0, 1]
train_images = train_images.reshape((60000, 22, 22, 1)).astype('float32') / 255
test_images  = test_images.reshape((10000, 22, 22, 1)).astype('float32') / 255

# Convertir los labels258 a one-hot encoding
train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels  = tf.keras.utils.to_categorical(test_labels)
```

3. Declara la estructura de la red neuronal: Creamos un modelo secuencias [que apila capas] y vamos añadiendo capa tras capa. Como podemos ver en este tipo de redes hay varios tipos diferentes de capas. Las *Conv2D*, *MaxPooling2D* y *Flatten* son típicas de las CNN.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(66, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

   También podemos ver que usamos la función de activación ReLu, y otra llamada *softmax*. Esta última convierte la salida de la red neuronal en un conjunto de probabilidades que juntas suman 1. Sin esta función de activación en la capa de salida, esta CNN emitiría 10 números, siendo el más grande el de la clase con más peso, y así el resto.

4. Compila en modelo creado. Es durante este proceso cuando se crea el modelo en realidad, iniciando sus pesos con valores aleatorios. Al tiempo elegimos el optimizador[283], la función de pérdida[284] y las métricas que usaremos para medir qué tal va el aprendizaje.

---

282 *Este código se sale de la norma. Ya nos entregan el dataset dividido, así que no lo hacemos "posteriormente", lo hacemos "anteriormente".*

283 *El optimizador es el encargado de "optimizar" los pesos para minimizar el error. Es el duende detrás de la magia de este tipo de aprendizaje. Por cierto, Adam significa: Adaptive Moment Estimation.*

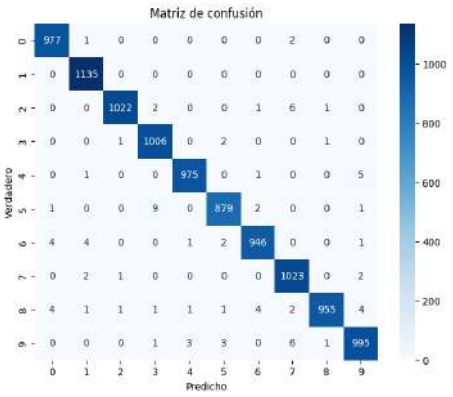
284 *De acuerdo, "categorical_crossentropy" no es un nombre atractivo ni es fácil de entender. Básicamente se calcula tomando el valor negativo del logaritmo de la probabilidad predicha para la clase verdadera. Quédate con que es adecuado para clasificación multiclase, como es el caso.*

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

5. Finalmente entrena el modelo: hacemos un *fit* con los datos de entrenamiento (*train_images* y *train_labels*, ambos forman el *trainset*), indicamos que queremos 10 *epochs* y cual va a ser el conjunto que usaremos para validar el aprendizaje (el *testset*).

```
history = model.fit(train_images,
                    train_labels,
                    epochs=10,
                    batch_size=64,
                    validation_data=(test_images, test_labels))
```

¿Y cual es el resultado? Tras 10 *epochs* tenemos una precisión media del 99.13%.



	precision	recall	f1-score
0	0.9909	0.9969	0.9939
1	0.9921	1.0000	0.9961
2	0.9971	0.9903	0.9937
3	0.9872	0.9960	0.9916
4	0.9949	0.9929	0.9939
5	0.9910	0.9854	0.9882
6	0.9916	0.9875	0.9895
7	0.9846	0.9951	0.9898
8	0.9969	0.9805	0.9886
9	0.9871	0.9861	0.9866
accuracy			0.9913
macro avg	0.9913	0.9911	0.9912
weighted avg	0.9913	0.9913	0.9913

Figura 94: Matriz de confusión multiclase 10x10 y métricas con CNN

En el Capítulo 16 conseguimos un 98% (ver la Figura 79). En ese mismo capítulo mostramos como el *dataset* (su calidad y cantidad) era determinante para obtener un buen modelo. Habíamos sometido al clasificador multiclase ganador a un *dataset* idéntico al *MNIST digits*, pero de menos calidad: el *MNIST fashion*. El resultado fue de una precisión media de 89.83%. Usando una arquitectura CNN obtuvimos 91,80% como máximo y en 6 *epochs*.

Redes neuronales recurrentes

Las redes neuronales recurrentes (RNN, por sus siglas en inglés) son arquitecturas diseñadas para modelar secuencias de datos, como frases (palabras, *tokens*) o series temporales. A diferencia de las redes neuronales tradicionales, las RNN tienen conexiones recurrentes, lo que significa que la información puede retroalimentarse en la red. No son por tanto *feedforward*. Esta propiedad permite que las RNN tengan memoria y capturen dependencias a largo plazo en los datos secuenciales.

Un ejemplo de aplicación de redes recurrentes es la generación de texto. Una RNN puede entrenarse con un conjunto de textos y luego generar nuevos textos basados en el patrón de los datos de

entrenamiento. Por ejemplo, se puede entrenar una RNN en textos de *Shakespeare* y luego generar nuevos textos que imiten el estilo y la estructura del autor.

> ***Nota**: Lo que sigue no son arquitecturas propiamente dichas, ya que se basa en una clasificación que se refiere a cómo se utiliza el modelo y cuál es su objetivo principal, no a la arquitectura subyacente específica, que puede ser suma de cualquiera de las anteriores.*

Redes neuronales generativas

Las redes generativas, como las **Redes Generativas Antagónicas** (GANs) o los **Autoencoders Variacionales** (VAEs), son arquitecturas diseñadas para modelar y generar datos que imitan la distribución de un conjunto de datos de entrenamiento. A través del aprendizaje no supervisado o semisupervisado, estas redes capturan características intrínsecas y patrones subyacentes en los datos, permitiendo la creación de nuevas instancias que son estadísticamente similares a los datos originales.

Por ejemplo, una GAN podría ser entrenada en un conjunto de datos de imágenes de rostros humanos y luego ser capaz de generar imágenes de caras que, aunque no son réplicas de las caras en el conjunto de entrenamiento, comparten las mismas características y variabilidades. Siendo más generalista, imagina que tu conjunto de datos es un conjunto de puntos en un espacio bidimensional (fotos de caras); una red generativa aprendería la *forma* general de esa nube de puntos y podría generar nuevos puntos que encajen de forma coherente dentro de esa forma aprendida (esto es, nuevas caras, ver Figura 45).

> ***Mi IA favorita** dice: GAN (Generative Adversarial Networks): Compuestas por dos redes, una generativa y otra discriminadora, que trabajan en conjunto para generar datos que sean indistinguibles de los datos reales.*

Redes autoencoder (o encoder/decoder)

Las redes autoencoder son diseñadas para aprender una representación compacta y eficiente de los datos de entrada, generalmente para tareas como la reducción de dimensionalidad o el preentrenamiento de una red neuronal. Estas redes constan de dos partes principales: el codificador, que comprime la entrada a un espacio latente más pequeño, y el decodificador, que reconstruye la entrada a partir de esta representación compacta. El objetivo del entrenamiento es minimizar la diferencia entre la entrada original y su reconstrucción, lo que lleva a que la red aprenda las características más importantes de los datos.

Por ejemplo, volvamos con el *dataset* MNIST *digits*. Queremos reducir la dimensionalidad de cada imagen manteniendo la mayor cantidad de información posible. Podríamos emplear un *autoencoder* que tome las imágenes como entrada, las comprima en una representación latente de menor dimensionalidad mediante el codificador, y luego las reconstruya mediante el decodificador. Al entrenar el *autoencoder* para minimizar la diferencia entre las imágenes originales y las reconstruidas, podemos extraer una representación compacta de cada imagen que capture sus características más importantes. Esta representación puede utilizarse posteriormente para tareas como la clasifi-

cación de dígitos, la detección de anomalías o incluso la generación de nuevos dígitos similares a los del conjunto de entrenamiento.

Figura 95: Ejemplo de predicción de CLIP

Fuente: OpenAI - https://openai.com/research/clip

Redes basadas en el concepto de atención

Las redes basadas en el concepto de atención [§12.1], particularmente los ***Transformers***, han revolucionado el campo del aprendizaje profundo y son especialmente poderosas en tareas de procesamiento del lenguaje natural, aunque no de forma exclusiva. Estas arquitecturas eliminan la necesidad de recurrencia [redes recurrentes] y se centran en mecanismos de atención para capturar las dependencias en los datos. A través de la atención ponderada, el modelo puede centrarse en diferentes partes de la entrada para realizar tareas como traducción, generación de texto, y clasificación de manera más efectiva y eficiente.

Pongamos como ejemplo el modelo BERT [*Bidirectional Encoder Representations from Transformers*] es un *Transformer* preentrenado que se ha utilizado con éxito en una variedad de tareas de procesamiento del lenguaje natural, como clasificación de texto, generación de resúmenes y respuesta a preguntas. BERT se entrena para predecir palabras faltantes en una oración, capturando así el contexto bidireccional que rodea a cada palabra.

En resumen, el mecanismo de atención [centrándonos en *transformers*] es parecido a un sistema para decidir qué palabras en un conjunto de datos son especialmente relevantes para las palabras en otro conjunto de datos, y luego usar esa información para generar una salida. Es una forma de permitir que el modelo se "enfoque" en la información más importante en un momento dado.

Estas son las arquitecturas o categorías más usadas o que más impacto ha tenido en el campo que nos ocupa. Pero no es lo único, más allá de las arquitecturas, se están llevando a cabo diseños mucho más complejos que combinan partes de las aquí expuestas. Llamémoslas **meta arquitecturas.**

Por ejemplo *DALL·E 3*, *Stable Diffusion*, *FLUX* o *Midjourney*, que a partir de un texto crean una imagen que cuadra con este; o [Open]CLIP [Figura 95, y ver más adelante] que a partir de una imagen proporciona las probabilidades para un conjunto de textos descriptivos dados.

17.5 Ejemplos de redes neuronales artificiales

A lo largo de este capítulo, y en capítulos anteriores, hemos hecho uso de redes neuronales artificiales, mostrando sólo sus resultados. Ahora vamos a ver cómo funcionan desde el punto de vista del código y comparar sus resultados con modelos anteriores; si es posible, dado que algunas de las arquitecturas que veremos es muy difícil entrenarlas en los ordenadores que habitualmente tenemos a nuestro alcance, incluida la infraestructura de Google Colab.

17.5.1 Autoencoders

Observa el MLP de la Figura 96, en él podemos observar que [1] la entrada tiene el mismo tamaño que la salida [16], y que las tres capas ocultas son del mismo tamaño. Imagina ahora que la entrenamos con un *dataset* de imágenes, a la entrada[285] le presentamos una imagen y a la salida esperamos que se muestre la misma imagen; el aprendizaje es pues que muestre a la salida lo mismo que le presentamos a la entrada.

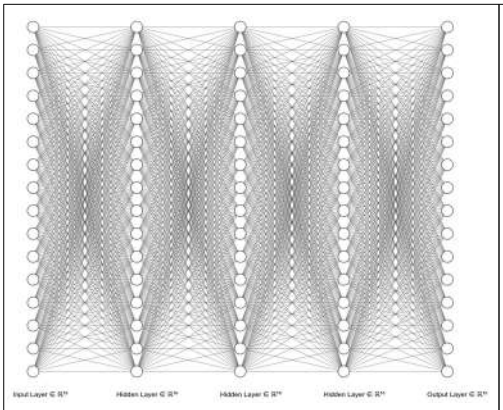


Figura 96: MLP con 5 capas (3 ocultas), todas de tamaño 16

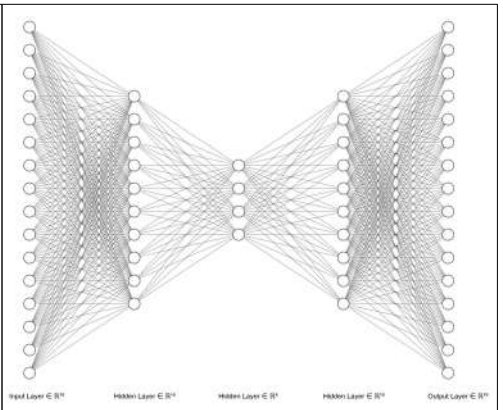


Figura 97: Autoencoder con 5 capas (3 ocultas)

En este tipo de redes, durante el entrenamiento, los pesos se distribuirán posiblemente igual en las tres capas ocultas; en cualquier caso este tipo de diseño no es muy útil, pero es un punto de partida para explicar los autocodificadores.

Observa ahora la Figura 97, la idea es la misma que en el diseño anterior, al igual que su *dataset* y entrenamiento con los que - imaginativamente - la estamos entrenando. Pero ahora hay un cambio: En este nuevo diseño hemos *estrechado* la capa central y ahora entre las capas de entrada y salida hay una especie de "cuello de botella. A la parte de la izquierda la llamamos **codificador** [*encoder*, de la capa de entrada a la más "estrecha", esta incluida] y a la de la derecha **decodificador** [*decoder*, desde la capa más "estrecha", no incluida, a capa de salida], ver la Figura 98.

285 *He propuesto una MLP de 16 entradas por motivos de claridad al imprimir la imagen. Insuficiente para imágenes. Imagina en este ejemplo que se trata de 784 (28x28) entradas e iguales salidas.*

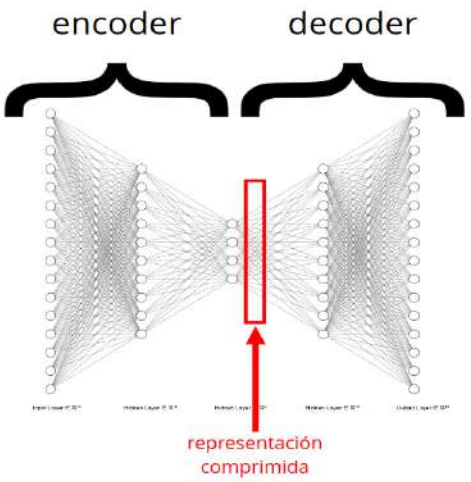


Figura 98: Esquema de un autoencoder

Seguimos imaginando que entrenamos este modelo alimentándolo con imágenes a la entrada y las mismas imágenes a la salida. Ahora la red, al hacer el esfuerzo de aprender que a partir de una entrada debe emitir la misma a la salida, necesita comprimir la información de la foto, quedándose sólo con las **características principales**: una representación comprimida de la imagen de la entrada.

Vamos a hacer un ejemplo real, con un *dataset* conocido. Nuestra arquitectura va a tener 784 entradas e igual número de salidas y cinco capas ocultas de tipo *dense*[286] [128, 64, 32, 64, 128]; la salida de la capa con 32 neuronas es la representación comprimida de la entrada, conteniendo sus características principales.

Caben dos preguntas: Dada la entrada, ¿cómo se ve la salida? pues así:

Figura 99: Entrada (arriba) y salida (abajo) de un autoencoder simple con el dataset MNIST digits

Como podemos ver, la red ha hecho su trabajo, y ha aprendido, pese al "cuello de botella", a propagar la entrada a la salida de forma bastante eficiente.

286 *Todas ReLu, la salida es sigmoid.*

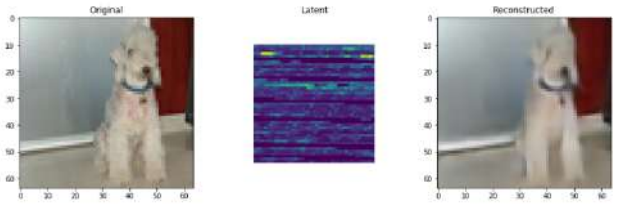


Figura 100: Autoencoder entrenado con imágenes (64x64) de perros.

Imagen original (izquierda), reconstruida (derecha) y el estado latente o representación comprimida (8x8).

Fuente: Chris Deotte - https://www.kaggle.com/code/cdeotte/dog-autoencoder/notebook

Y aquí viene la segunda pregunta: si la salida es tan parecida a la entrada ¿podemos decir que la red aprende a reconstruir la salida a partir de la representación comprimida? La respuesta es sí. Y esa es la magia de este tipo de arquitectura; entrenamos un modelo y a cambio obtenemos dos[287]:

1. El **codificador**, que a partir de una entrada del *dataset* es capaz de reducirlo a una representación latente o comprimida, un vector de características, que lo identifica de forma única dentro del *dataset*.
2. El **decodificador**, que a partir de una representación latente es capaz de reconstruir la información original. Esta representación latente no tiene porque formar parte del *dataset* original.

Aunque hemos puesto como ejemplo un *autoencoder* entrenado con imágenes, hay que recordar que para estas máquinas todo está formado por números: texto, imágenes, datos tabulados, etc. Así que le resulta indiferente[288] el tipo de información que procesa.

Estas dos herramientas son la base de la IA generativa (que merece un libro propio) y de muchas técnicas de arquitecturas más complejas. Por ejemplo: los *transformers* son, dentro de su complejidad, un codificador y un decodificador, como ya hemos visto [§12.1.2].

Por cierto, a este tipo de aprendizaje se le denomina **aprendizaje autosupervisado**, ya que es el propio algoritmo el que extrae de los datos las etiquetas que necesita para aprender. Se le considera un caso particular de aprendizaje supervisado.

17.5.2 Entrenando una arquitectura generativa: GAN

Vamos a ver una de las arquitecturas más apasionantes, y *madre* de muchas que han venido después. Una **Red Generativa Adversaria** (GAN, por sus siglas en inglés) consiste en dos redes neuronales, el generador y el discriminador, que se entrenan conjuntamente. El generador intenta producir datos falsos que parezcan reales, mientras que el discriminador intenta distinguir entre datos reales y datos falsos. Esencialmente, es un juego de gato y ratón donde el generador es el "falsifi-

287 *Sí, es posible "cortar" una red neurona artificial en partes como si usáramos una tijera. Recordemos que la unidad de esta es la capa, así que podemos eliminar, añadir modificar capas en estas y otras arquitecturas, incluso después de haber sido entrenadas.*

288 *Una matización: aunque son agnósticas respecto al tipo de datos hay arquitecturas que son preferibles según si hablamos de imágenes (redes convolucionales), texto (transformers),etc. Los autoencoders que estamos viendo aquí son los más generales.*

cador" y el discriminador es el "policía".

Otra analogía: el profesor y el alumno. Un profesor le pone un examen de una pregunta al alumno, siempre del mismo tema pero no siempre la misma. El alumno intenta aprobar el examen diciendo lo que se le ocurre, siempre de forma aleatoria. No ha estudiado nada. El profesor sólo corrige al alumno diciendo bien o mal, nada más. El alumno, tras muchos intentos, acaba aprendiendo qué decir para que el profesor no distinga su respuesta (aleatoria) y la de por buena.

Supongamos que trabajamos con imágenes. Esta arquitectura debe por tanto entrenar dos MLP: (1) la discriminadora, entrenada con un gran conjunto de imágenes para que haga clasificación binaria (si/no) de estas. Si, por ejemplo, hablamos de rostros el *dataset* estará compuesto por *n* imágenes con rostros y otras *n* con imágenes que no son rostros. La discriminadora aprende a diferenciar imágenes de un tipo u otro. (2) la generadora es un MLP de tipo *decoder*, una de las partes de una red *autoencoder*. Parte de una entrada aleatoria y como salida genera una imagen. Simple.

La clave es entrenar el MLP generador: Usando entradas aleatorias, dejamos que produzca imágenes y que la RNA discriminadora indique si es o no correcta. Si es incorrecta hacemos que esta aprenda, para que la próxima vez genere una solución menos incorrecta. Tras docenas de miles de *epochs* (o muchos más), tenemos un MLP generador que a partir de entradas aleatorias genera imágenes de rostros. Como los MLP tienden a generalizar, probablemente estos rostros serán originales, no existirán.

Figura 101: Rostros sintéticos producidos por StyleGAN

Fuente: NVIDIA research

La idea de enfrentar dos algoritmos entre sí provino de Arthur Samuel, un destacado investigador en el campo de la informática a quien se le atribuye haber popularizado el término *aprendizaje automático*. Mientras estaba en IBM, ideó un juego de damas que fue uno de los primeros en aprender por sí mismo con éxito, en parte al estimar las posibilidades de victoria de cada jugador en una configuración determinada del juego (recordemos, algoritmo *minimax*). Más tarde, Ian Goodfellow (et al.) en un artículo de investigación fundamental de 2014 titulado simplemente "*Redes generativas adversarias*", describen la primera implementación funcional de un modelo generativo basado

en redes adversarias, el que acabamos de describir.

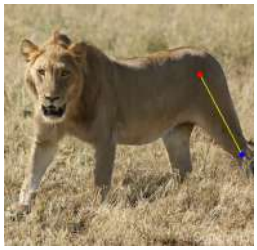
Lsa GAN son muy difíciles de entrenar, por la gran necesidad de potencia de cálculo y por la inestabilidad de su salida. Para imágenes de 1024x1024 (hoy en día no muy grandes) y con una GPU de más de cuatro mil euros, necesitaríamos más de 41 días de entrenamiento, unos 60k minutos.

Vídeo 14: Synthesizing High-Resolution Images with StyleGAN2

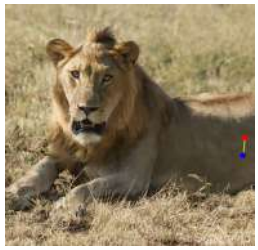
Fuente: NVIDIA Developer

StyleGAN es una arquitectura de red neuronal desarrollada por NVIDIA que genera imágenes artificiales de alta calidad utilizando GANs. Se utiliza principalmente para crear rostros, objetos y escenarios realistas pero sintéticos. StyleGAN se basa en la estructura de las GANs, que, como sabemos, consta de dos redes, el generador y el discriminador. A través de esta competencia, el generador mejora su capacidad para producir imágenes realistas. A diferencia de las GANs tradicionales donde se introduce un vector de ruido al comienzo, en StyleGAN, el vector de ruido se transforma en un "vector de estilo" que se introduce en múltiples etapas del generador. Esto permite controlar características específicas en diferentes niveles de detalle (por ejemplo, rasgos faciales gruesos versus detalles finos como el estilo del cabello).

Imagen original



Puntos de control



Resultado

Observar en el resultado, cuya imagen original es un rosto sonriente, como la expresión seria afectó también a los músculos de la cara.

17.5.3 Texto a imágenes

Dall·E [2021], Midjourney [2022], Stable Diffusion [2022] y otros muchos, aparecieron hace poco en la red como un avance de la IA generativa, con la capacidad de producir una imagen [ahora vídeos] a partir de un texto introducido como entrada [*prompt*]. ¿Cómo lo hacen? Los fundamentos para que funcionen los hemos visto a lo largo del libro, especialmente MLP, estados latentes [embeddings, vectores de características] y autoencoders.

Pero la pócima a veces no es revelada. De todas formas sí contamos con la arquitectura de DALL·E.

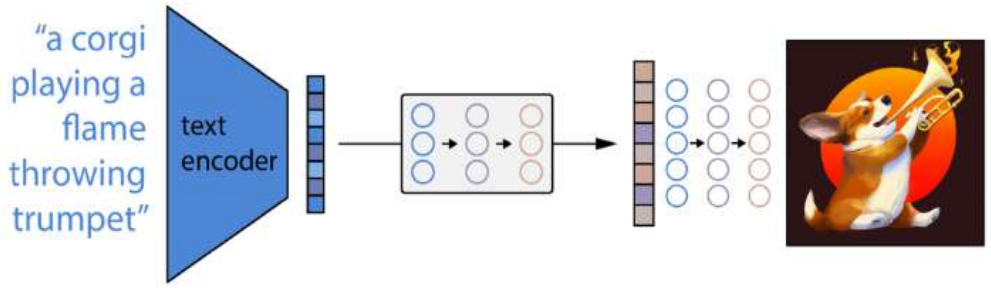


Figura 102: Esquema de la arquitectura de DALL·E 2

Fuente: "Hierarchical Text-Conditional Image Generation with CLIP Latents"; A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, M. Chen; 2022, arXiv.

A vista de pájaro, DALL-E funciona de forma muy sencilla [Figura 102]:

1. Se introduce un mensaje de texto [*prompt*] en un codificador [*encoder*] que está entrenado para transformar el mensaje a un espacio de estados latentes [*embeddings*].
2. A continuación, un modelo [MLP] llamado *prior* convierte el *embedding* de texto a una codificación de imagen, la cual captura la información semántica del mensaje contenido en la codificación de texto.
3. Finalmente, un decodificador [*decoder*] de imágenes genera estocásticamente una imagen[289] que es una representación visual de la información semántica que contiene el *prompt*.

Vídeo 15: How does DALL-E 2 actually work?

Fuente: AssemblyAI

289 *Similar a como lo hace el generador de GANs.*

DALL·E depende de CLIP (Figura 95), de OpenAI. CLIP se entrena con cientos de millones de imágenes y sus títulos asociados, y aprende en qué medida se relaciona un fragmento de texto determinado con una imagen. Es decir, en lugar de intentar predecir un título dado una imagen, CLIP simplemente aprende qué tan relacionado (en %) está un título determinado con una imagen concreta. Todo el modelo DALL-E 2 depende de la capacidad de CLIP para aprender la semántica del lenguaje natural.

DALL·E también depende de otro modelo anterior, GLIDE (Figura 103), para generar imágenes. Este modelo aprende a invertir el proceso de codificación de imágenes para decodificar estocásticamente incrustaciones de imágenes CLIP.

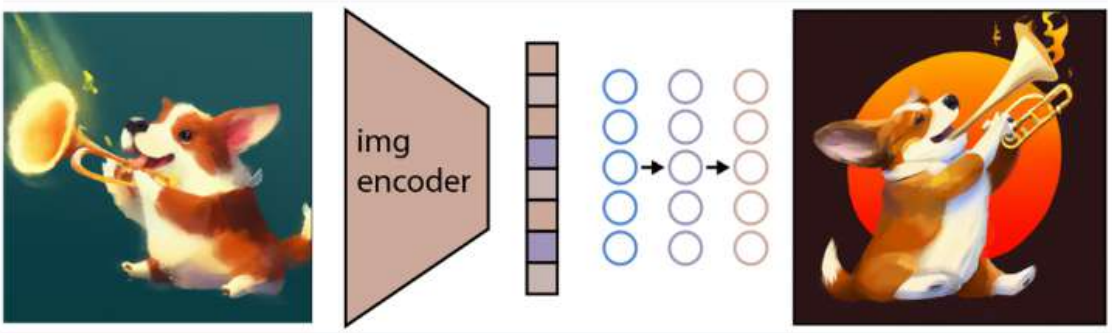


Figura 103: Arquitectura GLIDE para transformar una imagen

Fuente: "Hierarchical Text-Conditional Image Generation with CLIP Latents"; A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, M. Chen; 2022, arXiv.

Como se muestra en la imagen de arriba, cabe señalar que el objetivo no es construir un codificador automático y reconstruir exactamente una imagen dada su *embedded*, sino generar una imagen que mantenga las características principales de la imagen original dada su *embedded*. Para realizar esta generación de imágenes, GLIDE utiliza un **modelo de difusión**.

Imagina que tienes una imagen clara y, poco a poco, la "corrompes" con ruido hasta que es casi irreconocible. Luego, utilizando un modelo entrenado, intentas revertir este proceso paso a paso, eliminando el ruido en cada etapa hasta que vuelvas a tener una imagen original. Este proceso de revertir el ruido es esencialmente cómo funcionan los modelos de difusión.

Si el modelo de difusión se *corta a la mitad* después del entrenamiento, se puede utilizar para generar una imagen muestreando aleatoriamente ruido gaussiano y luego eliminándolo para generar una imagen fotorrealista (**difusión inversa**).

Ya tenemos todo, vamos a juntarlo de nuevo:

1. El codificador de texto CLIP asigna la descripción de la imagen al espacio de representación.
2. Luego, el modelo de difusión *prior* se asigna desde la codificación de texto CLIP a la codificación de imagen correspondiente.
3. Finalmente, el modelo de generación GLIDE modificado se asigna desde el espacio de re-

presentación al espacio de la imagen mediante difusión inversa, generando una de muchas imágenes posibles que transmite la información semántica dentro del título de entrada.

RETOS DEL CAPITULO 17

1. Dialoga con tu IA favorita, cuál es la diferencia entre una neurona biológica y una artificial.
2. Busca en la web, qué aplicaciones del mundo real utilizan redes neuronales artificiales.
3. Busca en la web, cómo se inspiran las redes neuronales artificiales en el cerebro humano.
4. Explica en qué consiste el proceso de entrenamiento de una red neuronal. Intenta llegar al mayor grado de entendimiento que puedas. Apóyate en una IA para ello.
5. Investiga y describe qué son las capas ocultas en una red neuronal.
6. ¿Qué significa la "propagación hacia atrás" en el contexto de las redes neuronales?
7. ¿Cómo se puede prevenir el sobreajuste en una red neuronal? Intenta deducirlo.
8. Busca y describe más arquitecturas de redes neuronales artificiales, además de las explicadas en el texto.
9. ¿Qué son los hiperparámetros en una red neuronal y cómo afectan a su desempeño?
10. Investiga y describe ejemplos de bibliotecas de Python que permiten implementar redes neuronales desde cero.
11. ¿Qué es la transferencia de aprendizaje y cómo se aplica en redes neuronales?
12. Investiga y describe un caso de estudio en el que las redes neuronales han tenido un impacto significativo.
13. ¿Qué son las redes neuronales recurrentes y en qué aplicaciones se utilizan?
14. ¿Cómo se puede explicar el funcionamiento de una red neuronal a alguien que no tiene conocimientos técnicos?
15. Investiga y comparte ejemplos de herramientas de software que permiten visualizar el entrenamiento de redes neuronales.
16. ¿Cómo se pueden usar las redes neuronales para el procesamiento de lenguaje natural? ¿y para la visión artificial?
17. Investiga y comparte ejemplos de recursos en línea que permiten a los principiantes aprender sobre redes neuronales.
18. Las redes GAN son muy interesantes. Busca más información sobre ellas.
19. Busca en la web ejemplos de lo que son capaces de hacer las redes basadas en modelos de difusión.
20. Tema de debate: ¿podríamos llegar a tener una IA fuerte en breve?