Trabajo de Laboratorio N°4:

Redes Neuronales Artificiales (ANN)

Materia: Inteligencia Artificial.

Carrera: Ingeniería en Informática.

Docente: Ing. Federico Gabriel D´Angiolo.

Alumnos: Cabot Lucas.

Calonge Federico.

Lew Imanol.

Índice

1-Objetivo.

El objetivo del presente trabajo de laboratorio consiste en estudiar el funcionamiento de un Perceptrón y el de una Red Neuronal, a partir de un ejemplo práctico haciendo uso del algoritmo de red neuronal que consiste en predecir prendas de moda, utilizando e incorporando los conceptos adquiridos durante el transcurso de la materia.

2-Introducción a las ANN.

2.1-Historia.

Entre las décadas de 1950 y 1960 el científico Frank Rosenblatt, inspirado en el trabajo de Warren McCulloch y Walter Pitts creó el Perceptrón, la unidad desde donde nacería y se potenciarían las redes neuronales artificiales. Un perceptrón toma varias entradas binarias x1, x2, etc. y produce una sola salida binaria. Para calcular la salida, Rosenblatt introduce el concepto de “pesos” w1, w2, etc., un número real que expresa la importancia de la respectiva entrada con la salida. La salida de la neurona será 1 o 0 si la suma de la multiplicación de pesos por entradas es mayor o menor a un determinado umbral. Sus principales usos son decisiones binarias sencillas, o para crear funciones lógicas como OR, AND.

El 1965 surgió el ‘multilayer perceptron’, el cual es una «ampliación» del percepción de una única neurona a más de una. Además, aparece el concepto de capas de entrada, oculta y salida. Pero con valores de entrada y salida binarios. Tanto el valor de los pesos como el de umbral de cada neurona lo asignaba manualmente el científico. Cuantos más perceptrones en las capas, mucho más difícil conseguir los pesos para obtener salidas deseadas.

En los ochentas surgen mejoras en mecanismos (como las redes Feedforward, donde las salidas de una capa son utilizadas como entradas en la próxima capa) y maneras de entrenar las redes (backpropagation). Estos conceptos se profundizarán en los siguientes puntos del informe. Además, en 1989 surgió la arquitectura de Redes Neuronales Convolucionales (CNN), la cual es útil en varias aplicaciones, principalmente procesamiento de imágenes. Esta arquitectura constó de varias capas que implementaban la extracción de características de la imagen y luego clasificar.

Pero se alcanza una meseta en la que no se puede alcanzar la «profundidad» de aprendizaje por la falta del poder de cómputo.

Luego, a partir de 2006 se logra superar esa barrera y se alcanza lo que se conoce como “Deep Learning” aprovechando el poder de las GPU y nuevas ideas se logra entrenar cientos de capas jerárquicas y dan una capacidad casi ilimitada a estas redes neuronales. Cada año surgen nuevos estudios acerca de las ANN y nuevas arquitecturas que (pueden o no) reemplazar a las anteriores.

2.2- Definición.

Las redes neuronales artificiales (ANN) son técnicas de Machine Learning que actualmente se están utilizando en varias áreas (además de Informática: en Medicina, Economía y Arte). Forman parte de la rama de **aprendizaje supervisado**, siendo un algoritmo de **Clasificación** (donde predecimos variables cualitativas; al igual que KNN y Naive Bayes). Aunque en algunos casos también se las usa en problemas de Regresión: por ejemplo las redes neuronales de Kohonen o "Mapas autoorganizados". Actualmente las ANN se están aplicando cada vez más en temas como el análisis de imágenes e interacción en forma hablada o escrita.

Una ANN, intenta simular el comportamiento de una red de neuronas biológicas de un ser humano. Una única **neurona biológica** es una estructura que procesa información; y está formada por:

- Un **cuerpo**.
- **Dendritas**: De donde les LLEGA la información de otras neuronas.
- **Axón**: De donde SALE la información hacia otras neuronas.

Entonces la información le llega a la neurona desde las dendritas, procesa la información y **sólo si le parece importante** la transmite al axón hacia la siguiente neurona (o hacia la siguiente **célula – no necesariamente tiene que ser una neurona**) mediante **Sinapsis** (proceso mediante el cual la neurona le pasa la información a la otra).

Cabe destacar que, si la neurona piensa que la información que mandó es muy importante, entonces refuerza esa información; y de esta manera no manda la misma cantidad que le llega, manda más señales a la que sigue, esto significa que le da un peso a la transmisión para la sinapsis (**Peso Sináptico**).

La **neurona artificial**, como dijimos previamente, intenta simular el comportamiento de su par biológico; de esta manera tiene una **serie de entradas** (como las dendritas) y tiene una **única salida** (que representaría al Axón). Procesa la información y dependiendo de cómo la procesa a veces puede aplicar una **FUNCIÓN** dentro de su cuerpo / soma.

Podemos observar ambos modelos de neuronas en la ***Imagen 1.*** La neurona artificial será explicada más detalladamente en la sección ***3.2-Modelo de un perceptrón.***

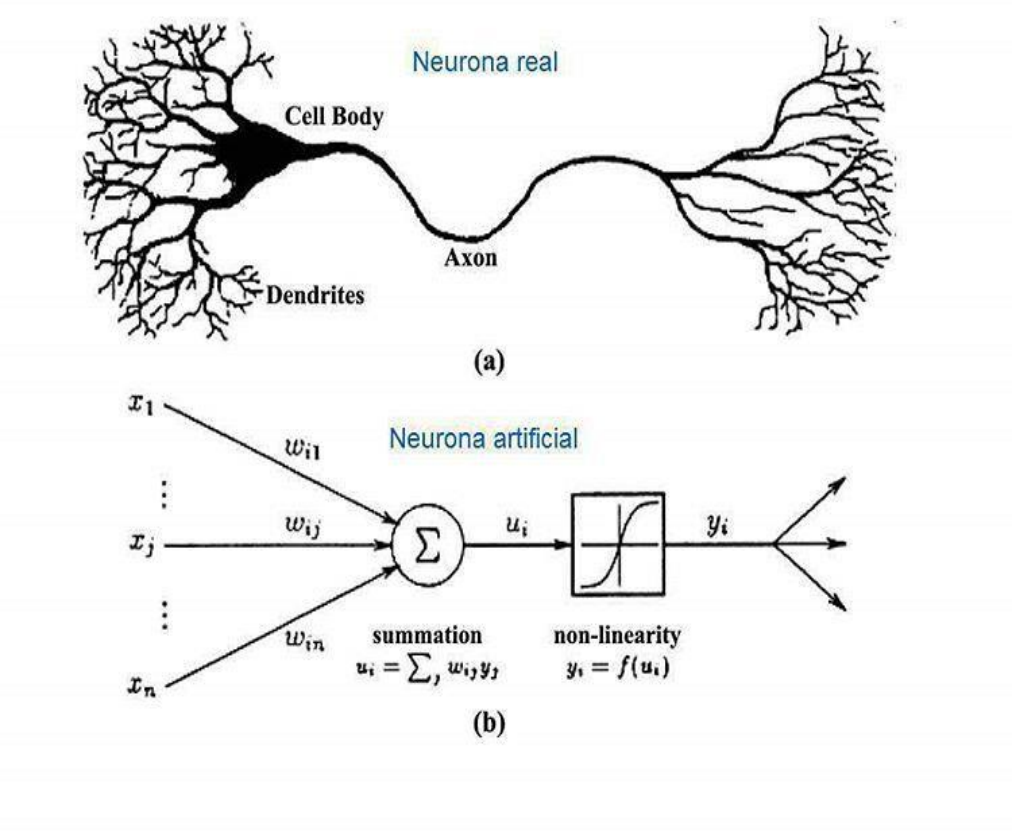


Imagen 1: Neurona Biológica y Neurona Artificial.

Una red neuronal biológica está formada por billones de estas neuronas conectadas entre sí (aproximadamente 2 billones), las cuales presentan conexiones de entradas de las cuales reciben estímulos externos (valores de entrada). En el caso del cerebro humano, las entradas están representadas por los 5 sentidos (el tacto, el olfato, el oído, la vista y el gusto), luego mediante las conexiones neuronales y el entrenamiento se adquiere una práctica que es traducida en conocimiento. En términos simples, **las redes neuronales artificiales consisten en un conjunto de unidades, llamadas neuronas artificiales o perceptrón, conectadas entre sí para transmitirse señales. La información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) produciendo unos valores de salida que nos servirán para realizar predicciones en problemas de clasificación.**

Estos sistemas aprenden y se forman a sí mismos, en lugar de ser programados de forma explícita, y sobresalen en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional.

De esta manera, cada neurona que resulta ser artificial debe aprender hasta obtener un modelo de red neuronal completo que sea capaz de clasificar eficientemente.

Para **problemas simples** de clasificación (**problemas LINEALES** o "linealmente separables": donde podemos colocar un hiperplano o una recta para separar nuestras clases → por ejemplo en problemas que se puedan modelar con compuestas OR o AND), podemos utilizar **técnicas más simples de Machine Learning** para resolverlos; por ejemplo, mediante técnicas de regresión logística. Pero, para el **caso complejo**, donde nosotros NO podemos separar las clases con un hiperplano o una recta (estaríamos ante un **problema NO LINEAL**: por

ejemplo, una compuerta XOR), en este caso necesitamos usar **redes neuronales** para resolverlos.

Dicho de otra forma, **las redes neuronales sirven para solucionar problemas lineales y no lineales**; pero si tenemos un problema lineal hay cosas más simples y menos costosas computacionalmente que podemos utilizar.

En los siguientes puntos profundizaremos más acerca de las redes neuronales y los elementos que la componen.

2.3-Aplicaciones.

Las ANN nos pueden servir para diversas aplicaciones, por ejemplo:

- Actualmente el traductor de Google utiliza ANN. En lugar de traducir palabra por palabra, traduce por contexto, utilizando redes neuronales para esto aplicadas en el campo de NLP. De esta manera está "entendiendo" la frase que uno quiere traducir por contexto.
- Gmail cuando nos da una sugerencia inteligente de qué respuesta dar, lo hace a través de redes neuronales donde está aprendiendo cuál sería la mejor respuesta a lo que nosotros estamos tratando de responder cuando hacemos el mail.
- Facebook usa ANN para el reconocimiento facial, detección de las personas y el etiquetado de las fotos.
- Actualmente los bancos (en Argentina y el mundo) utilizan redes neuronales para anticipar las subidas del dólar o caídas en las bolsas (en este caso serían problemas de Regresión).
- Se utilizan también para pasar de archivo de audio a texto. Por ejemplo este speach-to-text de IBM Watson: https://speech-to-text-demo.ng.bluemix.net.
- Y podemos predecir múltiples cosas más con ANNs: darle o no el préstamo a un cliente, darle o no medicación a una persona por una enfermedad determinada, etc.

3- Elementos básicos que componen una red neuronal

3.1-Modelo de Red Neuronal:

La distribución de neuronas dentro de la red se realiza formando niveles o capas, con un número determinado de dichas neuronas en cada una de ellas. A continuación, se presenta un modelo de conexión básico de redes neuronales:

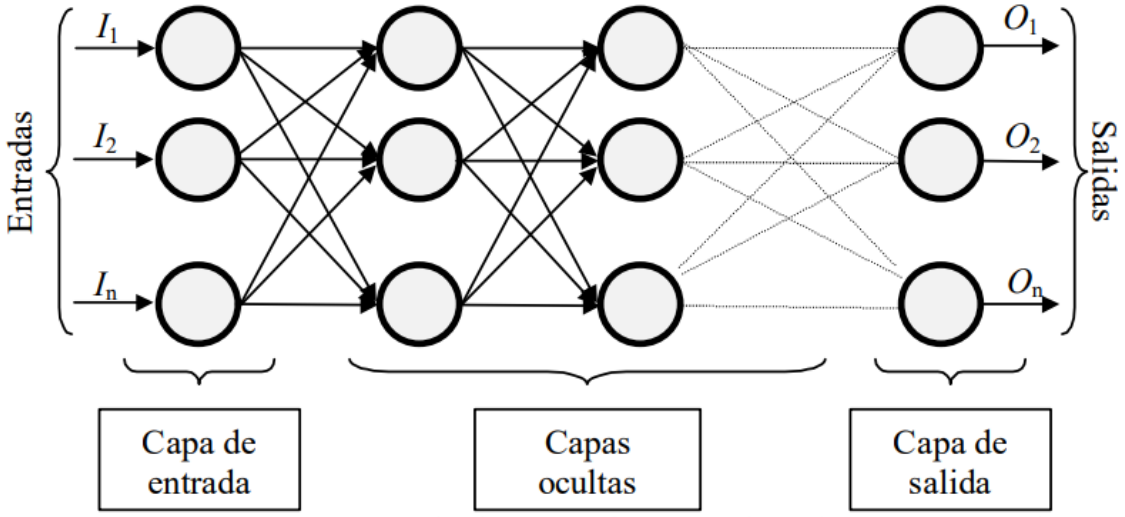


Imagen 2: Modelo de red neuronal

Como puede observarse, se pueden distinguir tres tipos de capas:

- **Capa de entrada:** es la capa que recibe directamente la información proveniente de las fuentes externas de la red.

- **Capas ocultas:** son internas a la red y no tienen contacto directo con el entorno exterior. El número de niveles ocultos puede estar entre cero y un número elevado. Las neuronas de las capas ocultas pueden estar interconectadas de distintas maneras, lo que determina, junto con su número, las distintas topologías de redes neuronales.

- **Capa de salida:** transfieren información de la red hacia el exterior. En la imagen 2 se puede ver el ejemplo de la estructura de una posible red multicapa, en la que cada nodo o neurona únicamente está conectada con neuronas de un nivel superior. Notar que hay más conexiones que neuronas en sí; en este sentido, se dice que una red es totalmente conectada si todas las salidas desde un nivel llegan a todos y cada uno de los nodos del nivel siguiente.

3.2-Modelo de un perceptrón:

Una única neurona o perceptrón en nuestro modelo de ANN se representa como un "**modelo de perceptrón simple**", en el cual tenemos solo 1 capa de entrada y 1 capa de salida. La capa de entrada son las variables predictoras (las características de un perro por ejemplo: altura, pelo, ojos, etc.) y la capa de salida tenemos la variable a predecir (por ejemplo qué raza de perro es). Un perceptrón simple solo puede resolver problemas simples de clasificación (problemas LINEALES o "linealmente separables": donde podemos colocar un hiperplano o una recta para separar nuestras clases → por ejemplo en problemas que se puedan modelar con compuestas OR o AND). Pero, para el **caso complejo**, donde nosotros NO podemos separar las clases con un hiperplano o una recta (estaríamos ante un **problema NO LINEAL**: por ejemplo una compuerta XOR), **necesitamos añadir más neuronas/perceptrones en la capa oculta.**

Las fórmulas y el modelo de perceptrón simple se pueden representar de la siguiente forma:

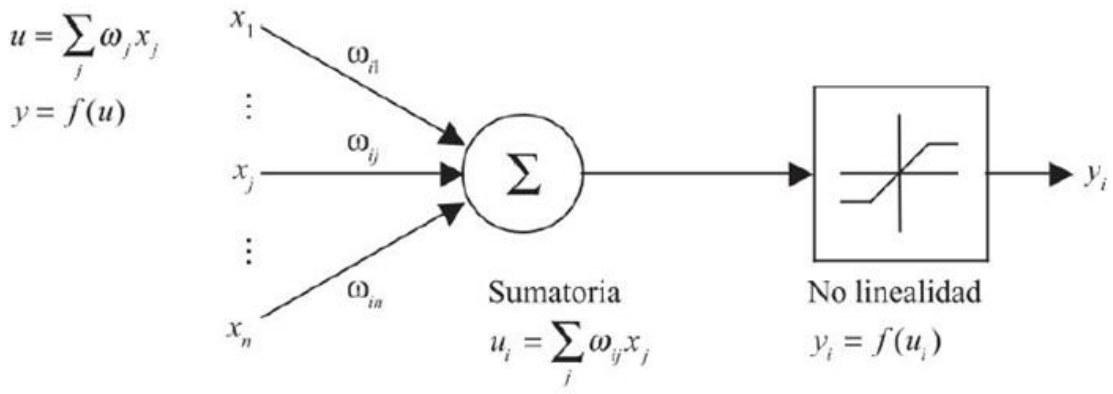


Imagen 3: Modelo de un perceptrón

-Donde:

xi = Entradas, que pueden ser datos de alguna imagen.
wi = Pesos o ponderaciones
u = Sumatoria que tiene en cuenta cada entrada y sus pesos
f(u) = Función de activación
y = Salida del Sistema

De esta manera a las entrada x1, x2, x3, etc... les damos un **PESO** (w1, w2, w3, etc.). Y así, **cada peso está asociada a 1 entrada**, a esto se lo llama "**ponderación de pesos**". De esta manera a cada entrada la multiplicamos por su peso y las sumamos dándonos la **salida u**. Esta salida u nos dará un valor dependiendo el valor de los pesos y las entradas. Este valor pasará por una "**función de activación**" (que generalmente es NO lineal). Por ej. podemos poner en dicha función... "si el valor de u que me llega es mayor a 0 que a la salida me devuelva 1, y si es menor a 0 que me devuelva 0". De esta manera a la salida "Y" obtenemos una clasificación: 1 o 0.

Los valores de Xi e Y ya los tenemos; lo que tiene que aprender la neurona en el entrenamiento son los pesos Wi. **La gracia (y toda la problemática) en una red neuronal es encontrar estos pesos Wi adecuados, que nos permitirán dar el resultado/salida "Y" correcta.** Cuando entrenamos a la ANN tenemos que configurar cuánto valen esos W. Para esto aplicaremos 2 técnicas que nos permitirán obtener estos valores de Wi: "Feedforward" y "backpropagation" (ver sección ***5.2- Feedforward*** y ***5.3 Backpropagation***).

Como dijimos previamente, nuestra u(t) era el resultado de la suma de cada peso W multiplicado por cada entrada X (y en algunas ocasiones suele aparecer un valor de **bias** que se le suma a esta expresión de **u(t)**). De esta manera nos queda la siguiente expresión:

$$u(t) = \sum_j \omega_j(t) x_j(t) + \theta(t).$$

Fórmula 1

Entonces, cada **neurona** o perceptrón tiene una tarea simple**: recibir la entrada de otras unidades o de fuentes externas y procesar la información para obtener una salida** que se propaga a otras unidades: por ejemplo, las neuronas de la capa oculta reciben la información de las capas de entrada, procesan esto (suman los pesos y multiplican por las entradas) y la salida va a las neuronas de la capa de salida.

La regla que logra **establecer el efecto de la entrada total u(t)** en la activación se denomina **función de activación (F),** las cuales veremos en la siguiente sección.

3.3-Funciones de Activación:

Como vimos anteriormente, en una red neuronal, los puntos de datos numéricos, llamados entradas, alimentan a las neuronas en la capa de entrada. Cada neurona tiene un peso, y al multiplicar el número de entrada con el peso se obtiene la salida de la neurona, que se transfiere a la siguiente capa.

La función de activación es una "puerta" matemática entre la entrada que alimenta a la neurona actual y su salida a la siguiente capa. Puede ser tan simple como una función de paso que activa y desactiva la salida de neuronas, dependiendo de una regla o umbral. O puede ser una transformación que mapea las señales de entrada en señales de salida que son necesarias para que la red neuronal funcione.

Dicho de otra forma, **la función de activación se encarga de devolver una salida a partir de un valor de entrada aplicando una transformación**, normalmente el conjunto de valores de salida está en un rango determinado como (0,1) o (-1,1). Como vimos anteriormente, nuestra entrada será "u". Se buscan funciones que las derivadas sean simples, para minimizar con ello el coste computacional.

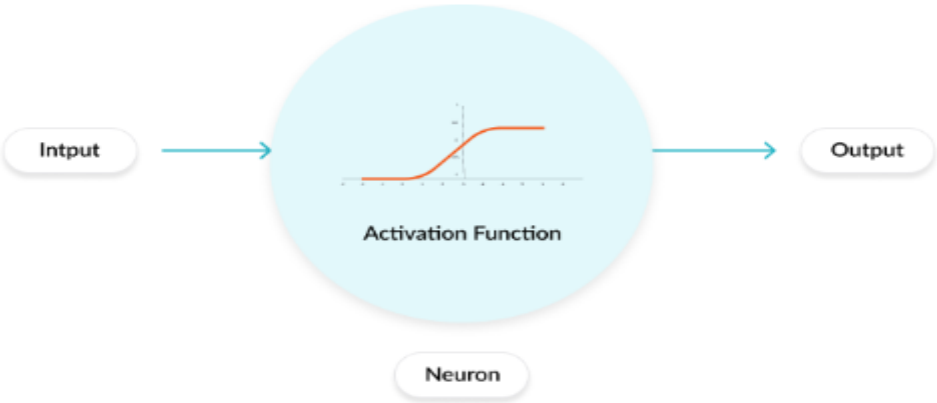


Imagen 3: Función de activación

En las ecuaciones que nombraremos abajo consideramos nuestra "u" como "x" (nuestras entradas).

3.3.1-Escalón.

Esta es la función de activación más sencilla, donde se asocia a neuronas binarias en las cuales, **cuando la suma de las entradas (nuestra U) es mayor o igual que el umbral de la neurona (en este caso si es mayor o igual a 0)**, la activación es 1; y si es menor a 0, la activación es 0 (o -1 en el caso de la variante al escalón). En el gráfico de abajo mostraremos la función escalón y, a la derecha, su variante:

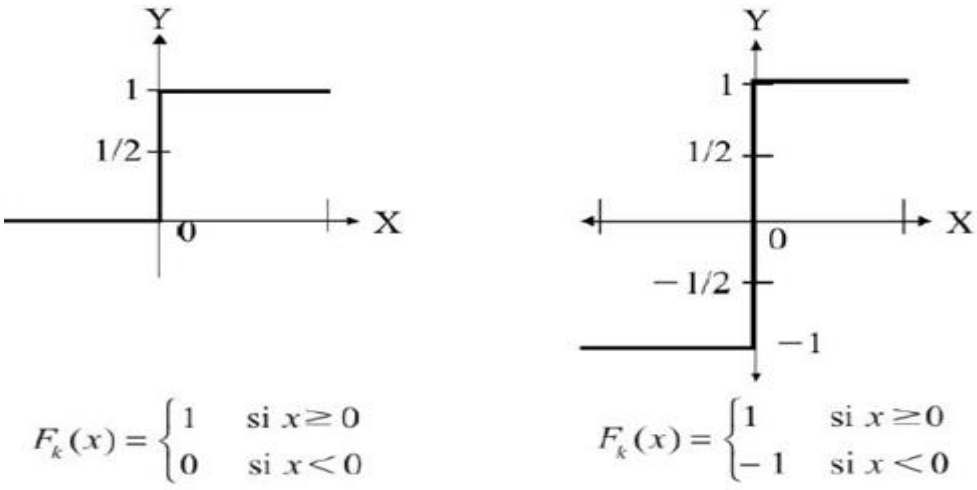


Imagen 4: Función escalón

Esta función de activación NO se utiliza habitualmente, ya que, como tenemos que derivar para hacer los cálculos de los pesos en el proceso de Backpropagation (Ver Sección ***5.3 Backpropagation),*** y la derivada del escalón es el IMPULSO, esto computacionalmente es muy difícil de realizar. Por esto, se utilizan las funciones enumeradas más abajo donde si se pueden derivar las ecuaciones correspondientes a cada función.

3.3.2-Sigmoidal.

Esta función es similar a la escalón pero más “suave”, también está acotada/oscila entre 0 y 1. Con esta función, el valor dado por la misma tiende a ser **asintótico**, de esta manera transformamos los valores introducidos a una escala (0,1)... donde los valores altos tienden de manera asintótica a 1 y los valores muy bajos tienden de manera asintótica a 0. Esto produce que **el valor de salida esté comprendido en la zona alta o baja del sigmoide**.

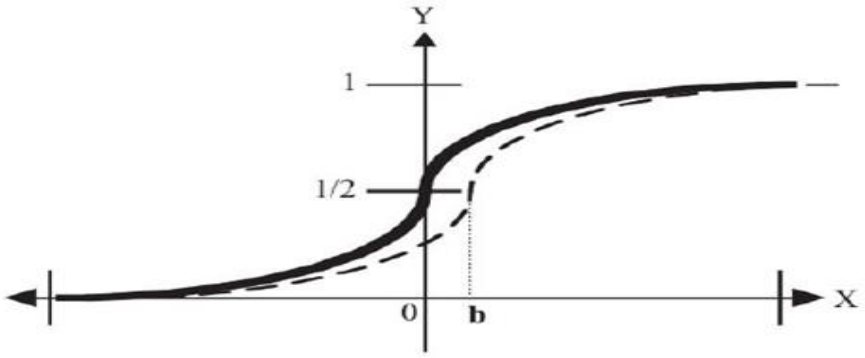


Imagen 5: Función sigmoidal

$$f(x) = \frac{1}{1 - e^{-x}}$$

Fórmula 2

3.3.3-Tanh - Tangente hiperbólica.

La función tangente hiperbólica transforma los valores introducidos a una escala (-1,1), donde los valores altos tienen de manera asintótica a 1 y los valores muy bajos tienden de manera asintótica a -1. A diferencia de la Sigmoidal, esta está centrada en 0.

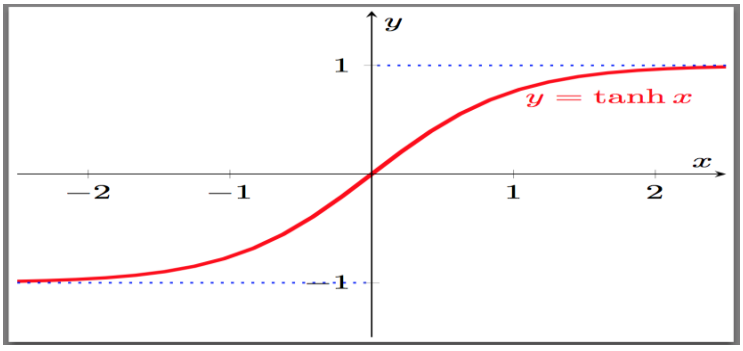


Imagen 6: Función tangente

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

Fórmula 3

3.3.4-Relu - Rectified Lineal Unit

La función Relu transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran.

Esta función es la que más se utiliza en ANN convolucionales, ya que son rápidas. Es parecida al escalón en el sentido que, para u negativos la salida vale 0 pero ahora para valores positivos (X>0) NO vale 1... sino que la salida toma los valores de “x”. De esta manera, si “x” es 3, “y” es 3; si “x” es 10, “y” es 10.

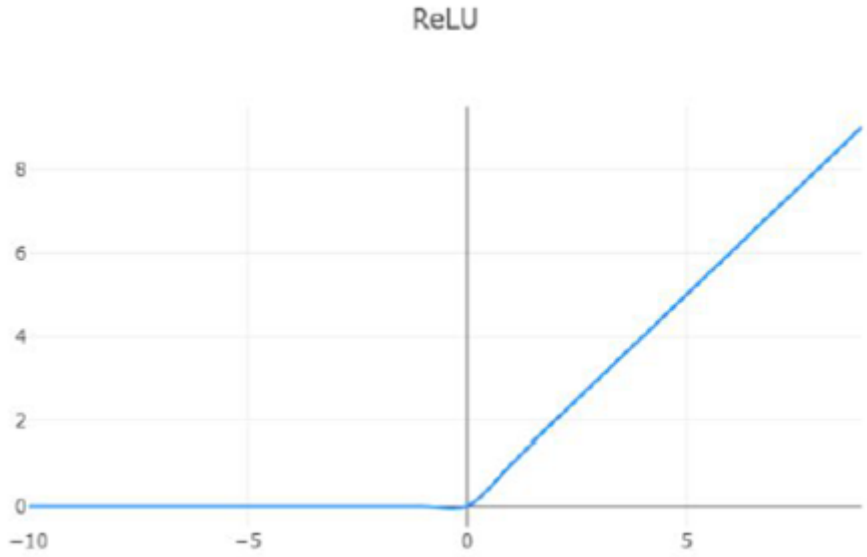


Imagen 7: Función Relu

$$f(x) = \max(0, x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$$

Fórmula 4

3.3.5-Softmax

La función Softmax transforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas de 1. Está acotada entre 0 y 1 y las utilizamos en las ANN para clasificar. En nuestro TL la utilizaremos para las neuronas de la capa de salida, para que puedan clasificar correctamente; ya que tienen un muy buen rendimiento en las últimas capas.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Fórmula 5

4-Clasificación de neuronas artificiales

Las neuronas artificiales se pueden clasificar de acuerdo con los valores que pueden tomar. De este modo podemos clasificar o distinguir a las neuronas de acuerdo con los valores que toman, siendo:

a) Neuronas binarias: solamente pueden tomar valores dentro del intervalo {0, 1} o {-1, 1}
b) Neuronas reales: pueden tomar valores dentro del rango [0, 1] o [-1, 1].

Es importante aclarar que los pesos normalmente no están restringidos a un cierto intervalo, aunque para aplicaciones específicas puede ser necesario limitar a un mismo rango.

5- Elección del conjunto inicial de pesos

5.1-Descenso del gradiente

Anteriormente, en los modelos de regresión lineal calculábamos el método de error cuadrático mínimo para que nos dijera exactamente cuál era el punto mínimo de la función de coste, en otras palabras, forzábamos a que la función de coste sea el error cuadrático medio, porque de esta manera podríamos obtener una función convexa, lo cual esto era algo fácil y sencillo de calcular. Por ejemplo, ¿si tenemos una función convexa, como podemos determinar el mínimo global de la función?

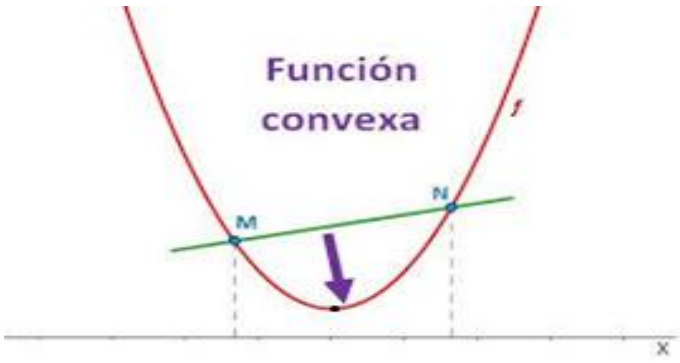


Imagen 8: Función convexa

Una forma de obtener el punto mínimo de la función es derivando ya que esta nos indica la pendiente de la función en dicho punto, es decir que buscaremos derivar todos los puntos cuando la pendiente sea igual a 0, y resolver dicha ecuación para encontrar el punto cuando la pendiente es nula.

Como puede observarse en la imagen 8, esta función convexa tiene una propiedad que nos dice que, de encontrar un punto mínimo, podemos asegurarnos que dicho punto será un mínimo global de la función. Es decir, no encontraremos un punto de la función que sea inferior a este punto. También, existe la función cóncava

que si la invertimos - f(x), obtendríamos una función convexa y podemos partir del mismo análisis.

¿Pero qué sucede con funciones que tienen más de un mínimo local?

La diversidad de modelos y funciones de coste en el campo de machine learning nos obliga a encontrar una solución para las funciones no convexas (funciones que presentan más de un mínimo local). El principal problema o limitación, es que aquí no se cumple la propiedad que veníamos analizando de funciones convexas, como se explicó anteriormente, en este tipo de funciones no convexas podremos encontrarnos con un punto que no sea el punto mínimo global de la función.

Por ejemplo, si tenemos la siguiente función de coste tridimensional, donde "theta 1" y "theta 2" son nuestros parámetros (X y Z), y nuestro error es ("Y")

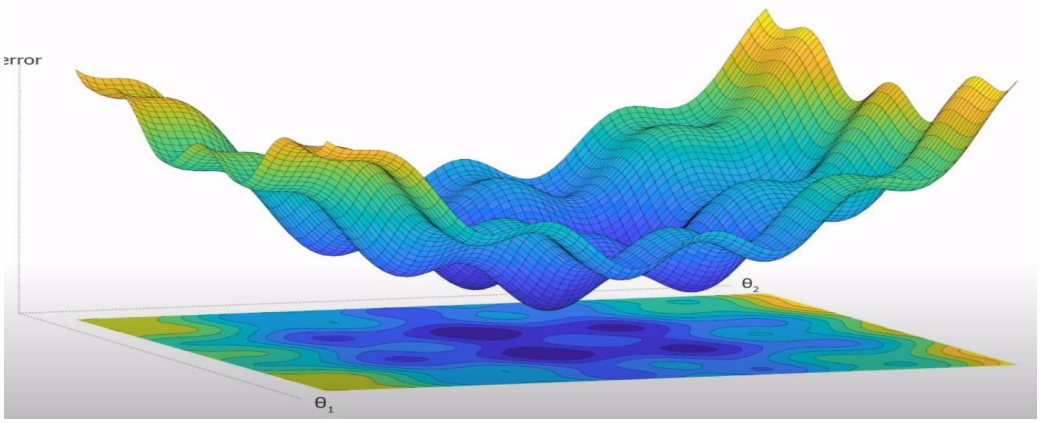


Imagen 9: Función no conexa

Suponemos que iniciamos nuestro entrenamiento del modelo con valores aleatorios de nuestros parámetros (theta 1, theta 2, error), esto equivale a iniciar en un punto cualquiera de nuestra función de coste, por ejemplo, se parte del siguiente punto que se observa a continuación.

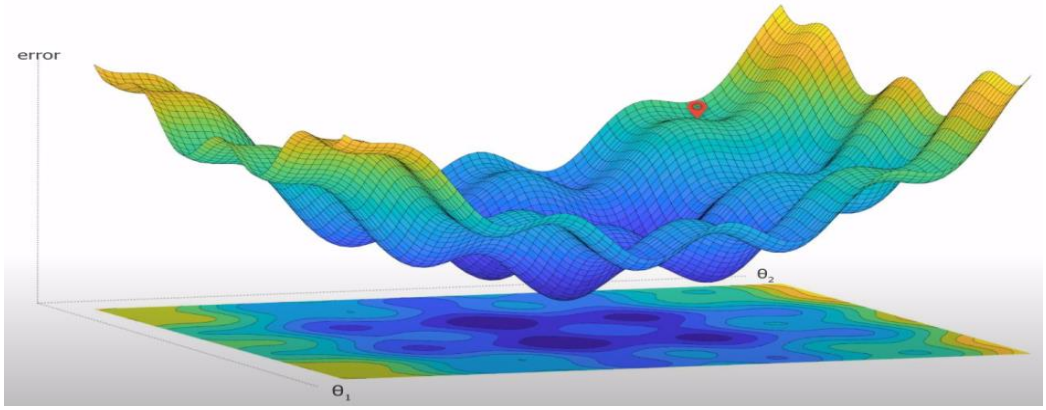


Imagen 10: Función no conexa

En este caso evaluamos la pendiente de la posición en la que se encuentra el punto, esto equivale a calcular la derivada parcial de la función en dicho punto. Como nuestra función es multidimensional tendremos que calcular las derivadas parciales de cada uno de nuestros parámetros (derivada del error en función de theta 1 y la derivada del error en función de theta 2), y cada uno de estos valores, nos dirá cuál es la pendiente en el eje de dicho parámetro, como puede observarse a continuación. Todas estas derivadas parciales conforman un vector que nos indica la dirección en el que vector asciende, este vector se denomina el gradiente. Como buscamos descender, ya que queremos obtener el mínimo de esta función, lo que se hará es tomar el sentido opuesto de este vector, es decir si el gradiente nos indica como tenemos que actualizar nuestros parámetros para ascender, lo que se hará es restarlo para tomar el sentido opuesto, para ello tendremos que hacer lo siguiente:

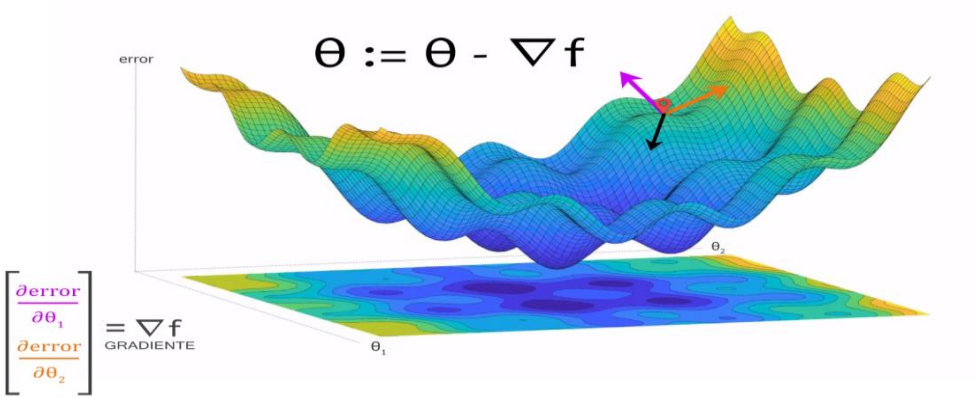


Imagen 11: Función no conexa

Este resultado nos indica un nuevo conjunto de parámetros, y por lo tanto nos indica una nueva posición o lugar dentro de la función con una pendiente menor a la anterior. Este proceso se repetirá n cantidad de veces hasta que ya realizar el procedimiento no suponga un cambio significativo o variación notable del coste. En otras palabras, la pendiente sea próxima o lo más cercano a un valor nulo, en este caso estaremos en un mínimo local y por lo tanto se minimizó el coste del modelo.

El último paso para finalizar con la obtención de este valor mínimo es incorporar el ratio de aprendizaje (alpha), este se multiplica con el gradiente, como se observa a continuación.

$$\theta := \theta - \alpha \nabla f$$

Fórmula 6

El ratio de aprendizaje indica cuanto afecta el gradiente a la actualización de nuestros parámetros en cada iteración, en otras palabras, cuanto avanzamos en cada paso. Es importante mencionar que se debe escoger un ratio adecuado de aprendizaje, ya que si no se determina correctamente se podría entrar en un bucle (por ejemplo, suponiendo un ratio alto) y nunca aproximarse al mínimo de la función de coste, esto es porque los pasos o saltos son tal altos que el punto es incapaz de introducirse en la zona de mínimo coste. También, podría ocasionar que tarde mucho en encontrarlo

(suponiendo un ratio bajo o pequeño) calculando de esta manera varias derivadas parciales innecesarias lo cual podría conducir a obtener un mínimo local que no sea el mínimo global de la función de coste, haciendo ineficiente al algoritmo.

A continuación, se adjunta un sitio de referencia donde se puede modificar el ratio de aprendizaje para distintas funciones de coste y ver de manera interactiva e intuitiva como se obtiene el mínimo de la función.

http://www.benfrederickson.com/numerical-optimization/

5.2- Feedforward

El término de feedforward se refiere al recorrido de "izquierda a derecha" o "hacia adelante" que hace el algoritmo de la red para dar valores a las entradas de las neuronas y así obtener los valores de salida.

5.3 Backpropagation

El nombre de backpropagation resulta de la forma en que el error es propagado hacia atrás a través de la red neuronal, en otras palabras, el error se propaga hacia atrás desde la capa de salida. Es una fórmula matemática creada a fines de los años 80 que permite colocarle los pesos a la red neuronal en forma automática.

A modo de ejemplo, supongamos que analizamos la cadena de responsabilidades de la producción de un combo de hamburguesa de Mc. Donald's, y el cliente va a premiar o castigar con una calificación de acuerdo al valor que le asigne a la atención y el producto que recibió, indicando en una pantalla de una tablet antes de retirarse, con una carita positiva o negativa. El combo de la hamburguesa lo podríamos ver como el resultado de salida de nuestra red y la evaluación del cliente como la función de coste. En este proceso cada tarea es un nodo especializado en una tarea determinada. Lo que se buscará es si el resultado no es el deseado (suponiendo la calificación negativa) buscar la/las neuronas/s que tengan responsabilidad en el peso del resultado final, de modo de poder analizar en que neurona se produce el error y poder corregirlo y de esta manera obtener un resultado positivo. Esto se logra mediante la retro-propagación de errores.

Supongamos que el cliente indicó el resultado con una carita negativa, la primera pregunta que nos podemos hacer es ¿dónde se presenta el error en esa cadena de responsabilidades? Quizás no le prepararon la hamburguesa con los ingredientes que el cliente solicitó, puede ser que la hamburguesa salió más cocida de lo que debía, o le llegó fría, o quizás la atención del empleado no fue buena, o las mesas estaban sucias, hay diferentes factores que pueden afectar el resultado final, y es aquí donde se introduce el concepto de backpropagation, buscando analizar en donde se produce el error y modificando los pesos de los parámetros para lograr un mejor resultado o el resultado deseado. Este análisis como su nombre lo indica se hace desde la capa de salida hacia atrás. Por ejemplo, el cliente indicó que calificó

negativamente porque le llegó una hamburguesa con otros ingredientes, bueno en este caso sabemos claramente que debemos analizar la cadena de responsabilidades desde la preparación de la hamburguesa hacia atrás, ya que los pasos siguientes no afectó en el resultado final. En este paso, se supone que la última capa es la de preparación del pedido, y la retropropagación de errores se analizará desde esta capa hacia las capas anteriores. Gracias a esto una vez que se llega a la capa inicial, se sabrá cual es el error para cada neurona y para cada uno de sus parámetros.

Esto permite que los pesos sobre las conexiones de las neuronas ubicadas en las capas ocultas cambien durante el entrenamiento. El cambio de los pesos en las conexiones de las neuronas además de influir sobre la entrada global influye en la activación y por consiguiente en la salida de una neurona. Por lo tanto, es de gran utilidad considerar las variaciones de la función activación al modificarse el valor de los pesos. Esto se llama sensibilidad de la función activación, de acuerdo con el cambio en los pesos.

Para ver cómo cambia el coste cuando variamos un parámetro, se obtiene con el descenso por el gradiente como se explicó en el punto anterior, derivando el coste por cada uno de los parámetros. El problema es que un cambio en el peso de un parámetro influye en el resultado de la información final que fluye por alguna de las conexiones de la red neuronal. A continuación, se muestra un esquema de conexión básico de una red neuronal totalmente conectada.

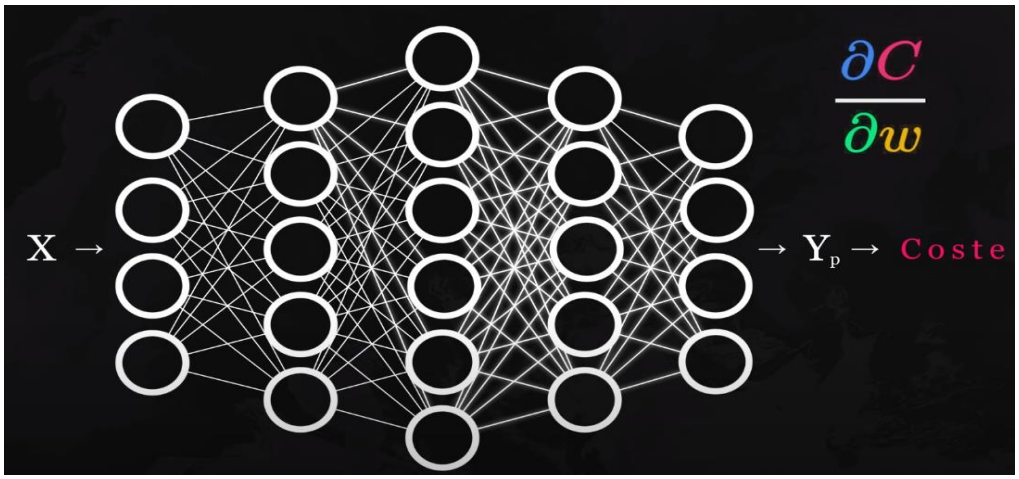


Imagen 12: backpropagation

Como puede observarse en la imagen 12, el algoritmo de backpropagation se utilizará para calcular las derivadas parciales del coste respecto a cada uno de los parámetros de nuestra red. Al igual que como hacíamos antes, se utilizará el descenso del gradiente para optimizar la función de coste haciendo uso de la técnica de backpropagation para calcular el vector de gradiente dentro de la complejidad de la arquitectura de la red neuronal.

A modo de resumen, en una red neuronal, existen 2 tipos diferentes de parámetros, los pesos (Wi) y el sesgo (también llamado bias, denominado b), como se observa a continuación:

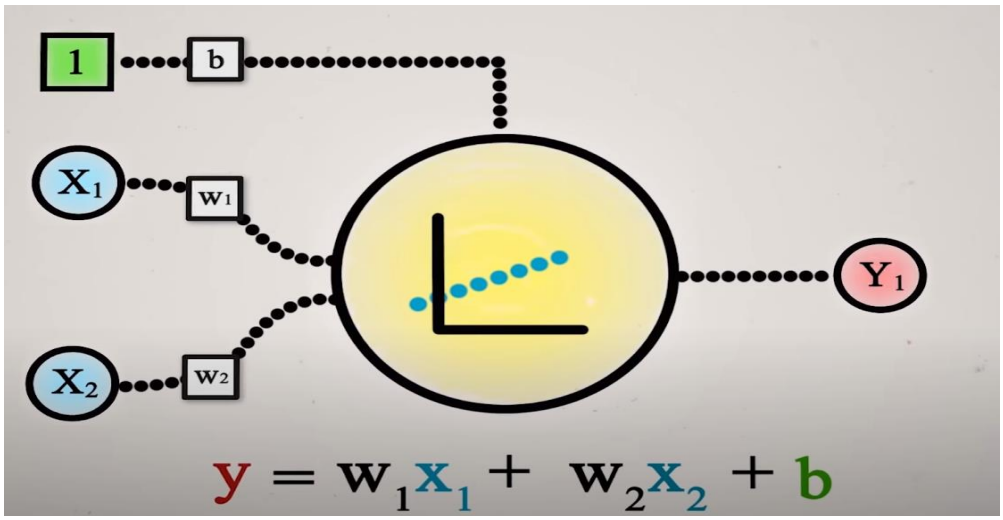


Imagen 13: resumen red neuronal

Esto indica que se deberá calcular en principio 2 derivadas parciales:

- dC/dw: derivada parcial del coste en función del peso (parámetro w)
- dC/dB: derivada parcial del coste en función del bias (parámetro b)

Para obtener las derivadas parciales de todos los parámetros de nuestra red neuronal artificial, debemos seguir los siguientes pasos:

1er paso: Calcular las derivadas de los parámetros de la última capa (L)

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$$

Fórmula 7

Donde:

- C es la función de coste
- a es la función de activación
- Z es la suma ponderada de la neurona:

$$Z^L = W^L a^{L-1} + b^L$$

Fórmula 8

En este paso se pide que se calcule la derivada de la función de coste con respecto al output de la red neuronal (función de activación) y calcular cómo

varía el output de la neurona cuando variamos la suma ponderada de la neurona (z).

Por otra parte, la derivada del coste en función de la última capa nos dirá qué responsabilidad tiene la neurona en el resultado final. Si en esta derivada el resultado es grande es que ante un pequeño cambio en el valor de la neurona este se verá reflejado en el resultado final. Sin embargo, si el resultado es pequeño no afectará al error de la red.

2do. paso: Retropropagamos el error a la capa anterior (L-1)

Utilizando el resultado obtenido anteriormente (fórmula 7), obtenemos el error de la capa anterior de la siguiente manera

$$\delta^{l-1} = W^{l}\delta^{l} \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}}$$

Fórmula 9

3er. Paso: Calculamos las derivadas de la capa actual usando el error para obtener las derivadas parciales para el parámetro de bias (b) y el parámetro de los pesos (W)

$$\frac{\partial C}{\partial b^{l-1}} = \delta^{l-1} \qquad \frac{\partial C}{\partial w^{l-1}} = \delta^{l-1} a^{l-2}$$

Fórmula 10

6- Ventajas que ofrecen las redes neuronales

- ✓ Aprendizaje Adaptativo: Capacidad de aprender a realizar tareas basadas en un entrenamiento o en una experiencia inicial.
- ✓ Auto-organización: Una red neuronal puede crear su propia organización o representación de la información que recibe mediante una etapa de aprendizaje.
- ✓ Tolerancia a fallos: La destrucción parcial de una red conduce a una degradación de su estructura; sin embargo, algunas capacidades de la red se pueden retener, incluso sufriendo un gran daño.
- ✓ Operación en tiempo real: Los cómputos neuronales pueden ser realizados en paralelo; para esto se diseñan y fabrican máquinas con hardware especial para obtener esta capacidad.

7- Limitaciones de las redes neuronales

Las redes neuronales cuentan con variadas limitaciones o desventajas.

- *Black box*: Una de las más conocidas desventajas, es que las redes neuronales tienen una naturaleza de caja negra. Es decir, si le damos inputs, y la red nos genera un output, no sabemos ciertamente él porque nos dio ese output específico. Lo que hace que sea difícil entender porque la red intentó clasificar nuestro input de cierta forma.
- Proceso de desarrollo: Si bien contamos con librerías como Keras que hacen el desarrollo de una red neuronal mucho más accesible, a veces se necesita tener más control sobre los detalles del algoritmo, sobre todo, cuando queremos resolver algo que nunca alguien ha resuelto. Para este caso utilizaremos tensorflow que nos da más opciones, pero es más complicado y el desarrollo puede ser muy prolongado.

- Cantidad de muestras: Las redes neuronales necesitan mucha más información que los algoritmos tradicionales de machine learning. Necesitan miles de muestras (e incluso hasta un millón de muestras.) y estas no son necesariamente fácil de conseguir.

- Computacionalmente costosas: Las redes neuronales son más costosas computacionalmente que los algoritmos tradicionales de ML. Incluso, algunos algoritmos de deep learning pueden tomar hasta semanas para ser entrenados. Claro que en parte también es debido a la inmensa cantidad de muestras que se necesitan en nuestro dataset para que la red pueda hacer su trabajo.

8-Explicación algoritmo de red neuronal.

8.1-Lectura y análisis de los datos.

Nuestro Data set consiste en distintas imágenes de ropa de moda las cuales trataremos de predecir Es un dataset que está incorporado en la librería Keras.

Data Set Imágenes Ropa de Moda

Las clases que utilizaremos para clasificar estas prendas serán las siguiente:

```
[29] class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Los datos no requieren preprocesamiento por nuestra parte dado que estos ya fueron procesados para una facilitación de uso, para todo aquel que quiera hacer uso de la librería Keras en este mismo ejemplo.

8.2-Implementación del algoritmo.

Procedemos a crear nuestro modelo de red neuronal, donde definimos:

-Cuantas neuronas tendremos en la entrada (Nota: como las imágenes son de 28 píxeles, la cantidad de neuronas de entrada debe ser 28x28, es decir, 784 neuronas)

-Cuantas neuronas tendremos en la capa oculta y que función de activación tendrán

-Cuantas neuronas tendremos a la salida y que función de activación tendrán

```
[32] model_epochs= keras.Sequential([
         keras.layers.Flatten(input_shape=(28, 28)),
         keras.layers.Dense(128, activation='relu'),
         keras.layers.Dense(10, activation='softmax')
     ])
```

A continuación, compilamos el modelo, el cual también lleva una configuración previa:

-Optimizer: Actualiza basado en el set de datos que ve y la función de pérdida

-Loss function: Mide qué tan exacto es el modelo durante el entrenamiento

-Metrics: Se usan para monitorear los pasos de entrenamiento y de pruebas

```
[33] model_epochs.compile(optimizer='adam',
                          loss='sparse_categorical_crossentropy',
                          metrics=['accuracy'])
```

Como de costumbre, el entrenamiento del modelo comienza con la función *fit*, la cual recibe las imágenes de entrenamiento, de prueba, y la cantidad de *epochs*. Los epochs representan las veces que será iterado nuestro modelo, y por ende, la cantidad de veces que será entrenado.

```
[34] mymodel=model_epochs.fit(train_images, train_labels, epochs=100)
```

8.3-Resultados.

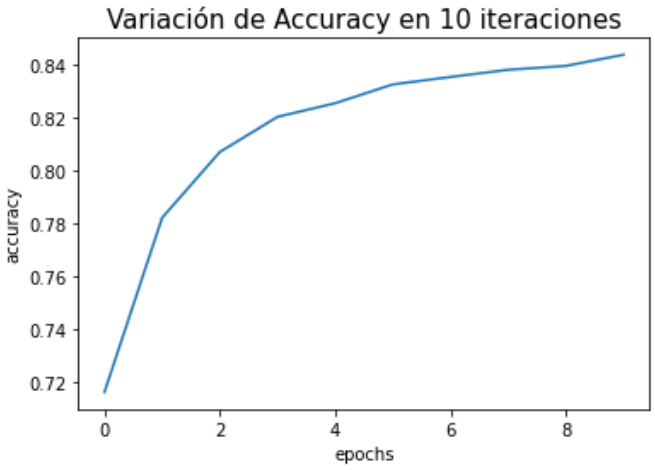
8.3.1-Variación de Epochs.

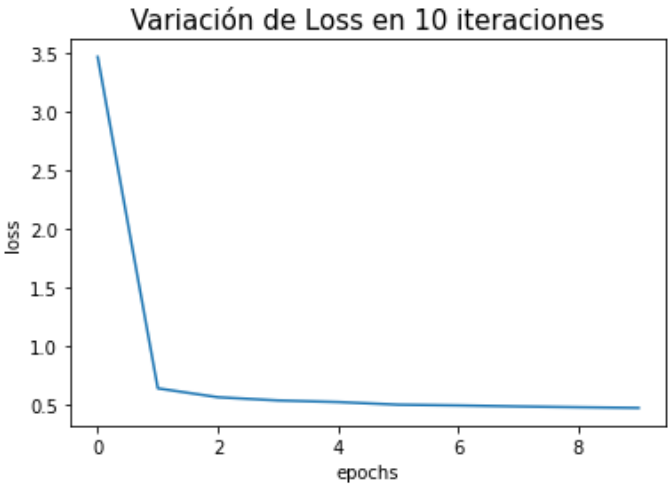
Una vez creado nuestro modelo, podemos empezar a analizarlo. A continuación, se puede visualizar la salida del algoritmo, que nos permite ver para cada epoch una eficacia y el resultado de la función de pérdida.

```
Epoch 1/100
1875/1875 [==============================] - 4s 2ms/step - loss: 3.4644 - accuracy: 0.7162
Epoch 2/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6332 - accuracy: 0.7821
Epoch 3/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5573 - accuracy: 0.8069
Epoch 4/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5297 - accuracy: 0.8202
Epoch 5/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5161 - accuracy: 0.8253
Epoch 6/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4937 - accuracy: 0.8324
Epoch 7/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4874 - accuracy: 0.8353
Epoch 8/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4785 - accuracy: 0.8380
```

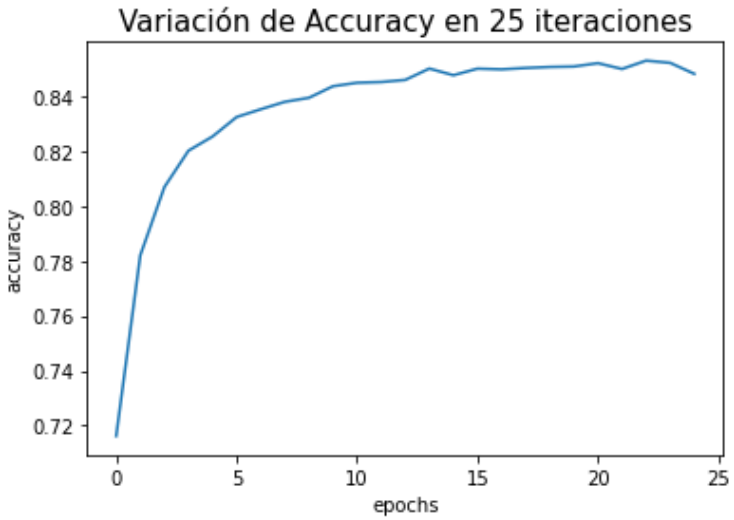
```
Epoch 94/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4014 - accuracy: 0.8645
Epoch 95/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4011 - accuracy: 0.8641
Epoch 96/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4167 - accuracy: 0.8625
Epoch 97/100
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4108 - accuracy: 0.8627
Epoch 98/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4076 - accuracy: 0.8625
Epoch 99/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4165 - accuracy: 0.8619
Epoch 100/100
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3981 - accuracy: 0.8655
```

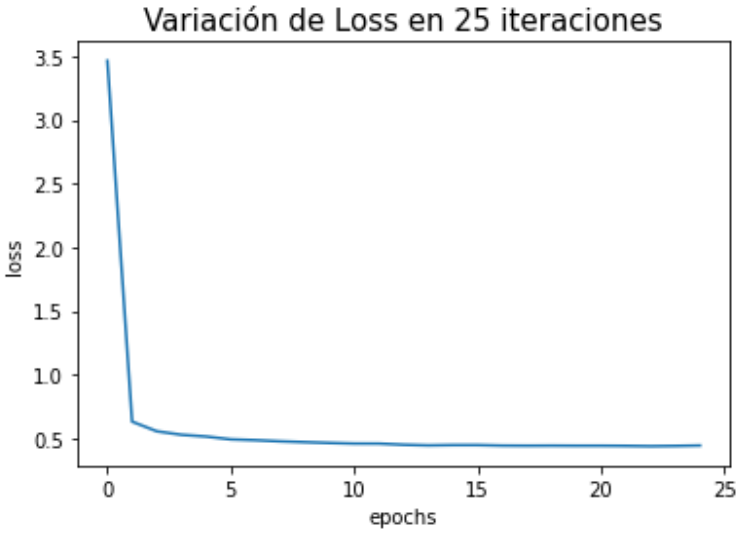
Variación de eficacia y Loss (o función de perdida) para 10 epochs:



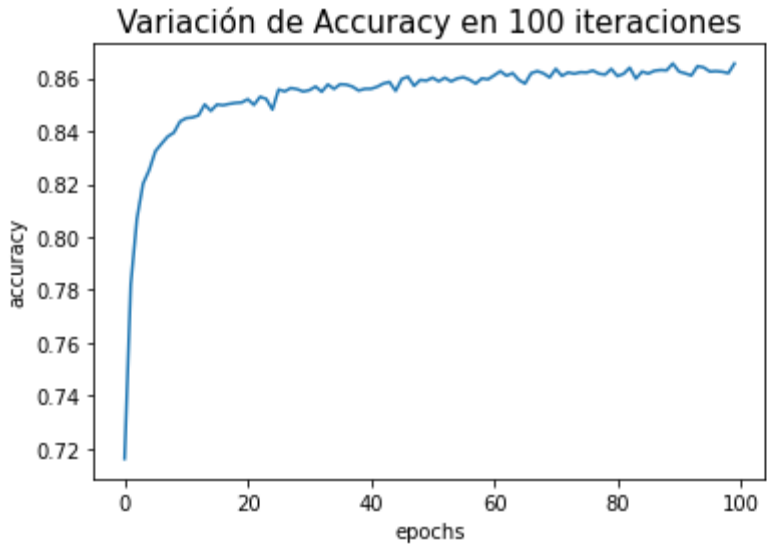


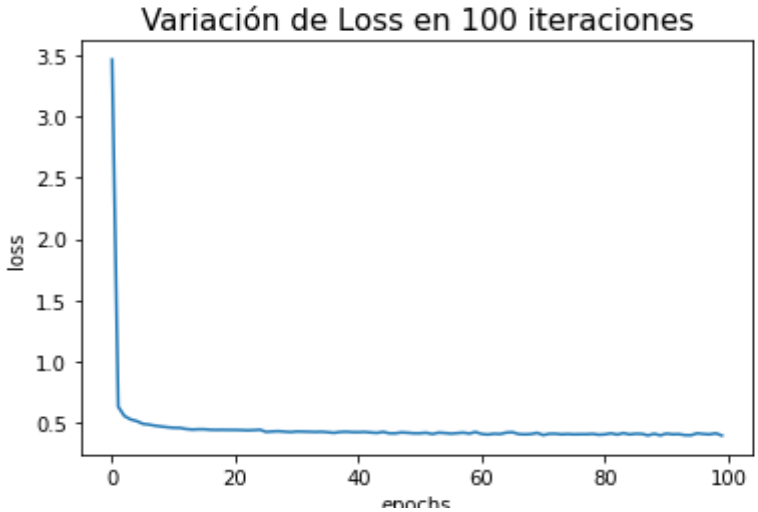
-Variación de eficacia y Loss para 25 epochs.





-Variación de eficacia y Loss para 100 epochs:





Valores mínimos de Loss para epoch=10, 25 y 100:

```
Loss para los primeros 10 epochs:  0.48073241114616394 en el epoch numero:  9
Loss para los primeros 25 epochs:  0.4388991594314575 en el epoch numero:  24
Loss para los primeros 100 epochs:  0.4081231951713562 en el epoch numero:  88
```

Valores máximos de Accuracy para epoch=10, 25 y 100:

```
Accuracy para los primeros 10 epochs:  0.8436499834060669 en el epoch numero:  9
Accuracy para los primeros 25 epochs:  0.8529833555221558 en el epoch numero:  22
Accuracy para los primeros 100 epochs:  0.8655666708946228 en el epoch numero:  89
0.8655666708946228
```

Como podemos observar, a mayor cantidad de epochs nuestra LOSS es menor y nuestra accuracy es mayor.

8.3.2-Variación de cantidad de neuronas en la capa oculta.

A continuación, se analizará cómo varía la pérdida si modificamos la cantidad de neuronas en la capa oculta. Anteriormente, se utilizaron 128. En este análisis se observará como es modificado el Loss para los casos de 64 y 256 neuronas en la capa oculta.

Recordemos que con 128 neuronas en la capa oculta obtuvimos un LOSS mínimo de 0.40 y un ACCURACY máximo de 0.86.

Empezamos con una ANN de 64 neuronas:

```
#Creamos:
model_epochs_64hidden= keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

model_epochs_64hidden.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

#Entrenamos:
mymodel_64hidden=model_epochs_64hidden.fit(train_images, train_labels, epochs=100)
```

El resultado que obtuvimos fue:

Loss mínimo de 0.49 y Accuracy máximo de 0.81.

Vemos que nuestra LOSS fue mayor y nuestra ACCURACY menor.

Procedemos para el caso de *hidden layers*=256

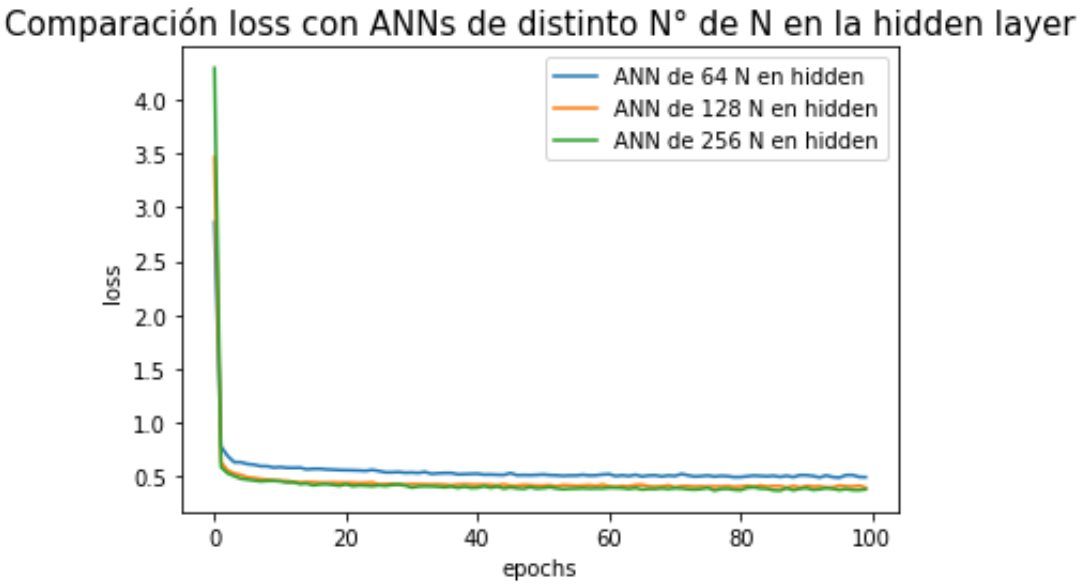
```
model_epochs_256hidden= keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
model_epochs_256hidden.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
mymodel_256hidden=model_epochs_256hidden.fit(train_images, train_labels, epochs=100)
```

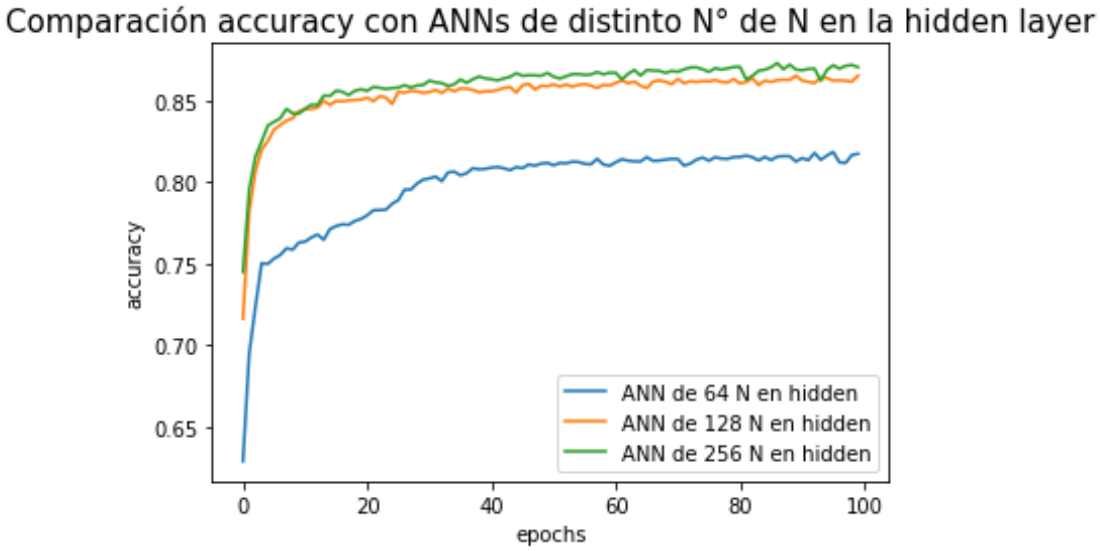
El resultado que obtuvimos fue:

Loss mínimo de 0.37 y Accuracy máximo de 0.87.

Vemos que nuestra LOSS fue menor que los 2 anteriores y nuestra ACCURACY tambien fue mayor que los anteriores.

Comparación de Loss y eficacia con hidden layers=(64,128,256)





Como podemos ver, hay un gran cambio si pasamos de 64 a 128 neuronas, y si bien también hay una ganancia de eficacia y Loss con 256, el cambio no es tan impactante.

8.3.3-Modificando función de activación.

Ahora, veremos cómo se desenvuelve el algoritmo si cambiamos la función de activación en la hidden layer. Actualmente se utiliza la función *relu*, y esta será reemplazada por las funciones *sigmoid* y *tanh*.

Se empezará con la función sigmoid:

```
model_sigmoid= keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='sigmoid'),
    keras.layers.Dense(10, activation='softmax')
])
model_sigmoid.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
mymodel_sigmoid=model_sigmoid.fit(train_images, train_labels, epochs=100)
```

-Los resultados que obtuvimos:

Obtuvimos un mínimo LOSS de 0.52 y un máximo ACCURACY de 0.81. Nos dió bastante peor que utilizando RELU.

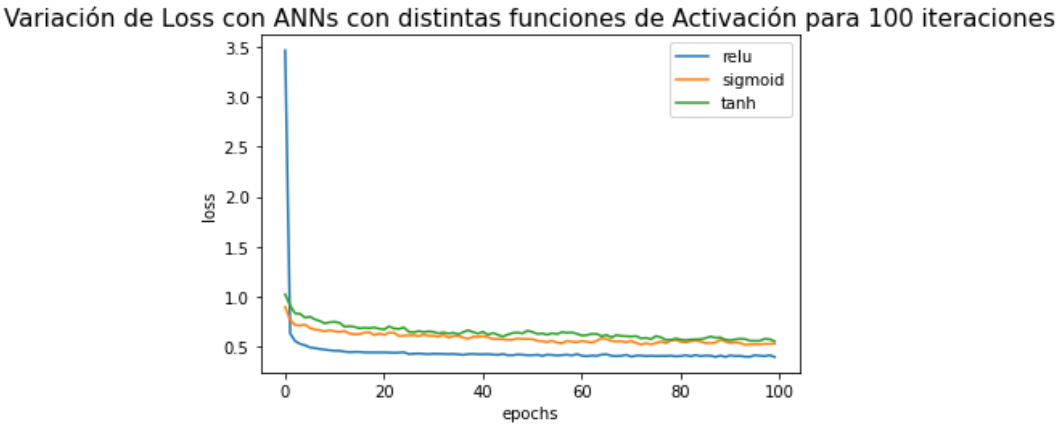
Por último, ahora utilizaremos la función tanh

```
model_tanh= keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='tanh'),
    keras.layers.Dense(10, activation='softmax')
])
model_tanh.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
mymodel_tanh=model_tanh.fit(train_images, train_labels, epochs=100)
```

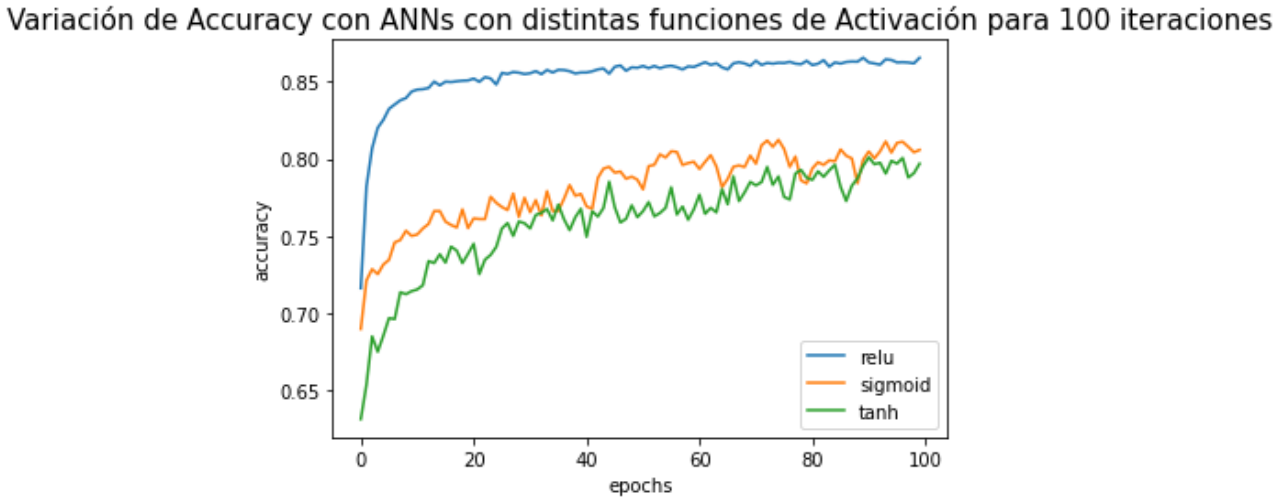
-Los resultados que obtuvimos:

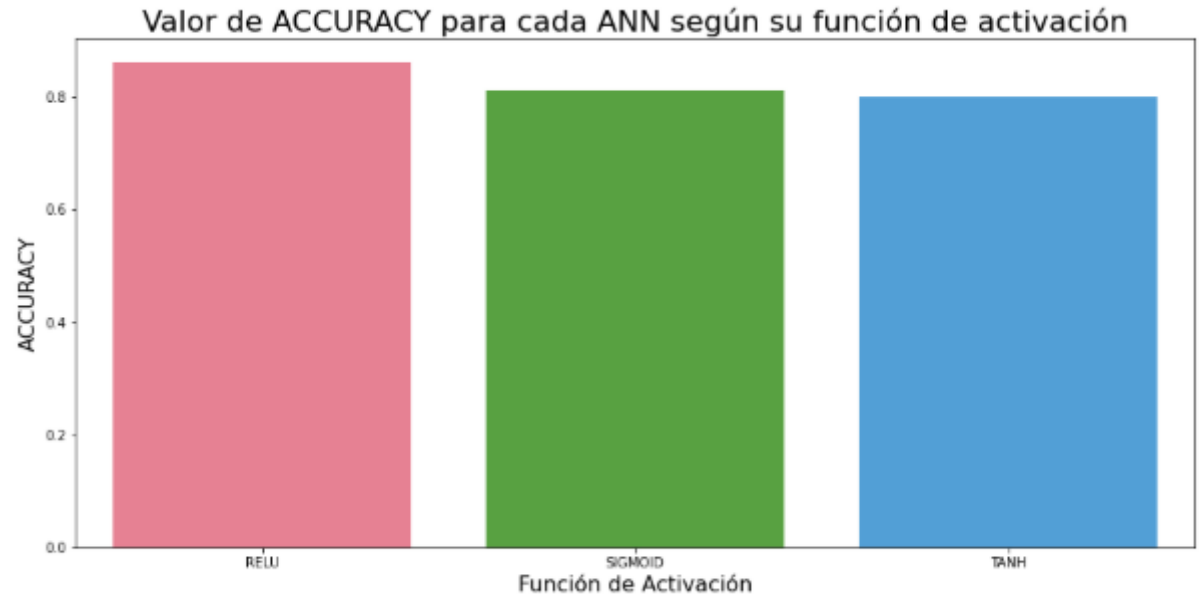
Obtuvimos un mínimo LOSS de 0.55 y un máximo ACCURACY de 0.80. Un poco mejor que con sigmoid.

Comparación de Loss entre relu, sigmoid y tanh



Comparación de precisión entre relu, sigmoid y tanh





De esta manera, observamos que utilizando la función de activación RELU obtenemos un mejor performance: logramos el MENOR LOSS y el MAYOR ACCURACY.

9-Conclusiones.

En este TL pudimos analizar los valores de **Loss** y de **Accuracy** para distintas modificaciones que le fuimos aplicando a la red neuronal. De esta manera, para este problema en particular de clasificación de imágenes de moda, observamos que nos resulta más conveniente:

- **Epochs**: utilizar la mayor cantidad de epochs posibles (sin llegar al estado de *overfitting*).
- **Número de neuronas en la capa oculta**: utilizar la mayor cantidad. Nos dio un mejor resultado agregando más neuronas.
- **Función de activación de las neuronas de la capa oculta**: Observamos que con RELU tenemos una mejor performance. Igualmente, esto depende para cada problema, en nuestro caso de clasificación de imágenes de moda con esta función de activación nos dio el menor LOSS y el mayor ACCURACY.

La desventaja de agregar más epochs y más neuronas a la capa oculta es el **tiempo de procesamiento**, que será mayor al aumentar estas métricas. Además, hay casos donde agregar muchos más epochs (lo cual puede llevar a overfitting si se usan muchos, o underfitting si hay pocos) o muchas más neuronas no nos afecta (disminuye) demasiado en el valor de LOSS ni nos aumenta mucho nuestro accuracy; en estos casos se deben utilizar cantidades “promedio” (ni muy bajas ni muy altas) y nuestro modelo de ANN predecirá correctamente.

De esta manera, como pudimos comprobar en el desarrollo del informe, **podemos modificar muchas variables al crear nuestra ANN**: epochs, número de neuronas en la capa oculta, funciones de activación, regularizadores, etc. **Para cada problema debemos ajustar esto según nuestro objetivo.**

10-Mejoras a desarrollar.

Dada las limitaciones mencionadas en el presente informe, y la gran cantidad de datos que puede llegar a requerir el algoritmo para ser entrenado y que pueda lograr una correcta predicción, hace que, al tener una gran cantidad de neuronas, a su vez se tienen demasiadas conexiones, esto en términos computacionales es muy costoso y demanda una gran cantidad de tiempo. Si bien para el ejemplo presentado, este algoritmo de red neuronal funciona correctamente, si el conjunto de datos fuera superior, el algoritmo quizás se vea afectado y no alcance a detectar algunos patrones de las fotos necesarios o si introducimos nuevas clases de prendas con detalles minuciosos a tener en cuenta en la clasificación, este algoritmo puede que no logre interpretarlos correctamente.

Una posible solución a este problema es el algoritmo de red neuronal convolucional (CNN), ya que este introduce varios conceptos u operaciones interesantes, como ser:

- Pooling o agrupación, es una operación utilizada para reducir el tamaño de nuestra imagen. Asimismo, el pooling hace la imagen más pequeña quitando número de conexiones y de esa manera también evitar sobre ajustar el modelo
- Convolución: Se puede asimilar como pasarles varios filtros a nuestras imágenes para detectar ciertos patrones relevantes dentro de ella.
- Profundidad de nuestra capa convolucional: número de filtros a aplicar a nuestra imagen en cada una de las convoluciones.
- stride o paso: se refiere a la cantidad de pixeles en la que se recorre el filtro. Mientras más alto sea nuestro paso, más alto vamos a reducir el tamaño de nuestra imagen.

Entonces si definimos capas que se encarguen de las convoluciones, podríamos entonces procesar ciertos parches o segmentos de las fotografías, por lo tanto, el número de conexiones será menor. Cada convolución va a recorrer toda la imagen, y nos va a dar una imagen de salida (esta nueva imagen va a tener una longitud y altura más pequeña, pero va a aumentar la profundidad). Además, con cada filtro definido se obtendrá un mejor resultado y por lo tanto mejorará la precisión de predicción del algoritmo.

11-Bibliografía.

- https://keras.io/guides/sequential_model/
- https://keras.io/examples/mnist_cnn/
- https://keras.io/api/layers/activations/
- https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
- https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/
- https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
- https://developers.google.com/machine-learning/glossary#logits
- http://www.benfrederickson.com/numerical-optimization/