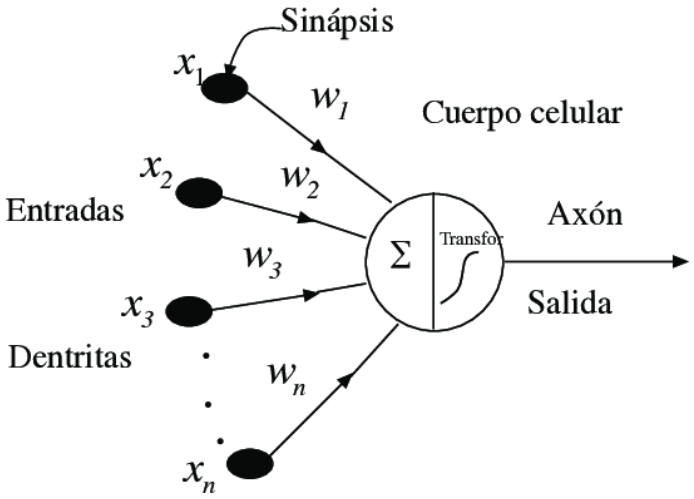
El Perceptrón

El perceptrón es una de las arquitecturas de Redes Neuronales artificiales más simples, esta fue inventada en 1957 por Frank Rosenblatt y se basó en una neurona artificial un poco diferente, conocida también como umbral lógico unitario TLU.



En este caso las entradas del TLU son números en vez de entradas binarias x_n, donde a cada una de las entradas se le asigna un peso W_n.

De esta manera el TLU ejecuta una suma producto de sus entradas $z = x_1w_1 + x_2w_2 \ldots x_nw_n$.

Una vez teniendo la suma producto, esta se transforma normalmente utilizando la función conocida como heavyside donde:

$$Heavyside\ (z) = \{0\ si\ z < 0\ \ 1\ si\ z \geq 0$$

Si *z* que es el resultado de nuestra suma producto es menor que 0 entonces es 0 y si *z* es mayor o igual a 0 entonces se clasifica como un 1, y esta sería la salida de nuestra neurona TLU.

Ya te estarás imaginando que una neurona artificial tipo TLU puede ser utilizada para clasificación binaria, ya que, al introducir datos, esta los clasifica entre 1 y 0.

En este caso entrenar un modelo TLU consiste en encontrar los valores de los pesos W_n adecuados.

Pues un perceptrón está simplemente compuesto de una sola capa de neuronas TLU conectadas a todas las entradas.

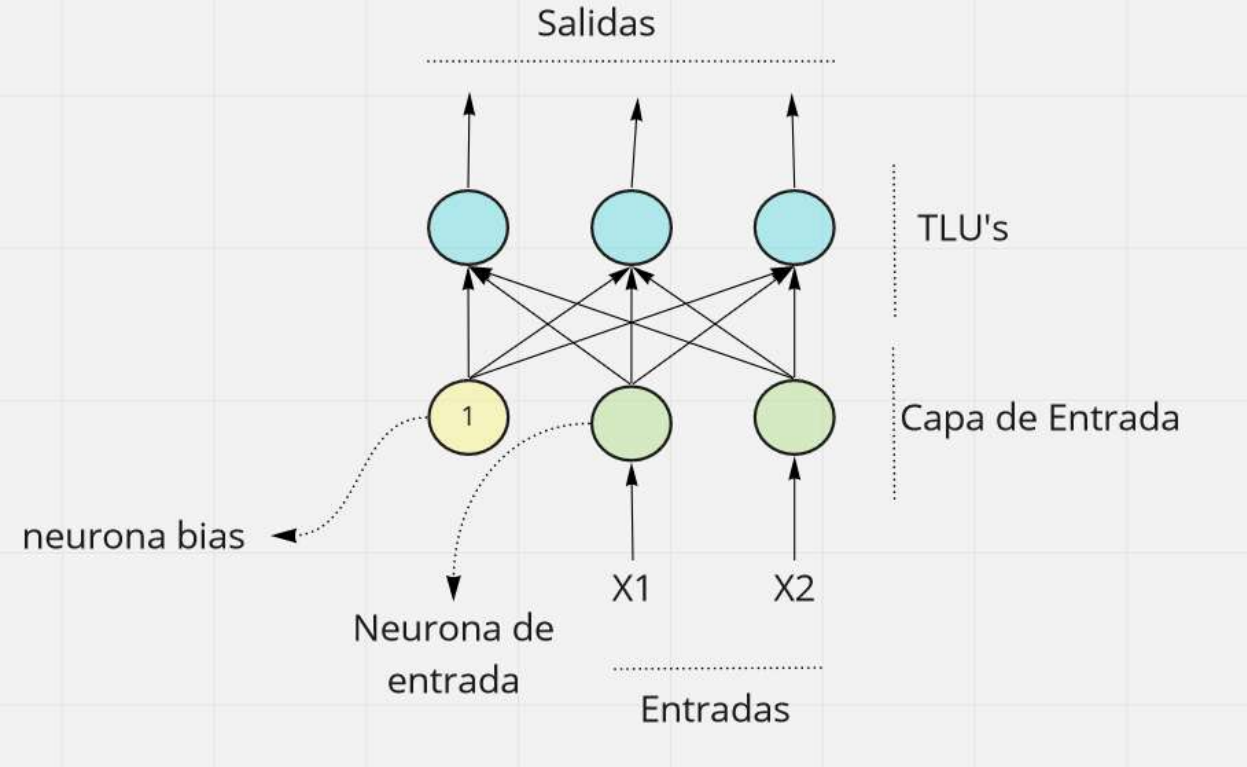
Cuando todas las neuronas de una capa están conectadas a todas las neuronas de la capa anterior, a la capa se le conoce como capa completamente conectada, o *capa densa*.

Las entradas de un perceptrón pasan por un tipo especial de neuronas conocidas como neuronas de entrada, estas neuronas tienen una salida la cual muestra cualquier entrada que se les haya asignado, por lo que todas las neuronas de entrada forman la conocida capa de entrada.

Los perceptrones también cuentan con una neurona conocida como la neurona de bias, que si recordamos el bias es un parámetro que normalmente acompaña a las funciones de peso W_n y en este caso el bias siempre será igual a 1.

Ejemplo:

Un perceptrón con 3 entradas y 3 salidas se vería de la siguiente manera.



En este caso este perceptrón clasifica las entradas de datos en 3 neuronas TLU diferentes, las cuales como ya vimos son clasificadores binarios, por lo que este perceptrón es capaz de clasificar en 3 clases diferentes haciéndolo un clasificador multinivel.

Como vemos tenemos nuestra capa de entrada, donde cierta cantidad de datos son introducidos, estos después son transformados por las neuronas en la capa TLU, y al final son clasificados.

De nuevo gracias al Álgebra Lineal, es posible computar este tipo de modelo de manera eficiente donde para calcular las salidas del perceptrón tenemos que:

$$Salidas = \phi(XW + b)$$

Donde:

- *X* es nuestra matriz de valores de entrada, o el set de datos de entrada.
- *W* es la matriz de pesos, la cual es la matriz que deseamos optimizar al final como hemos visto en varios modelos anteriormente.
- El vector bias es conocido como b y como sabemos es un parámetro que viene en las funciones relacionadas con peso, ejemplo ecuación de recta y = mx + b.

- ϕ Es la función de activación, aunque de esta función hablaremos un poco más adelante.

Te preguntarás ¿cómo es que se entrena un perceptrón? el algoritmo de entrenamiento propuesto por Rosenblatt fue inspirado por la regla de Hebb.

En su libro de 1949 la organización del comportamiento, Donald Hebb sugirió que cuando una neurona biológica activa otra neurona, la conexión de estas 2 neuronas se vuelva más poderosa que la de 2 neuronas que ya estaban activas cuando se conectaron.

Los perceptrones son entrenados utilizando una variante a esta regla, la cual toma en consideración el error generado en una red mientras se genera una predicción.

La regla de aprendizaje del perceptrón incrementa las conexiones que ayudan a reducir este error. En otras palabras, el perceptrón se enfoca más en las neuronas que generan menos error que las otras.

En cada una de las entradas, el perceptrón genera alguna predicción, por cada neurona que generó cierta predicción este incrementa el peso W_n de las neuronas que ayudaron a disminuir el error.

Esto se demuestra con la siguiente ecuación:

$$W_{i,j}^{Siguiente} = w_n + \eta(error)x$$

Donde:

- *W* es el peso de la neurona entrada con relación al peso de la neurona de salida.
- *X* es el valor de entrada.
- Error es la resta de la predicción generada en contra del valor real que tiene esa instancia.
- En este caso también se le asigna un ritmo de aprendizaje conocido como η.

Hay que mencionar que los límites de decisión de cada salida de las neuronas son lineales, por lo que el perceptrón es incapaz de aprender patrones complejos, justo como los clasificadores de regresión logística.

Aunque como ya sabemos, si nuestros datos son linealmente separables, se ha demostrado que este algoritmo podrá converger en una solución bastante buena.

Perceptrón en Scikit

Scikit provee un módulo de perceptrón el cual contiene una red de neuronas TLU. Se puede utilizar justo como hemos utilizado los modelos anteriores.

Si utilizamos como demostración nuestro set de datos de clasificación de candidatos a empleados.

```
candidates = {'gmat':
[780,750,690,710,680,730,690,720,740,690,610,690,710,680,770,610,580,650,540,590,620,600,
550,550,570,670,660,580,650,660,640,620,660,660,680,650,670,580,590,690],
        'gpa':
[4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,3.3,2,2.3,2.7,3,3.3,3.7,2.
3,3.7,3.3,3,2.7,4,3.3,3.3,2.3,2.7,3.3,1.7,3.7],
```

```
        'work_experience':
[3,4,3,5,4,6,1,4,5,1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,5,1,2,1,4,5],
        'admitted': [1,1,0,1,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1]
        }

df = pd.DataFrame(candidates,columns= ['gmat', 'gpa','work_experience','admitted'])
df.head()
```

	gmat	gpa	work_experience	admitted
0	780	4.0	3	1
1	750	3.9	4	1
2	690	3.3	3	0
3	710	3.7	5	1
4	680	3.9	4	0

Y separamos nuestros predictores de la variable a predecir:

```
x = df[['gpa','gmat','work_experience']]
y = df['admitted']
```

Entonces podemos usar el módulo de perceptrón importando la librería perceptrón con la cual entrenaremos los datos justo como lo hemos hecho en ocasiones anteriores.

```
from sklearn.linear_model import Perceptron

per = Perceptron
per.fit(x,y)
```

Para probarlo podemos generar una predicción con datos que queramos introducir.

```
y_pred = per.predict([[4,650,4]])
y_pred
```

```
array([1], dtype=int64)
```

Donde con un GPA de 4, un GMAT de 650 y 4 años de experiencia, el perceptrón clasificó a este empleado como admitido.

Como puedes ya haber notado al ver cómo funciona y se entrena un perceptrón, este se parece mucho a la forma de entrenar un gradiente descendente, ya que intenta ajustar los pesos de los datos conforme va midiendo el error.

De hecho, el utilizar esta librería de perceptrón es equivalente a usar la librería de Scikit de gradiente estocástico.

Hay que notar que, al contrario de la regresión logística, el perceptrón no trabaja utilizando probabilidades de que cada instancia pertenezca a cierta clase, si no que este utiliza un umbral de decisión y se va ajustando con el error, esto es una razón por la que se podría preferir usar la regresión logística en vez del perceptrón.

En su documento de 1969 perceptrones, Marvin Minsky y Saymour Papert resaltaron cierto número de debilidades en los perceptrones.

En particular se dieron cuenta de que el perceptrón fallo en resolver algunos problemas, que aunque algunos otros modelos también fallaron en resolver, se tenía tanta expectativa de los perceptrones que fue una gran decepción que mucha gente decidió detener sus investigaciones en Redes Neuronales y enfocarse en otro tipo de problemas.

Fue un tiempo después cuando se descubrió que muchas de las limitaciones de los perceptrones podrían ser erradicadas agregando múltiples perceptrones en un solo modelo.

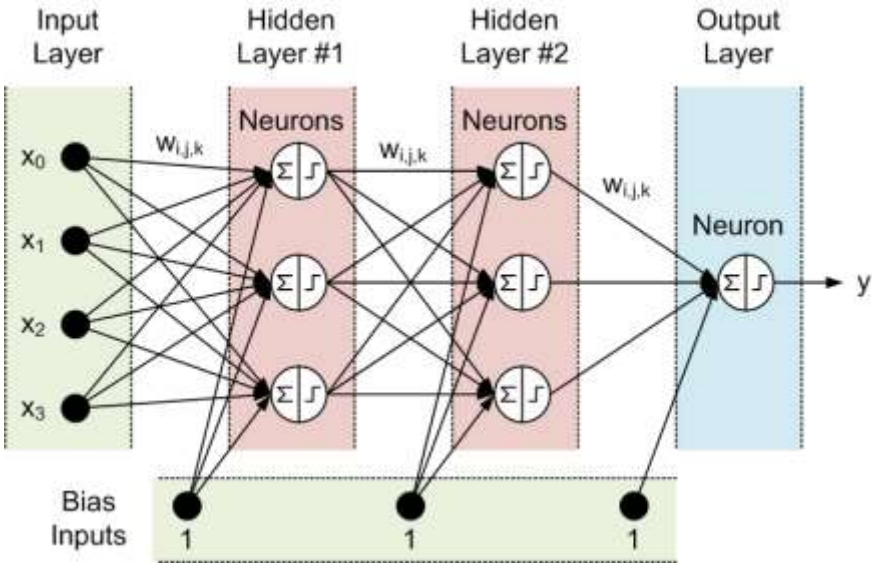
A esto se le llamó perceptrón multicapa, y los resultados fueron tan buenos que este tipo de perceptrón pudo resolver varios de esos problemas que en su momento habían decepcionado a mucha gente por no poder ser resueltos.

Perceptrón Multicapa (MLP) y Backpropagation

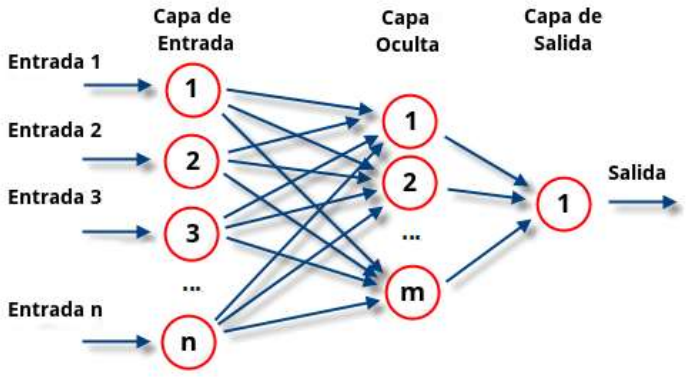
El perceptrón multicapa consiste en una capa de entrada y una o más capas de TLU's, también conocidas como capas escondidas, mientras que la última capa siempre es conocida como capa de salida.

A las capas que están más cerca de la capa de entrada se les conoce como capas bajas mientras que a las capas que están más cerca de la capa de salida se les conoce como capas altas.

Cada capa a excepción de la capa de salida incluye una neurona y está completamente conectada a la siguiente capa.



En esta imagen podemos observar cómo cada serie de capas incluye una neurona de bias, y en la siguiente imagen se puede observar cómo se ve un perceptrón multicapa de manera más simple.



Cuando una red neuronal contiene una cantidad profunda de capas escondidas se le conoce como red neuronal profunda o Deep Neural Network.

Pues el conocido término Deep Learning estudia estas Redes Neuronales profundas.

Por muchos años investigadores intentaron encontrar la forma de entrenar perceptrones multicapa o MLP's pero fallaron.

Fue en 1986 cuando se publicó un documento que introdujo el conocido algoritmo de entrenamiento backpropagation.

En resumen, este algoritmo es un gradiente descendente el cual utiliza una técnica computacional bastante eficiente para computar los gradientes automáticamente.

En solo 2 recorridos de la red neuronal uno hacia adelante y una hacia atrás, backpropagation pudo computar un gradiente para el error en la red, tomando en cuenta cada uno de los parámetros del modelo.

En otras palabras, fue capaz de reconocer en todas las instancias de la red, qué peso y qué bias tenía que ser modificado para reducir el error en toda la red en general.

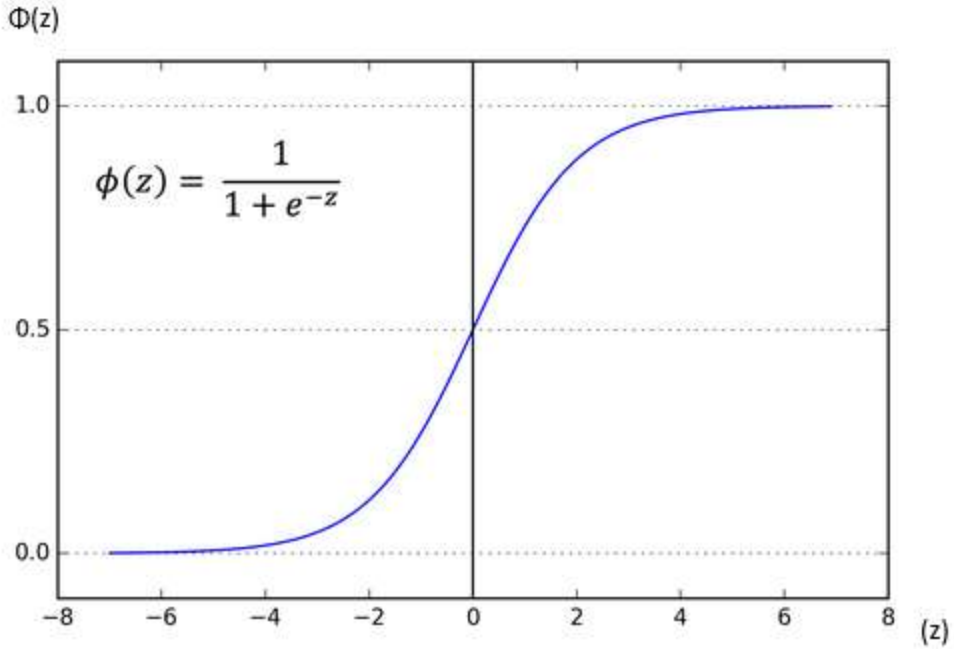
Vamos a hablar sobre este algoritmo con un poco más de detalle:

- Este funciona con un mini-batch a la vez, que como ya sabemos este es un pequeño segmento del set de datos, y al final este a través de muchos mini batches recorre el set de datos completo varias veces, donde cada toma de datos se conoce como epoch.
- Cada mini batch pasa por la capa de entrada, la cual envía los datos a la primera capa escondida, en esta capa cada neurona genera una predicción de salida y el resultado es enviado a la siguiente capa, esto se repite hasta que se llega la capa de salida, a lo se le conoce como paso hacia adelante, mientras que todos los resultados intermedios son guardados ya que serán necesarios para el conocido como paso hacia atrás.
- Después el algoritmo mide el error, comparando las predicciones con el set de datos que utilizamos para el entrenamiento.

- Después este computa de cada salida, cuanto atribuye cada una de estas al error en la red. Esto se hace analíticamente utilizando la regla de la cadena, la cual es muy útil en cálculo diferencial, y ya hemos analizado un poco antes.
- Después el algoritmo mide cuánto de este error fue provocado por cada conexión en las capas de abajo, otra vez utilizando la regla de la cadena. Haciendo esto con el paso hacia atrás hasta que se alcanza la capa de entrada de nuevo.
- Una vez medidas la influencia en el error de las salidas y de las conexiones en las capas, el algoritmo utiliza el sistema del gradiente descendente para modificar los pesos de cada una de sus conexiones y minimizar el error de la red.

Este algoritmo es tan importante actualmente que vale la pena resumirlo una vez más. Por cada instancia de entrenamiento el algoritmo utiliza el paso hacia adelante donde genera predicciones con las diferentes capas, después mide el error de estas predicciones y empieza a ir paso hacia atrás para medir la contribución que cada conexión tiene con el error generado, una vez medidas las contribuciones modifica los pesos de cada conexión en la red, y minimiza su error.

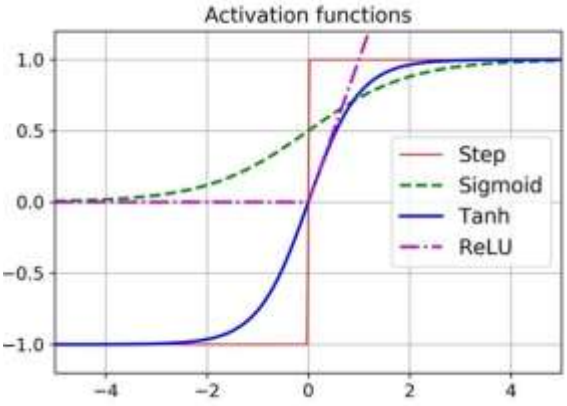
Para que este algoritmo funcionará, se tuvieron que hacer algunos cambios en la arquitectura de los MLP, se reemplazó la función de paso por una función sigmoide igual a la que vimos en la regresión logística $\sigma(z) = \frac{1}{1+e^{-z}}$.



Esto fue fundamental ya que la función de paso contenía sólo segmentos lineales, por lo que no era posible aplicar descenso de gradiente pues como sabemos este no funciona en superficies lineales ya que su derivada sería 0 en todas partes.

De hecho, backpropagation se ha probado ya en diferentes funciones, las que ahora conocemos como funciones de activación y se desarrolla bastante bien.

Las funciones de activación son:



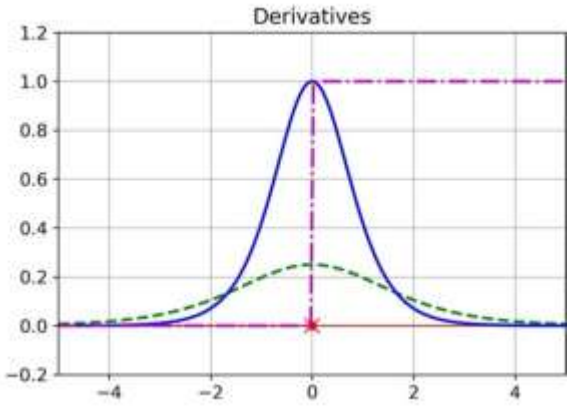
La función de tangente hiperbólica $tanh(z) = 2\sigma(2z) - 1$

Justo como la función logística, está tiene una forma de S y es continua y derivable, la diferencia es que en vez de ir de 0 a 1 esta va de -1 a 1. Este rango hace que las instancias estén menos cerca del 0, lo que ayuda a que el modelo haga medidas de peso más rápidas y acelere su convergencia.

Función Lineal Unitaria Rectificada $RELU(z) = max(0, z)$

Esta función, aunque es continua desafortunadamente no es derivable cuando *z = 0*, y su derivada cuando *z* es menor a 0 es 0 y cuando es mayor sube incrementalmente 1, esto computacionalmente es bastante útil, por lo que en la práctica esta función de activación es de las más utilizadas, por lo que hablaremos nuevamente de ella en el siguiente capítulo.

Aunque para que te des una idea, el gradiente descendente sabemos que trabaja con derivadas, por lo que te muestro una gráfica de las derivadas de cada función.



Como vemos las derivadas de ReLU son mucho más simples, y es por eso que en sets de datos muy grandes, utilizar esta función de activación hace que el código funcione mucho más rápido.

Ahora una vez que sabemos de dónde vienen las Redes Neuronales, como funciona su arquitectura, y cómo interpretar sus salidas, vamos a ver cómo se pueden utilizar.

Regresión con MLP's

Primero, vamos a ver que los MLP's se pueden utilizar para tareas de regresión si solo se quiere predecir una sola salida como el precio de una casa, entonces solo será necesario utilizar una sola neurona, donde su salida será el valor para predecir.

Para predicción con múltiples valores, se necesita una salida de neuronas por cada salida de dimensión.

Por ejemplo, para predecir el centro de una imagen plana 2D, se necesitan predecir coordenadas 2D, por lo que se necesitan 2 neuronas de salida, si fuera una imagen 3D se necesitan 3 neuronas de salida.

En general, cuando se desea realizar tareas de regresión con MLP's no es necesario utilizar funciones de activación para las neuronas de salida, esto para que sean libres de trabajar con cualquier rango de valores.

Si quieres garantizar que el resultado sea siempre positivo entonces puedes utilizar la función ReLU ya que el mínimo de su derivada es 0.

Si quieres asegurar que el resultado estará dentro de un rango de valores puedes utilizar la función logística o la función de tangente hiperbólica y moldear con sus parámetros para que estas queden dentro del rango deseado.

La función de pérdida o error que se recomienda es la del error medio cuadrático MSE, o si se tienen muchos outliers en el set de datos se recomienda usar el error absoluto medio MAE.

En esta tabla se tienen varias recomendaciones a la hora de utilizar la regresión con MLP's.

Hiperparámetros	Valor Normalmente Utilizado
Neuronas de entrada	Una por variable predictora, ejemplo si son 3 variables, 3 neuronas
Capas ocultas	Depende del problema, pero lo común es de 1 a 5.
Neuronas por capa oculta	Depende del problema, pero lo común es de 10 a 100.
Neuronas de salida	1 por cada dimensión de la predicción. Si solo se requiere un valor numérico entonces solo 1 neurona
Activación	Ninguna, aunque ReLU para números positivos o Logistica y TanH para rango de valores.
Función de error	MSE o MAE

Clasificación con MLP's

Los perceptrones multicapa también pueden ser utilizados para tareas de clasificación.

Para un problema de clasificación binario, sólo se necesita una neurona de salida, utilizando la función logística se obtendrá un número de 0 a 1, el cual se puede interpretar como la probabilidad de que cierta instancia pertenezca a dicha clase.

Los MLP's también pueden ejecutar tareas de clasificación multinivel binarias, por ejemplo, se puede clasificar una imagen para saber si es una motocicleta o un automóvil y al mismo tiempo clasificar si la imagen está a color o está en blanco y negro.

En este caso se necesitan 2 neuronas de salida utilizando la función de activación logística. La primera neurona calcularía la probabilidad de que el objeto sea una motocicleta y la segunda calcularía la probabilidad de que la imagen esté a color.

Este tipo de modelos da la entrada a diferentes posibilidades de combinación, por ejemplo, puede haber una imagen que sea de un automóvil y sea a color y una de una motocicleta que sea blanco y negro o una motocicleta que sea a color también etcétera.

En el caso de querer implementar una clasificación multiclase, por ejemplo, para clasificar si la imagen es verde, roja o azul entonces se necesita usar una neurona de salida por clase, en este caso serían 3.

Si recordamos en el capítulo 4 vimos un tipo de función llamada Softmax, el caso de clasificación multiclase se recomienda usar esta función de activación ya que hace que las probabilidades de las 3 neuronas en este caso sean igual a 1.

Por ejemplo, que se tenga una probabilidad de 0.20 que sea azul, 0.80 que sea verde y 0 que sea rojo, para que en total se tenga un 100%.

Para la función de pérdida o de error se recomienda utilizar una conocida como cross entropy o entropía cruzada también conocida como log loss, la cual vimos más a detalle en el capítulo 4.

A continuación, se muestra una tabla con las recomendaciones a la hora de realizar modelos de clasificación con MLP's.

Hiperparámetros	Clasificación Binaria	Clasificación multinivel binaria	Clasificación multiclase
Capas de entrada y ocultas	Las mismas que la regresión	Las mismas que la regresión	Las mismas que la regresión
Neuronas de salida	1	1 por nivel	1 por clase
Activación para capa de salida	Logística	Logística	SoftMax
Función de error	Log Loss	Log Loss	Log Loss

Ahora que conocemos todos estos conceptos es hora de ponerse manos a la obra y empezar a implementarlos con Python.