**Función de activación**

|  |  |
| --- | --- |
| Una función de activación no es más que una **función** que agregamos a las neuronas en una red neuronal artificial con el objetivo de que la red pueda aprender relaciones complejas en los datos.  La imagen muestra el uso de la función de activación en nuestra red neuronal. |  |

Podemos ver de la imagen que la función de activación (también llamada *función de transferencia*, si el rango de salida es limitado se conocen también como *función de aplastamiento*) define cómo la suma ponderada de la entrada (con pesos) se transforma en una salida de un nodo o nodos en una capa de la red.

* Una red neuronal tiene 3 tipos de capas: *entrada, oculta y salida*.
* Por lo general, todas las capas ocultas suelen utilizar la misma función de activación.
* La capa de salida normalmente utiliza una función de activación distinta y depende del tipo de problema de aprendizaje supervisado (regresión, clasificación, …).

Las funciones de activación deben ser diferenciables. Esto es necesario dado que las redes neuronales se entrenan generalmente utilizando el algoritmo [backpropagation](https://en.wikipedia.org/wiki/Backpropagation) y se requiere la derivada de primer orden del error de predicción para actualizar los pesos del modelo.

**¿Si no usamos una función de activación?**

El uso de una función de activación introduce un paso adicional en cada capa en nuestra red neuronal aumentando la complejidad. Sin embargo, esta función es muy importante.

Imaginemos una red neuronal que no tenga funciones de activación. En ese caso, cada neurona de nuestra red neuronal solo realizará transformaciones lineales de nuestras entradas usando los pesos y sesgos. Es decir, solo realizará sumas ponderadas.

**Una red neuronal sin una función de activación es prácticamente un modelo de regresión lineal.**

Es por lo que utilizamos una transformación no lineal a las entradas de la neuronal y esta no linealidad se consigue con la función de activación.

**Características**

Es deseable que toda función de activación cumpla estas características:

* [**Fuga de gradiente:**](https://es.wikipedia.org/wiki/Problema_de_desvanecimiento_de_gradiente) las redes neuronales se entrenan utilizando el descenso de gradiente y propagación hacia atrás. Eso quiere decir, que el gradiente de cada capa afecta a la capa anterior. Si el gradiente de la función de activación es cercano a cero, perjudicará el entrenamiento de la red neuronal (los pesos de las capas anteriores no podrán actualizarse correctamente).
* **Centrado en cero:** la función de activación debe ser simétrica en cero, de esta manera, los gradientes no se desplazan hacía una dirección particular.
* **Gasto computacional:** la función de activación se aplican después de cada capa y deben calcularse millones de veces, es por lo que deberían ser económicos computacionalmente.
* **Diferenciable:** como se entrenan las redes neuronales usando el descenso de gradiente, las capas del modelo deben ser diferenciables. Este es un requisito necesario que debe cumplir una función de activación.

**Funciones de activación más conocidas**

**Python**

|  |  |
| --- | --- |
|  | **import** matplotlib.pyplot as plt  **import** numpy as np  **import** tensorflow as tf      **def** graficar\_funcion\_activacion(funcion):  rango\_x **=** np.arange(**-**10, 10, 0.1)  salida **=** funcion(rango\_x)  plt.plot(rango\_x, salida)  plt.grid() |

**R**

|  |  |
| --- | --- |
|  | graficar\_funcion\_activation = **function**(funcion){  rango\_x = seq(-10, 10, 0.1)  salida = funcion(rango\_x)  plot(rango\_x, salida, type = 'l')  } |

**Sigmoide**

La función sigmoide o logística es una de las funciones de activación más utilizadas. Se define de la siguiente manera:

\sigma(z)=\frac{1}{1+e^{-z}}

La gráfica de la función de activación es la siguiente:

Chart, line chart

Description automatically generated

**Características:**

* La función es una curva en forma de S.
* La salida de la función se centra en 0.5, con un rango de 0 a 1
* Usada como capa final en un problema de clasificación binaria.
* Es una función diferenciable.
* Aunque la función es monótona, su derivada no lo es.

**Desventajas:**

* [Fuga de gradiente:](https://en.wikipedia.org/wiki/Vanishing_gradient_problem) cuando las entradas se vuelven pequeñas o grandes, la función se concentra en 0 o 1, trayendo una derivada cercana a 0. Esto trae que no tenga gradiente a propagarse a través de la red, por lo que casi no queda nada para las capas inferiores.
* Costoso computacionalmente: al tener operaciones exponenciales.

**Python**

|  |  |
| --- | --- |
|  | **def** sigmoide(z):  **return** 1 **/** (1 **+** np.exp(**-**z))  sigmoide(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
|  | sigmoide = **function**(z){  return(1 / (1 + exp(-1 \* z)))  }    sigmoide(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.activations.sigmoid(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.sigmoid) |

**Softmax**

Es la forma más generalizada de la función de activación sigmoide. Se utiliza en problemas de clasificación de múltiples clases. Similar a la sigmoide, los elementos del vector de salida están en el rango 0-1, la diferencia es que su suma es igual a 1. Se utiliza como capa final en los modelos de clasificación.

En este caso solo definiremos esta función de activación usando Tensorflow.

**Tensorflow**

|  |  |
| --- | --- |
|  | inputs **=** tf.random.normal(shape**=**(32, 10))  tf.keras.activations.softmax(inputs)    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.softmax) |

**Tangente hiperbólica (Tanh)**

|  |  |
| --- | --- |
| Es otra función de activación muy popular.  tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} | Chart, line chart  Description automatically generated |

**Características:**

* La función también tiene una curva en forma de S, similar a la función de activación logística.
* A diferencia de la función de activación sigmoide, esta función está centrada en 0, con un rango de -1 a 1.
* Similar a la función sigmoide, esta función es diferenciable.
* Similar a la función sigmoide, esta función es monótona, pero su derivada no lo es.

**Desventajas:**

* [Fuga de gradiente:](https://en.wikipedia.org/wiki/Vanishing_gradient_problem) similar a la función sigmoide. Cuando las entradas se vuelven pequeñas o grandes, la función se concentra en 0 o 1, trayendo una derivada cercana a 0. Esto trae que no tenga gradiente a propagarse a través de la red, por lo que casi no queda nada para las capas inferiores.
* Costoso computacionalmente: al tener operaciones exponenciales.

**Python**

|  |  |
| --- | --- |
|  | **def** tanh(z):  **return** (np.exp(z) **-** np.exp(**-**z)) **/** (np.exp(z) **+** np.exp(**-**z))    tanh(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
|  | tanh = **function**(z){  return((exp(z) - exp(-1 \* z)) / (exp(z) + exp(-1 \* z)))  }    tanh(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.activations.tanh(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.tanh) |

**ReLU**

ReLU (Rectified Linear Units) es la función de activación más utilizada en el aprendizaje profundo.

Chart, line chart

Description automatically generated

Ventajas:

* Activación escasa: por ejemplo, en una red inicializada aleatoriamente, alrededor del 50% de las unidades ocultas están activadas (tienen una salida distinto a cero).
* Mejor propagación del gradiente: menos problemas de [fuga de gradiente](https://en.wikipedia.org/wiki/Vanishing_gradient_problem" \t "_blank) en comparación con las funciones de activación sigmoide y thanh.
* Cálculo eficiente: ya que solo es comparación, suma y multiplicación.
* Invariante en escala: max(0,a*x)=a*max(0,x)\text{ para }a\geq0
* Es una función más rápida de cálcular (a diferencia con sigmoide y tanh)

**Desventajas:**

* No está centrado en cero.
* Es diferenciable en cualquier valor, pero no en 0, el valor de la derivada en este número puede elegirse arbitrariamente a ser 0 o 1.
* [ReLU moribundo:](https://en.wikipedia.org/wiki/Rectifier_(neural_networks)) Durante el entrenamiento, algunas neuronas mueren efectivamente, lo que significa que dejan de producir algo que no sea 0. En algunos casos, puede encontrar que la mitad de las neuronas de su red están muertas, especialmente si utilizó una tasa de aprendizaje alta. Una neurona muere cuando sus pesos se modifican de tal manera que la suma ponderada de sus entradas es negativa para todas las instancias del conjunto de entrenamiento. Cuando esto sucede, sigue generando ceros y el descenso del gradiente ya no lo afecta, ya que el gradiente de la función ReLU es 0 cuando su entrada es negativa.
* Explosión de activaciones: ya que no tiene límite superior, es infinito. Esto a veces conduce a nodos inutilizables

**Python**

|  |  |
| --- | --- |
|  | **def** relu(z):  **return** np.maximum(0, z)    relu(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
|  | relu = **function**(z){  return(pmax(0, z))  }    relu(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.activations.relu(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.relu) |

**Leaky ReLU**

Leaky ReLU es una mejora con respecto a la función de activación ReLU. Tiene todas las propiedades de ReLU, además, nunca tendrá el problema de ReLU moribundo. Se define de la siguiente manera:

LeakyReLU(z)=max(\alpha*z, z)

La gráfica de la función de activación es la siguiente para el valor de \alpha=0.1:Chart, line chart

Description automatically generated

Aquí \alpha es un hiperparámetro generalmente establecido en 0.01. La pequeña pendiente en Leaky ReLU asegura que la neurona nunca muera, resolviendo el problema de ReLU moribundo. Si establecemos \alpha=1 entonces Leaky ReLU se convertirá en una función lineal y no será de utilidad. Por lo tanto, el valor de \alpha nunca debe ser igual a 1.

**Python**

|  |  |
| --- | --- |
|  | **def** leaky\_relu(alpha):  **def** activacion(z):  **return** np.maximum(alpha **\*** z, z)  **return** activacion    leaky\_relu(0.1)(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
|  | leaky\_relu = **function**(alpha){  activacion = **function**(z){  return(pmax(alpha \* z, z))  }  return(activacion)  }    leaky\_relu(0.1)(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.layers.LeakyReLU(0.1)(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.layers.LeakyReLU(0.1)) |

**PReLU**

Es una variación de Leaky ReLU, donde \alpha en lugar de ser un hiperparámetro debe ser aprendido durante el entrenamiento.

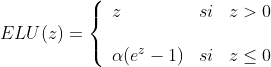
Se ha demostrado empíricamente que en grandes conjuntos de datos PReLU supera con creces a ReLU, pero en conjunto de datos más pequeños se corre el riesgo de sobreajustar el conjunto de entrenamiento.

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.layers.PReLU()(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.layers.PReLU()) |

**ELU (Exponencial Linear Units)**

Esta función de activación es una variante de ReLU que modifica la pendiente de la parte negativa de la función. A diferencia de Leaky ReLU y PReLU, en lugar de una línea recta, ELU usa una curva logarítmica para definir los valores negativos. Se define de la siguiente manera:



La gráfica de la función de activación es la siguiente para el valor de \alpha=0.3:

Chart, line chart

Description automatically generated

* ELU modifica el gradiente de la parte negativa de la función.
* A diferencia de Leaky ReLU y PReLU, en lugar de línea recta, ELU usa una curva logarítmica para valores negativos.
* Según sus [autores](https://arxiv.org/abs/1511.07289" \t "_blank), ELU supera a todas las variantes de ReLU en sus experimentos.

Desventajas:

* Es más lento de calcular que ReLU y sus variantes.

**Python**

|  |  |
| --- | --- |
|  | **def** elu(alpha):  **def** activacion(z):  **def** activacion\_(x):  **if** x &lt;**=** 0:  **return** alpha **\*** (np.exp(x) **-** 1)  **return** x  **return** np.vectorize(activacion\_)(z)  **return** activacion      elu(0.3)(np.arange(**-**10, 10, 0.1)) |

**R**

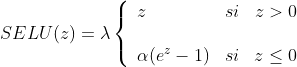
|  |  |
| --- | --- |
|  | elu = **function**(alpha){  activacion = **function**(z){  activation\_ = **function**(x){  if (x &lt;= 0)return(alpha \* (exp(x) - 1))  return(x)  }  return(unlist(lapply(z, activation\_)))  }  }    elu(0.3)(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.layers.ELU(0.3)(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.layers.ELU(0.3)) |

**SELU (Scaled Exponencial Linear Unit)**

Es otra variación de ReLU. Los autores demostraron que si construimos una red neuronal compuesta exclusivamente de capas con función de activación SELU, la red se auto-normalizará (la salida de cada capa tenderá a conservar la media 0 y desviación estándar 1 durante el entrenamiento) resolviendo el problema de fuga y explosión de gradientes. Por lo general, esta función de activación supera a otras funciones de activación. Se define de la siguiente manera:



Donde \alpha=1.6732632\text{ y }\lambda=1.05070098 son constantes predefinidas. La gráfica de la función de activación es la siguiente:

Chart, line chart

Description automatically generated

Desventajas:

* SELU funciona solo para redes neuronales compuestas por capas densas. Puede que no funcione para redes neuronales convolucionales.
* Los pesos de cada capa oculta deben inicializarse mediante la inicialización normal de LeCun.
* Las variables de entrada (inputs) deben estar normalizadas con media 0 y desviación estándar 1.

**Python**

|  |  |
| --- | --- |
|  | **def** selu(z):        alpha\_ **=** 1.67326324      lambda\_ **=** 1.05070098    **def** activacion(x):  **if** x &lt;**=** 0:  **return** lambda\_ **\*** alpha\_ **\*** (np.exp(x) **-** 1)  **return** lambda\_ **\*** x  **return** np.vectorize(activacion)(z)      selu(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
|  | selu = **function**(z){  alpha\_ = 1.67326324  lambda\_ = 1.05070098    activacion = **function**(x){  if (x &lt;= 0)return(lambda\_ \* alpha\_ \* (exp(x) - 1))  return(lambda\_ \* x)  }  return(unlist(lapply(z, activacion)))  }    selu(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.activations.selu(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.selu) |

**Swish**

Propuesto en 2017 por Ramachandran et.al. Se define de la siguiente manera:

Swish(z)=\frac{z}{1+e^{-z}}

La gráfica de la función de activación es la siguiente:

Chart, line chart

Description automatically generated

Tiene un rendimiento ligeramente mejor en comparación con ReLU, ya que su gráfico es similar a ReLU. Sin embargo, dado que la función no cambia abruptamente como lo hace ReLU en x=0, esto facilita la convergencia durante el entrenamiento.

Pero el inconveniente de Swish es que es computacionalmente costoso.

**Python**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **def** swish(z):    **return** z **/** (1 **+** np.exp(**-**z))      swish(np.arange(**-**10, 10, 0.1)) |

**R**

|  |  |
| --- | --- |
| 1  2  3  4  5 | swish = **function**(z){  return(z / (1 + exp(-1 \* z)))  }    swish(seq(-10, 10, 0.1)) |

**Tensorflow**

|  |  |
| --- | --- |
|  | tf.keras.activations.swish(np.arange(**-**10, 10, 0.1))    # Usando como capa  tf.keras.layers.Dense(100, activation**=**tf.keras.activations.swish) |

**Más funciones de activación**

En el siguiente enlace tenemos todas las funciones de activación implementadas en Tensorflow:

<https://www.tensorflow.org/api_docs/python/tf/keras/activations>

Las funciones de activación que dependen de hiperparámetros, algunos de ellos entrenables se encuentran en el siguiente enlace. Aquí podemos encontrar funciones de activación como Leaky Relu, PReLU, etc:

<https://www.tensorflow.org/api_docs/python/tf/keras/layers>

**Cree su propia función de activación en Tensorflow**

Supongamos que queremos implementar la función de activación PreLU. Esta función necesita un hiperparámetro \alpha que se actualiza en el proceso de entrenamiento:

f(z)=max(\alpha*z, z)

Vamos a implementar nuestra función de activación de dos maneras: usando funciones y mediante clases.

**Función**

El método más fácil de crear nuestra función de activación es mediante una función en Python. Esta función necesita dos argumentos: \alpha que indica el factor al que tenemos que multiplicar y z que indica la entrada. La desventaja de usar funciones es que no podemos aprender el hiperparámetro \alpha, por lo que en lugar de PReLU estamos generando Leaky ReLU. La función lo definimos de la siguiente manera:

|  |  |
| --- | --- |
|  | **def** leaky\_relu(alpha):  **def** activacion(z):  **return** tf.math.maximum(alpha **\*** z, z)  **return** activacion |

La función leaky\_relu depende de un hiperparámetro fijo \alpha. Probemos la función de activación con MNIST y para un valor de 0.1. La función de activación lo agregamos en la segunda capa de 128 neuronas:

|  |  |
| --- | --- |
|  | mnist **=** tf.keras.datasets.mnist  (x\_train, y\_train), (x\_test, y\_test) **=** mnist.load\_data()  x\_train, x\_test **=** x\_train **/** 255.0, x\_test **/** 255.0  model **=** tf.keras.models.Sequential([      tf.keras.layers.Flatten(input\_shape**=**(28, 28)),      tf.keras.layers.Dense(128, activation**=**leaky\_relu(0.1)),      tf.keras.layers.Dense(10, activation**=**'softmax')  ])  model.compile(optimizer**=**'adam',                loss**=**'sparse\_categorical\_crossentropy',                metrics**=**['accuracy'])  model.fit(x\_train, y\_train, epochs**=**5)  model.evaluate(x\_test, y\_test) |

Luego de entrenar por 5 épocas, la precisión en el conjunto de test es de aproximadamente 97%.

Si queremos que nuestro hiperparámetro \alpha aprenda en el entrenamiento, debemos usar clases.

**Clase**

Ahora vamos a definir nuestra función de activación pero usando clases. Este método tiene la ventaja de que nuestro hiperparámetro \alpha puede aprender en el proceso de entrenamiento.

Nuestra clase depende de la clase **tf.keras.layers.Layer** de **Keras** ya que haremos uso de los métodos de esta clase y nuestros hiperparámetros pueden aprender.

En el método **\_\_init\_\_** solicitamos un argumento \alpha. Este valor será el valor inicial que tendrá nuestro hiperparámetro alpha\_factor.

En el método build se crea la variable a entrenar alpha\_factor. Vamos a agregar la condición que esta variable no sea negativa solo para esta función de activación (en PReLU original no se cumple esta condición).

Por último en el método **call** definimos la función de activación.

|  |  |
| --- | --- |
|  | **class** PReLU(tf.keras.layers.Layer):      ''' funcion de activacion PReLU '''  **def** \_\_init\_\_(self, alpha):          super(PReLU, self).\_\_init\_\_()          self.alpha **=** alpha    **def** build(self, input\_shape):          ''' Definiendo la variable de factor '''          self.alpha\_factor **=** tf.Variable(self.alpha,                                          dtype**=**tf.float32,                                          constraint**=**tf.keras.constraints.non\_neg(),                                          name**=**'alpha\_factor')    **def** call(self, inputs, mask**=**None):          ''' Aplicando la función de activación PReLU '''  **return** tf.math.maximum(self.alpha\_factor **\*** inputs, inputs) |

Probemos la función de activación con MNIST. La función de activación lo agregamos en la segunda capa de 128 neuronas. He creado el objeto prelu con valor inicial de \alpha=1. Luego del entrenamiento confirmaremos que este hiperparámetro haya actualizado y que cumpla la condición de no ser negativo:

|  |  |
| --- | --- |
|  | mnist **=** tf.keras.datasets.mnist  (x\_train, y\_train), (x\_test, y\_test) **=** mnist.load\_data()  x\_train, x\_test **=** x\_train **/** 255.0, x\_test **/** 255.0  prelu **=** PReLU(alpha**=**1)  model **=** tf.keras.models.Sequential([      tf.keras.layers.Flatten(input\_shape**=**(28, 28)),      tf.keras.layers.Dense(128),      prelu,      tf.keras.layers.Dense(10, activation**=**'softmax')  ])  model.compile(optimizer**=**'adam',                loss**=**'sparse\_categorical\_crossentropy',                metrics**=**['accuracy'])  model.fit(x\_train, y\_train, epochs**=**5)  model.evaluate(x\_test, y\_test) |

Luego de entrenar por 5 épocas, la precisión también es cercana a 97%.

Veamos el valor de la variable alpha\_factor luego de entrenar. Recordemos que antes del entrenamiento su valor era igual a 1:

|  |  |
| --- | --- |
|  | print(prelu.alpha\_factor) |

El valor actual de la variable alpha\_factor es aproximadamente 3.5, cumpliendo la condición de que ha aprendido en el proceso de entrenamiento y cumple la condición de no ser negativo.

**¿Qué función de activación elegir?**

Hasta ahora hemos visto muchas funciones de activación e incluso podemos crear nuestra propia función, pero ¿Qué función de activación usar? Los siguientes consejos pueden ser muy útiles:

* Funciones sigmoide y softmax se recomiendan usar en problemas de clasificación y se añaden en la capa final.
* Las funciones sigmoides y tanh aveces se evitan usar en capas ocultas debido a la fuga de gradiente.
* La función ReLU es un función de activación general y se utiliza en la mayoría de casos actualmente.
* Si encontramos casos de neuronas muertas en nuestras redes, se pueden usar las funciones de activación Leaky ReLU o PReLU.
* La función ReLU solo debe usarse en las capas ocultas.

**Conclusión**

En esta publicación hemos aprendido qué son las funciones de activación, las principales funciones de activación, cómo crear su propia función de activación y consejos a la hora de implementarlos cuando construya su red neuronal.

Espero que este artículo ayude a su aprendizaje en Deep Learning. Cualquier comentario es bienvenido.

Algunos enlaces de interés:

<https://towardsdatascience.com/7-popular-activation-functions-you-should-know-in-deep-learning-and-how-to-use-them-with-keras-and-27b4d838dfe6>

<https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/>

<https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253>