Arquitectura mínima de una Red Neuronal

Alcance de la lectura

- Reconocer los principales componentes de una red neuronal.
- Entender cómo operan los sesgos y pesos en las neuronas.
- Conocer el rol de las funciones de activación y sus variantes.
- Implementar un modelo single layer con Keras.

En la sección de optimización, revisamos los principales mecanismos con los que podemos entrenar un modelo de aprendizaje estadístico para que pueda encontrar los valores de sus parámetros. También vimos la unidad básica que compone a las capas con las que se construyen las redes neuronales,el perceptrón, y exploramos sus ventajas y limitaciones en el caso de la clasificación.

En esta sección estudiaremos el nivel siguiente de abstracción: las capas de una red neuronal, las cuales, como veremos, son concatenadas de forma secuencial y mezcladas de diversas formas para poder formar las redes neuronales clásicas conocidas como redes feed-forward.

Motivación

Hasta el momento conocemos cómo se puede implementar una red neuronal básica cuando tenemos un elemento de entrada y un elemento de salida. Éste se conoce como un perceptrón. Una de las principales virtudes de este marco analítico es la posibilidad de flexibilizar la cantidad de atributos y parámetros a estimar. Por lo generar implementaremos una serie de capas intermedias conocidas como **capas escondidas (hidden layers)** que permiten realizar representaciones precisas de los datos que nuestro modelo ingiere.

Cuando las redes neuronales tienen una capa intermedia, presentan una característica deseable conocida como el **aproximador universal**: pueden aproximarse a **cualquier** función continua. Éste comportamiento mejora en la medida que incorporamos más neuronas en las capas escondidas, a riesgo de incurrir en overfitting del modelo. Dada esta característica, las redes neuronales son capaces de aprender atributos de manera independiente, relegando el involucramiento de investigador a un plano secundario.

Las redes neuronales también pueden aprender **representaciones distribuídas** sobre los datos. Cada neurona dentro de una capa puede capturar un atributo específico, y cada capa escondida puede representar distintos aspectos un conjunto de atributos. Mediante esta característica, los algoritmos pueden aprender no solo el mapeo de atributos a outputs, pero también el cómo se desarrolla el mapeo de datos. (Goodfellow, 2017).

Las capas neuronales y su rol en las Redes Neuronales Artificiales (RNA)

***Digresión**: Antes de comenzar esta lectura, se recomienda re-leer las definiciones dadas en la lectura sobre Tensores y Perceptrones sobre los componentes básicos de una red neuronal.*

Las redes neuronales se encuentran en el corazón de la disciplina de Deep Learning, son modelos que permiten lograr gran versatilidad, escalabilidad y efectividad en tareas de machine learning complejas como el reconocimiento de imágenes, video, audio, etc., la clave. Sin embargo, la principal virtud de las RNA es la capacidad de especificar detalladamente el comportamiento del modelo mediante la definición de una arquitectura neuronal. Por esto hacemos referencia a la configuración de la red en cuanto a cantidad de neuronas y capas, los tipos de capa y las funciones de activación asociadas.

No importa qué tan compleja sea nuestra función de activación, qué tan eficiente sea nuestro optimizador o cuán profunda sea nuestra red, si la combinación de capas no es la indicada para el fenómeno en estudio, obtendremos modelos con mal rendimiento o peor aún, mal rendimiento + largo tiempo de entrenamiento. Existen una serie de buenas prácticas respecto a cómo definir la arquitectura. Nuestra primera tarea es definir la cantidad de capas y la cantidad de neuronas contenidas en cada una.

Una capa neuronal, como mencionamos en la unidad anterior, puede ser catalogada en tres tipos principales:

- **Input layer**: Capa de entrada que representa el vector de datos de entrada.
- **Hidden layer**: Capa intermedia entre el input y output.
- **Output layer**: Es la capa que representa la salida de la capa hacia capas subsiguientes o como respuesta de la red neuronal.

Comencemos por revisitar un par de estructuras básicas:

Single-Layer perceptron (Perceptrón unicapa)

Recordemos la estructura de nuestro perceptrón de la sección anterior:

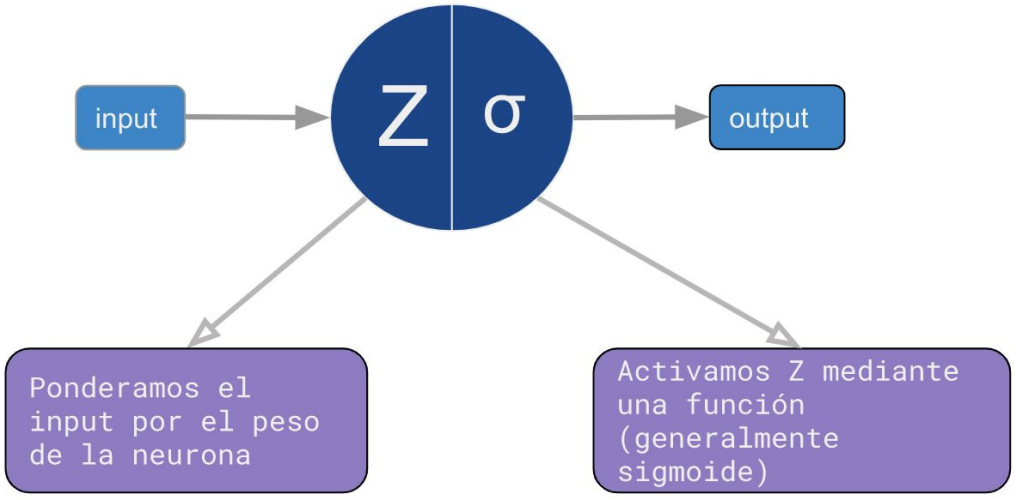


Imagen 1. Estructura del perceptrón.

En nuestro perceptrón básico, un cierto input es ingresado a la neurona posterior a su ponderación por el peso. Una vez ponderado, es pasado por una **función de activación** antes de ser arrojado como output. Ahora llevaremos nuestro modelo al siguiente paso, veremos cómo se conforma una capa neuronal básica y a partir de esta podremos conformar una red.

Un aspecto a considerar es el hecho que este es el comportamiento básico de una neurona dentro de una arquitectura neuronal. Si aumentamos la abstracción y cantidad de neuronas, generamos una capa neuronal. Ésta es un arreglo de neuronas, donde todas presentan una misma función de activación (aunque se puede especificar funciones distintas para cada neuronal). Hay que considerar que si bien definimos una función de activación para todas las neuronas, **los pesos de cada neurona no serán iguales**. Este se conoce como un *Single Layer Perceptron*: Una configuración de perceptrones ordenados de forma paralela.

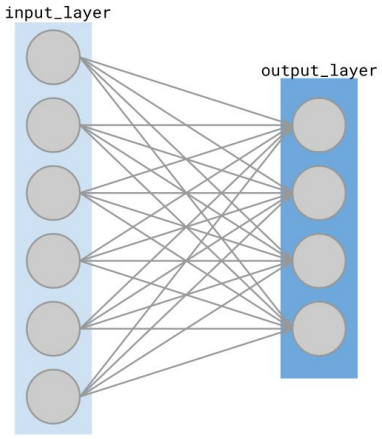


Imagen 2. Input-Output.

lti-layer perceptron (Perceptrón multicapa)Mu

Iteremos en el mismo ejercicio, ahora concatenando múltiples capas. Cuando agregamos una capa entre el input y el output, estamos implementando una capa oculta (o Hidden Layer). Cuando implementamos una capa oculta, se agrega una neurona adicional conocida como Sesgo. En la imágen de abajo se representa como b. Posteriormente revisaremos el rol del sesgo en la activación de neuronas.

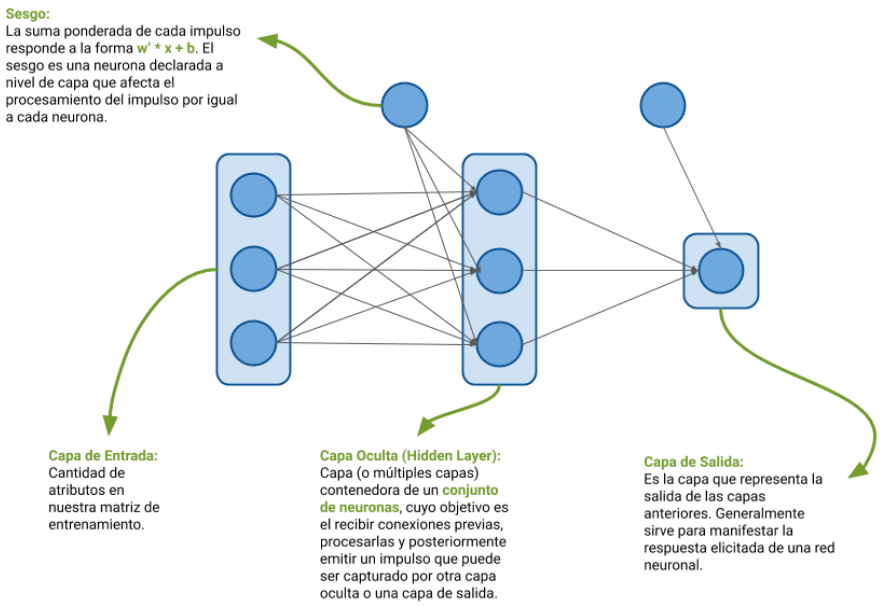


Imagen 3. lti-layer perceptron.

Por lo general, las capas están densamente conectadas, es decir, todas las neuronas de la capa anterior están conectadas a cada una de las neuronas de la capa siguiente.

Digresión: Refresher en la estructuración de una Red Neuronal con Keras

Recuerden que debemos importar tres elementos para definir la arquitectura de una red neuronal con Keras: El modelo secuencial de capas (que define el flujo y permite la concatenación de múltiples capas) con `keras.models.Sequential`. La definición de capas completamente conectadas con `keras.layers.Dense` y finalmente un método de optimización, en este caso será el método de Descenso de Gradiente Estocástico `keras.optimizer.SGD`.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizer import SGD
```

Para implementar nuestra arquitectura básica de un Multilayer Perceptron, debemos instanciar un objeto creado con Sequential. Con este vamos a poder concatenar múltiples capas.

```
model = Sequential()
```

Posteriormente, debemos asignar una capa con el método `add` de nuestro objeto instanciado. En esta parte definimos las características de las capas: *Cantidad de Neuronas, Dimensiones* del Input (ésta sólo es necesaria para la primera capa) y función de activación. En este caso generaremos una capa con 10 neuronas, que recibe vectores y con activación RELU. Cabe mencionar que es una buena práctica el asignarle un nombre a nuestra capa creada, para facilitar la identificación en la estructura.

```
model.add(Dense(10, input_dim=5, activaction='relu', name='hidden1'))
```

Si deseamos agregar otra capa a nuestra arquitectura, implementamos una sintáxis similar, sólo que sin la necesidad de declarar las dimensiones de entrada, dado que serán inferidas en base a la cantidad de neuronas de la capa previa.

```
model.add(Dense(10, activation='relu', name='hidden2'))
```

También se debe agregar la definición de la capa de egreso. En este caso imaginemos que trabajamos con solo una neurona de salida, que representa un problema de clasificación. Para ello vamos a implementar una definición donde existe una neurona, que se alimenta de la cantidad de neuronas declaradas en la capa previa y con una activación sigmoide.

```
model.add(Dense(1, activation='sigmoid', name='output'))
```

Al final, debemos compilar y generar el fit del modelo. En esta parte ingresamos nuestro método de optimización, así como nuestra función de pérdida y métricas asociadas.

```
model.compile(optimizer = SGD(lr = 1), loss = 'binary_crossentropy',
metrics = ["accuracy"])
```

El fit funciona de una manera similar a cómo funciona en scikit-learn. De manera adicional ingresamos la cantidad de épocas a iterar así como el batch size.

```
model.fit(X, y, epochs=50, batch_size=100, verbose=0)
```

Convenciones en la arquitectura de una red neuronal

Karpathy sugiere una serie de convenciones al momento de describir y reportar a otros la arquitectura de nuestra red neuronal.

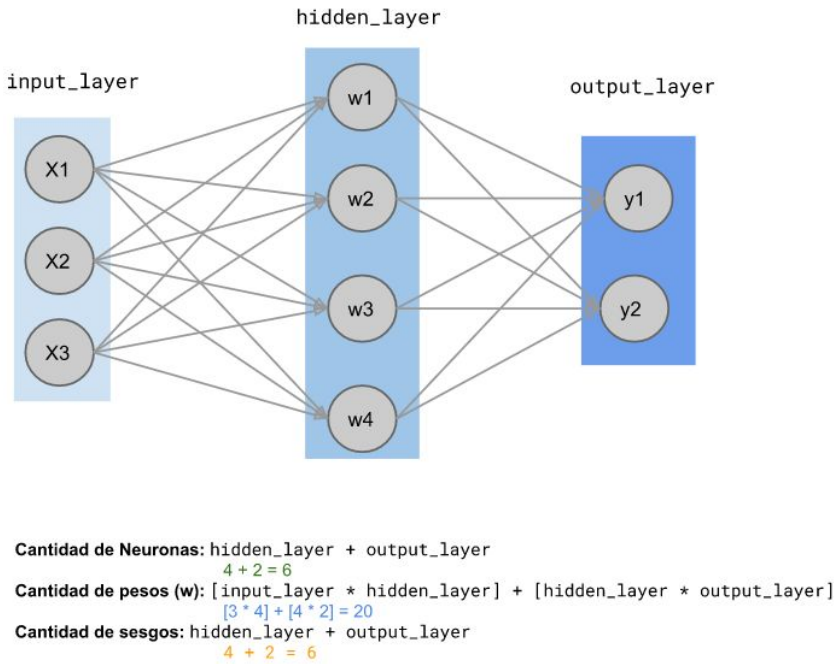


Imagen 4. Input-Hidden-Output.

- **Nombre de nuestra arquitectura**: Cuando reportarmos la arquitectura mínima de nuestra red neuronal, no se considera la capa de entrada. La siguiente red neuronal tendrá 2 capas, una escondida y una de salida.

- **Dimensionado de nuestra arquitectura**: A parte de mencionar a grandes rasgos la cantidad de capas existentes en nuestra red neuronal, es meritorio mencionar cuántas neuronas, pesos y sesgos existen en el modelo. Ésta parte es relevante al reportar dado que el tiempo de entrenamiento y mecánicas de regularización dependerán de la cantidad de parámetros estimables. Estimemos cuántos parámetros debemos estimar en una red con 3 neuronas de ingreso, 4 neuronas en la capa intermedia y 2 en la capa de salida.

El rol del sesgo en una capa neuronal

El sesgo de una capa es un valor constante (no depende de lo que hayan procesado las capas anteriores) que se incorpora a una capa con el objetivo de darle flexibilidad a los valores posibles del output de la capa.

Primero veamos un ejemplo de como funciona el **peso** de una conexión. Por simplicidad veremos el caso del Perceptron Unineuronal con una activación sigmoidal. Recordemos el comportamiento del perceptrón: En base a un input resultante de la multiplicación del peso por el valor, la neurona entregará un output pasado por la función sigmoidal.

$$z = \sigma(w_i + b_{capa})$$
$$Output = FuncionActivacion(Pesos \cdot Sesgo)$$

Comprobemos cómo afecta el peso que multiplica al input sobre el output del perceptrón:

```
import warnings
warnings.filterwarnings(action = 'ignore')
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import lec13_graphs as afx
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (10, 6)
```

```
Using TensorFlow backend.
```

```
afx.weight_bias_behavior()
```

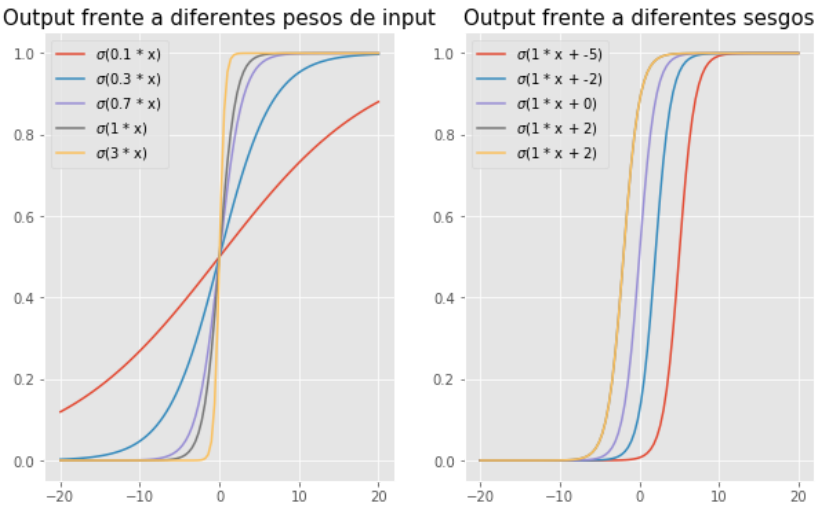


Imagen 5. Output.

La figura creada con `afx.weight_bias_behavior` demuestra dos comportamientos claves a entender:

- Los pesos controlan la inclinación/pendiente de la curva.
- Los sesgos controlan la traslación del output.

Es importante el sesgo de una capa porque le permite al output de esa capa llegar a condiciones que pueden ser necesarias para el problema.

Funciones de activación

Cuando hablamos de función de activación debemos tener claro que esta es una característica de las neuronas. Sin embargo, es común que todas las neuronas de una misma capa tengan la misma función de activación, por lo que en general se refiere a "la función de activación de la capa", lo cual, junto con la cantidad de neuronas de la capa y los pesos definen la arquitectura mínima de una red neuronal.

Elegir una correcta función de activación es crucial para el proceso de aprendizaje de la red. La potencia de las redes neuronales para resolver problemas complejos depende de su capacidad de encontrar variables latentes. Por ello es vital poder superar la linealidad del Perceptrón. Las redes neuronales escapan de las limitaciones lineales del perceptrón mediante la **concatenación de sucesivas capas con función de activación no lineal**, por lo tanto, las funciones de activación que veremos a continuación se basan en esta premisa: **implementar no linealidad al input.**

1. Sigmoidal

Una neurona con activación sigmoidal es similar a un perceptrón, sin embargo, mientras un perceptrón tiene un output binario (0 o 1) una neurona sigmoidal define un output en un rango continuo calculado según la función sigmoidal (también conocida como logística):

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Donde z corresponde a la ponderación de los inputs según el peso se asignado por la capa a dicha conexión, agregado al bias de la capa: $z = w \cdot x + b$.

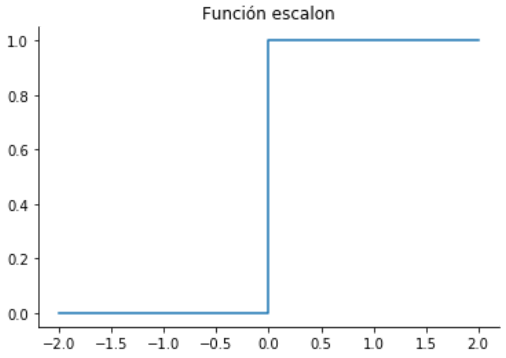


Imagen 6. Escalon.

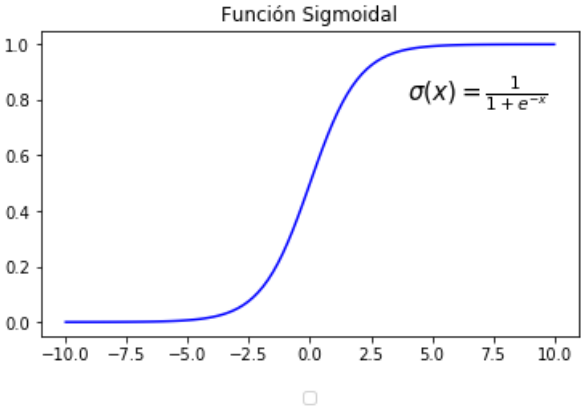


Imagen 7. Sigmoidal

La función sigmoidal tiene la particularidad de que nos permite interpretar el output de las capas anteriores de una red como si fueran probabilidades (por siempre arrojar valores entre 0 y 1), esta característica no se permitirá utilizar redes en problemas de clasificación.

Problemas asociados a la activación sigmoidal

La función sigmoidal presenta problemas serios al momento de utilizarla en redes neuronales:

- Dado las características morfológicas de la función, las gradientes que utilizabamos para actualizar los parámetros en SGD decrecen en magnitud. Esto conlleva a que las capas que se implementen con una función de activación sigmoidal demorarán más en aprender la representación de los datos, condicional al número de épocas de entrenamiento. Esto se conoce como el **problema de las gradientes desvanecientes.**

- Por otro lado, los valores entregados por la función sigmoidal no están centrados en cero. Esto presenta problemas al entrenar las capas de la red neuronal dado que en el caso que el input de la capa sea siempre positivo, el gradiente de los pesos será siempre positivo o siempre negativo lo cual puede generar inestabilidad en la actualización de los pesos.

Estos problemas, en especial el de los gradientes desvanecientes, hacen que utilizar la función sigmoidal como función de activación para capas ocultas en redes profundas sea una mala idea.

Digresión: Swamping

En computación científica existe un fenómeno para nada trivial llamado swamping y que tiene directa relación con las gradientes desvanecientes. Imaginemos que la memoria de nuestro sistema, dado su tamaño, solo puede representar números con un cierto rango de precisión y magnitud (mientras más grande o más decimales tenga un número, más espacio se necesita para almacenarlo).

¿Qué sucede cuando tenemos números demasiado precisos?: El sistema no es capaz de representarlos, por lo que los aproxima al número más cercano que sí puede representar. En el caso de números en el rango **[0,1]** demasiado pequeños, estos son aproximados a **0**, a este fenómeno se le conoce como swamping.

Recordemos la fórmula de actualización de parámetros en SGD: $\widehat{\theta_i^j} \leftarrow \widehat{\theta_i^{j-1}} - \alpha \nabla \theta J(\theta; x^{(i)}; y^{(i)})$.

La clave para encontrar los valores de los parámetros está en el gradiente de la función de pérdida $\nabla_\theta J(\theta)$ Por lo tanto, si las gradientes se hacen tan pequeños que ocurre swamping los parámetros se dejarán de actualizar y el modelo no aprenderá más! Respecto a las gradientes desvanecientes, no es necesario que ocurra swamping para comenzar a provocar problemas en el aprendizaje, basta cualquier número cercano a **0** para observar estancamientos en el entrenamiento.

Finalmente, en la catastrófica situación que el número sea demasiado grande en magnitud, este es representado con una codificación especial que representa ∞ o $-\infty$ lo cual es desastroso si estamos realizando cálculos debido a que no es algo que podamos tratar matemáticamente. Si el gradiente nos llega a dar ∞ entonces el valor del parámetro explota, al fenómeno en el cual los gradientes se hacen demasiado grandes, sin necesariamente llegar ∞ se le conoce como **gradientes explosivos**.

2. Tangente hiperbólica

Una alternativa para evitar el problema de la función sigmoidal de no tener un output centrado en cero es utilizar la función tangente hiperbólica, la cual entrega valores en el rango **[-1,1]**.

$$\tanh(x) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

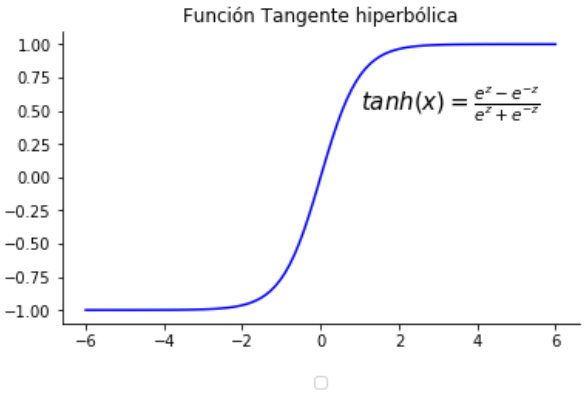


Imagen 8. Tangente.

Lamentablemente la función tangente hiperbólica, debido a su morfología, también sufre del problema de los gradientes desvanecientes.

3. ReLU (Rectified Linear Unit)

ReLU permite **amortizar** el problema de los gradientes desvanecientes gracias a que sus valores no se saturan llegando a un cierto máximo. Obviamente esto elimina la posibilidad de utilizarla en una interpretación directa de la probabilidad. Lamentablemente, ReLU presenta otro problema similar a las gradientes desvanecientes dado a su imagen en el dominio de $[-\infty, 0]$ que hace que el gradiente sea igual a cero en todo ese intervalo. Para solucionar esto, hay variaciones de ReLU que no veremos en este curso por cuestiones de tiempo.

$$f(x) = max(0, x)$$

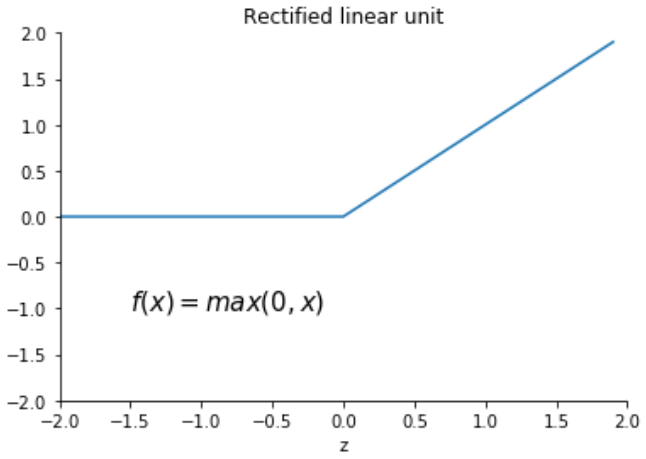


Imagen 9. Rectified linear.

4. Softmax

Esta función es usada por lo general en la capa de salida de la red. Al igual que la función sigmoidal tiene la particularidad de entregar valores positivos que suman 1 (Dado la propiedad aditiva del Axioma de Kolmogorov). Por esto es utilizada como una interpretación directa de la probabilidad en clasificación.

$$\sigma(z)_j = \frac{\exp(zj)}{\Sigma_{k-1}^{K} \exp(zk)}$$
$$\sigma : \mathbb{R}^K \rightarrow [0,1]^K$$

Categorical cross-entropy loss

Para poder evaluar el rendimiento de la red bajo el output entregado por la función softmax, debemos utilizar una función de costo adecuada, dicha función es la cross-entropy loss: $L_i = -\Sigma_j t_i, jLog(pi, j)$.

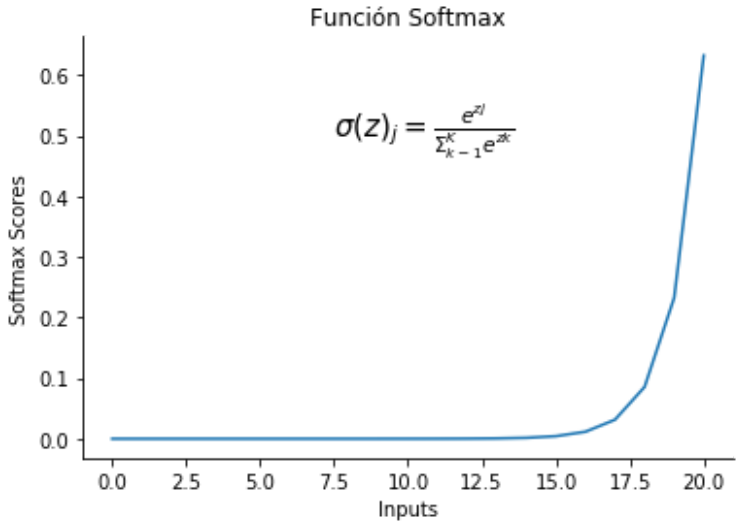


Imagen 10. Función Softmax

Con respecto a la función de costo a utilizar con softmax, es la misma que para la función sigmoidal: cross-entropy.

Un aspecto que quizás haya generado dudas en este punto es la similitud de las funciones softmax y sigmoidal en cuanto a su utilización/interpretación al ser usadas en la red. La razón de esto es que hay pequeñas diferencias sutiles:

Softmax es básicamente una generalización de sigmoidal, puesto que la salida de softmax suma 1, distribuye la probabilidad en cada neurona de salida. Por tanto, si asignamos a cada neurona una clase en un problema de clasificación, una salida softmax nos permite obtener la probabilidad de pertenecia a cada clase. En sigmoidal por otro lado, las probabilidades no necesariamente suman 1 debido a que sigmoidal no modela la probabilidad de una clase de forma dependiente de otra.

En el caso de que estemos trabajando con clasificación binaria, podemos comprobar que sigmoidal y softmax tienen el mismo comportamiento:

```
afx.softmax_sigmoid_behavior()
```

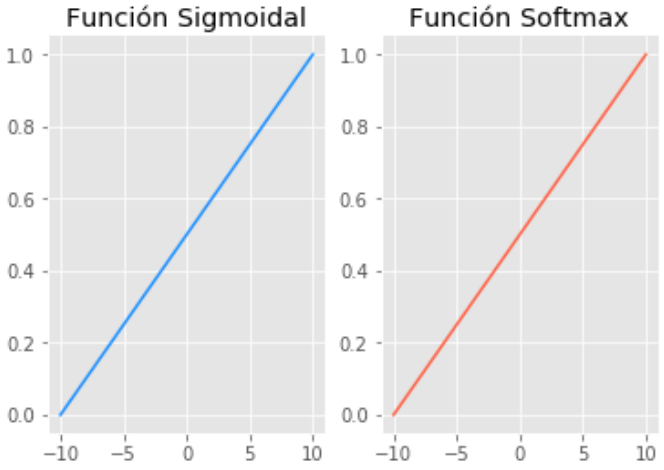


Imagen 11. Funciones.

Pesos de la capa

Antes de implementar nuestra red neuronal con Keras, hay que mencionar la importancia y complejidad de inicializar los pesos de cada capa. Si elegimos pesos que están lejos de un óptimo local o en una mala región del espacio de búsqueda, el entrenamiento será ineficiente en términos de optimización y en tiempo de ejecución. Una opción es inicializar todos los pesos de una red a lo largo de todas las capas con una constante arbitraria. Posteriormente la red se encargará de actualizarlos en cada época de entrenamiento. Esta no es una buena práctica, dado que estaremos asignando un valor idéntico para todas las neuronas dentro de todas las capas, lo que conlleva a que se refuerce una representación específica de los datos, ignorando las demás.

Entonces nuestra pregunta es cómo inicializar los pesos. Resulta que hay una infinidad de posibles valores a escoger para cada uno de los pesos $w_{\varepsilon}[-\infty, \infty]$

Dado que es problemático escoger pesos altos, una primera solución es escoger pesos en un rango pequeño $w_{\varepsilon}[0, 1]$ siguiendo una distribución uniforme. El problema asociado a esta alternativa es que sesgamos que los valores sean todos positivos, situación donde podemos representar erróneamente los datos. Otra opción es inicializar los pesos siguiendo una distribución normal estandarizada $Normal\,(0, 1)$ escalada por algún factor que fuerce pasos pequeños, de forma tal que $w_i \sim Normal\,(0, 1)$ y su esperanza es $E[w_i] = 0$.

Al confirmar que la esperanza del peso es cercana a cero, nos aseguramos que no existan sesgos asociados en la red y que ninguna neurona presente pesos sustancialmente mayores al resto. Esta técnica se conoce como **symmetry breaking**.

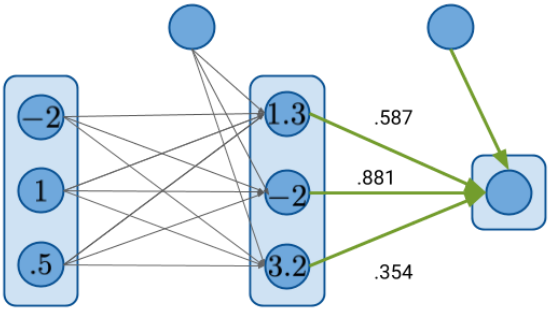
Evaluando el impulso de una red neuronal

Sabemos los elementos que componen una red neuronal, pero no tenemos conocimiento sobre cómo operan éstas en la predicción de observaciones. Partamos desde el siguiente supuesto: **los pesos nos son dados**, por tanto ignoraremos el proceso de entrenamiento y aprendizaje de la red neuronal. Cabe destacar que posteriormente hablaremos sobre cómo aprenden este tipo de modelos.

Desde el punto de inicio más básico, una red neuronal busca procesar una serie de datos mediante impulsos, y estos ser convertidos mediante la forma $\sigma(w^T x_i)$ que es la activación de la combinación lineal de parámetros. Un aspecto a considerar es que los datos de un ejemplo a evaluar sólo interactúan con la primera capa oculta.

En el diagrama se ejemplifica el efecto entre la última capa oculta y la capa de egreso. El proceso implica ponderar cada uno de los pasos anteriores. En este ejemplo, ponderamos los impulsos por los pesos `[1.3, -2, 3.2]` de la capa previa, los cuales son sumados más un término constante que representa el sesgo a nivel de capa de salida. Una vez que tenemos esta suma ponderada, se debe procesar mediante una función de activación.

Cabe destacar que en este diagrama emulamos el comportamiento en la capa de salida, por lo que posterior a la emisión del impulso en la capa de salida, el modelo debe realizar una decisión. Si estamos interesados en un problema de clasificación binaria, podemos evaluar este valor en función a un criterio de corte. Si estamos en un problema de multiclases, podemos incluir tantas neuronas como clases existan, y evaluar mediante máximo a posteriori.



$$z_4 = \sigma\Big(0.587 + 0.881 + 0.354 + 1\Big) = \sigma(2.822) = 0.9438$$

Imagen 12. Flux.

Ejemplo: Modelando no linealidades con un perceptron multicapas

Retomaremos el problema que tratamos de atacar con el perceptrón simple en la unidad anterior: ¿Cómo poder modelar casos donde los datos presentan no-linealidad? Para ello implementaremos una serie de simulaciones para ver cuál es el efecto de los parámetros estudiados en esta sesión: La cantidad de neuronas en una capa y la cantidad de capas en la arquitectura original

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
# importamos las librerías asociadas con redes neuronales
import tensorflow as tf
import keras
from keras import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
# generamos datos
X, y = make_moons(n_samples=500, random_state=11238, noise=.15)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33,
random_state =11238)
```

```
for i in np.unique(y_train):
    plt.scatter(X_train[y_train == i][:, 0],
                X_train[y_train == i][:, 1],
                label="Clase: {}".format(i))
    plt.legend()
```

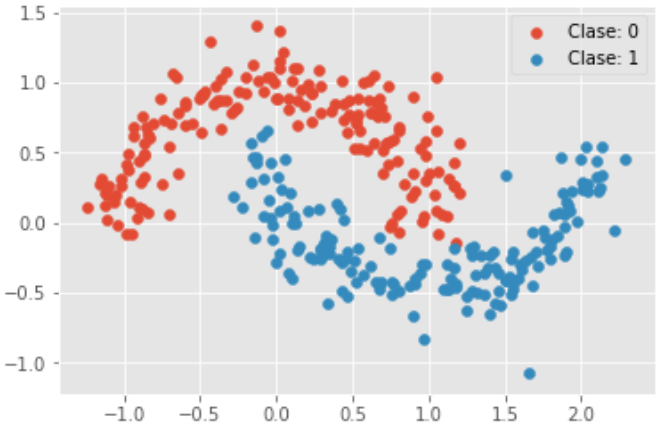


Imagen 13. Gráfico clase.

Cantidad de neuronas en una capa

El primer comportamiento a evaluar es la cantidad de neuronas en una capa. Para ello nuestra simulación considerará sólo una capa donde se incorporará neuronas de forma secuencial La capa de egreso se configura con 1 neurona y función de activación "sigmoid". La optimización se realiza con Descenso de Gradiente Estocástica, donde su learning rate $a = 1$ y una función de pérdida `"binary_crossentropy"`.

El siguiente código simula este comportamiento. A grandes rasgos se aprecia que en la medida que la cantidad de neuronas aumenta, la precisión y ajuste de la superficie de respuestas para mapear correctamente la representación de los datos mejora. Aún así, es meritorio considerar el caso cuando la cantidad de neuronas en una capa es sustancialmente grande.

Una buena regla de selección de cantidad de neuronas en una capa es guiarse por la cantidad de atributos. Éste número debe situarse entre 1 y la cantidad de atributos. Otra opción más conservadora es restarle la cantidad de outputs en la capa de egreso.

```
def ann_neurons_number(X_train=X_train, y_train=y_train, X_test = None,
y_test = None, n_neurons=12):

    """
    ann_neurons_number: train a neural net with n defined neurons
    """
    # Set sequential canvas
    tmp_model = Sequential()
    # add a fully connected layer
    tmp_model.add(
        # with user defined neurons
        Dense(n_neurons,
              # input layer
              input_dim = X_train.shape[1],
              # weights are randomly initialized following glorot normal
              kernel_initializer = 'glorot_normal',
              # weighted sum is relu activated
              activation='relu',
              name = 'hidden1')
    )
    # add a fully connected layer
    tmp_model.add(
        # with 1 neuron and glorot normal initialization
```

```
        Dense(1, kernel_initializer='glorot_normal',
              # weighted sum is sigmoid activated
              activation='sigmoid',
              name = 'output')
    )
    # arquitecture is compiled
    tmp_model.compile(
        # following a SGD optimizer
        optimizer = SGD(lr=1),
        # defined loss function is binary crossentropy
        loss='binary_crossentropy',
        # evaluation metric
        metrics=['accuracy']
    )
    # train model
    tmp_model.fit(X_train, y_train,
                  epochs=50, batch_size=100,
                  verbose=0)
    # return model
    return tmp_model
```

```
plt.figure(figsize=(8, 12))
x_mesh, y_mesh, joint_xy = afx.get_joint_xy(X_train, y_train)
for index, value in enumerate([5, 10, 20, 40, 50, 100]):
    plt.subplot(3, 2, index + 1)
    tmp_neuron_number = afx.ann_neurons_number(X_train=X_train,
y_train=y_train,  n_neurons=value)
    afx.plot_response_surface(tmp_neuron_number, X_train, y_train,
x_mesh, y_mesh, joint_xy)
    plt.title("Number of neurons: {}\nAccuracy: {} - Loss: {}".format(
            value,
            np.mean(tmp_neuron_number.history.history['acc']).round(3),

np.mean(tmp_neuron_number.history.history['loss']).round(3)))
    plt.tight_layout()
```

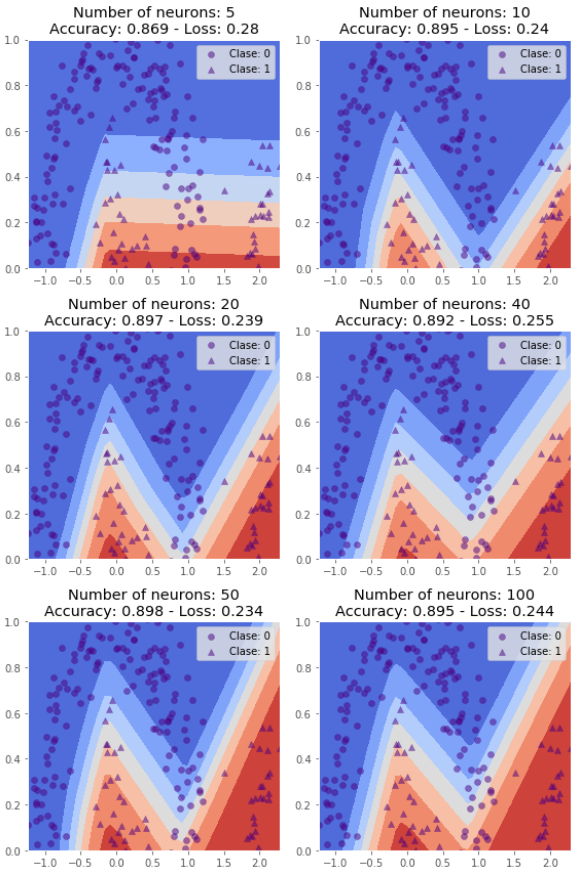


Imagen 14. Number of neurons.

Aspectos adicionales: Cantidad de capas escondidas con 20 neuronas cada una.

En la próxima unidad profundizaremos sobre el comportamiento de nuestras redes neuronales cuando aumentamos la cantidad de capas escondidas. Por ahora, observamos el comportamiento de las redes bajo un experimento adicional.

Caveat: Por razones expositivas parte substancial del código de esta figura se excluye. Pueden visitar el archivo auxiliar `lec13_graphs` para ver el detalle de la construcción de las redes con la función `ann_number_of_layers.`

En esta simulación evaluaremos el fit en la superficie de respuesta de los datos. Evaluaremos la exactitud general así como el puntaje de pérdida cuando incrementamos las cantidad de capas ocultas con una configuración idéntica (12 neuronas cada una, con inicialización `"glorot_normal"` y función de activación `'relu'`) y una capa de egreso con 1 neurona y función de activación `"sigmoid"`. La optimización se realiza con Descenso de Gradiente Estocástica, donde su learning rate $a = 1$ y una función de pérdida `"binary_crossentropy"`.

Quizás el efecto de la cantidad de capas sea más pernicioso para el ajuste general del modelo. Cuando el modelo tiene un número conservador de capas entre 1 y 3, la red neuronal parece aprender de buena manera la representación de los datos. Cuando aumenta de forma substancial, llegamos a una situación de overfitting donde cada capa aprende de forma innecesaria más capas de lo necesario.

```
plt.figure(figsize=(8, 8))
for index, value in enumerate([1, 3, 5, 15]):
    plt.subplot(2, 2, index + 1)
    tmp_layers = afx.ann_number_of_layers("demo2", X_train=X_train,
y_train=y_train, layers=value)

    afx.plot_response_surface(tmp_layers, X_train, y_train, x_mesh,
y_mesh, joint_xy)
    plt.title("Layers: {}\nAcc: {} - Loss: {}".format(
                value,
                np.mean(tmp_layers.history.history['acc']).round(3),
                np.mean(tmp_layers.history.history['loss']).round(3)))
    plt.tight_layout()
```

Bibliografía

- Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1). Cambridge: MIT press.
- Grus, J. (2015). Data science from scratch: first principles with python. " O'Reilly Media, Inc.".
- Chollet, F. (2017). Deep learning with python. Manning Publications Co..
- "Neural Nets".- Francis Tseng. https://frnsys.com/