Perceptrón y Tensores

Alcance:

- Entender que es un tensor como estructura de datos.
- Reconocer los orígenes y terminología asociada a las redes neuronales.
- Aprender el concepto de *perceptrón* y su importancia en la construcción de redes neuronales más complejas.
- Identificar los componentes básicos de una red neuronal.

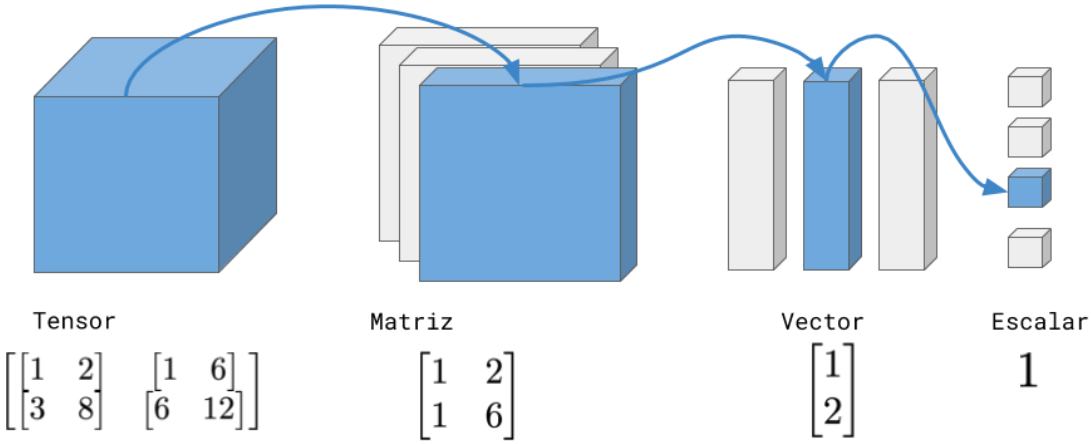
A lo largo del curso hemos aprendido diferentes algoritmos de aprendizaje estadístico y hemos explorado distintos conceptos relacionados con Machine Learning que en este momento nos permitirán introducir en un **tipo de modelo** que ha ganado una gran cantidad de atención en el último tiempo: Las redes neuronales.

Parte de los elementos fundacionales de las redes neuronales se remontan a los perceptrones, unidades creadas con la intensión de "simular el proceso neuronal". Antes de comenzar a trabajar con perceptrones, debemos aprender un tipo de estructura de datos clave para el rendimiento de estos modelos, los tensores.

Tensores

Un *tensor* es un arreglo multidimiensional de datos, muy similar a una matriz, solo que en más de dos dimensiones. En la actualidad todas las aplicaciones/modelos que utilizan grandes cantidades de datos, incluyendo los modelos de machine learning que hemos visto hasta ahora, emplean tensores para realizar sus operaciones. A bajo nivel, el cálculo utilizando tensores se hace en base a la teoría establecida por el álgebra lineal y el cálculo numérico.

Los tensores son generalizaciones de estructuras de datos contenedoras de datos más simples: Un número es un tensor cero-dimensional, un arreglo (o vector) es un tensor uni-dimensional, una matriz un tensor dos-dimensional, etc... Tomando en cuenta lo anterior, podemos definir la dimensionalidad del tensor en base al número de valores contenidos en él.



Una confusión común es el creer que el número de valores contenidos en un vector indica la dimensión del tensor, lo cual no es cierto, un tensor puede ser uni-dimensional (un vector) y contender tantos valores como se quiera:

```
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import lec12_graphs as gfx
plt.style.use('ggplot')
```

```
vec = np.array([1,2])
print('Tensor 1: {0}'.format(vec))
print('La dimension de este vector/tensor es: {0}'.format(vec.ndim))
print('Sin embargo contiene {0} valores.'.format(vec.size))
```

```
Tensor 1: [1 2]
La dimension de este vector/tensor es: 1
Sin embargo contiene 2 valores.
```

```
vec_2 = np.array([[1],[2],[3],[4],[5],[6],[7],[8]])
print('Tensor 2: \n{0}'.format(vec_2))
print('La dimension de este vector/tensor es: {0}'.format(vec_2.ndim))
print('Sin embargo contiene {0} valores.'.format(vec_2.size))
```

```
Tensor 2:
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]]
La dimension de este vector/tensor es: 2
Sin embargo contiene 8 valores.
```

Características de un tensor

- **Forma (shape):** Es una tupla de enteros que informa la cantidad de dimensiones que tiene el tensor. Se puede acceder a esta característica utilizando el método `shape`.
- **Rango:** Es común referirse al rango también como el número de ejes del tensor, este valor es el que nos entrega el comando `ndim` de la librería `Numpy`.
- **Tipo de dato:** Un tensor debe estar asociado al un cierto tipo de dato que será el que se almacenará, de esto depende qué tipo de operaciones es posible realizar con el tensor. Se puede saber el tipo de datos para el que está definido un tensor utilizando el método `dtype` sobre el tensor en la librería `Numpy`. (e.g. `vec.dtype`)

```
print('Tensor: \n{0}'.format(vec_2))
print('Shape: {0}'.format(vec_2.shape))
print('Rango: {0}'.format(vec_2.ndim))
print('Tipo de dato que almacena: {0}'.format(vec_2.dtype))
```

```
Tensor:
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]]
Shape: (8, 1)
Rango: 2
Tipo de dato que almacena: int64
```

En la vida real, un tensor puede venir de muchas formas y tamaños diferentes dependiendo del dominio en el que estemos trabajando. Condicional a su naturaleza, el procesamiento a implementar dependerá de nuestro objetivo y cómo deseamos ingresarlos en nuestro algoritmo.

Para ejemplificar la omniprescencia de los tensores, trabajaremos con imágenes. Para ello haremos uso de una imagen del telescopio Hubble. Para ingestar la imagen a un formato numérico utilizaremos la función `scipy.misc.imread`.

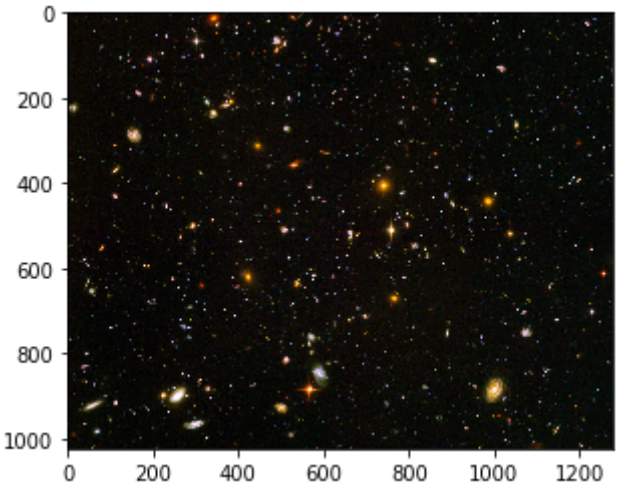
```
from scipy import misc

# importamos la imagen y la mostramos
hubble = misc.imread('img/hubbledeepfield.png')
```

Si preguntamos por la representación nativa de la imagen, Python nos dirá que ésta es un objeto `ndarray`.

```
type(hubble)
```

```
numpy.ndarray
```

```
# Para poder visualizar la representación nativa de la imagen, usamos imshow
plt.imshow(hubble);
```



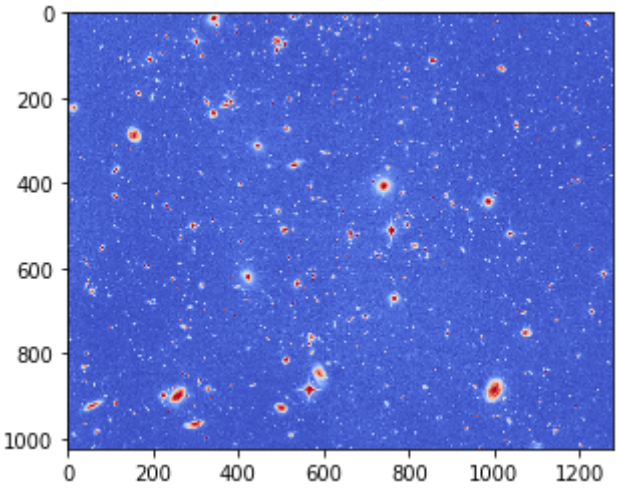
Con librerías como `matplotlib` podemos importar y visualizar imagenes, sin embargo, para tratarlas como tensores y realizar operaciones sobre ellas, debemos trabajar con `numpy`:

```
print('Tipo de datos que almacena la imagen: {0}'.format(hubble.dtype))
print('Forma del tensor: {0}'.format(hubble.shape))
print('Rango: {0}'.format(hubble.ndim))
```

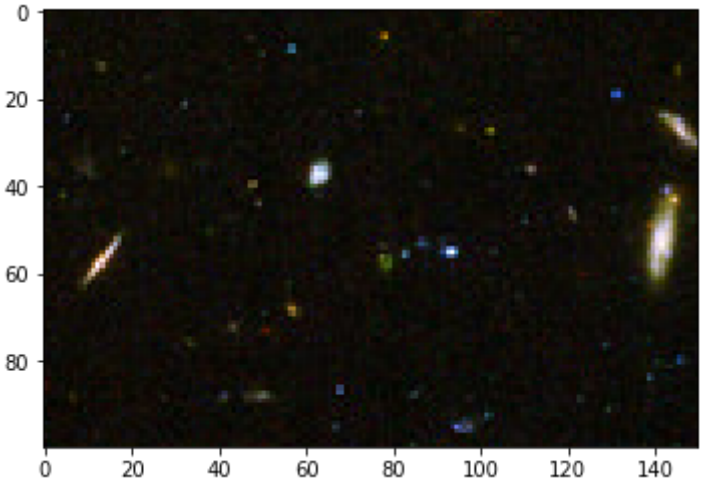
```
Tipo de datos que almacena la imagen: uint8
Forma del tensor: (1024, 1280, 4)
Rango: 3
```

Una parte importante de lo que haremos con tensores es la selección de ciertas dimensiones o el *slicing* de los mismos:

```
# Podemos seleccionar solo un plano de la 3ra dimensión del tensor
hubble_2 = hubble[:,:,0]
plt.imshow(hubble_2, cmap='coolwarm');
```




```
# tambien podemos seleccionar solo una parte del tensor
hubble_3 = hubble[700:800,750:900,:]
plt.imshow(hubble_3);
```



Una de las operaciones más comunes a realizar con tensores es redimensionar los datos en una dimensión más baja. Si originalmente nuestro tensor tenía una forma de **1024, 1280, 4**, podemos reexpresarlo en una matriz de filas y columnas. Generalmente implementaremos preprocesamientos similares cuando deseemos agregar los atributos de una imagen en modelos a entrenar.

```
# Podemos reordenar el tensor, cambiando su forma
import copy
hubble_4 = copy.copy(hubble)
hubble_4 = hubble_4.reshape(hubble.shape[0]*hubble.shape[1], hubble.shape[2])
hubble_4.shape
```

```
(1310720, 4)
```

Cuando cambiamos la forma de un tensor, siempre debemos entregar una dimensión tal que el número de valores contenidos en la nueva forma dimensional sea igual al numero de valores contenidos en la forma dimensional anterior, es decir, no podremos redimensionar un tensor de forma (4,4) en uno de forma (1,2), sin embargo, si podemos redimensionar el primero a (1,16).

Las mismas operaciones que hemos hecho sobre la imagen en este ejemplo son válidas también para cualquier tensor de datos, pues hemos trabajado sobre los datos como tensor y no solo como si fuesen una imagen.

Operando con tensores

Como toda estructura de datos, un tensor debe tener asociado un conjunto de operaciones elementales que son posibles de realizar utilizando como base elementos de la estructura. Estas operaciones, sin embargo, deben estar primero definidas matemáticamente. Para poder definir operaciones sobre espacios vectoriales debemos verificar que se cumplan las propiedades de pertenencia, cuerpo y orden de un espacio

$$\mathbb{N} - dimensional$$

definido sobre el tipo de datos con los que definimos el tensor.

Si tenemos un tensor

$$\mathbb{N} - dimensional$$

que almacena el tipo de datos **d**, entonces, si está definida la operación

$$f(\cdot)$$

sobre el tensor, entonces siempre debe estar definido el resultado de **f(d)**. Lo mismo se debe cumplir para operaciones que impliquen más de un tensor, como la suma o la multiplicación.

Existen dos tipos de operaciones elementales que debemos aprender a diferenciar: Las operaciones elemento-a-elemento y las operaciones matriciales o *tensoriales*.

- **Operación elemento-a-elemento (element-wise):** Es aquella operación que trabaja operando sobre cada uno de los valores del tensor por separado. Un ejemplo de este tipo de operaciones es la suma/resta vectorial.
- **Operación matricial o tensor-wise:** Es aquella operación que combina los elementos de el/los tensor/es para formar la salida. Ejemplos de este tipo de operaciones son el producto punto entre vectores y la multiplicación matricial.

```
vec_a = np.array([1,2,3,4])
vec_b = np.array([6,7,8,9])
print('Vector a: \n{0}'.format(vec_a))
print('Vector b: \n{0}'.format(vec_b))
print('Matriz: \n {0}'.format(vec_a.reshape(2,2)))
```

```
Vector a:
[1 2 3 4]
Vector b:
[6 7 8 9]
Matriz:
 [[1 2]
 [3 4]]
```

```
# Operación element-wise
print('Suma')
print('Operación element-wise: {1} + {2} =
{0}'.format(vec_a+vec_b,vec_a,vec_b))
```

```
Suma
Operación element-wise: [1 2 3 4] + [6 7 8 9] = [ 7  9 11 13]
```

```
# Operación tensor-wise
print('Producto punto')
print('Operación tensor-wise: {1} * {2} = {0}'.format(np.dot(vec_a,
vec_b),vec_a,vec_b))
```

```
Producto punto
Operación tensor-wise: [1 2 3 4] * [6 7 8 9] = 80
```

```
from numpy import tensordot
# Operación tensor-wise
print('Producto matricial')
tensorwise = np.tensordot(vec_b, vec_a, axes = 0)
print('Operación tensor-wise con la matriz:\
        vec_b (x) vec_a = \n{0}'.format(tensorwise))
```

```
Producto matricial
Operación tensor-wise con la matriz:        vec_b (x) vec_a =
[[ 6 12 18 24]
 [ 7 14 21 28]
 [ 8 16 24 32]
 [ 9 18 27 36]]
```

Como podrán notar, el producto punto es un caso especial del producto matricial en el que ambos elementos son unidimensionales.

Finalmente, hay que mencionar que el trabajo con tensores que hemos hecho es a través de la librería `numpy` que tiene esta particularidad de tratar sus objetos de tipo `ndarray` y `array`. Bajo ningún caso se debe asumir que este es un comportamiento nativo de Python!.

Set-up básico

Tensorflow

Tensorflow es un framework creado por Google para el trabajo con tensores que implementa de forma optimizada las operaciones matemáticas que involucra el cálculo vectorial/tensorial (operaciones matriciales y vectoriales principalmente). Dada la complejidad y vasta documentación existente para implementar TensorFlow de manera nativa, no trabajaremos de manera directa con éste. En su lugar utilizaremos **Keras**, una librería que funciona como una capa intermedia entre el usuario y TensorFlow, haciéndo uso de éste para implementar los cálculos necesarios en las redes neuronales.

Digresión: Instalando Tensorflow

Si tenemos Anaconda instalado, podemos incorporar fácilmente Tensorflow a nuestro entorno ejecutando el comando:

```
conda install -c conda-forge tensorflow
```

Esto instalará la versión CPU de TensorFlow. Si contamos con una tarjeta de video dedicada existe otra versión que hace uso de este hardware que se puede instalar adicionalmente con:

```
conda install -c conda-forge tensorflow-gpu
```

Keras

Keras es una librería que funciona utilizando **Tensorflow** para construir los modelos de redes neuronales, esto implica que tendremos que tener ambos instalados para poder trabajar con redes neuronales artificiales.

Digresión: Instalando Keras

Antes de instalar Keras se debe instalar un *backend*, en nuestro caso, utilizaremos Tensorflow como backed. De la misma forma que para Tensorflow, en anaconda podemos instalar Keras de la siguiente forma:

```
conda install -c conda-forge keras
```

Una vez instalado Keras, debemos asegurarnos de que esté ocupando Tensorflow como backend, para esto, se puede abrir un notebook cualquiera y ejecutar el siguiente código:

```
import tensorflow as tf
import keras
```

Podemos comprobar el backend en cualquier momento ejecutando lo siguiente:

```
keras.backend.backend()
```

¿Cómo estructurar modelos en Keras?

Para comprender como estructurar modelos neuronales en Keras hay que entender un par de conceptos en los que se basa la librería:

- **Sequential:** El modelo Sequential en Keras es un apilamiento lineal de capas, por lo tanto, cuando implementemos arquitecturas multi-capa le tendremos que entregar estas capas a Sequential para que las una en una sola estructura.
- **input_shape:** El modelo necesita saber el tamaño de lo que será el vector de entrada a la red, en nuestra analogía del plato de comida, necesitamos decirle al modelo el tamaño de la cuchara.
- **Compilar el modelo:** Compilar un modelo en Keras se refiere a armar la arquitectura, unir las capas y asociar las funciones de costo, agendas de entrenamiento, optimizador, etc. al modelo. Todo modelo debe ser compilado antes de poder ser entrenado.

Digresión: ¿Qué es un backend?

Es natural estar confundido con el paso anterior en el que se menciona que hay que especificar un *"backend"* para Keras siendo que hasta el momento siempre había bastado descargar una librería con un comando para poder utlizarla. La razón es que Keras es una API, o más específicamente, una librería de nivel de modelo. Esto quiere decir que los métodos que implementa son de alto nivel y están enfocados en hacernos más fácil el trabajo de armar modelos complejos como las redes neuronales.

Los métodos matemáticos que se encargan de realizar los cálculos no son implementados sino que son importados desde otra librería. Esta "otra librería" es lo que se conoce como *backend*. En nuestro caso, Keras se encarga de hacer fácil la construcción de arquitecturas de redes neuronales y deja la implementación matemática a TensorFlow, por esto necesitamos tener instalado TensorFlow antes que Keras y además debemos especificarle a Keras qué backend ocupar. Otros backend posibles de utilizar para nuestro propósito son *Theano* y *CNTK*.

Conceptos básicos de una red neuronal

Las redes neuronales son un modelo de aprendizaje estadístico que se desarrolló en su momento bajo la idea de imitar la anatomía de las redes neuronales observadas hasta el momento en el cerebro de los animales. A pesar que esta idea sonó prometedora, en la actualidad sabemos bien que el modelo matemático conocido como "red neuronal" dista bastante de la realidad de la configuración de nuestro cerebro.

La anatomía básica de una red neuronal se compone de los siguientes elementos:

- **Neuronas (*Neurons*):** Elemento básico de una red neuronal, recibe un cierto input y al procesarlo genera un output de acuerdo a una **función de activación**.
- **Capas (*Layers*):** Es una serie de neuronas que ocupan una determinada posición paralela en la secuencialidad de una red. Se pueden distinguir tres tipos de capas principales:
    - **Input:** Es la capa que representa el vector de entrada de datos a la red neuronal, estos datos son los ejemplos del conjunto de entrenamiento. **Cuidado!**, no confundir el tamaño de la capa de input con el tamaño del dataset, sin importar el tamaño del plato de comida que nos estemos sirviendo, lo que determina cuanto ingeriremos en cada cucharada es el tamaño de nuestra cuchara, no el tamaño del plato.
    - **Output:** Es la capa de salida que representa el resultado entregado por la red neuronal, puede ser una categoría (en el caso de que estemos resolviendo un problema de clasificación) o un número (regresión), o incluso puede ser un vector con valores (regresión/clasificación multiple).
    - **Hidden:** Son todas aquellas capas entre las capas de input y el output.

Las capas resultan ser la estructura fundamental de las redes neuronales. En Keras, cada una de éstas series de neuronas que componen las capas tiene distintas características. Vectores simples en dos dimensiones con la estructura `(muestras, atributos)` generalmente se preprocesan por capas *densas* con la clase `Dense` de Keras. Datos secuenciales o serie temporales se representan en tensores con tres dimensiones `(muestra, tiempo, atributos)` suelen preprocesarse con capas *recurrentes* con la clase `LSTM` (Long Short Term Memory).

Las imágenes pueden representarse mediante tensores de cuatro dimensiones, procesados con la clase `Conv2D`.

- **Función de activación:** Puede ser más de una, son las funciones con las cuales se computa el resultado (o *"activación"*) de cada neurona. Cada neurona tiene asociada una función de activación. Sin embargo, se asocianfunciones de activación a las capas y se asume que todas las neuronas en esa capa tienen la misma función de activación.
- **Modelos:** Siguiendo las convenciones de Chollet (2017) -el creador de Keras, éstas capas se conectan mediante **modelos**, grafos que ordenan el flujo de conexión entre ellas. El modelo más simple es uno donde existe un flujo único entre una capa de input con una neurona y una capa de salida con una neurona. Éste modelo se conoce como el Perceptrón (Rosenblatt, 1958), ejemplo que visitaremos posteriormente. A medida que nuestro requerimientos se tornan más complejos, las redes neuronales se pueden ajustar de gran manera a la naturaleza de nuestro

problema. Para ello debemos diseñar una topología adecuada que permita representar el espacio de posibles resultados.

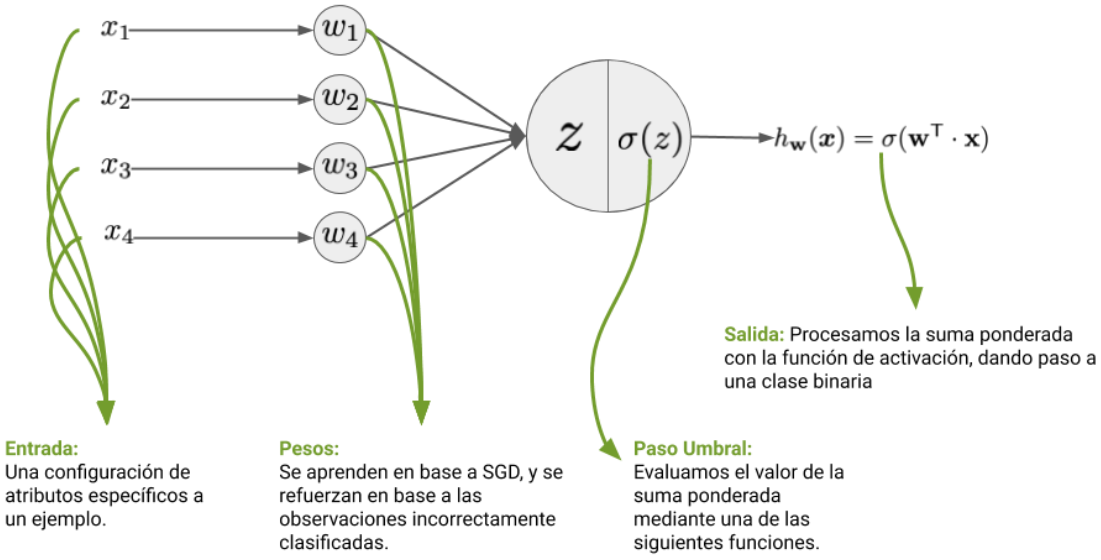
- **Funciones de pérdida y optimizadores**: Con los componentes que tenemos a mano sólo hemos hecho referencia a la arquitectura de nuestra red neuronal. Esto es el equivalente al paso de *feature engineering* en nuestros modelos clásicos de aprendizaje de máquinas. Resulta que ahora debemos implementar el modelo y juzgarlo. Para ello necesitamos configurar dos elementos que facilitarán el proceso de aprendizaje de la red neuronal: **Función de pérdida** y **optimizador**.
    - Ya tenemos conocimiento sobre lo que representa una función de pérdida: la cifra que se buscará minimizar en el proceso de entrenamiento de nuestra red neuronal.
    - También sabemos lo que representa el optimizador: norma que determina cómo se actualizarán los parámetros inferidos por la red neuronal en base a la función de pérdida. Por lo general se implementa alguna variante de la Gradiente Descendente Estocástica. Una red neuronal que tiene múltiples neuronas en la capa de output puede tener múltiples funciones de pérdida (una por cada neurona), pero el proceso de optimización **debe estar basado en una sola función de pérdida**. Para éstos casos se puede promediar la función de pérdida de cada neurona antes de optimizar.

La elección de la función de pérdida puede ser compleja y quitar parte substancial del tiempo disponible en un proyecto, por lo que Chollet (2017) sugiere algunas aproximaciones:

- Si nuestro vector objetivo es contínuo y nuestra estrategia es similar a una regresión, podemos implementar métricas de pérdidas como *mean squared error*.
- Si nuestro vector objetivo es categórico con dos clases, podemos incorporar métricas como *binary crossentropy*.
- Si nuestro vector objetivo es categórico con más de dos clases, generalmente implementaremos métricas como *categorical crossentropy*.

El Perceptrón

El modelo más simple de red neuronal es una sola neurona, a este modelo básico se le conoce también como **Perceptrón**:



En la figura se puede ver que la neurona recibe un cierto input y lo multiplica por un valor **W**, luego, el valor resultante es pasado por una función

$$f(\cdot)$$

llamada **función de activación** y el valor entregado por la función de activación es el que entrega como output el modelo. El perceptrón "disparará" una señal hacia la neurona del output sólo cuándo la suma de las neuronas de entrada alcance cierto umbral.

Como ya sabemos, cuando entrenamos un modelo en realidad lo que estamos haciendo es buscar los valores de sus parámetros que minimizan la función de costo, en el caso del perceptrón el modelo solo tiene un parámetro: El coeficiente **W** que aparece en la figura entre el input y el perceptrón,

Por el momento nos quedaremos solo con el perceptrón, exploraremos sus capacidades y limitantes. En las siguientes unidades comenzaremos a estudiar cómo construir y concatenar capas. Nuestro siguiente paso es estudiar las librerías que nos permiten implementar este tipo de modelos.

Nuestro objetivo es cómo crear un perceptrón mediante `Keras`:

```
import tensorflow as tf
import keras
 # El modelo Sequential es la base para construir la arquitectura
from keras.models import Sequential
# Las capas hay que importarlas por separado pues hay de diversos tipos, por
ahora nos preocuparemos solo
# de ver como se comporta el perceptron y después explicaremos el tipo de capa
utilizada
from keras.layers import Dense
print('Utilizando {0} como backend para
Keras.'.format(keras.backend.backend()))
```

```
Utilizando tensorflow como backend para Keras.
```

```
Using TensorFlow backend.
```

El primer paso es declarar cómo se conectarán las capas a declarar. Dado la relativa simpleza del perceptrón, optaremos por una arquitectura secuencial donde todas las capas se conectarán de manera lineal. Para ello generaremos una nueva instancia de la clase `Sequential`.

```
model = Sequential()
```

La instancia de `Sequential` que generamos en el objeto `model` tendrá varios atributos que utilizaremos. A grandes rasgos el proceso de montar un perceptrón en Keras implica:

- `model.add`: En esta etapa añadimos cada una de las capas de la red neuronal. Esto conlleva a definir su función de activación y cantidad de neuronas a generar. Cada capa se irá actualizando en el objeto.
- `model.compile`: En ésta etapa declaramos nuestra **agenda de entrenamiento**: ¿Qué función de pérdida y optimizador utilizaremos?
- `model.fit`: Posterior a la definición de nuestra agenda de entrenamiento, entrenamos el modelo.
- `model.predict`: Como todo flujo de trabajo en aprendizaje de máquinas, el desempeño del modelo debe evaluarse en datos previamente ignorados.

`model.add`

Sabemos que las características del perceptron es tener una neurona de entrada y una neurona de salida, por lo que nuestro objetivo es añadir dos capas. Por defecto implementaremos capas con `Dense`, dado que buscamos capturar todas las posibles conexiones entre neuronas (un poco trivial dado que sólo existe una conexión entre ambas).

Necesitamos definir los siguientes elementos dentro de nuestra clase `Dense`: Las unidades de entrada (referenciadas con `input_shape`), ver si incluímos un sesgo en nuestra función de activación (si incluímos un sesgo estamos imponiendo un umbral en ésta. Es un análogo a una combinación lineal de parámetros con forma

$$\beta + W_i X_i$$

También debemos ver la forma funcional de la función de activación, para éste caso será lineal.Por último podemos asignarle un nombre a las capas, para mantener más claro los componentes de nuestro modelo.

En las capa de entrada cada neurona declarada hace referencia a un atributo específico en la matriz

$$\mathbf{X}$$

mientras que en la capa de salida representa cada posible respuesta en nuestro vector objetivo. En este caso nuestro vector objetivo es contínuo, por lo que sólo existe una neurona. En aquellos casos donde existen categorías, la capa de salida tendrá tantas neuronas como categorías existan.

```
# capa de entrada
model.add(Dense(1, activation = 'linear',      # funcion de activacion lineal
                use_bias = False,              # ignoramos sesgo
                kernel_initializer='uniform',
                input_shape = [1],
                name="entrada"))
# capa de salida
model.add(Dense(activation='linear',
                input_dim=1,
                use_bias=False,
                name="salida",
                units=1))
```

Podemos ver la estructura de la red que tenemos hasta el momento utilizando el método `summary`:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
entrada (Dense)              (None, 1)                 1         
_________________________________________________________________
salida (Dense)               (None, 1)                 1         
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

Si nos fijamos, nos dice el número de parámetros totales del modelo en la parte inferior del output entregado

`model.compile`

Con nuestra estructura definida, ahora podemos implementar la agenda de entrenamiento. Para éste caso implementaremos un algoritmo de Gradiente Descendente Estocástico con una tasa de aprendizaje

$$\alpha$$

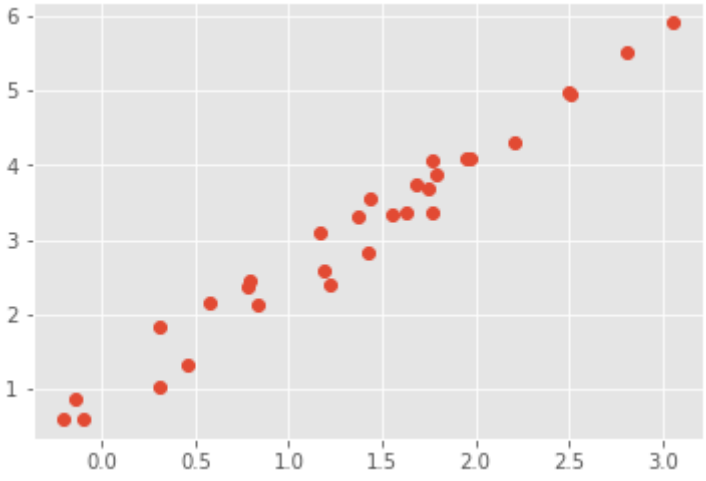
de 1. Para medir el desempeño del modelo utilizaremos la métrica del error cuadrático promedio. Un aspecto a considerar es que debemos incorporar de forma clara el optimizador en la fase de compilación. Para ello podemos ver los optimizadores disponibles en `keras.optimizer`.

```
from keras.optimizers import SGD
model.compile(optimizer = SGD(lr = 0.1), loss='mse')
```

Para nuestro primer perceptrón, generaremos un conjunto de datos artificiales. El patrón de datos sigue una forma lineal simple.

```
m = 30; intercept_true = 0.5; slope_true = 1.5
x = slope_true + np.random.randn(m, 1)
y = intercept_true + (slope_true * x) + np.random.rand(m, 1)
```

```
plt.plot(x, y, 'o')
```



`model.fit`

Finalmente, entrenaremos nuestro modelo con la orden `model.fit`. Debemos ingresar nuetra matriz de validación y vector objetivo. Uno de los aspectos que profundizaremos posteriormente son las épocas que implementamos en nuestro entrenamiento. Por último, implementamos la opción `verbose = 0` para que el modelo no presente su agenda de ejecución.

Si todo sale bien, podremos solicitar los parámetros de cada neurona en cada capa con la opción `get_weights`. Éstos reportarán sobre el peso de entrada y salida en cada neurona.

```
model.fit(x, y, epochs = 50, verbose = 0)
```

```
<keras.callbacks.History at 0x1a3111c6a0>
```

```
model.get_weights()
```

```
[array([[1.4329796]], dtype=float32), array([[1.1752721]], dtype=float32)]
```

Como todo modelo entrenado, el objeto representante tiene el método `predict` que permite predecir valores predichos en base a

```
model.predict([2.5])
```

```
array([[4.2103524]], dtype=float32)
```

Ejemplo: No linealidades con un perceptrón

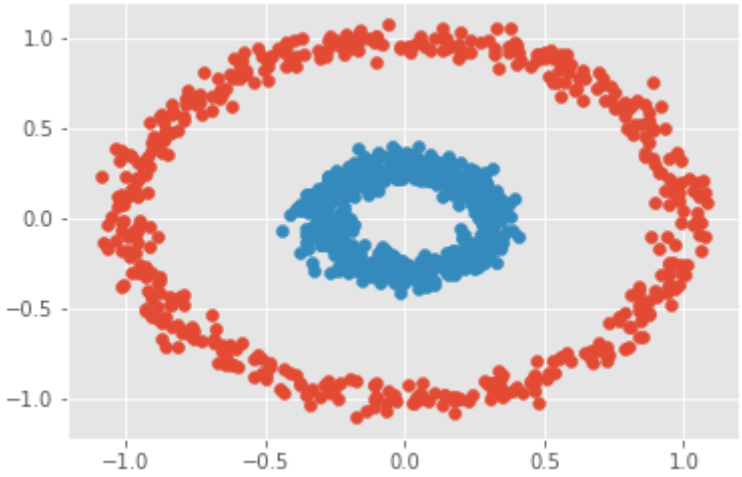
Veamos los límites que presenta el preceptrón con un problema bastante recurrente en la realidad y que demuestra que el perceptrón es muy simple como para ser tratado de forma individual. Trataremos el problema de la clasificación binaria sobre un dataset artificial.

```
X_train, y_train, X_test, y_test = gfx.circles() # Primero creamos el conjunto de datos
```

```
for i in [X_train, y_train, X_test, y_test]:
    print(i.shape)
```

```
(1000, 2)
(1000,)
(1000, 2)
(1000,)
```

```
for i in np.unique(y_test):
    subset = X_test[y_test==i]
    plt.scatter(subset[:,0], subset[:, 1])
```



Nuestro objetivo es generar un clasificador que pueda separar de forma adecuada los círculos presentes. Para ello montaremos nuestro perceptrón con los siguientes elementos:

```
# Definimos una serie de capas lineales como arquitectura
model = Sequential()
# Añadimos una capa densa (neuronas completamente conectadas) con la cantidad
de atributos en nuestra matriz de entrenamiento
model.add(Dense(1, input_dim = X_train.shape[1], kernel_initializer =
"uniform", activation = "relu", name='in'))
# Añadimos una capa densa con 1 neurona para representar el output
model.add(Dense(1, kernel_initializer = "uniform", activation = "sigmoid",
name='out'))
# compilamos los elementos necesarios, implementando gradiente estocástica y
midiendo exactitud de las predicciones como norma de minimización
model.compile(optimizer = SGD(lr = 1), loss = "binary_crossentropy", metrics =
["accuracy"])
# entrenamos el modelo
model.fit(X_train, y_train, epochs = 50, batch_size = 100, verbose = 0)
```

```
<keras.callbacks.History at 0x1a31714668>
```

Veamos cómo quedó implementado nuestro perceptrón con Keras al ejecutar `model.summary`. Un aspecto a considerar es que la capa de entrada tiene tres neuronas: dos por cada atributo en la matriz y uno más por el sesgo. Lo mismo se replica en la capa de salida.

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
in (Dense)                   (None, 1)                 3
_________________________________________________________________
out (Dense)                  (None, 1)                 2
=================================================================
Total params: 5
Trainable params: 5
Non-trainable params: 0
_________________________________________________________________
```

Podemos extraer los pesos de cada capa al solicitar información específica con la función `get_layer` la capa específica a inspeccionar. Veamos los pesos de la capa de entrada. Uno de los puntos que profundizaremos es la interpretación de cada uno de éstos parámetros en el contexto de las redes neuronales. Por ahora extraigamos los pesos del modelo.

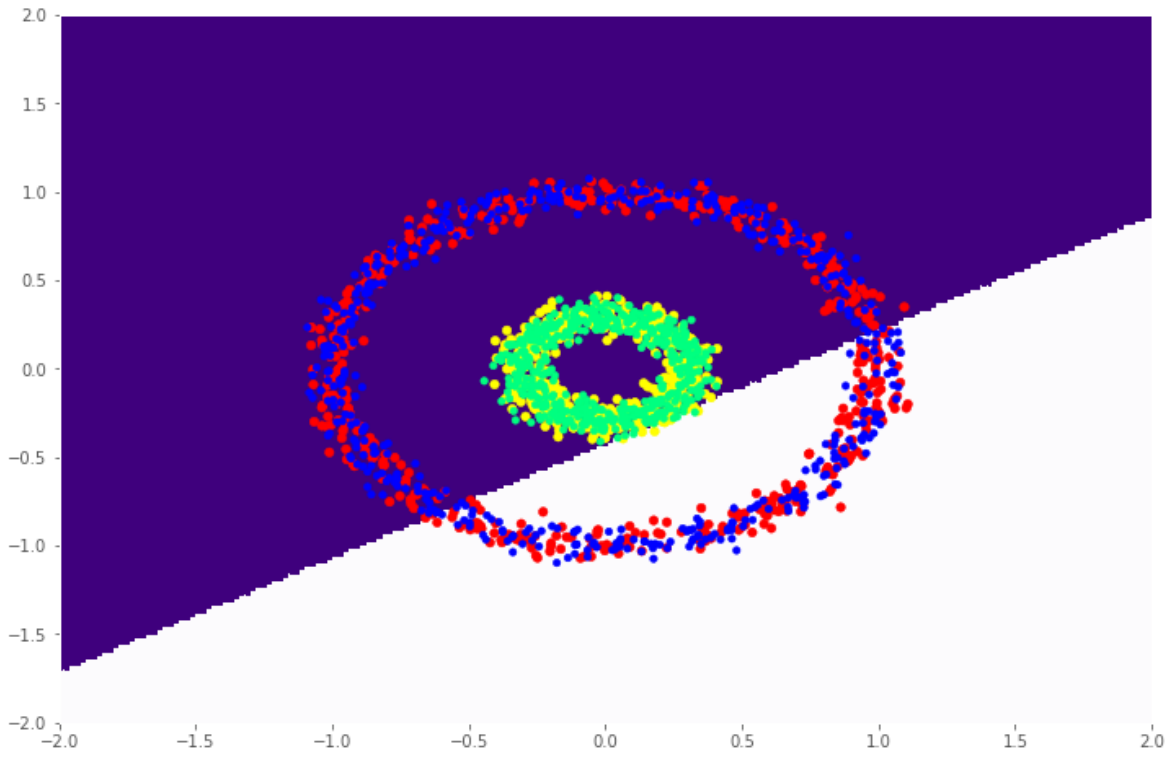
```
# guardaremos la información existente de la capa de entrada
capa_entrada = model.get_layer('in')
capa_entrada.get_weights()
```

```
[array([[-1.2983606],
        [-3.8069565]], dtype=float32), array([-1.3816038], dtype=float32)]
```

Generemos una visualización y evaluación del modelo respecto a los datos. Observamos que el clasificador de forma lineal no puede adaptarse de buena manera a la forma de los datos. En términos de exactitud, el perceptrón tiene un desempeño 18% superior que un clasificador aleatorio.

```
gfx.evaluate_network(model, X_train, y_train, X_test, y_test)
```

```
1000/1000 [==============================] - 0s 61us/step
Accuracy: 0.683000
```




```
0.683
```

Uno de los puntos a destacar sobre las redes neuronales es que no son una solución perfecta para todo problema. Para ello implementaremos un modelo que ya conocemos: un Clasificador de Soporte Vectorial con un kernel `rbf`. Si entrenamos el modelo implementamo los mismos datos de entrenamiento y validación, obtendremos que la capacidad predictiva en el conjunto de entrenamiento de datos es de 1. Haciendo salvedad a lo sospechoso que puede sonar este puntaje, sirve para destacar un elemento en particular: **las redes neuronales son simplemente una herramienta más en nuestras manos para poder generar modelos predictivos.**

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
support_vector_classifier = SVC(kernel='rbf').fit(X_train, y_train)
print("Capacidad predictiva en el testing set: ", accuracy_score(y_test,
support_vector_classifier.predict(X_test)))
```

```
Capacidad predictiva en el testing set:  1.0
```

Si bien el modelo no es capaz de clasificar bien el circulo central,veremos en la siguiente unidad el real potencial de las redes neuronales y que les permite resolver problemas complejos como este, nace de dos cosas principalmente:

1. La posibilidad de aplicar una función no lineal a los distintos atributos con los que trabaja.
2. La capacidad de utilizar las llamadas 'variables latentes' para hacer explotar el espacio de atributos y encontrar combinaciones de variables favorables para resolver el problema.

Bibliografía y referencias

- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.
- Friedman, J., Hastie, T., & Tibshirani, R. (2001). The elements of statistical learning (Vol. 1, No. 10). New York, NY, USA:: Springer series in statistics. Chicago
- Chollet, F. (2017). Deep Learning with Python. Manning Publications Co.
- Patterson, J., & Gibson, A. (2017). Deep Learning: A Practitioner's Approach.O'Reilly Media, Inc.
- Documentación oficial de Keras: Keras.io.
- Documentación oficial de Tensorflow: tensorflow.org