Simple keras regression. Aproximación 1.

Esta es una aplicación de nivel de entrada.
El ejemplo propuesto y resultados no muestran ni de cerca la potencialidad de las redes neuronales, pero es una primera aproximación para entender algunos de los conceptos que más adelante se tratarán en otros ejemplos algo más complicados. Digamos que es un primer escalón de los muchos que hay por delante y que nos va a servir para tener algunos conceptos iniciales como red neuronal, épocas, batch, funciones de optimización, de activación o función de pérdida.

Generación de datos para la red

En el ejemplo vamos a trabajar con un problema en el que la red neuronal tiene que encontrar la relación que existe entre tres variables diferentes de entrada. Estas tres variables podrían ser cualquier cosa. Tres parámetros de salud de una persona (nivel de azúcar en sangre, presión arterial y nivel de colesterol, o en otro contexto, nivel de nitrógeno de una gramínea, nivel de humedad del suelo donde crece y temperatura ambiente).

Nosotros generamos los valores de estas tres variables de forma aleatoria con la librería numpy, mediante el siguiente código que se encuentra en el archivo *data_generator.py*

```
observations = 1000
x = np.random.uniform(low=-10, high=10, size=(observations,1))
y = np.random.uniform(low=-10, high=10, size=(observations,1))
z = np.random.uniform(low=-10, high=10, size=(observations,1))
```

Ahora necesitamos proporcionar alguna relación entre estas variables. Puesto que estamos en un caso completamente inventado, vamos a inventar también la relación que tienen esas variables. Va a ser una relación lineal, una combinación lineal de los valores a la cual le vamos también a añadir un determinado ruido para que los resultados no sean completamente perfectos.

```
noise = np.random.uniform(low=-1, high=1, size=(observations,1))
targets = 2*x - 5*y + z + noise
```

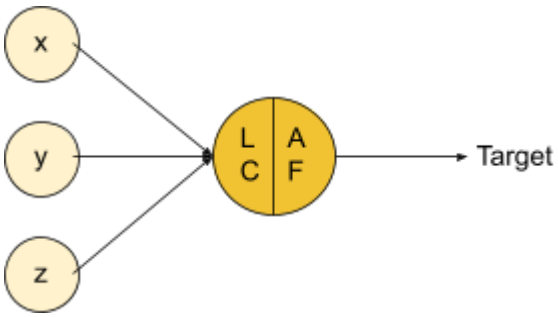
Finalmente agrupamos los valores de las tres variables independientes en un stack y se guardan en un formato npz, formato capaz de guardar los tensores tal y como los puede leer y entender la librería tensorflow.

```
var_inputs = np.column_stack((x,y,z))    #creates a 1000x3 matriz
np.savez('data_all', inputs= var_inputs, targets=targets)
```

Diseño de la red

Nuestro diseño es extremadamente simple, de hecho creo que el más simple. Nuestra red consta de una sola neurona. En cada neurona de una red neuronal ocurren dos operaciones. Una suma ponderada de las entradas y la aplicación de una función de activación.
En la suma ponderada se multiplica cada valor de entrada (x,y,z en el ejemplo) por un peso asociado w y se suman junto con un sesgo. Este es el valor neto de entrada a la neurona y no es más que una combinación lineal (LC en el gráfico).

A continuación se pasa por la función de activación (AF en el gráfico), que transforma ese valor neto de entrada en una salida. Nuestro ejemplo aplica por defecto una relación lineal, tiene sentido utilizar la misma función lineal (f(x) = x) pero el gran poder de las funciones de activación es proporcionar buenos resultados cuando las relaciones no son lineales.



Red neuronal con tres nodos en la capa de entrada y una sola neurona.

LC representa la combinación lineal entre variables

$t = xw_1 + yw_2 + zw_3 + bias$

AF representa la función de activación, por defecto se aplica la función identidad f(t)=t función que deja la salida tal y como viene de la combinación.

Explicación funcional de la red

La propuesta de esta red neuronal es la siguiente. Le vamos a entregar una serie de valores en las tres variables *x, y, z*. Y también le vamos a proporcionar los valores de salida, *targets*. La red es la encargada de encontrar la relación que hay entre estas tres variables. Nosotros ya la sabemos, puesto que la hemos creado con los valores target mediante la combinación lineal

```
2*x - 5*y + z + noise
```

Así que la red tiene que proporcionarnos los pesos 2, -5 y 1 o al menos valores muy próximos a éstos.
En un caso real no sabríamos qué relación hay entre las variables y es precisamente lo que estaríamos buscando y por eso utilizamos la red neuronal. Pero en este ejercicio vamos sobre seguro y usamos datos conocidos puesto que nuestra intención es más educativa que exploratoria.

También en un caso real necesitaríamos dividir el conjunto de datos en varios sets (train, test, validation) que nos sirven para ver la precisión con la que el modelo está trabajando. Pero en este ejemplo y puesto que ya sabemos el resultado, no haremos esa parte.

El entrenamiento real del modelo comienza con la siguiente línea:

```
model_A.fit (training_data['inputs'], training_data['targets'],
epochs=num_epochs_A ,verbose=0, callbacks=[history_A])
```

Le decimos, entrena con estos datos de entrada (*inputs*) para que se ajusten a estas salidas (*targets*), durante este número de épocas (*epochs*). Los otros dos parámetros son simplemente para ver o guardar los resultados que el entrenamiento va arrojando en cada época. verbose = 0 hace que no haya salida por pantalla, verbose = 2 que la haya. El *callback* hace que esos resultados se almacenen (en *history*).

Ahora bien, este es el comando básico, la orden final para decirle que se entrene. Pero antes de ordenarle el entrenamiento hemos tenido que definir algunos parámetros, reglas, del modelo para que ese entrenamiento se haga bajo estos parámetros. Las redes neuronales son super potentes, pero el precio a pagar es que son altamente configurables y por tanto tienen cientos de parámetros que fijar para que funcionen.

Entre otras hay que fijar el tipo de red (si es secuencial o funcional), el número de capas que va a tener, la profundidad de estas capas (número de neuronas), el tipo de cada capa (qué funcionalidad tiene cada capa), el algoritmo de optimización de los parámetros, la tasa de aprendizaje o la función de pérdida son algunos de ellos.

Cada uno de esos parámetros tiene a su vez otros hiperparámetros internos configurables.

La red se forma utilizando el algoritmo de retropropagación y se optimiza de acuerdo con el algoritmo de optimización y la función de pérdida especificados al compilar el modelo.

El algoritmo de retropropagación (backpropagation) requiere que la red sea entrenada para un número específico de **épocas** o exposiciones al conjunto de datos de entrenamiento.
Se encarga de cuantificar la influencia o responsabilidad de cada peso y el sesgo en la red y sus predicciones. Gracias a esto se puede identificar qué pesos de la red hay que modificar para mejorarla.

Batch size: Define el número de patrones a los que se expone la red antes de que se actualicen las ponderaciones dentro de una época. También es una optimización de la eficiencia, asegurando que no se carguen demasiados patrones de entrada en la memoria a la vez. Una ronda que

completa todas las iteraciones sobre todos los batch se llama época.

Reglas del modelo en el ejemplo

Esta es la pieza de código que define parte del modelo.

```
model_A = tf.keras.Sequential([tf.keras.layers.Dense(
    output_size_A,
    activation=activation_f_A,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    # kernel_initializer=tf.random_uniform_initializer (minval=-0.1,
maxval=0.1),
    #
bias_initializer=tf.random_uniform_initializer(minval=-0.1,maxval=0.1)
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,)])
```

Keras es una API que nos simplifica trabajar con tensorflow. Keras ofrece varias formas para construir redes neuronales, la del ejemplo es secuencial.

En **Sequential** creamos el modelo capa por capa. Es simple y sencillo, pero no permite compartir (*sharing*) o ramificar (*branching*). Tampoco permite tener múltiples entradas y salidas. Estos dos conceptos están todavía fuera del alcance de este ejemplo, pero puedes navegar por ellos en este enlace. Hay incluso una tercera forma, Model Subclassing, todavía más compleja. Puedes encontrar más aquí.

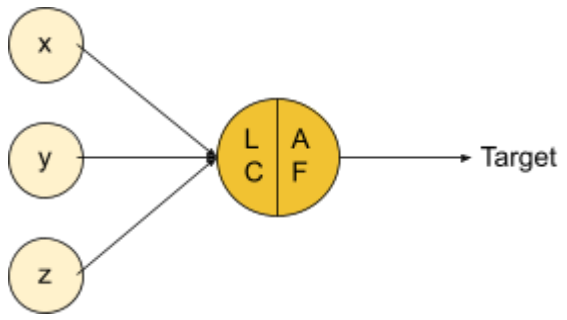
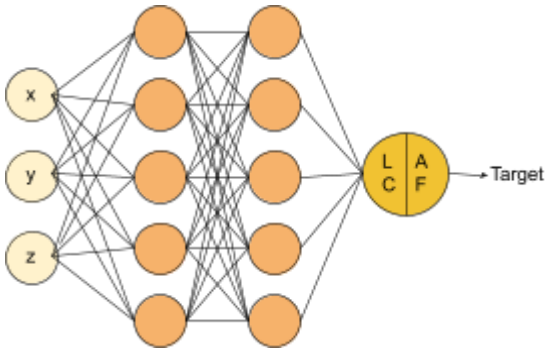
Básicamente Functional nos permite jugar más con las capas para conseguir otro tipo de resultados más avanzados. Sequential está pensado para definir las capas así, de forma secuencial, dentro de la forma de la red neuronal.

Dense indica que la capa, o neurona en nuestro ejemplo, toma inputs de todas las otras neuronas que están en la capa anterior. Funcionalmente lo que hace es la multiplicación matricial de todos los valores de entrada por los pesos para obtener la salida. Hay otras muchas, keras proporciona un montón de capas pre-diseñadas para diferentes arquitecturas de redes neuronales, como *flatten layer, dropout layer, reshape layer,* etc. Una breve descripción de otras capas aquí.

Además vamos a ver un par de hiperparámetros dentro de la definición del modelo, son ***unit*** **y** ***activation function*****.**

unit que el ejemplo es el valor de la variable *output_size*.

Es la dimensionalidad del espacio de salida de la capa. En este caso, solo tenemos una capa y va a dar la salida directamente, que es de dimensión 1, es decir, no tenemos capas ocultas entre medias. SOLO TENEMOS LA CAPA DE ENTRADA Y LA CAPA DE SALIDA, LA DE SALIDA SE HACE CON LA COMBINACIÓN LINEAL MÁS UNA FUNCIÓN DE ACTIVACIÓN QUE ES LINEAL. Se podrían poner capas intermedias de mayor dimensión, por ejemplo 5, tal y como se puede ver en la parte derecha del siguiente gráfico, y la última, antes de la de salida, ponerle de salida 1.

	
Una capa de entrada, ningúna capa oculta y una capa de salida que hace la combinación lineal y aplica la función de activación.	*Una capa de entrada, dos capas ocultas en las cuales el valor de unit debe ser 5 y una capa de salida cuyo valor de unit es 1*

activation function

La función de activación convierte el valor neto de entrada a la neurona (el valor neto es una combinación lineal de variables por pesos más un sesgo) a un nuevo valor. Hay muchas funciones de activación y una gran parte de ellas convierte los valores netos en salidas que están entre 0 y 1 o bien entre -1 y 1.

Por defecto la aplicación muestra una función lineal, es decir, f(x) = x. Por tanto no está haciendo nada sobre la entrada neta de la neurona. Es como si no hubiera función de activación, deja pasar lo que le viene en términos planos, es la función identidad.

Esto no es el caso normal, al menos en las capas ocultas de la red neuronal. Un caso en el que se puede utilizar una lineal es en la capa de salida de las regresiones. Pero en la de salida, no en las capas ocultas. Por tanto no es normal que se use esta lineal.

También, otro caso excepcional es el que nos ocupa, donde no hay capas ocultas, sino una sola capa que hace la combinación lineal y aplica la función de activación directamente para la salida.

Optimizador y tasa de aprendizaje

Las líneas de código donde aplicamos estos parámetros del modelo son estas:

```
model_A.compile (optimizer=optimizer_A, loss=losses_A)
learn_rate_A = st.slider ('Learning rate of optimizers A: ',
min_value=0.01, max_value=0.900, step=0.01, value=0.02)
```

donde el optimizador se elige entre estas opciones:

```
optimizer_list = ['SGD', 'RMSprop',
'Adam','AdamW','Adadelta','Adamax','Adafactor','Nadam','Ftrl' ]
```

y la tasa de aprendizaje (*learning rate*) lo toma de un slider que está configurado para dar valores entre 0.01 y 0.90

Cuánto y cómo modificar los pesos es tarea del **algoritmo de optimización**. Este algoritmo hace los cambios de forma que se minimice el error, es decir, la función de medición del error que también hemos tenido que especificar en los parámetros.

El descenso del gradiente y el descenso del gradiente estocástico (SGD) fueron los primeros métodos utilizados para entrenar redes neuronales. Ambos utilizan un gradiente (es decir, una tasa de cambio) para hacer los cambios que lleven a un menor error de predicción. Pero estos métodos pueden llevar a problemas en redes complejas en los que la optimización queda estancada en regiones de mínimos locales. Por eso se desarrollaron otros métodos más adaptativos.

La **tasa de aprendizaje** (*learning rate*) define cómo de grandes han de ser los cambios en los parámetros del modelo a medida que se va iterando. Si la tasa es muy grande podemos hacer saltos tan grandes que no se ajusten bien a los datos y por tanto perdamos precisión en el modelo. Si por el contrario es una tasa muy baja podemos consumir mucho poder de computación y hacer el proceso muy lento.

Por lo general usaremos valores pequeños mientras el tiempo de computación sea adecuado a nuestro propósito. Si es posible es mejor utilizar valores de tasa variables, más grandes al principio y menores al final.

La función de pérdida

Cada vez que el algoritmo de optimización realiza cambios en los parámetros (pesos, sesgo) tiene que haber algo que nos indique cómo de lejos o cerca estamos de los valores esperados, es decir, cómo de grande es el error que estamos cometiendo con ese valor de parámetros. De esta forma, cuando el algoritmo vuelva a cambiar dichos parámetros verá si el cambio va en la buena o mala dirección. Si comete un error mayor verá que está en el camino equivocado, si el error es menor, seguirá por ese camino.

En nuestro ejemplo la lista de posibles funciones de pérdida la obtenemos del siguiente código:

```
losses_list = ['mean_squared_error','mean_absolute_error',
'mean_absolute_percentage_error',
                'mean_squared_logarithmic_error', 'cosine_similarity']
```

Estas son unas cuantas opciones dentro de las funciones disponibles en keras para modelos de regresión (como es nuestro ejemplo). Un listado y definición se encuentra aquí.
Si el problema no fuera de regresión hay otras funciones de error disponibles. Por ejemplo para problemas de probabilidad: Binay Crossentropy, Categorical Crossentropy, etc. Dependiendo de

cada problema tendremos diferentes opciones para la función de pérdida. Un listado de más funciones y cuándo usarlas se puede encontrar aquí.