Simple keras regression. Aproximación 2.

Esta es la documentación para la segunda aproximación de la aplicación Simple keras regression. Si no conoces la aproximación 1, te sugiero que vuelvas a ella para entender lo que sigue a continuación.

Vamos a dar algunos pasos más alrededor del problema. En la aproximación 1 dejamos que el modelo entrenara con todo el conjunto de datos del que disponíamos, los 10.000 registros o número aleatorios que habíamos generado. Ésta es una forma de trabajar muy simplista y complicada ya que no tenemos forma de saber cómo de preciso o cómo de bien está trabajando nuestro modelo. Por ese motivo existe el procedimiento general de separar el conjunto inicial de datos en tres, cada una con una funcionalidad diferente, y que nos ayudan no sólo a generar el modelo sino también a comprobar su precisión.

División de los datos

train:

- Es el conjunto de datos sobre el que el modelo realmente se está entrenando, el único que se utiliza para la actualización de pesos durante la *back-propagation*.

validación

- Es un conjunto que podemos llamar de "desarrollo". Lo utilizamos para poder cambiar hiperparámetros del modelo, comprobar de nuevo cómo trabaja usando el conjunto de train y validar cómo de efectivo es el modelo. De esta forma el modelo nunca ve el conjunto de datos de test, queda separado del proceso de aprendizaje, re-parametrización y nuevos entrenamientos.
- Este conjunto no se usa para cambiar pesos, pero sí para validar el modelo, por lo que de forma indirecta está afectando a la creación del modelo.

test

- lo utilizamos para comprobar cómo ha aprendido nuestro modelo. Una vez probado nuestro modelo con el conjunto de test ya no tocamos nuevos hiperparámetros. Mientras trabajamos con el conjunto de validación deja de ser desconocido para el modelo e indirectamente aprende patrones de este subconjunto. Por este motivo se necesita un subconjunto "nuevo", uno que no se ha visto nunca

```
size = len(training_data['inputs'])
size_train = int(0.8 * size)
size_val = int(0.1 * size)
size_test = int(0.1 * size)

data_train= training_data['inputs'][0:size_train,]
data_val= training_data['inputs'][size_train:size_train+size_val,]
data_test= training_data['inputs'][size_train+size_val:,]

target_train= training_data['targets'][0:size_train,]
```

```
target_val= training_data['targets'][size_train:size_train+size_val,]
target_test= training_data['targets'][size_train+size_val:-1,]
```

Apéndice 1:
Tengamos en cuenta que keras también puede hacer una división automática en el proceso de división entre entrenamiento y validación. Esto se aplicaría en el momento del entrenamiento, mediante un código similar a este:

```
model.fit(features, labels, validation_split=0.33, epochs=150, batch_size=10)
```

Apéndice 2:
La división también se puede hacer mediante el uso de scikit-learn que tiene métodos específicos para ello. Sería de la forma:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2,
random_state=seed)
```

Métricas y función de pérdida

Algunos de los parámetros a configurar en el modelo antes de hacer el ajuste son el optimizador y la función de pérdida. Esto se hace en la línea de código:

```
model_A.compile (optimizer=optimizer_A,
                 loss='cosine_similarity',
                 )
```

El modelo va a trabajar con la función de pérdida que pongamos en *loss*
pero además podemos especificar otro parámetro, que es *metrics*, en el cual nos devuelve los resultados de esas medida para comprobar cómo evoluciona la pérdida en cada época.

Los valores opcionales de *loss* en este modelo de regresión son exactamente los mismos que podemos poner en *metrics*, así que podríamos aplicar una función de pérdida, pero tener la medición de otra. De esta forma:

```
model_A.compile (optimizer=optimizer_A,
                 loss=losses_A,
                 metrics='cosine_similarity')
```

El resultado sería algo como esto, donde el *verbose=2* nos muestra los valores de la función de pérdida aplicada (que en este caso es `mean_squared_error`
mientras que metrics nos muestra de `'cosine_similarity'`

```
Epoch 1/50
25/25 - 0s - loss: 75.0878 - cosine_similarity: 0.9275 - 465ms/epoch - 19ms/step
Epoch 2/50
25/25 - 0s - loss: 0.3967 - cosine_similarity: 0.9925 - 23ms/epoch - 936us/step
Epoch 3/50
25/25 - 0s - loss: 0.4373 - cosine_similarity: 0.9900 - 24ms/epoch - 946us/step
Epoch 4/50
25/25 - 0s - loss: 0.4594 - cosine_similarity: 0.9950 - 24ms/epoch - 964us/step
```

Ajustando junto con el conjunto de validación

Un aspecto diferenciador de esta segunda aproximación es la incursión de los subconjuntos de validación y test. Su funcionalidad está explicada arriba.
El de validación se añade al ajuste del modelo en esta parte de código:

```
model_A.fit (data_train, target_train,
             epochs=num_epochs_A ,
             verbose=2,
             validation_data=(data_val,target_val),
             callbacks=[history_A])
```

y el subconjunto de test no lo incluimos en ninguna parte del modelo porque básicamente debe permanecer fuera de él, ni olerlo.

Como podemos ver en la gráfica de pérdida del grupo de entrenamiento frente a la del grupo de validación el hecho de que la pérdida caiga rápidamente en las primeras épocas y luego se mantengan cercanas y bajas las pérdidas de entrenamiento yo validación indican que el modelo ajusta bien y no está sobre ajustando (*overfitting*).

Evaluación

Evaluate permite hacer una evaluación entre los valores que predice el modelo y los valores reales, utilizando el set de test específicamente reservado para ello.

```
score = model_A.evaluate(data_test, target_test, verbose = 2)
```

Cuando en

```
model_A.compile (optimizer=optimizer_A,
                 loss=losses_A,
                 metrics=['mse','mae']
                 )
```

no ponemos ninguna *metrics*, tanto *history* y luego también el *evaluate* sólo tiene valores de *loss*, que será el que corresponda a la función de error que hayamos elegido para entrenar el modelo. Pero si añadimos parámetros a *metrics* éstos aparecen tanto en history como en el valor de *evaluate*. En el segundo caso los valores aparecen como una lista, en el orden que se han solicitado en *metrics*.

Evaluation A

- *Loss by selected function: 0.4032*
- *Loss by mean_squared_error: 0.4032*
- *Loss by mean_absolute_errors: 0.5612*
- *Loss by cosine proximity: 0.94*

Conclusiones

Hasta aquí la segunda aproximación de este proceso de creación de un modelo de aprendizaje automático basado en redes neuronales para el tratamiento de una regresión multivariante.

Empezamos con tres variables independientes y una función target conocida. De esta forma podemos estudiar el comportamiento del modelo sabiendo de antemano qué resultados debería producir.

Hemos construido una sencilla aplicación usando streamlit en la que podemos comparar este modelo utilizado con parámetros diferentes. Es decir, partiendo del mismo set de datos y el mismo tipo de red neuronal podemos jugar con parámetros como la función de optimización o la función de pérdida para ver cómo se comportan estos parámetros sobre la precisión y eficacia del modelo.

Estas son algunos de las conclusiones más evidentes que podemos sacar jugando con la aplicación:

1. Es fundamental graficar la función de pérdida frente al número de épocas, sobre todo en momentos de desarrollo. El motivo es que cuantas más épocas tengamos, más tiempo de computación necesitamos, mientras que como vemos en este caso, con unas pocas épocas es suficiente para que el modelo de resultados buenos y no mejorables por más épocas que incluyamos.
2. En el caso de que esperemos una combinación lineal de las variables la única función que da resultados precisos es *lineal*. *Relu* también funciona bien y finalmente nos proporciona unos pesos de la función target muy buenos, sin embargo y por su naturaleza no nos

permite predecir valores negativos que en nuestro caso de ejemplo sí que se pueden dar. Algo parecido ocurre con *elu* o *softplus*.

3. Es conveniente probar para un problema dado diferentes opciones en la función de pérdida y del optimizador ya que algunos funcionan mejor que otros dependiendo del caso. Para ello es buena idea hacer la gráfica de la pérdida del conjunto de entrenamiento y del conjunto de validación y ver si convergen o no.