Redes neuronales en Tensorflow Keras - inicio

```
import tensorflow as tf
import numpy as np
import os
```

Hacer propios conjuntos de datos

- Para pequeñas pruebas, hay que preparar los datos de entrada con sus datos de salida esperados
- Se recomienda utilizar arrays del módulo numpy:

```
x = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

y = np.array([
    [0],
    [1],
    [1],
    [0]
])
```

Crear modelos (redes ya vistas)

- Existen varias maneras, pero esta es la más intuitiva y con menos que escribir. No se pone la capa de entrada, depende de los datos de entrada (recuerda, los nodos de la capa de entrada no son neuronas). La última capa de la lista es la capa de salida.
- Para inicializar los pesos, es necesario hacer propagación hacia delante al menos una vez. Los datos de entrada son una lista de tantos elementos como nodos en la capa de entrada

Cuándo usar un modelo secuencial

A `Sequential` modelo es apropiado para **una pila llanura de capas,** donde cada capa tiene **exactamente un tensor de entrada y un tensor de salida.**

Esquemáticamente, la siguiente `Sequential` modelo:

```
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
# Call model on a test input
x = tf.ones((3, 3))
y = model(x)
```

Crear modelos (redes ya vistas)

- Se pueden poner nombres a las capas, no es obligatorio
- Se puede indicar la función de activación de las neuronas en cada capa
- Existen diferentes tipos de capas, ¡pero algunas no son ni conjuntos de neuronas! Otras son redes densas pero que ya utilizan una función de activación, y otras incluso son conjuntos de capas. Puedes encontrar todas en: https://www.tensorflow.org/api_docs/python/tf/keras/layers
- La capa básica es Dense, que aparte permite asignar la función de activación a utilizar.
- Flatten sirve para tomar una capa que tenga varias dimensiones y convertirla en una capa de una dimensión

```
(None, 1, 10, 64)
```

```
>>> model.add(Flatten())
>>> model.output_shape
(None, 640)
```

- Dropout aleatoriamente con tiempo cambia los valores de la red a 0 (peso o bias?), y a los valores que no les hace eso, tomando un parámetro llamado rate, multiplica a todos los elementos por 1/(1-rate)
- En el enlace anterior también se pueden encontrar algunas funciones útiles

Qué hacer con los modelos

- Existen varias funcionalidades, se enumeran las más básicas:
- compile: configura el modelo para el entrenamiento, permitiendo definir si se quiere utilizar un optimizador y cuál, la función de pérdida que se desea utilizar en la capa de salida, métricas a utilizar (básicamente, la función de coste de la red, accuracy, mse, ...), …
- fit: Ejecuta un cierto número de épocas de entrenamiento en la red
- evaluate: comprobar métricas de coste, como la precisión, de un conjunto de test
- predict: intenta predecir una salida para unos ciertos datos de entrada
- save: guarda la red en disco (variante save_weights?)
- load: carga la red de disco (también puede ser solo configs. Del modelo)
- Se pueden encontrar todas en este enlace: https://www.tensorflow.org/api_docs/python/tf/keras/Model

Ejemplo para 1-0 problema XOR

```
[4]  x = np.array([
         [0, 0],
         [0, 1],
         [1, 0],
         [1, 1]
     ])

     y = np.array([
         [0],
         [1],
         [1],
         [0]
     ])

     model = tf.keras.models.Sequential([
       tf.keras.layers.Dense(2, activation= 'relu'),
       #tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(1, activation= 'sigmoid')
     ])

     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.3), #'adam',
                   loss='mean_squared_error',
                   metrics=['mse'])

     model.fit(x, y, epochs=1000, verbose=0)#verbose = 0 para que no enseñe el progreso de entrenamiento

     <keras.callbacks.History at 0x7f7aa0eed810>

[5]  model.evaluate(x, y)
     print( model.predict(np.array([[0, 0]])) )
     print( model.predict(np.array([[0, 1]])) )
     print( model.predict(np.array([[1, 0]])) )
     print( model.predict(np.array([[1, 1]])) )
```

Ejemplo para 1-1 clasifiación

Softmax + Cross Entropy 1-1

```
[6]  x = np.array([
         [0, 0],
         [0, 1],
         [1, 0],
         [1, 1]
     ])

     y = np.array([
         [1, 0, 0, 0],
         [0, 0, 1, 0],
         [0, 1, 0, 0],
         [0, 0, 0, 1]
     ])
```

```
[9]  model = tf.keras.models.Sequential([
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(4, activation= 'softmax')
     ])

     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.3), #'adam',
                   loss='categorical_crossentropy',
                   metrics=['categorical_crossentropy'])

     model.fit(x, y, epochs=1000, verbose=0)#verbose = 0 para que no enseñe el progreso de entrenamiento
```

```
     <keras.callbacks.History at 0x7f7a8dd21d80>
```

```
[10] model.evaluate(x, y)
     print( model.predict(np.array([[0, 0]])) )
     print( model.predict(np.array([[0, 1]])) )
     print( model.predict(np.array([[1, 0]])) )
     print( model.predict(np.array([[1, 1]])) )
```

Ejemplo para 1-2 clasificación multi-label

```
[25] x = np.array([
         [0, 0],
         [0, 1],
         [1, 0],
         [1, 1]
     ])

     y = np.array([
         [1, 1, 0, 0],
         [1, 0, 1, 0],
         [1, 0, 1, 0],
         [0, 0, 1, 1]
     ])
```

```
[26] model = tf.keras.models.Sequential([
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(4, activation= 'sigmoid')
     ])

     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.3), #'adam',
                   loss='binary_crossentropy',
                   metrics=['binary_crossentropy'])

     model.fit(x, y, epochs=1000, verbose=0)#verbose = 0 para que no enseñe el progreso de entrenamiento
```

```
<keras.callbacks.History at 0x7f7a8d563d00>
```

```
model.evaluate(x, y)
print( model.predict(np.array([[0, 0]])) )
print( model.predict(np.array([[0, 1]])) )
print( model.predict(np.array([[1, 0]])) )
print( model.predict(np.array([[1, 1]])) )
```

Ajustar más parámetros

- Podemos en vez de definir algunas cosas como una cadena de texto (optimizador, función de activación, …), crear su clase correspondiente y poner los parámetros que nos interesen
- Para los pesos de la red, se pueden definir en cada capa con los atributos que se ven en estas imágenes. Se puede indicar como cadena de texto o como un objeto de tipo initializer

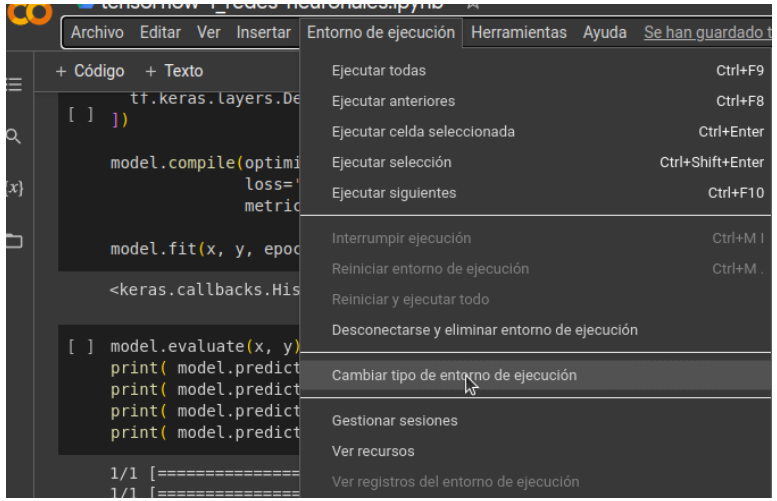
```
tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
```

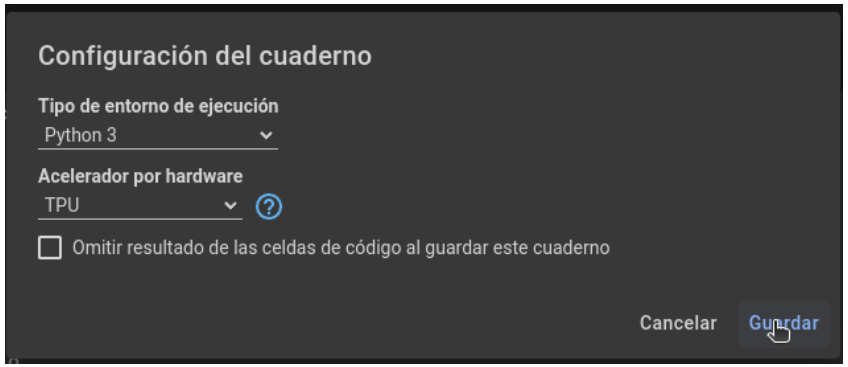
```
layer = layers.Dense(
    units=64,
    kernel_initializer='random_normal',
    bias_initializer='zeros'
)
```

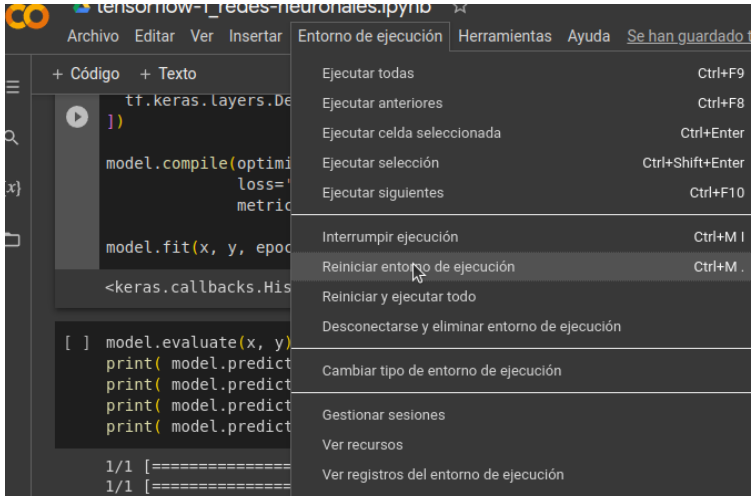
- Puedes crear tus propias funciones de pérdida: https://towardsdatascience.com/creating-custom-loss-functions-in-tensorflow-understanding-the-theory-and-practicalities-383a19e387d6
- Ni idea, pero seguro puedes crear tus propias funciones de activación
- Podemos crear las capas de salida que queramos, con las neuronas que deseemos

Más potencia de entrenamiento! Uso de TPU

- Ojo, aquí se explica cómo utilizarlo de manera básica en colab, más adelante se verá cómo crear tu propio bucle, que acelerará las cosas
- En colab, hay un límite de tiempo de uso de una sesión de 12 horas, podemos aprovechar toda esta potencia a nuestro favor y guardar el modelo antes de terminar. Ahora vamos a lo básico, pero se menciona esto para hacerse una idea de cómo aprovechar esta potente ayuda computacional
- Primero hay que cambiar el tipo de entorno de ejecución para que tenga TPU e iniciarlo. Después si es necesario instalar las librerías de Python necesarias y reiniciar la sesión (ojo no terminarla y volverla a empezar)







Más potencia de entrenamiento! Uso de TPU

- Lo siguiente, hay que pedirle a la sesión preparar esta TPU

```
import tensorflow as tf

import os
```

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='')
tf.config.experimental_connect_to_cluster(resolver)
# This is the TPU initialization code that has to be at the beginning.
tf.tpu.experimental.initialize_tpu_system(resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))
```

- Luego hay que crear una estrategia, en este caso, para la TPU. La estrategia para la TPU es única, pero para GPU hay varias, aunque teniendo una TPU de 8 núcleos qué más quieres...

```
strategy = tf.distribute.TPUStrategy(resolver)
```

Más potencia de entrenamiento! Uso de TPU

- Finalmente, tenemos que crear el modelo utilizando la estrategia. No es necesario luego utilizarlo para entrenarlo como se puede ver en la imagen.

```
with strategy.scope():
  model = create_model()
  model.compile(optimizer='adam',
                # Anything between 2 and `steps_per_epoch` could help here.
                steps_per_execution = 50,
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                metrics=['sparse_categorical_accuracy'])

model.fit(train_dataset,
          epochs=5,
          steps_per_epoch=steps_per_epoch,
          validation_data=test_dataset,
          validation_steps=validation_steps)
```

Más potencia de entrenamiento! Uso de TPU

- Ejemplo para 1-0 (steps-per-execution se puede quitar):

```
[ ]  strategy = tf.distribute.TPUStrategy(resolver)
```

```
[ ]  #with strategy.scope():
     model = tf.keras.models.Sequential([
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(10, activation= 'relu'),
       tf.keras.layers.Dense(1, activation= 'sigmoid')
     ])
     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.3), #'adam',
                   loss='mean_squared_error',
                   steps_per_execution = 50,
                   metrics=['mse'])

     model.fit(x, y, epochs=100, verbose=0)#verbose = 0 para que no enseñe el progreso de entrenamiento
```

```
     <keras.callbacks.History at 0x7f86c8eff370>
```

```
[ ]  model.evaluate(x, y)
     print( model.predict(np.array([[0, 0]])) )
     print( model.predict(np.array([[0, 1]])) )
     print( model.predict(np.array([[1, 0]])) )
     print( model.predict(np.array([[1, 1]])) )
```

Más potencia de entrenamiento! Uso de TPU

- Se puede crear un bucle personalizado y aprovechar más la potencia, pero eso se verá más adelante (aunque sale en el siguiente enlace)
- Hmmm, ojito, ¿ha llegado a tardar más que sin aceleración? Si, solamente ejecutar fit para TPU tarda más que hacerlo sin aceleración.
- Cuidado con esto, las TPU hacen productos matriciales rapidísimo, pero el resto no. Cuando la red sea grande pero no demasiado, mejor utilizar la GPU. Si es enorme la red, a lo mejor TPU llega a ser mejor. Si es pequeña, no utilizar ni GPU ni TPU
- Todo eso y más aquí: https://www.tensorflow.org/guide/tpu?hl=es-419

Más potencia de entrenamiento! Uso de GPU

- Ejemplo para 1-0 (antes has tenido que cambiar el entorno de ejec. con GPU)
- Importante, primero hacer lo de arriba, y luego crear el modelo y configurarlo con with strategy.scope(). Fit no necesita estar dentro
- Existen estrategias que utilizan varias GPUs, como Mirrored, pero no creo que las vayamos a utilizar (hay que pagar para más GPUs)

```
tf.debugging.set_log_device_placement(False)
gpu = tf.config.list_logical_devices('GPU')[0]
strategy = tf.distribute.OneDeviceStrategy(gpu)


with strategy.scope():
  model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, activation= 'relu'),
    tf.keras.layers.Dense(100, activation= 'relu'),
    tf.keras.layers.Dense(1000, activation= 'relu'),
    tf.keras.layers.Dense(10000, activation= 'relu'),
    tf.keras.layers.Dense(1000, activation= 'relu'),
    tf.keras.layers.Dense(100, activation= 'relu'),
    tf.keras.layers.Dense(10, activation= 'relu'),
    tf.keras.layers.Dense(1, activation= 'sigmoid')
  ])
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.3),
                loss='mean_squared_error',
                metrics=['mse'])


model.fit(x, y, epochs=1000, verbose=0)#verbose = 0 para que no enseñe
```

Guardar y cargar modelos de disco

- Es interesante que, una vez entrenado nuestro modelo, lo podemos guardar en disco en una carpeta, que luego podemos comprimir en un archivo .zip
- Además, luego podemos cargar desde disco esta red ya entrenada
- Para guardar un modelo en disco, tenemos que hacer esto:

```
!mkdir -p saved_model
model.save('saved_model/my_model')
```

- Para cargar el modelo, basta con hacer:

```
modelo_cargado = tf.keras.models.load_model('saved_model/my_model')
```

- Podemos comprobar que la carga ha sido exitosa:

```
modelo_cargado.evaluate(x, y)
print( modelo_cargado.predict(np.array([[0, 0]])) )
print( modelo_cargado.predict(np.array([[0, 1]])) )
print( modelo_cargado.predict(np.array([[1, 0]])) )
print( modelo_cargado.predict(np.array([[1, 1]])) )
```

- En collab podemos comprimir la carpeta en un archivo zip y descargarla en nuestro equipo

```
model_name = 'saved_model'
!zip -r {model_name}.zip {model_name}
```

- Y después descomprimirla

```
model_name = 'saved_model'
!unzip {model_name}.zip
```

¡Prueba el código!

- ¡Ejecuta por ti mismo los cuadernos de google collab online!: https://colab.research.google.com/drive/1Lt1IWkAcKI5WW_Lz_f3EXd2H7P8sgorB?usp=sharing
- Dependiendo del hardware de aceleración utilizado, cambia el entorno de ejecución tal y como se menciona en esta presentación