# Uso de tensorflow keras en collab studio avanzado para crear y utilizar modelos personalizados

Podemos crear modelos con varias entradas? "capas" personalizadas que pueden tener más capas e incluso tener varias entradas?

# Por qué complicarse más la vida

- En la introducción a tensorflow hemos visto que se pueden crear modelos a partir de un modelo especial, Sequential, pero tiene un problema, que sólo tiene una capa de entrada y una capa de salida, ¿y si necesito que mi red tenga varias capas de entrada, cómo paso los datos y en qué orden?
- Por ello, hay que crear modelos con tensorflow keras de una manera más compleja

# Usar capas

- Para permitir construir modelos más personalizados, primero necesitamos poder usar capas como deseemos, es decir, manejar cada capa de manera independiente
- Esto se consigue creando la capa a partir de una clase de keras ya existente, y después llamarla como una función, pasando como parámetro la entrada, que puede ser igual como un vector
- Todas las capas en: https://www.tensorflow.org/api_docs/python/tf/keras/layers

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

```
[13] densa_1 = tf.keras.layers.Dense(10, activation='sigmoid')
     res= densa_1(x)
     print(res)
```

```
tf.Tensor(
[[0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000
  0.50000]], shape=(1, 10), dtype=float32)
```

# Usar capas

- Pueden pasarse varios ejemplos a la vez (x tiene más de un elemento, una lista con los valores de entrada para cada ejemplo), el resultado será un tensor que tenga dentro de su matriz tantas filas como ejemplos

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

```
densa_1 = tf.keras.layers.Dense(10, activation='sigmoid')
res= densa_1(x)
print(res)
```

```
tf.Tensor(
[[0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000
  0.50000]
 [0.52646 0.47577 0.36528 0.36387 0.45951 0.53436 0.54320 0.58201 0.63159
  0.56174]
 [0.50811 0.39448 0.49455 0.34894 0.55333 0.66235 0.39412 0.58526 0.54200
  0.49210]
 [0.53454 0.37156 0.36024 0.23464 0.51295 0.69241 0.43615 0.66272 0.66984
  0.55394]], shape=(4, 10), dtype=float32)
```

# Funciones que utilizan las capas

- Se pueden crear funciones que manejan estas capas, las cuales pueden recibir varias entradas como parámetros, y escoger el orden en el que se obtienen las entradas. En esta imagen sólo se muestra un ejemplo simple con una entrada

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

```
def capa_personalizada_1(entrada):
  densa_1 = tf.keras.layers.Dense(10, activation='sigmoid')
  densa_2 = tf.keras.layers.Dense(1, activation='sigmoid')
  r1 = densa_1(entrada)
  r2 = densa_2(r1)
  return r2

res= capa_personalizada_1(x)
print(res)
```

```
tf.Tensor([[0.28889]], shape=(1, 1), dtype=float32)
```

# Funciones que utilizan las capas

- Ahora mostramos un ejemplo para utilizar varias entradas. Para "unir" dichas entradas, utilizaremos un producto matricial. La función sería el producto de las matrices a*b, permite transponerlas antes de hacer los productos.

```
tf.linalg.matmul(
    a,
    b,
    transpose_a=False,
    transpose_b=False,
    adjoint_a=False,
    adjoint_b=False,
    a_is_sparse=False,
    b_is_sparse=False,
    output_type=None,
    name=None
)
```

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

```
def capa_personalizada_2(entrada_1, entrada_2):
  densa_1 = tf.keras.layers.Dense(10, activation='sigmoid')
  densa_2 = tf.keras.layers.Dense(10, activation='sigmoid')
  r1 = densa_1(entrada_1)
  r2 = densa_2(entrada_2)
  print("r1:",r1)
  print("r2:",r2)
  mm = tf.matmul(r1, r2, transpose_b=True)
  return mm

res= capa_personalizada_2(x, x)
print(res)
```

```
r1: tf.Tensor(
[[0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000
  0.50000]], shape=(1, 10), dtype=float32)
r2: tf.Tensor(
[[0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000 0.50000
  0.50000]], shape=(1, 10), dtype=float32)
tf.Tensor([[2.50000]], shape=(1, 1), dtype=float32)
```

# Crear un modelo

- Creamos una clase que extienda de la clase tensorflow.keras.Model, en el constructor creamos las capas, y creamos un método call, el cual será el que gestionará la propagación hacia delante de las entradas en la red
- Se pueden crear aquí dentro las funciones de capas personalizadas, simplemente añadimos como primer parámetro self, y después llamamos a las capas creadas en la clase con self. a la izquierda. En ese caso, ojo llamarlas con self. a la izda.
- Para crear una instancia del modelo, solamente llamar a la clase en una variable

```
52] class MyModel(tf.keras.Model):
      def __init__(self):
        super(MyModel, self).__init__()
        self.densa_1 = tf.keras.layers.Dense(10, activation='sigmoid')
        self.densa_2 = tf.keras.layers.Dense(10, activation='sigmoid')
        self.out = tf.keras.layers.Dense(1, activation='sigmoid')

      def capa_personalizada_2(self, entrada_1, entrada_2):
        r1 = self.densa_1(entrada_1)
        r2 = self.densa_2(entrada_2)
        mm = tf.matmul(r1, r2, transpose_b=True)
        return mm

      def call(self, entrada_1, entrada_2):
        mm = self.capa_personalizada_2(entrada_1, entrada_2)
        return self.out(mm)
```

# Entrenar el modelo

- En este caso, como call tiene más de un parámetro, al utilizar los métodos compile y fit vistos anteriormente, se producirá un error de ejecución. Tenemos que personalizar estas acciones.
- Primero tendremos que crear la función de pérdida que se desea utilizar y el optimizador a utilizar. Por ejemplo puede ser Descenso de Gradiente normal, Descenso de Gradiente estocástico, ADAM, ...

```
funcion_perdida = tf.keras.losses.MeanSquaredError()

optimizador = tf.keras.optimizers.Adam()
```

# Entrenar el modelo

- Después escribimos una función que se encargará de hacer un paso en el entrenamiento (es decir, aprender recibiendo un conjunto de entrenamiento).
- Primero hacemos una predicción, llamando al modelo como si fuese con su método call con las entradas. Después calculamos con la función de pérdida los errores con respecto a las salidas esperadas. Finalmente se calculan los gradientes y se hacen aprender en el modelo con el optimizador escogido.

```
@tf.function
def train_step(x1, x2, y):
  with tf.GradientTape() as tape:
    predictions = model(x1, x2)
    loss = funcion_perdida(y, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizador.apply_gradients(zip(gradients, model.trainable_variables))
```

# Entrenar el modelo

- Finalmente, llamamos a esta función tantas veces como épocas deseemos. Podemos crear un bucle personalizado

```
[44] x1 = np.array([
         [0, 0],
         [0, 1],
         [1, 0],
         [1, 1]
     ])

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

```
[69] EPOCHS = 1000

     for epoch in range(EPOCHS):
       train_step(x1, x1, y)
```

# Hacer una predicción con el modelo

- Lo mismo que hacíamos durante el entrenamiento, llamar al modelo como si fuese llamar a su método call
- En la imagen, se han hecho más de 1000 épocas, pero como se puede observar, aunque a este extraño modelo le cuesta más aprender, lo hace bastante bien con las suficientes épocas

```
print(model(x1,x1))
```

```
tf.Tensor(
[[0.06603]
 [0.91944]
 [0.92112]
 [0.07877]], shape=(4, 1), dtype=float32)
```

# Otras pijotadas con el entrenamiento

- Se puede personalizar para que muestre el error cometido por época, precisión, etc etc etc. Se puede encontrar en este enlace: https://www.tensorflow.org/tutorials/quickstart/advanced?hl=es-419

# Utilizar la GPU en un modelo personalizado

- Funciona exactamente igual que como se vio en la presentación de introducción
- Ojo, sólo funciona en este caso si la estrategia es de un solo dispositivo, como en la introducción, en otro caso, la implementación es más compleja
- Creamos la red, densa, para que sea notable la diferencia computacional

```
class MyBigModel(tf.keras.Model):
  def __init__(self):
    super(MyBigModel, self).__init__()
    self.d1 = tf.keras.layers.Dense(10, activation= 'relu')
    self.d2 = tf.keras.layers.Dense(100, activation= 'relu')
    self.d3 = tf.keras.layers.Dense(1000, activation= 'relu')
    self.d4 = tf.keras.layers.Dense(10000, activation= 'relu')
    self.d5 = tf.keras.layers.Dense(1000, activation= 'relu')
    self.d6 = tf.keras.layers.Dense(100, activation= 'relu')
    self.d7 = tf.keras.layers.Dense(10, activation= 'relu')
    self.o = tf.keras.layers.Dense(1, activation= 'sigmoid')

  def capa_personalizada_2(self, entrada):
    r1 = self.d1(entrada)
    r2 = self.d2(r1)
    r3 = self.d3(r2)
    r4 = self.d4(r3)
    r5 = self.d5(r4)
    r6 = self.d6(r5)
    r7 = self.d7(r6)
    ro = self.o(r7)
    return ro

  def call(self, entrada):
    res = self.capa_personalizada_2(entrada)
    return res
```

# Utilizar la GPU en un modelo personalizado

- Después creamos la estrategia que vamos a utilizar, en este caso, la de un solo dispositivo

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

- Después, tenemos que crear el modelo, la función de pérdida y el optimizador con la estrategia:

```
with strategy.scope():
  model = MyBigModel()
  funcion_perdida = tf.keras.losses.MeanSquaredError()
  optimizador = tf.keras.optimizers.Adam()
```

- Ahora creamos la función de entrenamiento, que es casi igual

```
@tf.function
def train_step(x, y):
  with tf.GradientTape() as tape:
    predictions = model(x)
    loss = funcion_perdida(y, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizador.apply_gradients(zip(gradients, model.trainable_variables))
```

# Utilizar la GPU en un modelo personalizado

- El bucle de entrenamiento en este caso es prácticamente igual

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

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

```
EPOCHS = 1000

for epoch in range(EPOCHS):
  train_step(x, y)
```

- Para predecir, es igual

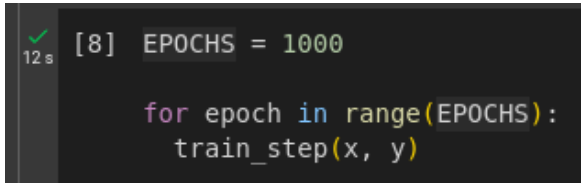
```
print(model(x))
```

```
tf.Tensor(
[[0.00000]
 [1.00000]
 [1.00000]
 [0.00000]], shape=(4, 1), dtype=float32)
```

# Utilizar la GPU en un modelo personalizado

- Tiempo de entrenamiento sin GPU (llegó a 3 minutos y medio, pero se ve que tarda más):


- Tiempo de entrenamiento con GPU:



# Guardar y cargar el modelo de disco

- Funciona exactamente igual que como se vio en la presentación de introducción
  - 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:

```
print(modelo_cargado(x1,x1))
```

```
tf.Tensor(
[[0.00349]
 [0.99493]
 [0.99372]
 [0.00689]], shape=(4, 1), dtype=float32)
```

    - 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