Brandon Alain Cruz Ruiz 198573

In [ ]:

```
pip install d2l
```

Suma de numeros binarios

El objetivo de esta práctica es diseñar un modelo recurrente basado en modelos de redes neuronales (RNN). Empecemos por cargar las librerías necesarias.

In [2]:

```
import os
import tensorflow as tf
from d2l import tensorflow as d2l

import tensorflow.keras as keras
import numpy as np
```

La idea es sencilla. Todo número entero tiene una representación binaria. Por ejemplo, el número `43` tiene una representación binaria igual a `1101010`. La cual podemos representar como un expansión en potencias de 2. Es decir,

$$1 \times 2^0 + 1$$
$$\times 2^1 + 0$$
$$\times 2^2 + 1$$
$$\times 2^3 + 0$$
$$\times 2^4 + 1$$
$$\times 2^5 + 0$$
$$\times 2^6$$

El objetivo es, dados dos números en binario (x_1, x_2) queremos entrenar una RNN para producir el resultado $y = x_1 + x_2$.

Por ejemplo, tenemos los siguientes dos números:

In [3]:

```
np.random.seed(1)

x1 = np.random.randint(0, 2**(7-1))
x2 = np.random.randint(0, 2**(7-1))

print("[%d, %d]"%(x1, x2))
```

```
[37, 43]
```

Que si sumamos tenemos como resultado:

In [4]:

```
print("x1 + x2 = %d"%(x1+x2))
```

```
x1 + x2 = 80
```

Lo que queremos hacer es encontrar una función (RNN) que "sepa" sumar en representación binaria. Es decir, utilizando la representación:

In [5]:

In [5]:

```
sequence_len = 7
format_str = '{:0' + str(sequence_len) + 'b}'

print("x1 = ", ''.join(list(reversed(format_str.format(x1)))))
print("x2 = ", ''.join(list(reversed(format_str.format(x2)))))
```

```
x1 =  1010010
x2 =  1101010
```

In [6]:

```
print("x1 + x2 = ", ''.join(list(reversed(format_str.format(x1 + x2)))))
```

```
x1 + x2 =  0000101
```

Nota que estamos utilizando la convención de escribir un número binario en potencias crecientes de 2. La suma binaria en este caso es una operación que va de izquierda a derecha. Con las reglas usuales:

$$0 + 0 = 0,$$
$$1 + 0 = 1,$$
$$0 + 1 = 1.$$

El caso $1 + 1 = 10$ es el mas interesante, pues en un modelo secuencial que predice un digito a la vez tendría que asignar un `0` y "saber" que "lleva" un `1`.

Vamos a escribir el modelo mas sencillo que podamos. Para esto consideramos lo siguiente. La suma binaria la entendemos digito por digito y con esto tratamos de decidir si emitimos un `0` o un `1`. Para esta parte supondremos que sumamos número pequeños.

Q1 ¿Cuál es el número más grande que podemos encontrar si generamos numeros binarias de longitud igual a 7?

2^8 = 256

Q2 Llena los espacios que faltan en el codigo de abajo para definir un modelo RNN simple.

In [7]:

```
MAX_BIT = 7


input_dim = 2 # El numero de dimensiones de entrada.
activation = "relu" # Una cadena de texto especificando la función de activación en la ca
pa de salida.

model = keras.models.Sequential()
model.add(keras.layers.SimpleRNN(
    4,
    input_dim        = input_dim,
    return_sequences = True
))
model.add(keras.layers.Dense(2, activation='relu'))
model.add(keras.layers.Dense(1, activation=activation))

print(model.input_shape)
print(model.output_shape)
```

```
(None, None, 2)
(None, None, 1)
```

Q3 Prueba que el modelo funciona con un ejemplo sencillo.

In [8]:

```
# test if the prediction shape are expected
input_array = [[0,1],[1,0],[1,0]]#
```

```
x = np.array([input_array])
print(x.shape)
model.predict(x)
```

```
(1, 3, 2)
```

Out[8]:

```
array([[[0.10251028],
        [0.        ],
        [0.5249908 ]]], dtype=float32)
```

Para evitar que el codigo sea demasiado complejo, puedes utilizar las siguientes funciones que modifican las entradas para representarlas en binarios. Nota que necesitamos una función adicional para hacer *padding* y rellenar con ceros la secuencia cuando el número es muy pequeño relativo a la longitud de potencias que estamos utilizando.

In [9]:

```
import keras.preprocessing.sequence

def to_seq(i):
    return list(reversed(list(map(float, "{0:b}".format(i)))))

def pad_seq(a, b, c, maxlen=None):
    return keras.preprocessing.sequence.pad_sequences(
        [a, b, c],
        padding='post',
        dtype='float32',
        maxlen=maxlen
    )
```

Abajo encontrarás dos funciones mas para generar conjuntos de datos para entrenar.

In [10]:

```
def gen_sample(a = None, b = None):
    maxlen = None
    if a is None and b is None:
        a = np.random.randint(2 ** MAX_BIT)
        b = np.random.randint(2 ** MAX_BIT)
        maxlen = MAX_BIT + 1
    c = a + b
    a, b, c = pad_seq(to_seq(a), to_seq(b), to_seq(c), maxlen=maxlen)

    return np.array(list(zip(a, b))), c
def gen_mass_samples(n = 50):
    x = np.zeros((n, MAX_BIT + 1, 2))
    y = np.zeros((n, MAX_BIT + 1, 1))
    for i in range(n):
        x_, y_ = gen_sample()
        x[i, :, :] = x_
        y[i, :, :] = y_.reshape(1, -1, 1)
    return x, y
```

In [11]:

```
x, y = gen_mass_samples()
print(x.shape)
print(y.shape)
list(zip(x[0], y[0]))
```

```
(50, 8, 2)
(50, 8, 1)
```

Out[11]:

```
[(array([0., 0.]), array([0.])),
 (array([0., 0.]), array([0.])),
 (array([1., 0.]), array([1.])),
 (array([1., 1.]), array([0.])),
```

```
 (array([0., 0.]), array([1.])),
 (array([0., 0.]), array([0.])),
 (array([0., 1.]), array([1.])),
 (array([0., 0.]), array([0.]))]
```

El siguiente pedazo de código entrena el modelo con elecciones *default*.

In [22]:

```python
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
N = 10000
MAX_BIT = 7
for i in range(8):
    x, y = gen_mass_samples(N)
    model.fit(x.reshape(N, -1, 2), y.reshape(N, -1, 1), batch_size=50, epochs=3)
```

```
Epoch 1/3
200/200 [==============================] - 1s 2ms/step - loss: 0.2718 - accuracy: 0.8874
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 0.2323 - accuracy: 0.9069
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 0.2197 - accuracy: 0.9141
Epoch 1/3
200/200 [==============================] - 1s 3ms/step - loss: 0.2021 - accuracy: 0.9210
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1984 - accuracy: 0.9173
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1998 - accuracy: 0.9187
Epoch 1/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1911 - accuracy: 0.9189
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1876 - accuracy: 0.9223
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1820 - accuracy: 0.9230
Epoch 1/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1736 - accuracy: 0.9309
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 0.1297 - accuracy: 0.9577
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 0.0433 - accuracy: 0.9993
Epoch 1/3
200/200 [==============================] - 0s 2ms/step - loss: 0.0121 - accuracy: 1.0000
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 0.0035 - accuracy: 1.0000
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 0.0012 - accuracy: 1.0000
Epoch 1/3
200/200 [==============================] - 1s 3ms/step - loss: 5.4748e-04 - accuracy: 1.0
000
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 2.7769e-04 - accuracy: 1.0
000
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 1.3560e-04 - accuracy: 1.0
000
Epoch 1/3
200/200 [==============================] - 0s 2ms/step - loss: 8.0795e-05 - accuracy: 1.0
000
Epoch 2/3
200/200 [==============================] - 1s 3ms/step - loss: 4.3741e-05 - accuracy: 1.0
000
Epoch 3/3
200/200 [==============================] - 0s 2ms/step - loss: 2.9942e-05 - accuracy: 1.0
000
Epoch 1/3
200/200 [==============================] - 0s 2ms/step - loss: 1.4617e-05 - accuracy: 1.0
000
Epoch 2/3
200/200 [==============================] - 0s 2ms/step - loss: 1.1939e-05 - accuracy: 1.0
000
Epoch 3/3
```

```
200/200 [==============================] - 1s 3ms/step - loss: 9.5519e-06 - accuracy: 1.0
000
```

In [23]:

```
x, y = gen_mass_samples(N * 3)
model.evaluate(x.reshape(N * 3, -1, 2), y.reshape(N * 3, -1, 1))
```

```
938/938 [==============================] - 2s 1ms/step - loss: 1.1199e-05 - accuracy: 1.0
000
```

Out[23]:

```
[1.1198524589417502e-05, 1.0]
```

Q4 Escribe un ejemplo para verificar la capacidad predictiva del modelo.

Hice un entrenamiento del modelo y obtuve un accuracy que andaba por el 0.5, volví a entrenar el modelo y obtuve un accuracy de casi 0.9, corrí el modelo por tercera vez para finalmente obtener un accuracy de 1

In [24]:

```
x, y = gen_mass_samples(1)
print(f"Entrada:\n  {x}")

print(f"Resultado esperado:\n {y}")
model.predict(x.reshape(1,-1,2))
```

```
Entrada:
  [[[0. 0.]
  [0. 0.]
  [0. 0.]
  [1. 0.]
  [1. 1.]
  [1. 1.]
  [1. 0.]
  [0. 0.]]]
Resultado esperado:
 [[[0.]
  [0.]
  [0.]
  [1.]
  [0.]
  [1.]
  [0.]
  [1.]]]
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7f19c5805f80> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

Out[24]:

```
array([[[0.        ],
        [0.        ],
        [0.        ],
        [1.7333345],
        [0.        ],
        [1.2926437],
        [0.        ],
        [1.3067518]]], dtype=float32)
```

Q5 Copia y pega el código necesario y entrena el modelo con números mas grandes. Evalúa la capacidad predictiva del modelo en este escenario (números grandes, por ejemplo longitud en binario = 30). ¿Porqué crees que ocurra esto?

In [25]:

```
MAX_BIT = 30


input_dim = 2 # El numero de dimensiones de entrada.
activation = "relu" # Una cadena de texto especificando la función de activación en la ca
pa de salida.

model_30 = keras.models.Sequential()
model_30.add(keras.layers.SimpleRNN(
    4,
    input_dim        = input_dim,
    return_sequences = True
))
model_30.add(keras.layers.Dense(2, activation='relu'))
model_30.add(keras.layers.Dense(1, activation=activation))

print(model_30.input_shape)
print(model_30.output_shape)
```

```
(None, None, 2)
(None, None, 1)
```

In [27]:

```
model_30.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
N = 10000
MAX_BIT = 30
for i in range(8):
    x, y = gen_mass_samples(N)
    model_30.fit(x.reshape(N, -1, 2), y.reshape(N, -1, 1), batch_size=50, epochs=3)
```

```
Epoch 1/3
200/200 [==============================] - 2s 5ms/step - loss: 7.7065 - accuracy: 0.5004
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6995 - accuracy: 0.5008
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7032 - accuracy: 0.5006
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6949 - accuracy: 0.5011
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6949 - accuracy: 0.5011
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6949 - accuracy: 0.5011
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7146 - accuracy: 0.4999
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7146 - accuracy: 0.4999
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7146 - accuracy: 0.4999
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7179 - accuracy: 0.4996
Epoch 2/3
200/200 [==============================] - 1s 4ms/step - loss: 7.7179 - accuracy: 0.4996
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7179 - accuracy: 0.4996
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7283 - accuracy: 0.4990
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7283 - accuracy: 0.4990
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7283 - accuracy: 0.4990
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6922 - accuracy: 0.5013
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6922 - accuracy: 0.5013
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.6922 - accuracy: 0.5013
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7297 - accuracy: 0.4989
Epoch 2/3
```

```
200/200 [==============================] - 1s 5ms/step - loss: 7.7297 - accuracy: 0.4989
Epoch 3/3
200/200 [==============================] - 1s 4ms/step - loss: 7.7297 - accuracy: 0.4989
Epoch 1/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7138 - accuracy: 0.4999
Epoch 2/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7138 - accuracy: 0.4999
Epoch 3/3
200/200 [==============================] - 1s 5ms/step - loss: 7.7138 - accuracy: 0.4999
```

A pesar de realizar los mismos pasos que con el ejercicio anterior (entrenarlo 3 veces), para este caso el accuracy se mantuvo alrededor de 0.5. Encontré esta posible razón en internet. "So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory."

Phi, M. (2018). *Illustrated Guide to LSTM's and GRU's: A step by step explanation* . https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

In [17]:

```python
x, y = gen_mass_samples(N * 3)
model_30.evaluate(x.reshape(N * 3, -1, 2), y.reshape(N * 3, -1, 1))
```

```
938/938 [==============================] - 2s 2ms/step - loss: 0.6975 - accuracy: 0.4960
```

Out[17]:

```
[0.6975104212760925, 0.49599784612655564]
```

In [18]:

```python
x, y = gen_mass_samples(1)
print(f"Entrada:\n  {x}")

print(f"Resultado esperado:\n {y}")
model_30.predict(x.reshape(1,-1,2))
```

```
Entrada:
  [[[1. 1.]
  [1. 1.]
  [0. 1.]
  [0. 0.]
  [0. 0.]
  [1. 1.]
  [0. 0.]
  [1. 0.]
  [1. 1.]
  [0. 0.]
  [1. 0.]
  [0. 1.]
  [1. 1.]
  [0. 1.]
  [1. 0.]
  [1. 0.]
  [1. 1.]
  [1. 0.]
  [1. 1.]
  [0. 1.]
  [0. 1.]
  [1. 1.]
  [0. 1.]
  [1. 1.]
  [0. 0.]
  [0. 1.]
  [0. 1.]
  [1. 0.]
  [1. 1.]
  [1. 0.]
  [0. 0.]]]
Resultado esperado:
```

```
Resultado esperado:
 [[[0.]
  [1.]
  [0.]
  [1.]
  [0.]
  [0.]
  [1.]
  [1.]
  [0.]
  [1.]
  [1.]
  [1.]
  [0.]
  [0.]
  [0.]
  [0.]
  [1.]
  [0.]
  [1.]
  [0.]
  [0.]
  [1.]
  [0.]
  [1.]
  [1.]
  [1.]
  [1.]
  [1.]
  [0.]
  [0.]
  [1.]]]
```

Out[18]:

```
array([[[0.2644159 ],
        [0.38858163],
        [0.46506295],
        [0.49951443],
        [0.5331684 ],
        [0.5352827 ],
        [0.4987791 ],
        [0.5466404 ],
        [0.5036161 ],
        [0.45428535],
        [0.5279045 ],
        [0.47411638],
        [0.5221865 ],
        [0.4964233 ],
        [0.51705456],
        [0.51292443],
        [0.5010302 ],
        [0.5227158 ],
        [0.5112114 ],
        [0.4802697 ],
        [0.53676987],
        [0.5256149 ],
        [0.50718737],
        [0.53249305],
        [0.49879476],
        [0.53572875],
        [0.5116632 ],
        [0.52450204],
        [0.4929862 ],
        [0.4547787 ],
        [0.5272448 ]]], dtype=float32)
```

Q6 Extiende el modelo para incorporar celdas de memoria y evalúa si el fenomeno que observaste en la pregunta anterior sigue persistiendo (posiblemente tengas que incrementar el número de epocas).

In [146]:

```python
MAX_BIT = 30


input_dim = 2 # El numero de dimensiones de entrada.
activation = "sigmoid" # Una cadena de texto especificando la función de activación en la
capa de salida.

model_30_cell = keras.models.Sequential()
model_30_cell.add(keras.layers.LSTM(
    8,
    input_dim        = input_dim,
    return_sequences=True,
))
model_30_cell.add(keras.layers.LSTM(
    4,
    input_dim        = input_dim,
    return_sequences=True,
))
model_30_cell.add(keras.layers.Dense(2, activation="relu"))
model_30_cell.add(keras.layers.Dense(1, activation=activation))


print(model_30_cell.input_shape)
print(model_30_cell.output_shape)
```

```
(None, None, 2)
(None, None, 1)
```

In [149]:

```python
model_30_cell.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']
)
N = 10000
MAX_BIT = 30
for i in range(8):
    x, y = gen_mass_samples(N)
    model_30_cell.fit(x.reshape(N, -1, 2), y.reshape(N, -1, 1), batch_size=50, epochs=15
)
```

```
Epoch 1/15
200/200 [==============================] - 6s 15ms/step - loss: 0.3183 - accuracy: 0.8716
Epoch 2/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3192 - accuracy: 0.8711
Epoch 3/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8716
Epoch 4/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3192 - accuracy: 0.8713
Epoch 5/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3204 - accuracy: 0.8704
Epoch 6/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3204 - accuracy: 0.8703
Epoch 7/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3199 - accuracy: 0.8705
Epoch 8/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3212 - accuracy: 0.8698
Epoch 9/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3196 - accuracy: 0.8707
Epoch 10/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3171 - accuracy: 0.8726
Epoch 11/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3187 - accuracy: 0.8714
Epoch 12/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3193 - accuracy: 0.8711
Epoch 13/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 14/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3195 - accuracy: 0.8709
Epoch 15/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3199 - accuracy: 0.8705
Epoch 1/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3190 - accuracy: 0.8710
Epoch 2/15
```

```
200/200 [==============================] - 3s 16ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 3/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 4/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 5/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 6/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 7/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 8/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 9/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 10/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 11/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 12/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 13/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 14/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 15/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3189 - accuracy: 0.8711
Epoch 1/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 2/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 3/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 4/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 5/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 6/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 7/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 8/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 9/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 10/15
200/200 [==============================] - 4s 19ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 11/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 12/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 13/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 14/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 15/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3177 - accuracy: 0.8719
Epoch 1/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 2/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 3/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 4/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 5/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 6/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 7/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 8/15
```

```
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 9/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 10/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 11/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 12/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 13/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 14/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 15/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3191 - accuracy: 0.8711
Epoch 1/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 2/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 3/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 4/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 5/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 6/15
200/200 [==============================] - 3s 15ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 7/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 8/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 9/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 10/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 11/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 12/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 13/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 14/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 15/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3183 - accuracy: 0.8715
Epoch 1/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 2/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 3/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 4/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 5/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 6/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 7/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 8/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 9/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 10/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 11/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 12/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 13/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 14/15
```

```
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 15/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3185 - accuracy: 0.8715
Epoch 1/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 2/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 3/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 4/15
200/200 [==============================] - 4s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 5/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 6/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 7/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 8/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 9/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 10/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 11/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 12/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 13/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 14/15
200/200 [==============================] - 3s 16ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 15/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3186 - accuracy: 0.8716
Epoch 1/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3207 - accuracy: 0.8698
Epoch 2/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 3/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3207 - accuracy: 0.8698
Epoch 4/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 5/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 6/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 7/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 8/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 9/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 10/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 11/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 12/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 13/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3207 - accuracy: 0.8698
Epoch 14/15
200/200 [==============================] - 3s 17ms/step - loss: 0.3206 - accuracy: 0.8698
Epoch 15/15
200/200 [==============================] - 4s 18ms/step - loss: 0.3206 - accuracy: 0.8698
```

In [151]:

```
x, y = gen_mass_samples(N * 3)
model_30_cell.evaluate(x.reshape(N * 3, -1, 2), y.reshape(N * 3, -1,1))
```

```
938/938 [==============================] - 5s 5ms/step - loss: 0.3201 - accuracy: 0.8703
```

Out[151]:

```
[0.3200721740722656, 0.8702860474586487]
```

A decir verdad, no sabría decir si este nuevo comportamiento (mejora de la precisión del modelo) se debe a que utilicé un modelo Long Short Term Memory o el hecho de haber agregado una capa adicional al modelo, pero hubo un punto donde la precisión se volvió a estancar

In [152]:

```
x, y = gen_mass_samples(1)
print(f"Entrada:\n  {x}")

print(f"Resultado esperado:\n {y}")
model_30_cell.predict(x.reshape(1,-1,2))
```

```
Entrada:
  [[[1. 1.]
  [1. 1.]
  [1. 1.]
  [0. 1.]
  [0. 1.]
  [1. 0.]
  [0. 0.]
  [0. 0.]
  [1. 1.]
  [0. 1.]
  [0. 1.]
  [0. 1.]
  [1. 1.]
  [0. 0.]
  [0. 0.]
  [1. 0.]
  [1. 1.]
  [0. 1.]
  [0. 1.]
  [0. 0.]
  [0. 0.]
  [1. 1.]
  [1. 0.]
  [0. 0.]
  [1. 1.]
  [0. 1.]
  [0. 0.]
  [1. 1.]
  [0. 0.]
  [0. 0.]
  [0. 0.]]]
Resultado esperado:
 [[[0.]
  [1.]
  [1.]
  [0.]
  [0.]
  [0.]
  [1.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [1.]
  [1.]
  [0.]
  [1.]
  [0.]
  [0.]
  [0.]
  [1.]
  [0.]
  [0.]
  [0.]
  [1.]
```

```
  [0.]
  [0.]
  [1.]
  [0.]
  [1.]
  [0.]
  [0.]]]
WARNING:tensorflow:6 out of the last 6 calls to <function Model.make_predict_function.<lo
cals>.predict_function at 0x7f19c4074050> triggered tf.function retracing. Tracing is exp
ensive and the excessive number of tracings could be due to (1) creating @tf.function rep
eatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

Out[152]:

```
array([[[0.20782042],
        [1.        ],
        [1.        ],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [1.        ],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [1.        ],
        [1.        ],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [1.        ],
        [0.20782042],
        [0.20782042],
        [0.20782042],
        [1.        ],
        [0.20782042],
        [0.20782042],
        [1.        ],
        [0.20782042],
        [1.        ],
        [0.20782042],
        [0.20782042]]], dtype=float32)
```