David Rodriguez Fragoso
A01748760
Momento de Retroalimentación: Módulo 2 Implementación de un modelo de deep learning

El objetivo de esta entrega es analizar el desempeño de la implementación de un modelo de deep learning haciendo uso de frameworks y datasets reales (no usados anteriormente en clase). Para esto haremos uso del framework de tensorflow e implementaremos una red neuronal convolucional que sea capaz de clasificar imágenes de ropa de manera efectiva.

El dataset que utilizaremos será “fashion_mnist” obtenido de los datasets que tensorflow nos ofrece a través de su portal: https://www.tensorflow.org/datasets/community_catalog

Lo primero que haremos será descargar el dataset en nuestra instancia de google colab, para esto haremos uso de la función load de la librería tensorflow_datasets. Nuestro dataset cuenta con información adicional que nos puede ser útil, tales como los nombres de las clases que existen en el dataset(metadata) además de nuestra data que usaremos para entrenar y probar nuestro modelo.

```
#obtenemos el dataset de los datasets que nos ofrece tensorflow
data, metadata = tfds.load('fashion_mnist', as_supervised=True, with_info=True)
```

Ahora separaremos nuestros datos en training y test y luego visualicemos los nombres de las clases que hay en nuestro dataset:

```
#separar en training y test
training_data, tests_data = data['train'], data['test']
```

```
class_names = metadata.features['label'].names
print(class_names)
```

```
['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Ya que estamos trabajando con imágenes de 255px, normalizaremos sus valores de 0-255 a 0-1 para que el entrenamiento sea más rápido.

```python
#Normalizar las imagenes de 0-255 a 0-1
def normalize(images, tags):
    images = tf.cast(images, tf.float32)
    images /= 255 #normalización
    return images, tags

training_data = training_data.map(normalize)
tests_data = tests_data.map(normalize)
```

Ahora procederemos a crear nuestro modelo. Este será un primer entrenamiento, por lo que más adelante probaremos cambiando los hiperparámetros como el número de capas y el número de neuronas en cada una de ellas:

```python
'''
Hay 789 neuronas en la capa de entrada (28x28)px
Hay 50 neuronas en cada una de las dos capas intermedias
Hay 10 neuronas en la capa de salida (una por cada clase)

'''

# Creamos el modelo con las especificaciones del docstring de arriba
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28,1)),
    tf.keras.layers.Dense(50, activation=tf.nn.relu),
    tf.keras.layers.Dense(50, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax) #funcion de activacion softmax

])
```

Aún así, podemos obtener información del modelo con la función summary:

```
model.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 50)                39250

 dense_1 (Dense)             (None, 50)                2550

 dense_2 (Dense)             (None, 10)                510

=================================================================
Total params: 42,310
Trainable params: 42,310
Non-trainable params: 0
_________________________________________________________________
```

Haremos uso de la función shuffle para mezclar nuestros datos de entrenamiento y asegurarnos que este sea completamente aleatorio y así reducir la probabilidad de hacer overfitting. Además, separaremos esta etapa por lotes para eficientar el entrenamiento.

```
ntraining = metadata.splits['train'].num_examples
ntests = metadata.splits['test'].num_examples

print(f'Ntraining: {ntraining}')
print(f'Ntests: {ntests}')
```

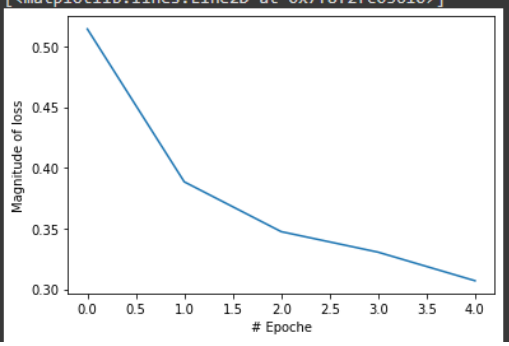
```
Ntraining: 60000
Ntests: 10000
```

```
# definiremos un chunk size para optimizar el entrenamiento
chunk_size = 32

# Usaremos la funcion shuffle para asegurarnos que nuestro entrenamiento sea aleatorio
training_data = training_data.repeat().shuffle(ntraining).batch(chunk_size)
tests_data = tests_data.batch(chunk_size)
```

```
#training
record = model.fit(training_data, epochs=5, steps_per_epoch=math.ceil(ntraining/chunk_size))
```

Entrenaremos nuestro modelo con 5 épocas (recordemos que este es un número arbitrario y podemos tratar de alterarlo para obtener mejores resultados) y graficaremos nuestra función de pérdida para comprobar si nuestro modelo ha mejorado con cada época de entrenamiento.

```
[13] #training
     record = model.fit(training_data, epochs=5, steps_per_epoch=math.ceil(ntraining/chunk_size))

     Epoch 1/5
     1875/1875 [==============================] - 13s 4ms/step - loss: 0.5144 - accuracy: 0.8201
     Epoch 2/5
     1875/1875 [==============================] - 7s 4ms/step - loss: 0.3885 - accuracy: 0.8592
     Epoch 3/5
     1875/1875 [==============================] - 7s 4ms/step - loss: 0.3474 - accuracy: 0.8719
     Epoch 4/5
     1875/1875 [==============================] - 9s 5ms/step - loss: 0.3305 - accuracy: 0.8791
     Epoch 5/5
     1875/1875 [==============================] - 8s 4ms/step - loss: 0.3070 - accuracy: 0.8864

     # graficaremos nuestra funcion de perdida
     plt.xlabel("# Epoche")
     plt.ylabel("Magnitude of loss")
     plt.plot(record.history["loss"])

     [<matplotlib.lines.Line2D at 0x7f8f2fc03610>]
```



Podemos observar que nuestra pérdida se reduce con cada época y que al contrario de esta, nuestra precisión aumenta con cada época. Este es un buen indicador de que nuestro modelo está siendo bien entrenado, pero para estar seguros, aún necesitamos probarlo con datos que no hayan sido utilizados para el entrenamiento, es decir, nuestro dataset de pruebas. Para este fin, haremos uso de la función predict.

Crearemos una función que nos permita visualizar varias imágenes en una matriz y haciendo uso de la función format desplegaremos el nivel de confianza que tiene nuestro modelo sobre su predicción (como usamos softmax, simplemente multiplicamos por 100). Para visualizar mejor las imágenes, usaremos el método binary y así nuestras imágenes se verán en blanco y negro.

```python
for test_images, test_labels in training_data.take(1):
  test_images = test_images.numpy()
  test_labels = test_labels.numpy()
  predictions = model.predict(test_images)

def graph_image(i, predictions_array, real_labels, images):
  predictions_array, real_label, img = predictions_array[i], real_labels[i], images[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img[...,0], cmap=plt.cm.binary)

  predict_label = np.argmax(predictions_array)

  plt.xlabel("{} {:2.0f}% ({})".format(
      class_names[predict_label],
      100*np.max(predictions_array),
      class_names[real_label]
  ))


rows = 5
columns = 5
img_n = rows*columns
plt.figure(figsize=(2*2*columns, 2*rows))

for i in range(img_n):
  plt.subplot(rows,2*columns, 2*i+1)
  graph_image(i, predictions, test_labels, test_images)
```

Podemos observar que en la mayoría de los casos hemos acertado, por lo que podemos concluir que nuestro modelo funciona correctamente. No obstante, realizaremos 2 pruebas más variando los hiperparámetros mencionados anteriormente para asegurarnos de que tengamos el mejor modelo posible.

Prueba 1:

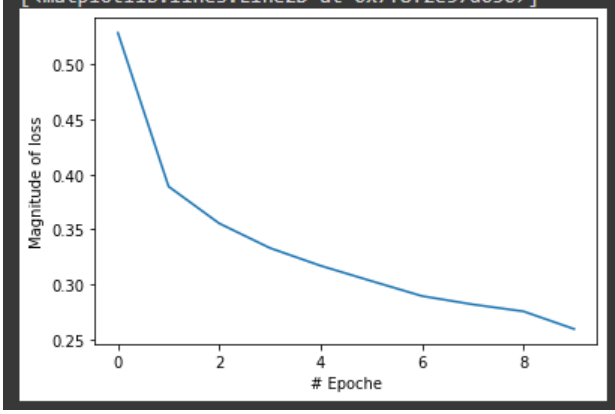
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_3 (Dense)             (None, 50)                39250

 dense_4 (Dense)             (None, 50)                2550

 dense_5 (Dense)             (None, 50)                2550

 dense_6 (Dense)             (None, 50)                2550

 dense_7 (Dense)             (None, 10)                510

=================================================================
Total params: 47,410
Trainable params: 47,410
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
1875/1875 [==============================] - 13s 4ms/step - loss: 0.5286 - accuracy: 0.8092
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3890 - accuracy: 0.8608
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3554 - accuracy: 0.8693
Epoch 4/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3331 - accuracy: 0.8774
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3170 - accuracy: 0.8823
Epoch 6/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3031 - accuracy: 0.8881
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2894 - accuracy: 0.8922
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2818 - accuracy: 0.8943
Epoch 9/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2755 - accuracy: 0.8967
Epoch 10/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2595 - accuracy: 0.9024
```

```
# graficaremos nuestra funcion de perdida
plt.xlabel("# Epoche")
plt.ylabel("Magnitude of loss")
plt.plot(record.history["loss"])
```

```
[<matplotlib.lines.Line2D at 0x7f8f2e57d050>]
```



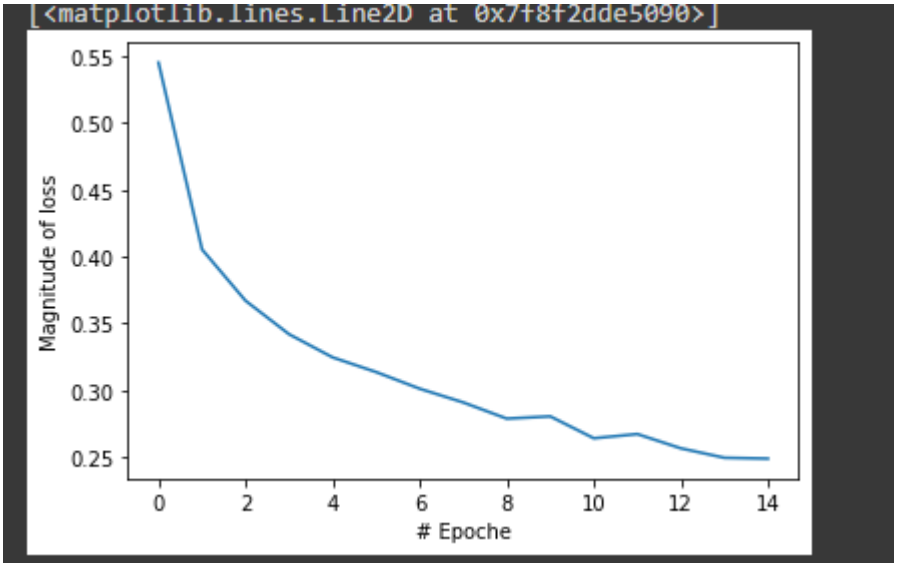
Prueba 2:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_8 (Dense)             (None, 70)                54950

 dense_9 (Dense)             (None, 70)                4970

 dense_10 (Dense)            (None, 70)                4970

 dense_11 (Dense)            (None, 70)                4970

 dense_12 (Dense)            (None, 70)                4970

 dense_13 (Dense)            (None, 70)                4970

 dense_14 (Dense)            (None, 70)                4970

 dense_15 (Dense)            (None, 70)                4970

 dense_16 (Dense)            (None, 10)                710

=================================================================
Total params: 90,450
Trainable params: 90,450
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/15
1875/1875 [==============================] - 14s 5ms/step - loss: 0.5453 - accuracy: 0.8007
Epoch 2/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.4053 - accuracy: 0.8531
Epoch 3/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3669 - accuracy: 0.8658
Epoch 4/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3418 - accuracy: 0.8760
Epoch 5/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3245 - accuracy: 0.8816
Epoch 6/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3134 - accuracy: 0.8862
Epoch 7/15
1875/1875 [==============================] - 11s 6ms/step - loss: 0.3010 - accuracy: 0.8913
Epoch 8/15
1875/1875 [==============================] - 10s 5ms/step - loss: 0.2907 - accuracy: 0.8927
Epoch 9/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2786 - accuracy: 0.8987
Epoch 10/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2802 - accuracy: 0.8970
Epoch 11/15
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2639 - accuracy: 0.9036
Epoch 12/15
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2670 - accuracy: 0.9011
Epoch 13/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2563 - accuracy: 0.9065
Epoch 14/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2492 - accuracy: 0.9079
Epoch 15/15
1875/1875 [==============================] - 9s 5ms/step - loss: 0.2486 - accuracy: 0.9083
```

```
[<matplotlib.lines.Line2D at 0x7f8f2dde5090>]
```



Podemos observar que ambas pruebas tuvieron mejores resultados que nuestro modelo original, aún así, para la segunda prueba el modelo tardó significativamente más tiempo en entrenarse que la primera prueba y sus resultados no variaron mucho (0.9024 vs 0.9083). Esto obviamente se debe a que el modelo de la segunda prueba es mucho más complicado que el de la primera prueba y además se entrenó por más tiempo.

Es así que podemos concluir que un modelo más complicado y con más tiempo de entrenamiento no siempre vale la pena, ya que por lo regular los recursos computacionales tienen un límite y para expandir este se tiene que justificar con resultados.