1. **Checking the TF version and availability of physical devices**

a. Get the version of TensorFlow running on your machine?

```
import tensorflow as tf
print(tf.__version__)
```

b. Get the type & number of physical devices available on your machine, print what are they, and test whether the GPU is available?

```
import tensorflow as tf
# Get the list of available physical devices
devices = tf.config.list_physical_devices()
print("Available physical devices:")
for device in devices:
    print(device)
# Check if GPU is available
if tf.test.is_gpu_available():
    print("GPU is available")
else:
    print("GPU is NOT available")
```

2. **Random number generator**

a. What is the need for setting a 'seed' value in any random number generation?

Setting a seed value in random number generation is also important in deep learning for the same reason as in any other context: to ensure reproducibility of results. In deep learning, random number generators are often used for tasks such as initializing weights and shuffling data during training.

If we don't set a seed value for these random number generators, the results of our model training can vary each time we run the code. This can make it difficult to debug issues, reproduce results, or compare performance between different models.

For example, when we initialize the weights of a neural network with random numbers, we want the same starting weights each time we train the model. This is important because different starting weights can result in different model performance, and we want to be able to compare the performance of different models on an equal footing.

Therefore, setting a seed value in deep learning is important for ensuring reproducibility and consistency of results, making it easier to debug and compare different models.

b. Create two random number generators using TensorFlow with the same seed of 42, create two random gaussian tensors of shape 2x3, and verify that the both tensors are identical.

```
import tensorflow as tf
# Set the seed value
tf.random.set_seed(42)
# Create two random Gaussian tensors of shape 2x3
tensor1 = tf.random.normal(shape=(2, 3))
tensor2 = tf.random.normal(shape=(2, 3))

# Verify that the tensors are identical
if tf.reduce_all(tf.equal(tensor1, tensor2)):
   print("The two tensors are identical")
else:
   print("The two tensors are NOT identical")
```

c. Create two random number generators using TensorFlow with two different seed values say 42 & 11, create two random gaussian tensors of shape 2x3, and verify that the both tensors are not identical.

```
import tensorflow as tf
# Set the seed value
tf.random.set_seed(42)
# Create two random Gaussian tensors of shape 2x3
tensor1 = tf.random.normal(shape=(2, 3))
tensor2 = tf.random.normal(shape=(2, 3))
# Verify that the tensors are identical
if tf.reduce_all(tf.equal(tensor1, tensor2)):
   print("The two tensors are identical")
else:
   print("The two tensors are NOT identical")
```

3. Shuffling of Tensors

a. Shuffle the given Tensor with and without an operation seed value. Write down your observations.

```
import tensorflow as tf
# Create a Tensor with values 0 to 9
tensor = tf.constant([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# Shuffle the Tensor without a seed value
shuffled_tensor = tf.random.shuffle(tensor)
print("Shuffled tensor without seed value:")
print(shuffled_tensor)
# Shuffle the Tensor with a seed value
tf.random.set_seed(42)
shuffled_tensor_with_seed = tf.random.shuffle(tensor)
print("Shuffled tensor with seed value:")
print(shuffled_tensor_with_seed)
```

When we shuffle the Tensor without a seed value, the order of the elements in the Tensor is randomized, but the order will be different every time we run the code. This is because the random number generator used by TensorFlow to shuffle the Tensor is not seeded, so it produces a different sequence of random numbers each time it is called.

When we shuffle the Tensor with a seed value, we set the seed of the random number generator to a specific value (in this case, 42). This ensures that the same sequence of random numbers is used every time we shuffle the Tensor, so the resulting shuffled Tensor will always be the same.

In the code above, we can see that the shuffled Tensor without a seed value and the shuffled Tensor with a seed value are different. This is because they were shuffled using different sequences of random numbers. If we run the code again, the shuffled Tensor without a seed value will be different again, but the shuffled Tensor with a seed value will be the same as before because the seed value is fixed.

In summary, using a seed value in TensorFlow's random number generators can help ensure reproducibility of results.

---

b. Show that 'operation seed' in 'tf.random.shuffle' and the 'global seed' in 'tf.random.set_seed' are different? Illustrate that having both gives the tensor in same order every time after shuffling?

```
import tensorflow as tf
# Create a Tensor with values 0 to 9
tensor = tf.constant([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# Shuffle the Tensor with a global seed value
tf.random.set_seed(42)
shuffled_tensor_with_global_seed = tf.random.shuffle(tensor)
print("Shuffled tensor with global seed value:")
print(shuffled_tensor_with_global_seed)
# Shuffle the Tensor with an operation seed value
shuffled_tensor_with_op_seed = tf.random.shuffle(tensor, seed=11)
print("Shuffled tensor with operation seed value:")
print(shuffled_tensor_with_op_seed)
# Shuffle the Tensor with both global and operation seed values
tf.random.set_seed(42)
shuffled_tensor_with_both_seeds = tf.random.shuffle(tensor, seed=11)
print("Shuffled tensor with both global and operation seed values:")
print(shuffled_tensor_with_both_seeds)
```

In the code above, we first shuffle the Tensor using only a global seed value of 42. We then shuffle the same Tensor using only an operation seed value of 11. Finally, we shuffle the same Tensor again using both a global seed value of 42 and an operation seed value of 11.

The output of the code shows that the shuffled Tensor with the global seed value is different from the shuffled Tensor with the operation seed value, because they were shuffled using different sequences of random numbers.

However, when we shuffle the Tensor with both the global and operation seed values, the shuffled Tensor is the same every time. This is because the global seed value sets the initial state of the random number generator used by TensorFlow, and the operation seed value sets the seed value for the shuffle operation specifically. By using both seeds, we ensure that the same sequence of random numbers is used every time we shuffle the Tensor, so the resulting shuffled Tensor will always be the same.

In summary, using both the global seed and operation seed in TensorFlow's random number generators can help ensure reproducibility of results, and can ensure that the same sequence of random numbers is used for a specific operation even if other operations in the graph use different random number generators

4. **Reshaping the tensors**
   a. (i) Construct a vector consisting of first 24 integers using 'numpy'.

```
import numpy as np
# Create a vector of the first 24 integers
vector = np.arange(1, 25)
print(vector)
```

o/p:

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

(ii) Convert that numpy vector into a Tensor of rank 3.

```
import tensorflow as tf
import numpy as np
# Create a NumPy vector of the first 24 integers
vector = np.arange(1, 25)
# Convert the NumPy vector to a TensorFlow Tensor of rank 3
tensor = tf.reshape(vector, (2, 3, 4))
print(tensor)
```

o/p:

```
tf.Tensor(
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]
 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]], shape=(2, 3, 4), dtype=int64)
```

(iii) Write your observations on how the elements of the vector got rearranged in the rank 3 tensor.

Sure! In the previous example, we converted a NumPy vector of the first 24 integers into a TensorFlow Tensor of rank 3 with shape `(2, 3, 4)`. This means that the resulting Tensor has 2 elements along the first dimension, 3 elements along the second dimension, and 4 elements along the third dimension.

To see how the elements of the vector got rearranged in the rank 3 Tensor, let's compare the original vector with the corresponding elements in the rank 3 Tensor:

Original vector: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
Rank 3 Tensor:
```
[ [   [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
  ],
  [   [13, 14, 15, 16],
    [17, 18, 19, 20],
    [21, 22, 23, 24]
  ]
]
```

We can see that the elements of the original vector have been rearranged into the rank 3 Tensor such that the first 4 elements form the first row of the first 2D matrix, the next 4 elements form the second row of the first 2D matrix, and so on. Similarly, the next 12 elements form the second 2D matrix, with the first 4 elements forming the first row, and so on.

Overall, we can observe that the elements of the original vector have been rearranged in the rank 3 Tensor to form a 2D matrix at each of the two higher dimensions. The first higher dimension contains two such 2D matrices, while the second higher dimension contains three such matrices.

Write a Program for text processing to identify the POS tags from the input text?

pip install nltk

```
import nltk
nltk.download('punkt') # download the necessary datasets
# input text
text = "The quick brown fox jumps over the lazy dog."
# tokenize the text into words
words = nltk.word_tokenize(text)
```

```
# identify the POS tags of the words
pos_tags = nltk.pos_tag(words)
# print the POS tags
print(pos_tags)
```

Write a Program for Text Processing to identify the common tags for the parsing?

```
import nltk
nltk.download('punkt') # download the necessary datasets
# read the text corpus
corpus = nltk.corpus.gutenberg.words('shakespeare-macbeth.txt')
# identify the POS tags of the words in the corpus
pos_tags = nltk.pos_tag(corpus)
# create a frequency distribution of the POS tags
fdist = nltk.FreqDist(tag for (word, tag) in pos_tags)
# print the 10 most common tags
print(fdist.most_common(10))
```

Write a program for Text Processing to identify the Named Entity Recognition from the given Text?

```
pip install spacy
```

```
import spacy
# load the English language model
nlp = spacy.load('en_core_web_sm')
# input text
text = "Apple is looking at buying U.K. startup for $1 billion."
# apply NER on the text
doc = nlp(text)
# iterate over the entities in the text
for entity in doc.ents:
    print(entity.text, entity.label_)
```

Write the Code for implementing the LSTM model for the sentiment Analysis on IMDB Movie Reviews dataset with Sigmoid Optimizer

```
import numpy as np
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing import sequence
# set the maximum number of words to use
max_features = 5000
# load the IMDB movie reviews dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=max_features)
# truncate and pad the input sequences
maxlen = 100
```

```
X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)
# create the LSTM model
model = Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# compile the model with the Sigmoid optimizer
model.compile(loss='binary_crossentropy', optimizer='sigmoid',
metrics=['accuracy'])
# train the model
batch_size = 32
epochs = 10
model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs,
validation_data=(X_test, y_test))
# evaluate the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Write the Code for implementing the LSTM model for the sentiment Analysis on IMDB Movie Reviews dataset with Adam Optimizer

```
import numpy as np
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing import sequence
# set the maximum number of words to use
max_features = 5000
# load the IMDB movie reviews dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=max_features)
# truncate and pad the input sequences
maxlen = 100
X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)
# create the LSTM model
model = Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# compile the model with the Adam optimizer
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
# train the model
```

```
batch_size = 32
epochs = 10
model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs,
validation_data=(X_test, y_test))
# evaluate the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Write the Code for implementing the LSTM model for the sentiment Analysis on IMDB Movie Reviews dataset with 'RMSProp' Optimizer?

```
import numpy as np
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing import sequence
# set the maximum number of words to use
max_features = 5000
# load the IMDB movie reviews dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=max_features)
# truncate and pad the input sequences
maxlen = 100
X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)
# create the LSTM model
model = Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# compile the model with the RMSprop optimizer
model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
# train the model
batch_size = 32
epochs = 10
model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs,
validation_data=(X_test, y_test))
# evaluate the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

7a)

i) NumPy: NumPy is a fundamental library for scientific computing in Python, and it is used for array manipulation, linear algebra operations, and numerical computations.

Pandas: Pandas is a data manipulation library that provides data structures for data analysis, including data frames and series.

Matplotlib: Matplotlib is a plotting library that is used to create visualizations, including graphs, charts, and histograms.

Keras/TensorFlow: Keras is a high-level neural networks API that can run on top of TensorFlow, which is an open-source software library for dataflow and differentiable programming across a range of tasks.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow import keras
```

ii)

```
from tensorflow import keras
from tensorflow.keras import layers

# create the model
model = keras.Sequential([
    layers.Dense(units=4, input_shape=(2,), activation='relu'),
    layers.Dense(units=2, activation='softmax')])

# compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# print the model summary
model.summary()
```

In this example, the model has two layers: a fully connected layer with four units and a ReLU activation function, and a fully connected layer with two units and a softmax

activation function. The input shape of the first layer is (2,), which corresponds to the two input features.

The model is then compiled with an optimizer of 'adam', a loss function of 'categorical_crossentropy' (assuming a classification problem), and an accuracy metric.

Finally, the model summary is printed to show the architecture of the model and the number of trainable parameters

iii)

Based on the figure provided, the appropriate loss function for this binary classification task would be binary_crossentropy. Here's an example of how to compile the model with the specified optimizer, loss function, and metrics, and print the model summary

```
from tensorflow import keras

from tensorflow.keras import layers

# create the model

model = keras.Sequential([

 layers.Dense(units=4, input_shape=(2,), activation='relu'),

  layers.Dense(units=1, activation='sigmoid')])

# compile the model

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# print the model summary

model.summary()
```

In this example, the model has two layers: a fully connected layer with four units and a ReLU activation function, and a fully connected layer with one unit and a sigmoid activation function, which is suitable for binary classification tasks.

The model is then compiled with an optimizer of 'adam', a loss function of 'binary_crossentropy', and an accuracy metric.

Finally, the model summary is printed to show the architecture of the model and the number of trainable parameters, which should match the figure provided.

b ii)

```
from tensorflow import keras

from tensorflow.keras import layers

# define the input layer

inputs = keras.Input(shape=(2,))

# define the hidden layer

hidden = layers.Dense(units=4, activation='relu')(inputs)

# define the output layer

outputs = layers.Dense(units=2, activation='softmax')(hidden)

# create the model

model = keras.Model(inputs=inputs, outputs=outputs)

# compile the model

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# print the model summary

model.summary()
```

In this example, we define the input layer with an input shape of (2,). We then define a hidden layer with four units and a ReLU activation function, and connect it to the input layer using the functional API. Finally, we define an output layer with two units and a softmax activation function, and connect it to the hidden layer.

The model is then created by specifying the input and output layers, and compiled with an optimizer of 'adam', a loss function of 'categorical_crossentropy' (assuming a classification problem), and an accuracy metric.

Finally, the model summary is printed to show the architecture of the model and the number of trainable parameters.

8a

i)from tensorflow import keras

from tensorflow.keras import layers

additional libraries for loading and preprocessing image data

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.preprocessing.image import ImageDataGenerator

We import Keras and its layers module, which contains all the necessary layers for building a CNN. We also import NumPy and Matplotlib for working with image data and visualizing the results, and we import the ImageDataGenerator class from Keras' preprocessing module for loading and preprocessing the image data.

ii)from tensorflow import keras

from tensorflow.keras import layers

define the model

model = keras.Sequential([

    # convolutional layer with 32 filters and a 3x3 kernel

    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),

```
    # max pooling layer with a 2x2 pool size
    layers.MaxPooling2D((2, 2)),
    # flatten the feature maps to a 1D vector
    layers.Flatten(),
    # fully connected layer with 128 units
    layers.Dense(128, activation='relu'),
    # output layer with 10 units (one for each class)
    layers.Dense(10, activation='softmax')
])
# print the model summary
model.summary()
```

In this example, we define a simple CNN architecture for image classification on the MNIST dataset. The input images have a shape of (28, 28, 1), representing grayscale images with a size of 28x28 pixels.

The model consists of a convolutional layer with 32 filters and a 3x3 kernel, followed by a max pooling layer with a 2x2 pool size. We then flatten the feature maps into a 1D vector and pass them through a fully connected layer with 128 units, followed by an output layer with 10 units (one for each class).

The input_shape argument in the first layer specifies the shape of the input images. The activation argument in each layer specifies the activation function used for that layer.

Finally, we print the model summary to show the architecture of the model and the number of trainable parameters.

iii)from tensorflow import keras

```
from tensorflow.keras import layers
# define the model
model = keras.Sequential([
    # convolutional layer with 32 filters and a 3x3 kernel
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    # max pooling layer with a 2x2 pool size
    layers.MaxPooling2D((2, 2)),
    # flatten the feature maps to a 1D vector
    layers.Flatten(),
    # fully connected layer with 128 units
    layers.Dense(128, activation='relu'),
    # output layer with 10 units (one for each class)
    layers.Dense(10, activation='softmax')
])
# compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# print the model summary
model.summary()
```

8b)ii

```
from tensorflow import keras

from tensorflow.keras import layers

# define the input shape

inputs = keras.Input(shape=(28, 28, 1)

# add convolutional layer with 32 filters and a 3x3 kernel

conv1 = layers.Conv2D(32, (3, 3), activation='relu')(inputs)

# add max pooling layer with a 2x2 pool size

pool1 = layers.MaxPooling2D((2, 2))(conv1)

# add flatten layer

flatten = layers.Flatten()(pool1)

# add fully connected layer with 128 units

dense1 = layers.Dense(128, activation='relu')(flatten)

# add output layer with 10 units (one for each class)

outputs = layers.Dense(10, activation='softmax')(dense1)

# create the model

model = keras.Model(inputs=inputs, outputs=outputs)

# print the model summary

model.summary()
```

In this example, we define the input shape as (28, 28, 1) and create a `Input` layer. We then add a convolutional layer with 32 filters and a 3x3 kernel to the input layer. Next,

we add a max pooling layer with a 2x2 pool size, a flatten layer, a fully connected layer with 128 units, and an output layer with 10 units (one for each class).

After defining the layers, we create the model using the `Model` class, specifying the input and output layers. Finally, we print the model summary to show the architecture of the model.

8ci)

```
import tensorflow as tf

# load CIFAR-100 dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar100.load_data()

# print the shape of the training and test datasets

print("Training data shape:", x_train.shape)

print("Training labels shape:", y_train.shape)

print("Test data shape:", x_test.shape)

print("Test labels shape:", y_test.shape)
```

In this example, we use the `load_data` function to load the CIFAR-100 dataset. The dataset is split into training and test sets, with 50,000 and 10,000 images, respectively. The shape of the training data is `(50000, 32, 32, 3)` which means there are 50,000 images with a width and height of 32 pixels and 3 color channels (RGB). The shape of the training labels is `(50000, 1)` which means there are 50,000 corresponding labels, each with a single integer value representing the class of the image. Similarly, the shape of the test data is `(10000, 32, 32, 3)` and the shape of the test labels is `(10000, 1)`.

8 c ii)

```
import tensorflow as tf

import matplotlib.pyplot as plt
```

```
# load CIFAR-100 dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar100.load_data()

# define class names for CIFAR-100

class_names = ['aquatic mammals', 'fish', 'flowers', 'food containers', 'fruit and vegetables',

        'household electrical devices', 'household furniture', 'insects', 'large carnivores',

        'large man-made outdoor things', 'large natural outdoor scenes', 'large omnivores and herbivores',

        'medium-sized mammals', 'non-insect invertebrates', 'people', 'reptiles', 'small mammals',

        'trees', 'vehicles 1', 'vehicles 2']

# plot some images from the training set

plt.figure(figsize=(10, 10))

for i in range(25):

    plt.subplot(5, 5, i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(x_train[i], cmap=plt.cm.binary)

    plt.xlabel(class_names[y_train[i][0]])

plt.show()
```

```
# plot some images from the test set
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_test[i][0]])
plt.show()
```

8c iii) The CIFAR-100 dataset consists of 32x32 color images, with 3 channels (RGB). Therefore, the resolution of each image in the dataset is 32 pixels by 32 pixels.

8c iv)

```
import tensorflow as tf
# load CIFAR-100 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar100.load_data()
# normalize the pixel values between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0
# add an extra dimension for the batches
x_train = x_train.reshape(x_train.shape[0], 32, 32, 3)
```

```
x_test = x_test.reshape(x_test.shape[0], 32, 32, 3)

# convert class vectors to binary class matrices

y_train = tf.keras.utils.to_categorical(y_train, 100)

y_test = tf.keras.utils.to_categorical(y_test, 100)
```

In this example, we first load the CIFAR-100 dataset using the `load_data` function. Then, we normalize the pixel values of the images by dividing them by 255.0, which scales them to the range [0, 1]. We also add an extra dimension to the image arrays using the `reshape` function, so that the shape of the training and test data becomes `(num_samples, 32, 32, 3)`, where `num_samples` is the number of images in the dataset. Finally, we convert the class labels to binary class matrices using the `to_categorical` function from Keras. This is necessary for multi-class classification tasks, and ensures that the labels are in the correct format for training the neural network.