1.Checking the TF version and availability of physical devices

In [1]:
```python
import tensorflow as tf
print(tf.__version__)
# 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")
```

```
2.11.0
Available physical devices:
PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')
WARNING:tensorflow:From C:\Users\dasar\AppData\Local\Temp\ipykernel_16732\79825394.py:10: is_gpu_available (from tensorflow.python.framework.test_util) is deprecated and will be removed in a future version.
Instructions for updating:
Use `tf.config.list_physical_devices('GPU')` instead.
GPU is NOT available
```

2.Random number generator

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

Ans.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.

In [3]:
```python
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))
```

In [4]:
```python
tf.random.set_seed(42)
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")
```

```
The two tensors are 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.

In [5]:
```python
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")
```

```
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.

In [6]:
```python
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)
#Observations:-
 '''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
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 sequen
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 shuffle
In summary, using a seed value in TensorFlow's random number generators can help ensure reproducibility of results.
'''
```

```
Shuffled tensor without seed value:
tf.Tensor([4 9 3 6 5 7 2 0 8 1], shape=(10,), dtype=int32)
Shuffled tensor with seed value:
tf.Tensor([7 6 3 0 8 9 5 4 1 2], shape=(10,), dtype=int32)
```

a. 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?

In [7]:
```python
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)
#Illustration:-
'''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. Fina
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 wer
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
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
'''
```

```
Shuffled tensor with global seed value:
tf.Tensor([7 6 3 0 8 9 5 4 1 2], shape=(10,), dtype=int32)
Shuffled tensor with operation seed value:
tf.Tensor([7 0 5 2 8 3 4 6 1 9], shape=(10,), dtype=int32)
Shuffled tensor with both global and operation seed values:
tf.Tensor([7 0 5 2 8 3 4 6 1 9], shape=(10,), dtype=int32)
```

4. Reshaping the tensors

a. (i) Construct a vector consisting of first 24 integers using 'numpy'.

In [8]:
```python
import numpy as np
# Create a vector of the first 24 integers
vector = np.arange(1, 25)
print(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]
```

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

In [9]:
```python
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)
```

```
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=int32)
```

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

Ans.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.

b(i) Create a tensor of rank 2. (ii) Convert that tensor into another tensor of shape 2x2x1 using 'tf.newaxis'

In [10]:
```python
import numpy as np

# Create a 2x3 tensor (i.e., a matrix)
tensor = np.array([[1, 2, 3], [4, 5, 6]])

# Print the tensor
print(tensor)
import tensorflow as tf

# Create the original tensor
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])

# Reshape the tensor using tf.newaxis
new_tensor = tensor[:,:,tf.newaxis]

# Print the original and new tensors
print("Original tensor:\n", tensor)
print("\nNew tensor:\n", new_tensor)
```

```
[[1 2 3]
 [4 5 6]]
Original tensor:
 tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)

New tensor:
 tf.Tensor(
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]], shape=(2, 3, 1), dtype=int32)
```

c (i) Create a tensor of rank 2. (ii) Convert that tensor into another tensor of shape 2x2x1 using 'tf.expand_dims'.

In [11]:
```python
import tensorflow as tf

# Create a 2x3 tensor (i.e., a matrix)
tensor = tf.constant([[1, 2, 3], [4, 5, 6]])

# Print the original tensor
print("Original tensor:\n", tensor)

# Use tf.expand_dims to add an extra dimension along the third axis
new_tensor = tf.expand_dims(tensor, axis=-1)

# Print the new tensor
print("\nNew tensor:\n", new_tensor)
```

```
Original tensor:
 tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)

New tensor:
 tf.Tensor(
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]], shape=(2, 3, 1), dtype=int32)
```

Linear Regression full experiment on boston housing prediction deep learning

In [3]:
```python
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the Boston Housing dataset
boston = load_boston()

# Extract the features and target from the dataset
X = boston.data
y = boston.target

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Define the model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(1, input_dim=13, activation='linear')
])

# Compile the model
model.compile(optimizer='adam', loss='mse')

# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)

# Evaluate the model on the test set
loss = model.evaluate(X_test, y_test, verbose=0)
print("Test loss:", loss)

# Make predictions on the test set
y_pred = model.predict(X_test)
model.summary()

# Calculate the coefficient of determination (R^2)
r_squared = np.corrcoef(y_test, y_pred.squeeze())[0,1]**2
print("R^2:", r_squared)
```

```
C:\Users\dasar\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function load_boston is deprecated; `load_bos
ton` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np

        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
Test loss: 472.87054443359375
4/4 [==============================] - 0s 1ms/step
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 1)                 14

=================================================================
Total params: 14
Trainable params: 14
Non-trainable params: 0
_________________________________________________________________
R^2: 0.5360499810778805
```

Regularization full experiment

https://colab.research.google.com/drive/1r6dz-iqo2uwAnMeugI1GFfPT9U344GKr?usp=sharing

In [16]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
# create the model
model = Sequential([
layers.Dense(units=4, input_shape=(2,), activation='relu'),
layers.Dense(units=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()
from tensorflow import keras
from tensorflow.keras import layers
# define the input layer
inputs = layers.Input(shape=(2,))
# define the hidden layer
hidden = layers.Dense(units=4, activation='relu')(inputs)
# define the output layer
hidden1 = layers.Dense(units=2, activation='relu')(hidden)
outputs = layers.Dense(units=1, activation='softmax')(hidden1)
# create the model
model = keras.Model(inputs=inputs, outputs=outputs)
# compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_53 (Dense)            (None, 4)                 12

 dense_54 (Dense)            (None, 2)                 10

 dense_55 (Dense)            (None, 1)                 3

=================================================================
Total params: 25
Trainable params: 25
Non-trainable params: 0
_________________________________________________________________
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 2)]               0

 dense_56 (Dense)            (None, 4)                 12

 dense_57 (Dense)            (None, 2)                 10

 dense_58 (Dense)            (None, 1)                 3

=================================================================
Total params: 25
Trainable params: 25
Non-trainable params: 0
_________________________________________________________________
```

Reguralization

In [17]:
```python
import tensorflow as tf
from sklearn.datasets import load_boston
from sklearn.linear_model import Lasso,Ridge,LinearRegression
from sklearn.model_selection import train_test_split
import pandas as pd
boston=load_boston()
```

```
C:\Users\dasar\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function load_boston is deprecated; `load_bos
ton` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np

        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
```

In [29]:
```python
boston_df=pd.DataFrame(boston.data,columns=boston.feature_names)
boston_df["Price"]=boston.target
features = boston_df.columns[0:11]
target = boston_df.columns[-1]
X=boston_df[features].values
y=boston_df[target].values
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3)
```

In [30]:
```python
lr=LinearRegression()
lr.fit(X_train,y_train)
lr.score(X_test,y_test)
```

Out[30]:
```
0.6759669577382478
```

In [31]:
```python
#Ridge Regression Model
ridgeReg = Ridge(alpha=100)

ridgeReg.fit(X_train,y_train)

#train and test scorefor ridge regression
train_score_ridge = ridgeReg.score(X_train, y_train)
test_score_ridge = ridgeReg.score(X_test, y_test)
print("\nRidge Model............................................\n")
print("The train score for ridge model is {}".format(train_score_ridge))
print("The test score for ridge model is {}".format(test_score_ridge))
```

```
Ridge Model............................................

The train score for ridge model is 0.6032618520675852
The test score for ridge model is 0.629897830799812
```

In [32]:
```python
#Lasso Regression Model
lassoReg = Lasso(0.1)

lassoReg.fit(X_train,y_train)

#train and test scorefor ridge regression
train_score_lasso = lassoReg.score(X_train, y_train)
test_score_lasso = lassoReg.score(X_test, y_test)
print("\nLasso Model............................................\n")
print("The train score for ridge lasso is {}".format(train_score_lasso))
print("The test score for ridge lasso is {}".format(test_score_lasso))
```

```
Lasso Model............................................

The train score for ridge lasso is 0.6416303394912597
The test score for ridge lasso is 0.6593403138128935
```

In [ ]: