Deep Learning

Experiment 01

TENSORFLOW AND KERAS

**Tensorflow:**

TensorFlow is an open-source deep learning library developed by Google. It is one of the most widely used libraries for building and training machine learning models, particularly neural networks. TensorFlow supports both CPU and GPU computing, making it suitable for a wide range of hardware configurations. The library is highly flexible and can be used for various machine learning tasks, such as image recognition, natural language processing, and more.

**Tensorflow Methods:**

TensorFlow offers a vast array of methods and functions for tensor manipulation, mathematical operations, and deep learning model building.

1) Tensor Creation Methods: These methods are used to create tensors, which are the fundamental data structures in TensorFlow. A Tensor is much the same as a multidimensional array. A Tensor contains values in one or more dimensions.

* tf.constant: Creates a constant tensor with a specific value.

Syntax:

import tensorflow as tf

tensor\_a = tf.constant([1, 2, 3])

print(tensor\_a)

* tf.Variable: Creates a variable tensor that can be updated during training.

Syntax:

import tensorflow as tf

tensor\_b = tf.Variable([4, 5, 6])

print(tensor\_b)

2) Mathematical Operations: TensorFlow supports various mathematical operations that can be performed on tensors.

* tf.add: Element-wise addition of tensors.

import tensorflow as tf

tensor\_a = tf.constant([1, 2, 3])

tensor\_b = tf.Variable([4, 5, 6])

result = tf.add(tensor\_a, tensor\_b)

print(result)

* tf.matmul: Matrix multiplication of tensors.

import tensorflow as tf

tensor\_a = tf.constant([[1, 2], [3, 4]])

tensor\_b = tf.constant([[5, 6], [7, 8]])

result = tf.matmul(tensor\_a, tensor\_b)

print(result)

* tf.reduce\_sum: Computes the sum of elements along specified axes.

import tensorflow as tf

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

sum\_axis\_0 = tf.reduce\_sum(tensor, axis=0)

sum\_axis\_1 = tf.reduce\_sum(tensor, axis=1)

print("Sum along axis 0:", sum\_axis\_0)

print("Sum along axis 1:", sum\_axis\_1)

* tf.reduce\_mean: Computes the mean of elements along specified axes.

import tensorflow as tf

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

mean\_axis\_0 = tf.reduce\_mean(tensor, axis=0)

mean\_axis\_1 = tf.reduce\_mean(tensor, axis=1)

print("Mean along axis 0:", mean\_axis\_0)

print("Mean along axis 1:", mean\_axis\_1)

3) Activation Functions: These methods apply various activation functions to the output of neural network layers.

* ReLU (Rectified Linear Activation):

import tensorflow as tf

# Define input data

x = tf.constant([-2.0, -1.0, 0.0, 1.0, 2.0])

# Apply ReLU activation

relu\_output = tf.nn.relu(x)

print("ReLU Output:")

print(relu\_output.numpy()) # Convert the tensor to a NumPy array for printing

ReLU Output:

[0. 0. 0. 1. 2.]

* Sigmoid activation function.

# Apply sigmoid activation

sigmoid\_output = tf.nn.sigmoid(x)

print("Sigmoid Output:")

print(sigmoid\_output.numpy())

Sigmoid Output:

[0.11920292 0.26894143 0.5 0.7310586 0.8807971 ]

* Softmax activation function:

# Apply softmax activation

softmax\_output = tf.nn.softmax(logits, axis=-1)

print("Softmax Output:")

print(softmax\_output.numpy())

Softmax Output:

[[0.09003057 0.24472848 0.66524094]

[0.09003057 0.24472848 0.66524094]]

4) Loss Functions: TensorFlow provides various loss functions that are used to compute the difference between predicted values and true labels during training.

* Mean Squared Error (MSE) Loss (Regression):

import tensorflow as tf

# Simulated target values and predicted values

target\_values = tf.constant([3.0, 4.0, 5.0])

predicted\_values = tf.constant([2.8, 3.9, 5.1])

# Compute the mean squared error loss

mse\_loss = tf.reduce\_mean(tf.square(target\_values - predicted\_values))

print("Mean Squared Error Loss:")

print(mse\_loss.numpy())

Mean Squared Error Loss:

0.019999994

* Binary Cross-Entropy Loss (Binary Classification):

import tensorflow as tf

# Simulated target labels (0 or 1) and predicted probabilities

target\_labels = tf.constant([1, 0, 1], dtype=tf.float32)

predicted\_probs = tf.constant([0.9, 0.2, 0.8], dtype=tf.float32)

# Compute the binary cross-entropy loss

bce\_loss = tf.keras.losses.binary\_crossentropy(target\_labels, predicted\_probs)

print("Binary Cross-Entropy Loss:")

print(bce\_loss.numpy())

Binary Cross-Entropy Loss: 0.1838824

5) Optimizers: TensorFlow offers different optimization algorithms to update model parameters during training.

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import SGD, Adam, RMSprop

# Simulated data

x\_train = tf.constant([[1.0], [2.0], [3.0], [4.0]])

y\_train = tf.constant([[2.0], [4.0], [6.0], [8.0]])

# Create and compile a Sequential model

def create\_model(optimizer):

model = Sequential([

Dense(units=1, input\_shape=(1,))

])

model.compile(optimizer=optimizer, loss='mean\_squared\_error')

return model

# Train the model with different optimizers

optimizers = [SGD(learning\_rate=0.01), Adam(learning\_rate=0.01), RMSprop(learning\_rate=0.01)]

for optimizer in optimizers:

model = create\_model(optimizer)

print(f"Training with {type(optimizer).\_\_name\_\_} optimizer:")

model.fit(x\_train, y\_train, epochs=100, verbose=0)

Training with SGD optimizer:

Training with Adam optimizer:

Training with RMSprop optimizer:

**Keras:**

Keras is an open-source high-level neural networks API written in Python and integrated with TensorFlow as its official high-level API since TensorFlow 2.0. It provides an easy-to-use interface for building, training, evaluating, and deploying deep learning models. Keras allows rapid prototyping of neural network architectures and enables both beginners and experts to work efficiently with neural networks.

Methods:

Here are some common methods and their syntax in Keras:

1. Sequential Model: The `Sequential` model is the simplest type of Keras model, where layers are stacked sequentially. It is appropriate for most feedforward neural networks.

Syntax:

import tensorflow as tf

from tensorflow.keras import models, layers

# Define the input dimension

input\_dim = 784 # Example input dimension for an image with 28x28 pixels

# Create a Sequential model

model = models.Sequential([

layers.Dense(units=64, activation='relu', input\_shape=(input\_dim,)),

layers.Dense(units=10, activation='softmax')

])

# Print the model summary

model.summary()

Output:

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_2 (Dense) (None, 64) 50240

dense\_3 (Dense) (None, 10) 650

=================================================================

Total params: 50,890

Trainable params: 50,890

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. Functional API: The functional API allows for more complex model architectures, including multi-input and multi-output models.

Syntax:

import numpy as np

from tensorflow.keras import models, layers

# Example data

x\_train = np.random.rand(100, 10) # Replace with your actual training data

y\_train = np.random.randint(0, 2, size=(100,)) # Replace with your actual training labels

x\_val = np.random.rand(30, 10) # Replace with your actual validation data

y\_val = np.random.randint(0, 2, size=(30,)) # Replace with your actual validation labels

# Create a Sequential model

model = models.Sequential([

layers.Dense(units=64, activation='relu', input\_shape=(10,)),

layers.Dense(units=10, activation='softmax')

])

3. Model Compilation: Before training the model, you need to compile it with an optimizer, a loss function, and optional evaluation metrics.

Syntax:

# Compile the model

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

4. Model Training : Train the model on your training data using the `fit` method.

Syntax:

# Train the model with actual data

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_val, y\_val))

5. Model Evaluation : Evaluate the model on your test data using the `evaluate` method.

Syntax:

# Model Evaluation

loss, accuracy = model.evaluate(x\_val, y\_val)

print("Validation loss:", loss)

print("Validation accuracy:", accuracy)

Output:

1/1 [==============================] - 0s 32ms/step - loss: 1.3806 - accuracy: 0.4667 Validation loss: 1.3805994987487793 Validation accuracy: 0.46666666865348816

6. Model Prediction : Make predictions using the trained model on new data.

Syntax:

# Model Prediction

x\_new\_data = np.random.rand(5, 10) # Replace with your actual new data

predictions = model.predict(x\_new\_data)

print("Predictions:", predictions)

Output:

1/1 [==============================] - 0s 85ms/step Predictions: [[0.27624795 0.23622085 0.02619928 0.07588329 0.09317679 0.05301536 0.05519771 0.04599153 0.07468914 0.06337814] [0.22535299 0.20943052 0.0351066 0.08885428 0.10418931 0.07151973 0.05858627 0.05650515 0.07561583 0.07483944] [0.20169051 0.3009162 0.03441966 0.07517613 0.11240764 0.04545456 0.06047007 0.03680461 0.07440072 0.05825995] [0.19390357 0.33111084 0.02663614 0.07554521 0.12630741 0.04490591 0.04362112 0.04056433 0.06849369 0.04891184] [0.27216634 0.1713839 0.03246921 0.10107397 0.12578835 0.051087 0.05424258 0.05313843 0.0716437 0.06700654]]

7. Model Saving and Loading : Save the trained model to disk and load it back.

Syntax:

# Model Saving and Loading

model.save('my\_model.h5')

# Load the model

loaded\_model = models.load\_model('my\_model.h5')

These are some common methods and their syntax in Keras (TensorFlow 2.x). Keras provides many other functionalities and layers to build more complex models, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), transfer learning, and more.