

DOCUMENTATION

TENSORFLOW

Shuhaib | Tensorflow practical test | 28/06/2024

ACTIVITY – 1

AIM:

Create a 3*3 matrix of ones and a 3*3 matrix of zero. Add them together using TensorFlow.

REQUIREMENTS:

-  Pc
-  Jupyter lab/ vs code

PROCEDURE/CODE:

- Step-1

```
import tensorflow as tf
```

- Step-2

```
# Create a 3x3 matrix of ones  
ones_matrix = tf.constant([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
```

- Step-3

```
# Create a 3x3 matrix of zeros  
zeros_matrix = tf.constant([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
```

- Step-4

```
# Add the two matrices together  
result_matrix = tf.add(ones_matrix, zeros_matrix)  
  
print(result_matrix)
```

OUTPUT:

```
tf.Tensor(  
[[1 1 1]  
 [1 1 1]  
 [1 1 1]], shape=(3, 3), dtype=int32)
```

RESULT:

The resulting matrix is printed, showing a 3x3 matrix filled with ones.

ACTIVITY – 2

AIM:

implement a custom layer in tensorflow that performs a specific operations (eg.customactivation, function) use this layer as a simple model.

REQUIREMENTS:

- 🖥️ Pc
- 📄 jupyternotebook / vs code
- 📦 TensorFlow library
- 🔧

PROCEDURE/CODE:

- Step-1

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Step 1: Define the custom activation function
def custom_relu(x):
    return tf.minimum(tf.maximum(0.0, x), 6.0)
```

- Step-2

```
# Step 2: Create a custom Layer that applies this activation function
class CustomActivationLayer(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(CustomActivationLayer, self).__init__(**kwargs)

    def call(self, inputs):
        return custom_relu(inputs)
```

- Step-3

```
# Step 3: Build a simple model using the custom layer
model = Sequential([
    Dense(10, input_shape=(5,)), # Input Layer
    CustomActivationLayer(),      # Custom activation Layer
    Dense(1)                      # Output Layer
])

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

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

- Step-4

```
# Step 4: Generate some random data and train the model
import numpy as np

X_train = np.random.rand(100, 5)
y_train = np.random.rand(100, 1)

# Train the model
model.fit(X_train, y_train, epochs=5)
```

OUTPUT:

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 10)	60
custom_activation_layer (CustomActivationLayer)	(None, 10)	0
dense_5 (Dense)	(None, 1)	11

Total params: 71 (284.00 B)

Trainable params: 71 (284.00 B)

Non-trainable params: 0 (0.00 B)

Epoch 1/5

4/4 ————— 4s 4ms/step - loss: 0.5804

Epoch 2/5

4/4 ————— 0s 5ms/step - loss: 0.5093

Epoch 3/5

4/4 ————— 0s 14ms/step - loss: 0.4535

Epoch 4/5

4/4 ————— 0s 11ms/step - loss: 0.4030

Epoch 5/5

4/4 ————— 0s 3ms/step - loss: 0.3951

RESULT:




Successfully executed program and displayed the output.

ACTIVITY – 3

AIM:

implement a simple example of distributed training using tf.distribute strategy to train a model on multiple GPUs

REQUIREMENTS:

-  Pc
-  jupyter notebook / vs code
-  TensorFlow library

PROCEDURE/CODE:

▪ Step-1

```
import tensorflow as tf

# Step 1: Set up the distribution strategy
strategy = tf.distribute.MirroredStrategy()

print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

▪ Step-2

```
# Step 2: Prepare the dataset
# Load and preprocess the MNIST dataset
def preprocess(x, y):
    x = tf.cast(x, tf.float32) / 255.0
    y = tf.cast(y, tf.int64)
    return x, y

batch_size = 64

(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()
train_images = train_images[..., tf.newaxis]
test_images = test_images[..., tf.newaxis]

train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
train_dataset = train_dataset.map(preprocess).shuffle(60000).batch(batch_size)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
test_dataset = test_dataset.map(preprocess).batch(batch_size)
```

- Step-3

```
# Step 3: Create the model inside the strategy scope
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D(64, 3, activation='relu'),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
```

- Step-4

```
# Step 4: Compile the model
model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

- Step-5

```
# Step 5: Train the model
model.fit(train_dataset, epochs=5, validation_data=test_dataset)
```

- Step-6

```
# Step 6: Evaluate the model
test_loss, test_acc = model.evaluate(test_dataset)
print(f'Test accuracy: {test_acc}')
```

OUTPUT:

```
Epoch 1/5
938/938 ————— 37s 30ms/step - accuracy: 0.8947 - loss: 0.3552 - val_accuracy: 0.9839 - val_loss: 0.0495
Epoch 2/5
938/938 ————— 30s 30ms/step - accuracy: 0.9849 - loss: 0.0500 - val_accuracy: 0.9885 - val_loss: 0.0331
Epoch 3/5
938/938 ————— 29s 29ms/step - accuracy: 0.9891 - loss: 0.0330 - val_accuracy: 0.9895 - val_loss: 0.0305
Epoch 4/5
938/938 ————— 29s 29ms/step - accuracy: 0.9926 - loss: 0.0234 - val_accuracy: 0.9888 - val_loss: 0.0329
Epoch 5/5
938/938 ————— 28s 28ms/step - accuracy: 0.9941 - loss: 0.0197 - val_accuracy: 0.9913 - val_loss: 0.0292
157/157 ————— 2s 13ms/step - accuracy: 0.9891 - loss: 0.0375
Test accuracy: 0.9912999868392944
```

```
157/157 ————— 3s 14ms/step - accuracy: 0.9891 - loss: 0.0375
Test accuracy: 0.9912999868392944
```

RESULT:

Program executed successfully and display the output

ACTIVITY – 4

AIM:

Write a TensorFlow function to calculate precision, recall, and F1-score for a multi-class classification problem

REQUIREMENTS:

- ✚ Pc
- ✚ jupyternotebook/ vs code
- ✚ TensorFlow library

PROCEDURE/CODE:

▪ Step-1

```
import tensorflow as tf

def simple_multi_class_metrics(y_true, y_pred, num_classes):
    # Convert predictions and true labels to one-hot encoding
    y_pred_onehot = tf.one_hot(y_pred, depth=num_classes)
    y_true_onehot = tf.one_hot(y_true, depth=num_classes)
```

▪ Step-2

```
# Calculate true positives, false positives, and false negatives
tp = tf.reduce_sum(y_true_onehot * y_pred_onehot, axis=0)
fp = tf.reduce_sum((1 - y_true_onehot) * y_pred_onehot, axis=0)
fn = tf.reduce_sum(y_true_onehot * (1 - y_pred_onehot), axis=0)
```

▪ Step-3

```
# Calculate precision, recall, and F1-score for each class
precision = tp / (tp + fp + tf.keras.backend.epsilon())
recall = tp / (tp + fn + tf.keras.backend.epsilon())
f1_score = 2 * precision * recall / (precision + recall + tf.keras.backend.epsilon())
```

▪ Step-4

```
# Return metrics as a dictionary
metrics = {
    'Precision': precision.numpy(),
    'Recall': recall.numpy(),
    'F1-Score': f1_score.numpy()
}

return metrics
```

▪ Step-5

```
# Example usage:
y_true = tf.constant([0, 1, 2, 0, 1, 2]) # Example true labels
y_pred = tf.constant([0, 2, 1, 0, 0, 1]) # Example predicted labels
num_classes = 3 # Number of classes in the classification problem

metrics = simple_multi_class_metrics(y_true, y_pred, num_classes)
print("Precision: ", metrics['Precision'])
print("Recall: ", metrics['Recall'])
print("F1-Score: ", metrics['F1-Score'])
```

OUTPUT:

```
Precision: [0.6666667 0.          0.          ]
Recall:    [1.  0.  0.]
F1-Score:  [0.79999995 0.          0.          ]
```

RESULT:

Program executed successfully and displayed the output.

