



28-06-2024

TENSORFLOW PRACTICALS

AKSHAY SURENDRAN



1. create a 3*3 matrix of ones and a 3*3 matrix of.add them together using tensorflow.

Steps:

Creating Matrices and Performing Addition

This code snippet demonstrates how to create matrices of ones and zeros using TensorFlow and perform element-wise addition.

Step-by-step Explanation:

1. Import TensorFlow:

```
```python
import tensorflow as tf
```
```

Imports the TensorFlow library.

2. Create Matrices:

```
```python
Create a 3x3 matrix of ones
matrix1 = tf.ones(shape=(3, 3), dtype=tf.float32)

Create a 3x3 matrix of zeros
matrix2 = tf.zeros(shape=(3, 3), dtype=tf.float32)
```
```

- `matrix1`: Creates a 3x3 matrix initialized with ones.

- `matrix2`: Creates a 3x3 matrix initialized with zeros.

2. Perform Addition:


```
```python
Add the matrices together
result = tf.add(matrix1, matrix2)
```
```

Computes element-wise addition of `matrix1` and `matrix2`, storing the result in `result`.

4. Print Matrices and Result:

```
```python
Print the matrices and their sum
print("Matrix of Ones:")
print(matrix1.numpy())
print("\nMatrix of Zeros:")
print(matrix2.numpy())
print("\nSum of matrices:")
print(result.numpy())
```
```

- `matrix1.numpy()`: Prints the values of `matrix1` as a NumPy array.
- `matrix2.numpy()`: Prints the values of `matrix2` as a NumPy array.
- `result.numpy()`: Prints the values of the sum `result` as a NumPy array.

```
 import tensorflow as tf

# Create a 3x3 matrix of ones
matrix1 = tf.ones(shape=(3, 3), dtype=tf.float32)

# Create a 3x3 matrix of zeros
matrix2 = tf.zeros(shape=(3, 3), dtype=tf.float32)

# Add the matrices together
result = tf.add(matrix1, matrix2)

# Print the matrices and their sum
print("Matrix of Ones:")
print(matrix1.numpy())
print("\nMatrix of Zeros:")
print(matrix2.numpy())
print("\nSum of matrices:")
print(result.numpy())
```

- Output:

Matrix of Ones:

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Matrix of Zeros:

```
[[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]
```

Sum of matrices:

```
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]  
...
```

Visualization of output:



```
Matrix of Ones:
```

```
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]
```

```
Matrix of Zeros:
```

```
[[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]
```

```
Sum of matrices:
```

```
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]
```

Explanation

Purpose: This code snippet demonstrates basic matrix creation and addition using TensorFlow operations.

Usage: Useful for initializing matrices with specific values (ones or zeros) and performing basic element-wise operations like addition.

-TensorFlow Functions Used:

- ``tf.ones()``: Creates a tensor filled with ones.

- `tf.zeros()`: Creates a tensor filled with zeros.
- `tf.add()`: Performs element-wise addition of tensors.

This example is straightforward and can be expanded upon for more complex matrix operations or integrated into larger TensorFlow workflows for machine learning and data processing tasks.

2. Implement a custom layer in tensor flow that performs a specific operation (for example a custom activation function) use this layer in a simple mode.

Steps:

1. Custom Activation Layer:

- A new layer class `CustomActivation` is created, inheriting from `tf.keras.layers.Layer`.
- The `call` method defines the layer's forward pass, applying the sigmoid activation function.

2. Model Definition:

- A sequential model is constructed with a dense layer, the custom activation layer, and another dense layer.

3. Compilation:

- The model is compiled with the Adam optimizer and mean squared error loss.

4. Training:

- Random dummy data is generated and used to fit the model for 10 epochs.

This demonstrates how to create and incorporate custom layers within TensorFlow models.

```

import tensorflow as tf

class CustomActivation(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(CustomActivation, self).__init__(**kwargs)

    def build(self, input_shape):
        # No trainable weights to be defined here
        super(CustomActivation, self).build(input_shape)

    def call(self, inputs):
        return tf.math.sigmoid(inputs) # Replace with your custom activation logic

# Simple model to demonstrate the custom layer
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, input_shape=(5,)),
    CustomActivation(),
    tf.keras.layers.Dense(1)
])

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

# Example usage with dummy data
import numpy as np
X = np.random.randn(100, 5)
y = np.random.randn(100, 1)

# Fit the model
model.fit(X, y, epochs=10, batch_size=32)

```

Visualization of Output:

```

Epoch 1/10
4/4 [=====] - 1s 8ms/step - loss: 0.9254
Epoch 2/10
4/4 [=====] - 0s 6ms/step - loss: 0.9103
Epoch 3/10
4/4 [=====] - 0s 5ms/step - loss: 0.8994
Epoch 4/10
4/4 [=====] - 0s 6ms/step - loss: 0.8871
Epoch 5/10
4/4 [=====] - 0s 5ms/step - loss: 0.8742
Epoch 6/10
4/4 [=====] - 0s 6ms/step - loss: 0.8603
Epoch 7/10
4/4 [=====] - 0s 7ms/step - loss: 0.8501
Epoch 8/10
4/4 [=====] - 0s 6ms/step - loss: 0.8401
Epoch 9/10
4/4 [=====] - 0s 6ms/step - loss: 0.8333
Epoch 10/10
4/4 [=====] - 0s 7ms/step - loss: 0.8262
<keras.src.callbacks.History at 0x7e60e465ea0>

```

3. implement a simple example of distributed training `tf.distribute.strategy` to train a model on multiple GPU's

1. **Check for GPUs:** It first checks for available GPUs using `tf.config.list_physical_devices('GPU')`.
2. **Model Definition:** A simple sequential model is defined with a dense layer (256 neurons, ReLU activation), dropout for regularization, and a final dense layer (10 neurons, softmax activation for classification).
3. **Distribution Strategy:** `tf.distribute.MirroredStrategy` is employed for potential distribution of training across available GPUs.
4. **Model Compilation:** Within the distribution strategy's scope, the model is compiled with Adam optimizer, sparse categorical crossentropy loss (suited for integer-encoded labels), and accuracy metric.
5. **Dummy Data:** Random data is generated for training (features `x_train` and labels `y_train`).
6. **Dataset Creation:** A TensorFlow Dataset is created from the dummy data and batched for efficient training.
7. **Model Training:** The model is trained on the dataset for 10 epochs using `model.fit`.


```

import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# Check available GPUs
gpus = tf.config.list_physical_devices('GPU')
print("Available GPUs:", gpus)

# Define a simple model
def create_model():
    model = models.Sequential([
        layers.Dense(256, activation='relu', input_shape=(784,)),
        layers.Dropout(0.2),
        layers.Dense(10, activation='softmax')
    ])
    return model

# Define distribution strategy
strategy = tf.distribute.MirroredStrategy()

# Create and compile the model under the strategy scope
with strategy.scope():
    model = create_model()
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

# Dummy data generation
X_train = np.random.rand(1000, 784)
y_train = np.random.randint(0, 10, size=(1000,))

# Create datasets
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(32)

# Train the model
model.fit(train_dataset, epochs=10)

```

Visualization of output:

```

Available GPUs: []
Epoch 1/10
32/32 [=====] - 2s 11ms/step - loss: 2.4870 - accuracy: 0.1070
Epoch 2/10
32/32 [=====] - 0s 11ms/step - loss: 2.2839 - accuracy: 0.1460
Epoch 3/10
32/32 [=====] - 0s 10ms/step - loss: 2.2558 - accuracy: 0.1680
Epoch 4/10
32/32 [=====] - 0s 9ms/step - loss: 2.2083 - accuracy: 0.2010
Epoch 5/10
32/32 [=====] - 0s 9ms/step - loss: 2.1595 - accuracy: 0.2350
Epoch 6/10
32/32 [=====] - 0s 10ms/step - loss: 2.0987 - accuracy: 0.2620
Epoch 7/10
32/32 [=====] - 0s 9ms/step - loss: 2.0200 - accuracy: 0.3090
Epoch 8/10
32/32 [=====] - 0s 9ms/step - loss: 1.9590 - accuracy: 0.3570
Epoch 9/10
32/32 [=====] - 0s 9ms/step - loss: 1.8670 - accuracy: 0.4190
Epoch 10/10
32/32 [=====] - 0s 11ms/step - loss: 1.7785 - accuracy: 0.4490
<keras.src.callbacks.History at 0x7b05685c9bd0>

```

4. write a tensorflow function to calculate precision,recall,and F1-score for a multiclass classification problem

1. **Precision:** Measures the proportion of correctly predicted positive instances out of all instances predicted as positive.
 - It compares the `argmax` (index of the highest value) of `y_true` and `y_pred` to determine true positives.
 - It divides the number of true positives by the total number of predicted positives.
2. **Recall:** Measures the proportion of correctly predicted positive instances out of all actual positive instances.
 - Similar to precision, it calculates true positives.
 - It divides the number of true positives by the total number of actual positives.
3. **F1-score:** Provides a balanced evaluation by combining precision and recall.
 - It calculates the harmonic mean of precision and recall.

Usage:

- The code demonstrates how to use these metrics with sample tensors `y_true` and `y_pred`.
- It calculates and prints the precision, recall, and F1-score values.

These metrics are crucial for assessing the performance of a classification model, especially when dealing with imbalanced datasets or when different types of errors have varying costs.

```

import tensorflow as tf

def precision(y_true, y_pred):
    """Function to calculate precision for a multiclass classification problem."""
    # Cast y_true to int64 to match the type of argmax output
    y_true = tf.cast(y_true, dtype=tf.int64) # Change to int64
    true_positives = tf.reduce_sum(tf.cast(tf.math.logical_and(tf.equal(y_true, tf.argmax(y_pred, axis=1)),
                                                                tf.equal(tf.argmax(y_pred, axis=1), tf.argmax(y_true, axis=1))), dtype=tf.float
    predicted_positives = tf.reduce_sum(tf.cast(tf.equal(tf.argmax(y_pred, axis=1), tf.argmax(y_pred, axis=1)), dtype=tf.float32))
    return true_positives / (predicted_positives + tf.keras.backend.epsilon())

def recall(y_true, y_pred):
    """Function to calculate recall for a multiclass classification problem."""
    # Cast y_true to int64 to match the type of argmax output
    y_true = tf.cast(y_true, dtype=tf.int64) # Change to int64
    true_positives = tf.reduce_sum(tf.cast(tf.math.logical_and(tf.equal(y_true, tf.argmax(y_pred, axis=1)),
                                                                tf.equal(tf.argmax(y_pred, axis=1), tf.argmax(y_true, axis=1))), dtype=tf.float
    actual_positives = tf.reduce_sum(tf.cast(tf.equal(y_true, tf.argmax(y_true, axis=1)), dtype=tf.float32))
    return true_positives / (actual_positives + tf.keras.backend.epsilon())

def f1_score(y_true, y_pred):
    """Function to calculate F1-score for a multiclass classification problem."""
    prec = precision(y_true, y_pred)
    rec = recall(y_true, y_pred)
    return 2 * ((prec * rec) / (prec + rec + tf.keras.backend.epsilon()))

# Example usage:
# Assuming y_true and y_pred are TensorFlow tensors or arrays
y_true = tf.constant([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
y_pred = tf.constant([[0.9, 0.1, 0.0], [0.0, 0.8, 0.2], [0.1, 0.2, 0.7]])

precision_val = precision(y_true, y_pred).numpy()
recall_val = recall(y_true, y_pred).numpy()
f1_score_val = f1_score(y_true, y_pred).numpy()

print("Precision:", precision_val)
print("Recall:", recall_val)
print("F1-score:", f1_score_val)

```

Visualization of Output:

Precision: 1.0

Recall: 1.0

F1-score: 1.0