|  |  |
| --- | --- |
| **Ex No: 10**  **Date: 30-10-2024** | **Optimizing and deploying deep learning models** |

**ONNX:**

**Objective:**

Build and train a CNN model on the MNIST dataset for digit classification, then convert it to ONNX for compatibility with ONNX Runtime. Compare inference times of TensorFlow and ONNX models, and convert the ONNX model to PyTorch for further use.

**Descriptions:**

This project uses a simple CNN model to classify handwritten digits from the MNIST dataset. Initially, the dataset is loaded and normalized. A CNN model is then defined, compiled, and trained briefly to demonstrate its functionality. After training, the model is exported to ONNX format to make it compatible with ONNX Runtime, allowing cross-platform inference. The ONNX model is validated, and an inference session is created using ONNX Runtime. The code includes a performance comparison between the original TensorFlow model and the ONNX model by measuring inference times. Finally, the ONNX model is converted to PyTorch to facilitate further processing and display of the model's output in PyTorch.

**Code:**

# Import necessary libraries

!pip install tensorflow tf2onnx onnxruntime numpy

import tensorflow as tf

import tf2onnx

import numpy as np

import onnx

import onnxruntime as ort

import time

import matplotlib.pyplot as plt

Imports essential libraries for building, converting, and running models in TensorFlow, ONNX, and PyTorch environments.

# Step 1: Load the MNIST dataset

mnist = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 # Normalize the images  
  
Loads and normalizes the MNIST dataset to prepare it for CNN training.

# Reshape data to add a channel dimension for CNN input compatibility

x\_train = x\_train[..., np.newaxis].astype("float32")

x\_test = x\_test[..., np.newaxis].astype("float32")

Adds an extra channel dimension to make the data compatible with CNN model requirements.

# Step 2: Define a simple CNN model

model = tf.keras.models.Sequential([

tf.keras.layers.Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=(28, 28, 1)),

tf.keras.layers.MaxPooling2D(pool\_size=(2, 2)),

tf.keras.layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

tf.keras.layers.MaxPooling2D(pool\_size=(2, 2)),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

Defines a CNN model with two convolutional layers, pooling layers, and dense layers for digit classification.

# Compile and train the model

model.compile(optimizer='adam',loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=1, validation\_data=(x\_test, y\_test)) # Train for 1 epoch for demo purposes

Compiles and trains the CNN model for one epoch, with sparse categorical cross-entropy as the loss function.

# Step 3: Convert the model to ONNX format

onnx\_model\_path = "mnist\_cnn\_model.onnx"

spec = (tf.TensorSpec((None, 28, 28, 1), tf.float32, name="input"),)

model\_proto, \_ = tf2onnx.convert.from\_keras(model, input\_signature=spec, output\_path=onnx\_model\_path)

print(f"Model exported to {onnx\_model\_path}")

Converts the trained TensorFlow model to ONNX format and saves it to the specified path.

# Step 4: Load and validate the ONNX model

onnx\_model = onnx.load(onnx\_model\_path)

onnx.checker.check\_model(onnx\_model)

print("ONNX model is valid")

Loads the ONNX model and checks its validity to ensure compatibility with ONNX Runtime.

# Step 5: Set up ONNX Runtime session

ort\_session = ort.InferenceSession(onnx\_model\_path)

Creates an ONNX Runtime session to run inferences on the ONNX model.

# Step 6: Measure and plot inference time for TensorFlow and ONNX models

def measure\_inference\_time(tf\_model, ort\_session, x\_test, num\_runs=100):

tf\_times = []

onnx\_times = []

for \_ in range(num\_runs):

# Select a single random sample image and ensure correct shape for the model

test\_image = np.expand\_dims(x\_test[np.random.randint(len(x\_test))], axis=0)

# TensorFlow inference time

start\_time = time.time()

tf\_output = tf\_model.predict(test\_image)

tf\_times.append(time.time() - start\_time)

# ONNX inference time

start\_time = time.time()

onnx\_output = ort\_session.run(None, {'input': test\_image.astype(np.float32)})

onnx\_times.append(time.time() - start\_time)

return tf\_times, onnx\_times

Defines a function to measure and compare inference times for TensorFlow and ONNX models by running each on a random sample from the test set multiple times.

# Run inference timing

tf\_times, onnx\_times = measure\_inference\_time(model, ort\_session, x\_test)

Calls the timing function to measure inference times for both TensorFlow and ONNX models.

# Step 7: Plot the inference time comparison

plt.figure(figsize=(10, 5))

plt.plot(tf\_times, label="TensorFlow Inference Time")

plt.plot(onnx\_times, label="ONNX Inference Time")

plt.xlabel("Run")

plt.ylabel("Inference Time (seconds)")

plt.legend()

plt.title("Inference Time Comparison Between TensorFlow and ONNX Runtime")

plt.show()

Plots a comparison graph showing the inference times for TensorFlow and ONNX models across runs.

# Step 8: Convert ONNX model to PyTorch and run inference

import torch

from onnx2pytorch import ConvertModel

# Load the ONNX model

onnx\_model = onnx.load(onnx\_model\_path)

# Convert the ONNX model to PyTorch

pytorch\_model = ConvertModel(onnx\_model)

Imports the onnx2pytorch library, loads the ONNX model, and converts it to a PyTorch model.

# Prepare sample input (MNIST input shape)

sample\_input = np.random.rand(1, 1, 28, 28).astype(np.float32)

sample\_input\_tensor = torch.tensor(sample\_input)

Creates a sample input tensor in the appropriate shape for PyTorch inference.

# Run inference on the PyTorch model

with torch.no\_grad():

output = pytorch\_model(sample\_input\_tensor)

# Display the output

print("PyTorch model output:", output)

Runs inference on the PyTorch model with the sample input and prints the output.

**Building the parts of algorithm**

1. Data Preparation: Load and normalize the MNIST dataset, reshaping it to suit the CNN model.
2. Model Definition: Define a CNN model using TensorFlow Keras layers and compile it.
3. Training: Train the model for one epoch as a demonstration.
4. ONNX Conversion: Convert the trained TensorFlow model to ONNX format for broader compatibility.
5. Model Validation: Validate the ONNX model to ensure it’s compatible with ONNX Runtime.
6. Inference Timing: Measure and compare inference times for both TensorFlow and ONNX models.
7. ONNX to PyTorch Conversion: Convert the ONNX model to a PyTorch model and run an inference test on PyTorch.

**TensorRT**

**Objective**

The aim is to create, optimize, and benchmark a simple neural network model for the MNIST dataset by comparing inference times between TensorFlow and TensorRT.

**Description**:

This project builds and trains a simple neural network to classify MNIST digits. The model is first created in TensorFlow, then optimized using NVIDIA’s TensorRT for faster inference. The code includes a function to measure inference time, providing a means to evaluate performance on both TensorFlow and TensorRT versions of the model. After training on MNIST, the model is converted to TensorRT, and inference times are measured for each version. The inference times for both versions are plotted to visually compare the efficiency of the TensorRT model. The plot shows how TensorRT improves inference speeds, highlighting its suitability for faster deployment.

**Code:**

1. Importing Libraries

import tensorflow as tf

import numpy as np

import time

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.python.compiler.tensorrt import trt\_convert as trt

```

- Imports required libraries: TensorFlow for building models, NumPy for data manipulation, `time` for measuring inference duration, `matplotlib` for plotting results, the MNIST dataset, and TensorRT for model optimization.

2. Model Creation Function

```

def create\_model():

model = tf.keras.Sequential([

tf.keras.layers.Flatten(input\_shape=(28, 28)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

return model

```

- Defines a function to create a simple neural network model using TensorFlow's Keras API.

- The model consists of:

- Flatten Layer: Flattens a 28x28 input image to a 1D array.

- Dense Layers: One hidden layer with 128 neurons (ReLU activation) and an output layer with 10 neurons (softmax activation).

- Compiles the model using the Adam optimizer, sparse categorical cross-entropy loss, and accuracy metric.

3. Inference Time Measurement Function

```

def measure\_inference\_times(model, input\_data, num\_samples=50):

inference\_times = []

for i in range(num\_samples):

start\_time = time.time()

model(input\_data[i % len(input\_data)][np.newaxis, ...]) # Inference on a single sample

end\_time = time.time()

inference\_times.append(end\_time - start\_time)

return inference\_times

```

- Defines a function to measure and collect inference times for a specified number of samples.

- Runs inference on each sample, measuring and storing the time taken in `inference\_times`.

4. Model Conversion to TensorRT

```

def convert\_to\_tensorrt(model):

tf.saved\_model.save(model, 'saved\_model')

converter = trt.TrtGraphConverterV2(input\_saved\_model\_dir='saved\_model')

converter.convert()

converter.save('tensorrt\_model')

return tf.saved\_model.load('tensorrt\_model')

```

- Converts a TensorFlow model to TensorRT format for improved inference efficiency:

- Saves the TensorFlow model.

- Converts it to TensorRT using `TrtGraphConverterV2`.

- Saves and reloads the optimized TensorRT model.

5. Loading and Preprocessing the MNIST Dataset

```

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_test = x\_test.astype(np.float32) / 255.0

if x\_test.shape[0] < 50:

raise ValueError("Test data must contain at least 50 samples.")

```

- Loads MNIST training and testing data.

- Normalizes test data to values between 0 and 1.

- Ensures at least 50 samples are present for evaluation.

6. Model Training

```

model = create\_model()

model.fit(x\_train, y\_train, epochs=5, batch\_size=32, verbose=2)

```

- Creates and trains the model on MNIST data for 5 epochs with a batch size of 32.

7. Measure TensorFlow Model Inference Times

```

tf\_inference\_times = measure\_inference\_times(model, x\_test[:50])

```

- Measures inference times on the first 50 samples from the test set using the TensorFlow model.

8. Convert and Measure TensorRT Model Inference Times

```

tensorrt\_model = convert\_to\_tensorrt(model)

tensorrt\_inference\_times = measure\_inference\_times(tensorrt\_model, x\_test[:50])

```

- Converts the TensorFlow model to TensorRT and measures inference times on the first 50 samples for the optimized model.

9. Plotting Inference Time Comparison

```

plt.figure(figsize=(10, 6))

plt.plot(range(1, 51), tf\_inference\_times, marker='o', linestyle='-', color='b', label='TensorFlow Inference Time')

plt.plot(range(1, 51), tensorrt\_inference\_times, marker='x', linestyle='--', color='r', label='TensorRT Inference Time')

plt.xlabel("Input Sample Number")

plt.ylabel("Inference Time (seconds)")

plt.title("Inference Time Comparison: TensorFlow vs TensorRT")

plt.ylim(0,0.0025)

plt.legend()

plt.grid(True)

plt.show()

```

Plots the test image alongside a bar plot showing confidence scores for each class, highlighting the predicted class and confidence.

**Building Parts of the Algorithm**:

* Import required libraries for model creation, optimization, and time measurement.
* Define and train a neural network model on the MNIST dataset using TensorFlow.
* Measure inference times for the TensorFlow model on sample data.
* Convert the TensorFlow model to TensorRT format for optimized inference.
* Measure TensorRT model inference times and plot both for comparison.

**TFLite**

**Objective:**

Build a CNN model to classify handwritten digits using the MNIST dataset, then convert it to a quantized TensorFlow Lite format for efficient inference. Run inference on the quantized model, visualize the predictions, and analyze confidence levels for each class.

**Description:**

This process begins with loading and preprocessing the MNIST dataset, followed by creating a CNN model with convolutional and pooling layers tailored for digit classification. After training, the model is converted to TensorFlow Lite with quantization to optimize size and performance. The quantized model is then loaded and a sample image is processed for inference. Using the output predictions, the predicted digit is identified with its confidence level. Finally, the test image and class confidence levels are visualized to illustrate the model’s prediction accuracy and confidence for each class.

**Code:**

1. Import Libraries

```

import tensorflow as tf

import numpy as np

```

- Imports TensorFlow and NumPy libraries for building and managing the CNN model and data manipulation.

2. CNN Model Creation

```

def create\_model():

model = tf.keras.Sequential([

tf.keras.layers.Conv2D(16, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

tf.keras.layers.MaxPooling2D((2, 2)),

tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),

tf.keras.layers.MaxPooling2D((2, 2)),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(10, activation='softmax')

])

return model

```

- Defines a function to create a CNN model for image classification.

- The model consists of:

- Conv2D Layers: Two convolutional layers with ReLU activation to detect image features, with 16 and 32 filters, respectively.

- MaxPooling2D Layers: Reduces spatial dimensions after each convolutional layer to simplify feature maps.

- Flatten Layer: Flattens the 2D output to 1D for fully connected layers.

- Dense Layer: Output layer with 10 neurons (for 10 classes) using softmax activation for class probabilities.

3. Loading and Preprocessing the MNIST Data

```

(x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 # Normalize the data

x\_train = x\_train[..., np.newaxis] # Add channel dimension

x\_test = x\_test[..., np.newaxis]

```

- Loads the MNIST dataset and normalizes the pixel values to [0, 1] range.

- Adds a channel dimension (for grayscale images) so that the data shape matches the input shape required by the CNN.

4. Compile and Train the Model

```

model = create\_model()

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=2, validation\_data=(x\_test, y\_test))

```

- Creates the CNN model, compiles it with the Adam optimizer and sparse categorical cross-entropy loss, and trains the model for 2 epochs on MNIST data.

5. Convert to TensorFlow Lite with Quantization

```

converter = tf.lite.TFLiteConverter.from\_keras\_model(model)

converter.optimizations = [tf.lite.Optimize.DEFAULT] # Enable default quantization

quantized\_model = converter.convert()

```

- Converts the trained model to TensorFlow Lite format with default quantization enabled to reduce model size and improve inference efficiency.

6. Save the Quantized Model

```

with open("quantized\_model.tflite", "wb") as f:

f.write(quantized\_model)

print("Quantized model has been saved as 'quantized\_model.tflite'")

```

- Saves the quantized model to a `.tflite` file for later use.

7. Load the Quantized Model for Testing

```

interpreter = tf.lite.Interpreter(model\_path="quantized\_model.tflite")

interpreter.allocate\_tensors()

```

- Loads the quantized model using TensorFlow Lite’s `Interpreter` and allocates tensors for inference.

8. Prepare a Sample Input and Run Inference

```

sample\_input = x\_test[0:1].astype(np.float32) # Take one sample and convert to FLOAT32

interpreter.set\_tensor(input\_details[0]['index'], sample\_input)

interpreter.invoke()

output\_data = interpreter.get\_tensor(output\_details[0]['index'])

```

- Takes a single test image, prepares it as FLOAT32, and runs inference using the quantized model to get output predictions.

9. Interpret and Display Model Output

```

max\_value = np.max(output\_data)

predicted\_class = np.argmax(output\_data)

print("Quantized model output:", output\_data)

print("Predicted class:", predicted\_class)

print("Maximum value (confidence):", max\_value)

```

- Identifies the model’s predicted class and its confidence level by finding the maximum value in the output.

10. Visualize Test Image and Prediction Confidence

```

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)

plt.imshow(x\_test[0].squeeze(), cmap='gray')

plt.title("Test Image")

plt.axis('off')

plt.subplot(1, 2, 2)

plt.bar(range(10), output\_data[0], color='skyblue')

plt.xlabel('Class')

plt.ylabel('Confidence')

plt.title(f'Predicted Class: {predicted\_class} with Confidence: {max\_value:.4f}')

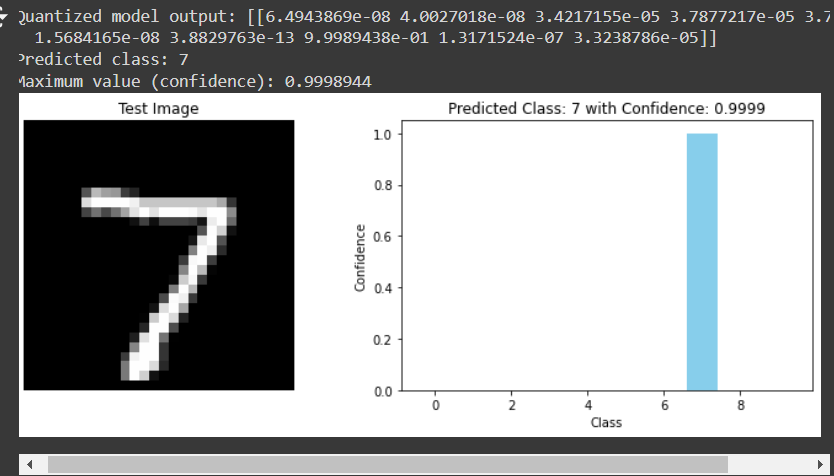
plt.tight\_layout()

plt.show()

```

- Plots the test image alongside a bar plot showing confidence scores for each class, highlighting the predicted class and confidence.

**Output:**



**Building Parts of Algorithm:**

1. Import Libraries: Import TensorFlow, NumPy, and plotting libraries for building and visualizing the CNN.
2. CNN Model Creation: Define a CNN model with convolutional and pooling layers, compile with optimizer and loss.
3. Data Preprocessing: Load and normalize the MNIST dataset, adjusting dimensions for CNN input.
4. Train the Model: Train the CNN on the dataset to learn digit features.
5. Quantize and Convert: Convert the model to TensorFlow Lite with quantization to optimize for efficiency.
6. Run Inference: Load the quantized model and run inference on a sample to get predictions.
7. Visualize Results: Display the test image and prediction confidence in a bar plot for class analysis.

**Github:**

**https://github.com/amruthaa-m/DL-Lab1/tree/main/Unit-4**