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

**Objective:**

The objective of this lab is to explore methods for optimizing and deploying deep learning models for efficient inference on different hardware platforms. This involves building a Convolutional Neural Network (CNN) using the MNIST dataset and converting it into various optimized formats—ONNX, TensorRT, and TensorFlow Lite—to enhance the model's inference speed and reduce computational overhead. The lab also aims to compare the performance of these optimized models by measuring and plotting inference times. Through this exercise, we aim to understand how each optimization technique can be applied to accelerate model inference, thereby enabling deep learning models to be deployed effectively on resource-constrained environments.

**Description:**

This lab focuses on the process of optimizing and deploying deep learning models for efficient inference across various platforms. Using the MNIST dataset, we will build and train a Convolutional Neural Network (CNN) in TensorFlow and then explore multiple optimization techniques, including conversion to ONNX, TensorRT, and TensorFlow Lite formats. Each format provides unique advantages tailored to different deployment environments, such as cloud, edge devices, or mobile platforms.

The lab involves evaluating and comparing the inference performance of each model format. The ONNX model is optimized for cross-platform compatibility, while TensorRT leverages NVIDIA’s hardware accelerations, and TensorFlow Lite enables deployment on mobile and IoT devices with quantization. By measuring and plotting inference times for these models, we will gain insight into how optimizations impact model efficiency and performance in real-world applications.

This lab is designed to provide hands-on experience with model conversion, deployment, and performance benchmarking, helping students understand the practical aspects of deploying optimized deep learning models in resource-constrained environments.

This lab involves:

1. Training a CNN on the MNIST dataset using TensorFlow.
2. Converting the trained model to ONNX format and running it with ONNX Runtime for inference.
3. Optimizing the model for NVIDIA hardware using TensorRT.
4. Converting the model to TensorFlow Lite format with quantization for deployment on mobile or embedded devices.
5. Measuring and visualizing inference time comparisons across TensorFlow, ONNX, TensorRT, and TensorFlow Lite models.

**ONNX: Converting and Optimizing with ONNX Runtime**

import tensorflow as tf

import tf2onnx

import numpy as np

import onnx

import onnxruntime as ort

import time

import matplotlib.pyplot as plt

# 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

# Reshape data to add a channel dimension

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

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

# 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')

])

# 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

# 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}")

# 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")

# Step 5: Set up ONNX Runtime session

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

tf.config.run\_functions\_eagerly(True)

 **Import Libraries**: Imports tensorflow for model creation, tf2onnx for model conversion, onnx and onnxruntime for ONNX operations, and numpy for data handling.

 **Load and Preprocess MNIST Data**: Loads the MNIST dataset, normalizes pixel values to [0, 1], and reshapes the data to (28, 28, 1) for CNN compatibility.

 **Define and Compile CNN Model**:

* Builds a CNN with:
  + Two Conv2D layers (32 and 64 filters) with ReLU activation.
  + MaxPooling2D layers for spatial downsampling.
  + A Flatten layer to convert 2D features to 1D.
  + Dense layers (128 neurons with ReLU, and 10 neurons with softmax for output).
* Compiles with the Adam optimizer and cross-entropy loss.

 **Train Model**: Trains the model for 1 epoch on the training set.

 **Convert to ONNX**: Converts the trained model to ONNX format using tf2onnx and saves it as mnist\_cnn\_model.onnx.

 **Validate ONNX Model**: Loads and validates the ONNX model with onnx.checker.

 **ONNX Runtime Inference Setup**: Initializes an InferenceSession in onnxruntime to run predictions on the ONNX model.

# Step 8: 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

# Run inference timing

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

# Step 9: 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()

 **Measure Inference Time for TensorFlow and ONNX**:

* measure\_inference\_time function:
  + Runs inference for num\_runs (default 100) on randomly chosen test images.
  + Measures time taken for each inference using TensorFlow (tf\_model.predict) and ONNX Runtime (ort\_session.run).
  + Collects each runtime into tf\_times and onnx\_times lists for comparison.

 **Run Inference Timing**:

* Calls measure\_inference\_time to get timing results for TensorFlow and ONNX models on the test dataset.

 **Plot Inference Time Comparison**:

* Plots tf\_times and onnx\_times using matplotlib for a side-by-side comparison of TensorFlow vs. ONNX inference times across multiple runs.
* Labels axes and adds a title for clarity.

# Import necessary libraries

import onnx

import torch

import numpy as np

from onnx2pytorch import ConvertModel

# Step 1: Load the ONNX model

onnx\_model\_path = "mnist\_cnn\_model.onnx" # Update this path if your model has a different name

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

# Step 2: Convert the ONNX model to a PyTorch model

pytorch\_model = ConvertModel(onnx\_model)

# Step 3: Prepare sample input (similar to the original input shape)

# For the MNIST dataset, the input should be in the shape (batch\_size, 1, 28, 28)

sample\_input = np.random.rand(1, 1, 28, 28).astype(np.float32) # Replace with actual test data as needed

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

# Step 4: Run inference on the PyTorch model

with torch.no\_grad(): # No need to calculate gradients during inference

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

# Step 5: Process and display the output

print("PyTorch model output:", output)

**TensorRT: Converting to TensorRT for Optimized GPU Inference**

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

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

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

inference\_times = []

# Measure inference time for each sample

for i in range(num\_samples):

start\_time = time.time()

# Use the model directly for inference

model(input\_data[i % len(input\_data)][np.newaxis, ...]) # Call the model directly

end\_time = time.time()

# Calculate inference time for the current sample

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

return inference\_times

def convert\_to\_tensorrt(model):

# Save the TensorFlow model

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

# Use TensorRT to optimize the model

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

converter.convert()

converter.save('tensorrt\_model')

# Load the TensorRT model

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

# Load the MNIST dataset

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

# Normalize the data to [0, 1]

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

# Ensure test\_data has at least 50 samples

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

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

# Create and train the model

model = create\_model()

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

# Measure inference times for the first 50 inputs using TensorFlow

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

# Convert the model to TensorRT

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

# Measure inference times for the first 50 inputs using TensorRT

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

# Plot the inference times for 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()

1. **Model Creation (create\_model() function):**
   * Defines a simple neural network with an input layer, one hidden layer with 128 neurons, and an output layer of 10 neurons (for 10 digits).
   * Compiles the model with Adam optimizer and categorical cross-entropy loss.
2. **Inference Time Measurement (measure\_inference\_times() function):**
   * Measures inference time for each sample by calling the model on each input, storing each inference duration in inference\_times.
3. **Convert Model to TensorRT (convert\_to\_tensorrt() function):**
   * Saves the trained TensorFlow model, then optimizes it with TensorRT using TrtGraphConverterV2.
   * Saves the optimized model for faster inference.
4. **Data Loading and Preprocessing:**
   * Loads MNIST, normalizes test data to a [0, 1] range, and checks that it has at least 50 samples.
5. **Model Training:**
   * Trains the model on MNIST for 5 epochs with batch size 32.
6. **Inference Time Comparison:**
   * Measures and stores inference times on the first 50 samples with the original TensorFlow model (tf\_inference\_times) and the TensorRT model (tensorrt\_inference\_times).
7. **Plotting Inference Times:**
   * Compares inference times by plotting them on a graph to visualize the speed difference between TensorFlow and TensorRT models.

This setup demonstrates how TensorRT optimization can reduce inference time, especially for large-scale inference tasks.

**TensorFlow Lite (TFLite): Converting to Lightweight Format for Mobile/Edge Deployment**

import tensorflow as tf

import numpy as np

# Create a simple CNN model

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

# Load and preprocess the 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]

# Compile and train the model briefly

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))

# Convert the trained model to TensorFlow Lite format with quantization

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

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

# Convert the model

quantized\_model = converter.convert()

# 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'")

# Optional: Load and test the quantized model for inference

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

interpreter.allocate\_tensors()

 **Define a Simple CNN Model (create\_model() function)**:

* Creates a CNN model with two convolutional layers (Conv2D) and two max-pooling layers (MaxPooling2D).
* Flattens the feature maps before adding a dense (fully connected) output layer with 10 neurons, each representing a digit (0-9) for classification.

 **Data Loading and Preprocessing**:

* Loads the MNIST dataset and normalizes pixel values to the range [0, 1] for easier training.
* Reshapes training and test data by adding a channel dimension ([..., np.newaxis]), which is necessary for compatibility with the CNN model’s expected input shape.

 **Compile and Train the Model**:

* Compiles the model with the Adam optimizer and sparse categorical cross-entropy loss, then trains it for 2 epochs on the MNIST dataset.

 **Convert the Model to TensorFlow Lite (TFLite) Format**:

* Uses TFLiteConverter to convert the trained Keras model to TFLite format, enabling optimizations (including quantization) to reduce model size and potentially improve inference speed.
* Converts the model, storing it in the variable quantized\_model.

 **Save the Quantized Model**:

* Writes the quantized model to a file named quantized\_model.tflite, which can be used for efficient inference on edge devices.

 **Load and Allocate Tensors for Inference (Optional)**:

* Loads the saved TFLite model into a tf.lite.Interpreter for inference.
* Calls allocate\_tensors() to prepare the model for running predictions, which allocates memory for the model’s input and output tensors.

import numpy as np

import matplotlib.pyplot as plt

# Ensure the input sample is in FLOAT32 format

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'])

# Find the maximum value in the model output and its index (predicted class)

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)

# Plot the test image and the output data

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

# Plot the test image

plt.subplot(1, 2, 1)

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

plt.title("Test Image")

plt.axis('off')

# Plot the output confidence values

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()

 **Prepare the Input for Inference**:

* sample\_input = x\_test[0:1].astype(np.float32): Selects the first image in the x\_test array, reshapes it for model compatibility, and ensures it is in FLOAT32 format.
* interpreter.set\_tensor(input\_details[0]['index'], sample\_input): Loads the sample\_input into the TFLite model by setting it at the model’s input tensor index.

 **Run the Model Inference**:

* interpreter.invoke(): Executes the inference using the loaded TFLite interpreter, producing output data at the model’s output tensor.

 **Extract the Model Output**:

* output\_data = interpreter.get\_tensor(output\_details[0]['index']): Retrieves the prediction output, which contains confidence scores for each class (0–9).

 **Identify the Predicted Class and Confidence**:

* max\_value = np.max(output\_data): Finds the highest confidence score in the output, which indicates how confident the model is in its prediction.
* predicted\_class = np.argmax(output\_data): Determines the class index with the highest confidence, representing the predicted digit.

 **Display the Prediction Results**:

* Prints the output data, predicted class, and confidence score, providing insight into the model’s decision-making for this sample.

 **Visualize the Test Image and Output Confidence**:

* **Test Image**:
  + plt.subplot(1, 2, 1): Sets up the first subplot.
  + plt.imshow(x\_test[0].squeeze(), cmap='gray'): Displays the test image in grayscale.
  + plt.title("Test Image"): Titles the image for context.
* **Output Confidence Values**:
  + plt.subplot(1, 2, 2): Sets up the second subplot for the bar plot of confidence scores.
  + plt.bar(range(10), output\_data[0], color='skyblue'): Plots each class (0–9) on the x-axis with confidence values on the y-axis.
  + plt.title(f'Predicted Class: {predicted\_class} with Confidence: {max\_value:.4f}'): Displays the predicted class and confidence score in the title.

**Conclusion:**

In this lab, we successfully implemented a quantized TensorFlow Lite (TFLite) model for digit classification using a simple Convolutional Neural Network (CNN) on the MNIST dataset. After training and optimizing the model with quantization, we converted it to TFLite format, enabling reduced model size and potentially faster inference on edge devices. We tested the quantized model’s predictions on sample images, examining both the predicted class and confidence levels, and visualized the model's performance. The quantization process demonstrated how model optimization can maintain classification accuracy while significantly reducing computational demands, underscoring the practical value of TFLite for deploying efficient machine learning models on resource-limited devices. This approach is particularly valuable for mobile and IoT applications, where maintaining accuracy with optimized performance is essential.

**Github Link:**

**https://github.com/Bhargava-Srinivasan-26/Deep\_learning\_elective/tree/main/Unit%203/Lab%209**