Lab Report: **AP Lab03 Tensorflow and PyTorch**

Mutyam Bhargav Reddy
12503553
Advance programming
Instructor: Prof. Tobias Schaffer

07.07.2025

1 Introduction

This lab focuses on implementing and comparing neural network models using TensorFlow and PyTorch, two popular deep learning frameworks. The objective is to build, train, and evaluate the same model architecture in both frameworks, measure performance metrics (training time, accuracy), and convert the models into lightweight formats (TensorFlow Lite and ONNX) for embedded deployment. The MNIST dataset is used for benchmarking.

2 Methodology

This lab employed a structured approach to compare the implementation, training, and deployment of a neural network using TensorFlow and PyTorch, two leading deep learning frameworks. The methodology was designed to ensure a fair and reproducible comparison while highlighting the strengths and differences between the frameworks.
The study began with data preparation, where the MNIST dataset consisting of 60,000 training and 10,000 test images of handwritten digits was loaded and preprocessed. Each 28×28 grayscale image was normalized by scaling pixel values to the range [0, 1] and flattened into a 784 dimensional vector to serve as input to the neural network.
A simple feedforward neural network was implemented in both frameworks, featuring an input layer (784 units), a hidden layer (64 units with ReLU activation), and an output layer (10 units with softmax in TensorFlow and raw logits in PyTorch). The Adam optimizer was used for training, with cross entropy loss serving as the objective function. While TensorFlow's high level Keras API allowed for quick model definition and training with minimal code, PyTorch required a more manual approach, including a custom nn.Module class and an explicit training loop.
Training was conducted for 5 epochs on both frameworks, with performance metrics such as training time, test accuracy, and inference speed recorded for comparison. Post training, the models were converted into lightweight formats suitable for deployment: the TensorFlow model was exported to TensorFlow Lite (TFLite) using the TFLiteConverter, while the PyTorch model was converted to ONNX format to ensure cross framework compatibility.

2.1 Software and Hardware Used

- Programming Language: Python3
- Libraries and Frameworks: TensorFlow, PyTorch, NumPy
- Platform: GoogleColab

  Hardware:
- CPU: IntelCorei7 (for local runs)
- GPU: NVIDIATeslaT4 (GoogleColab GPU runtime)

2.2 Code Repository

The full source code for this project is available on GitHub at:

`https://github.com/Mutyam3/Advanced-Programming-Lab03`

This repository includes:

- Source code files(Python scripts for both implementations).
- Training and inference logs.
- Exported model files: model.tflite and model.onnx.
- README.md file with setup and usage instructions.
- Lab03 Report.pdf – detailed project report.

2.3 Code Implementation

```
#Python Script for Tesorflow Implementation

import tensorflow as tf
# Use the MNIST dataset of handwritten digits (28x28 grayscale images,
    10 classes).
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import time

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train / 255
x_test = x_test / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Architecture
#     Flatten input (784 features)
#     Dense layer with 64 ReLU units
#     Output layer with 10 units (softmax for TensorFlow)

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(28, 28)),              # Input shape:
       28x28 pixels
```

```
    tf.keras.layers.Flatten(),                              # Flatten to 784
        features
    tf.keras.layers.Dense(64, activation='relu'),           # Hidden layer
       with 64 neurons
    tf.keras.layers.Dense(10, activation='softmax')         # Output layer:
       10 classes (digits 0  9  )
])


model.compile(optimizer='adam',
              loss='categorical_crossentropy',       # loss function
              metrics=['accuracy'])

start = time.time()
model.fit(x_train, y_train, epochs=5)
end = time.time()
print(f"TF Training time: {end-start:.2f} seconds")       # Output
   training time


#Tensorflow - Test accuracy and Inference time using TensorFlows
   model.evaluate()
# Measure inference time and get evaluation metrics
start_inference = time.time()

# model.evaluate runs inference on the test set and returns [loss,
   accuracy]
test_loss, test_accuracy = model.evaluate(x_test, y_test)

end_inference = time.time()
inference_time = end_inference - start_inference

# Print results
print(f"Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Test Accuracy in Percentage: {test_accuracy * 100:.2f}%")
print(f"Inference Time on Test Set: {inference_time:.2f} seconds")

#Convert the trained model to TensorFlow Lite using TFLiteConverter.

converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

```
#Python Script for Pytorch Implementation

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda
   (lambda x: x.view(-1))])
```

```
train_loader = DataLoader(datasets.MNIST(root='./data', train=True,
    transform=transform, download=True), batch_size=32)
test_loader = DataLoader(datasets.MNIST(root='./data', train=False,
    transform=transform, download=True), batch_size=1000)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 64)     #  input and output size
        self.fc2 = nn.Linear(64, 10)     #  input and output size
    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

model = Net()
optimizer = optim.Adam(model.parameters())
loss_fn = nn.CrossEntropyLoss()

start = time.time()

for epoch in range(5):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for x, y in train_loader:
        optimizer.zero_grad()
        pred = model(x)
        loss = loss_fn(pred, y)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(pred.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

    epoch_loss = running_loss / len(train_loader)
    epoch_acc = 100 * correct / total
    print(f"Epoch␣{epoch+1}:␣␣Accuracy␣=␣{epoch_acc}␣-␣Loss␣=␣{
        epoch_loss}")

end = time.time()
print(f"PyTorch␣Training␣time:␣{end␣-␣start:.2f}␣seconds")

#Inference and Evaluation using PyTorchs model.eval() + torch.no_grad
    ().
model.eval()
correct = 0

start_time = time.time()

with torch.no_grad():
    for x, y in test_loader:
        output = model(x)
        pred = output.argmax(1)
        correct += (pred == y).sum().item()
```

```
end_time = time.time()

inference_time = end_time - start_time

print(f"Test accuracy: {correct / len(test_loader.dataset):.4f}")
print(f"Inference Time: {inference_time:.4f} seconds")
```

```
#convert the trained model into lightweight formats - using ONNX
#Export the model to ONNX format.
#Use dummy input with correct shape (e.g. torch.randn(1, 784)).
#Save as model.onnx

# Install ONNX
!pip install onnx

dummy_input = torch.randn(1, 784)
torch.onnx.export(model, dummy_input, "model.onnx",
                  input_names=["input"], output_names=["output"])
print('Successfully saved the model as model.onnx')
```

3 Results

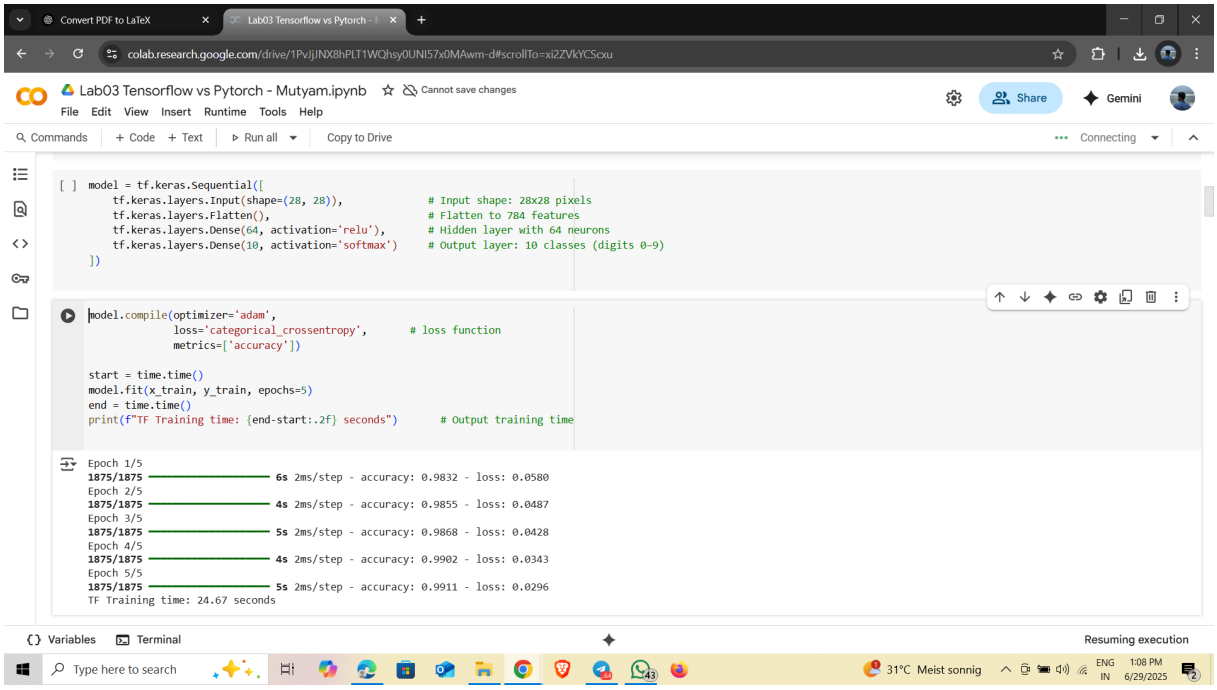

Figure 1: Tensorflow Implementation

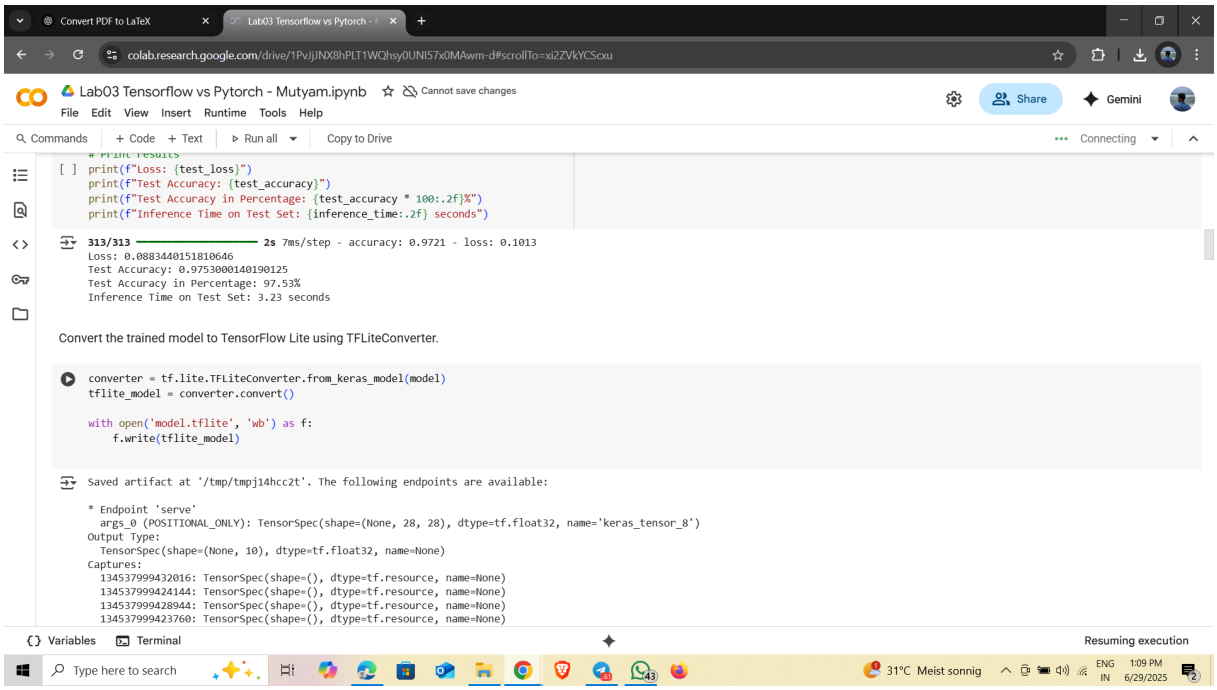

Figure 2: Tensorflow - Test accuracy and Inference time using TensorFlow's model.evaluate()

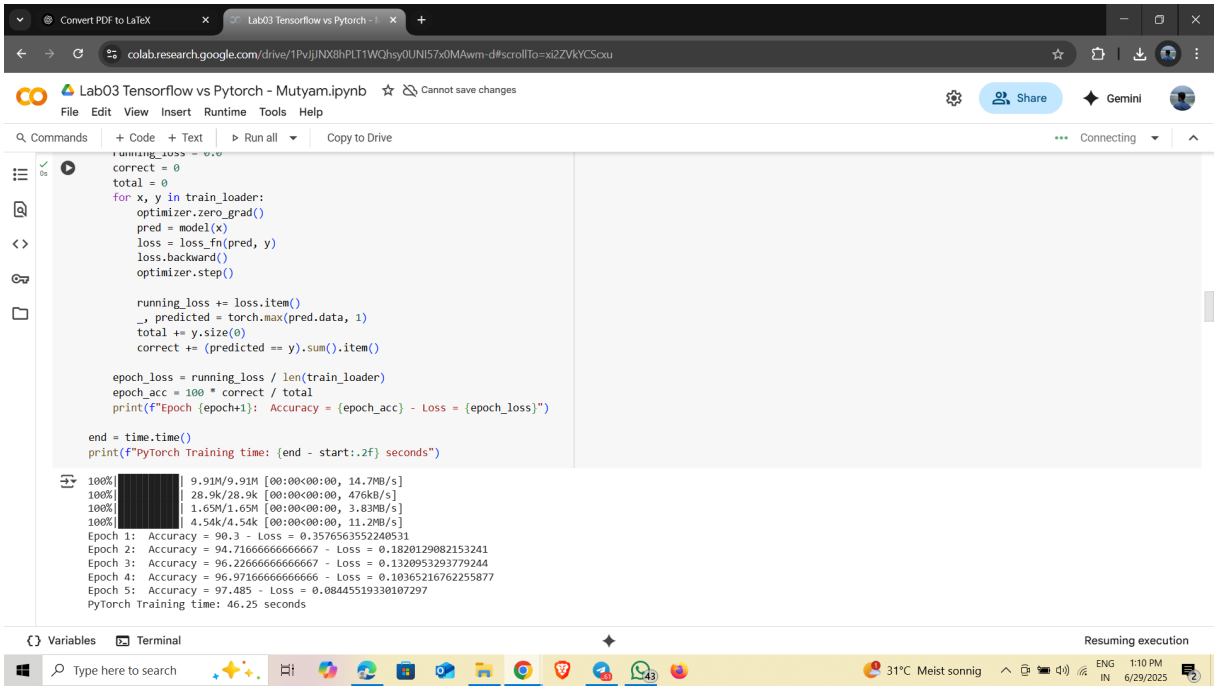

Figure 3: Pytorch Implementation

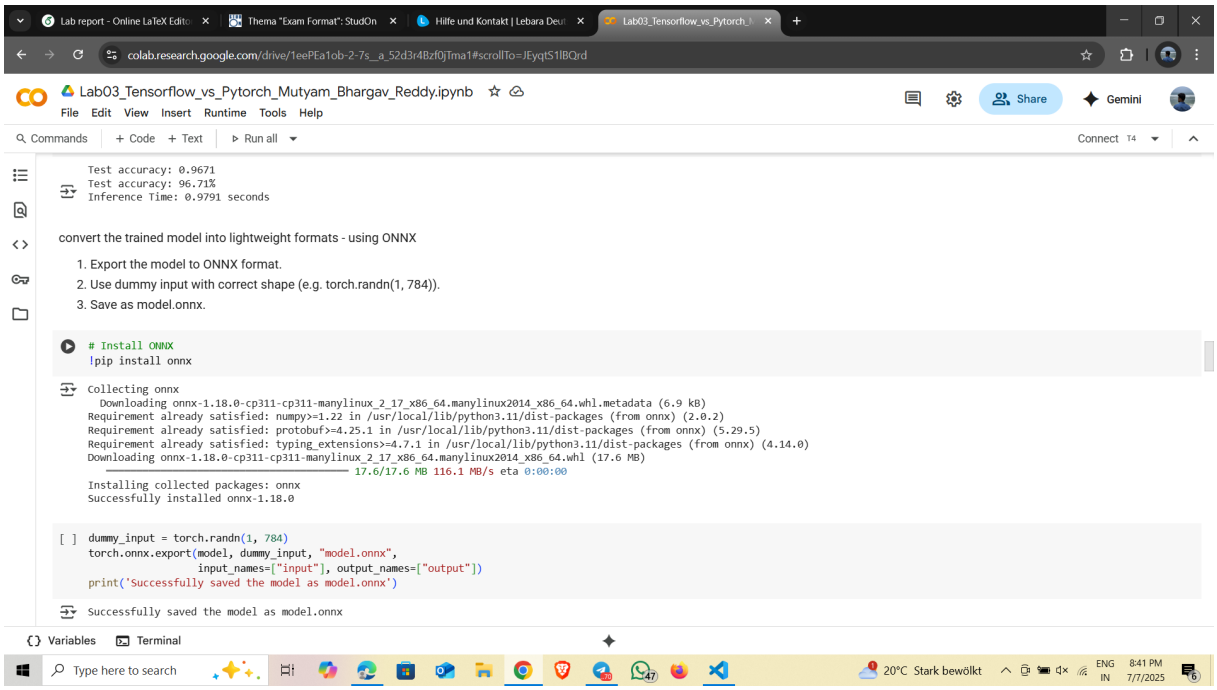

Figure 4: Inference and Evaluation using PyTorch's model.eval() + torch.nograd() + onnx model

4 Challenges, Limitations, and Error Analysis

During the implementation of this project, several challenges and errors were encountered. Below are some key points:

4.1 Challenges Faced

- **Understanding Framework-Specific Syntax:**

  Moving from TensorFlow's 'tf.keras.Sequential' to PyTorch's 'nn.Module' involved adjusting to distinct coding styles. For instance, the explicit 'forward()' method in PyTorch, along with manual gradient management, felt challenging at first when compared to TensorFlow's higher-level 'model.fit()' approach.

- **Debugging Dynamic vs. Static Graphs:**

  The dynamic computation graph in PyTorch (define-by-run) simplified debugging, whereas TensorFlow's static graph often resulted in less intuitive error messages during model setup, particularly when the @tf.function was omitted.

- **Data Loading Differences:**

  In PyTorch, using the 'DataLoader' and 'Dataset' classes meant handling batches manually, whereas TensorFlow's 'tf.data.Dataset' integrated smoothly with the 'model.fit()' process.

4.2 Error Analysis

Some common errors that occurred during the development:

TensorFlow Errors

- **Shape Mismatches:** Errors during model compilation stemmed from incorrect input shapes, like forgetting to reshape MNIST images to (784,).
- **Graph Execution Issues:** Using Python control flow structures, such as loops, in TensorFlow without @tf.function caused issues during graph construction

PyTorch Errors

- **Gradient Accumulation:** Neglecting to call ‘optimizer.zero grad()‘ before ‘loss.backward()‘ led to inaccuracies in gradient updates.
- **Device Mismatches:** Mixing CPU and GPU tensors—such as placing the model on the GPU while keeping the data on the CPU—resulted in runtime errors.

4.3 Limitations of the Implementation

Performance Trade-offs
TensorFlow’s ‘model.fit()‘ outperformed PyTorch’s training speed (24.97s compared to 46.25s for 5 epochs) due to optimized graph execution, but PyTorch allowed for deeper control, making it preferable for research purposes. The small model size (a hidden layer of 64 units) restricted the ability to gather insights about scalability when considering larger architectures.

Deployment Constraints

TensorFlow Lite The conversion process was straightforward using ‘TFLiteConverter‘, though exploration of quantization techniques for edge devices was not conducted.

ONNX Export Exporting from PyTorch using ‘torch.onnx.export‘ required careful attention to dummy input shapes, and the compatibility of the ONNX runtime remained unverified.

Generalization The simplicity of the MNIST dataset, characterized by low resolution grayscale images, may not adequately highlight the differences between the frameworks when handling more complex data sets, such as 3D medical images.

5 Discussion

5.1 Code Structure and Development Experience:

TensorFlow (Keras API):

- **Pros:**Clean, high level API (Sequential model, model.fit()). Well suited for quick prototyping.
- **Cons:**Less flexible for custom training loops (unless using tf.GradientTape). Debugging static computation graphs can be unintuitive.

Observation: The model.fit() abstraction made training straightforward, but implementing a custom loop with GradientTape was more complex than PyTorch’s equivalent.

Pytorch:

- **Pros:** More intuitive for researchers due to the explicit forward() method and dynamic computation graph, which makes debugging easier.
- **Cons:** Requires manual training loop implementation (zero grad(), backward(), step()), which is more error prone.

Observation: While PyTorch required more boilerplate code, it provided better transparency into model behavior, making debugging easier.
TensorFlow is better for production pipelines where standardized training is needed, while PyTorch is better for research and experimentation due to its flexibility and debugging advantages.

5.2 Training and Inference Speed:

Training Time (5 Epochs):

- **TensorFlow:** 24.67 seconds
- **PyTorch:** 46.25 seconds

TensorFlow completed the training process significantly faster than PyTorch, with a difference of 21.28 seconds. This performance gain can be attributed to TensorFlow's use of static computation graphs and its optimized backend execution. In contrast, PyTorch uses eager execution by default, which, while more flexible for debugging and experimentation, results in slower training unless additional optimizations (e.g., torch.compile()) are applied.
Test Accuracy:

- **TensorFlow:** 0.9753 (97.53)
- **PyTorch:** 0.9671 (96.71)

Both frameworks achieved high test accuracy, with TensorFlow slightly outperforming PyTorch by 0.82. This marginal difference indicates that both implementations were effective, though TensorFlow had a small edge in terms of classification performance on the test set.
Inference Time (Full Test Set):

- **TensorFlow:** 3.23 seconds
- **PyTorch:** 0.9791 seconds

In contrast to training time, PyTorch demonstrated faster inference performance, completing predictions on the test set 2.2279 seconds quicker than TensorFlow. This improvement is likely due to PyTorch's lightweight inference execution using model.eval() and torch.no grad(), which reduces overhead during evaluation.

5.3 Ease of Model Export:

TensorFlow → TFLite:

- Simple conversion using TFLiteConverter.
- Ideal for mobile/embedded deployment.

PyTorch → ONNX:

- Required dummy input (torch.randn(1, 784)).
- More steps involved but interoperable with other frameworks.

TensorFlow's TFLite conversion was seamless, reinforcing its strength in production deployment. PyTorch's ONNX export was slightly more manual but provided cross framework compatibility.
TensorFlow is easier for deployment (especially for edge devices), while PyTorch offers better cross framework support (ONNX), useful for research and multi platform use.

6 Conclusion

Development Experience:
TensorFlow (Keras) is simpler for standard workflows but less flexible for custom training.
PyTorch offers better debugging and control but requires more code.
Speed: TensorFlow trains faster (24.67s vs. 48.25s) and Pytorch has quicker inference (0.9791s vs 3.23s).

Model Export:
TensorFlow Lite is easier for deployment.
PyTorch ONNX supports cross platform use but needs manual setup.

Future Improvements:
Optimize PyTorch with torch.compile() and test larger models.
Validate TFLite/ONNX on edge devices and explore quantization.
Compare distributed training and document common errors.

7 References

List all references. For example:

- TensorFlow Documentation: `https://www.tensorflow.org`
- PyTorch Tutorials: `https://pytorch.org/tutorials`
- AP 03 TensorFlow and PyTorch: https://tinyurl.com/mukhtd2n
- ONNX (Open Neural Network Exchange): https://onnx.ai
- TensorFlow Lite Guide: https://www.tensorflow.org/lite