Machine Learning
LABORATORY: LSTM Homework

NAME: 黄仕翔	**STUDENT ID#:** 313513054.

Objectives:

- The goal of this assignment is to deepen your understanding of the Long Short-Term Memory (LSTM) architecture by implementing a manual LSTM cell from scratch.
- You will apply your manual LSTM to the MNIST digit classification task by treating each 28×28 image as a sequence of 28 time steps.
- Through this assignment, you will:
  - o Understand how the LSTM gates (forget, input, and output) interact to update the hidden and cell states.
  - o Implement the LSTM forward pass manually based on the given mathematical formulas.
  - o Train a deep learning model using PyTorch.
  - o Evaluate the model's performance on unseen data.
  - o Tune hyperparameters to improve accuracy.
- This assignment directly connects theoretical concepts (LSTM equations) with practical implementation for real-world applications.

Part 1. Instruction

In this assignment, you will implement a manual Long Short-Term Memory (LSTM) cell for sequence classification using PyTorch, without using any high-level RNN modules (no nn.LSTM, no optim.SGD, etc.).

You will manually implement:

- A step-by-step update of the hidden state and cell state based on the LSTM equations.
- A simple output layer to classify handwritten digits (0-9) from the MNIST dataset.
- Training using manual forward computation for each time step.

The general LSTM computations for each time step are as follows:

$$i_t = \sigma\left(W_i h_{t-1} + U_i x_t + b_i\right)$$

$$f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$$

$$o_t = \sigma\left(W_o h_{t-1} + U_o x_t + b_o\right)$$

$$\tilde{c}_t = tanh\left(W h_{t-1} + U x_t + b\right)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$h_t = o_t \odot tanh\left(c_t\right)$$

After the final time step, you apply an output layer:

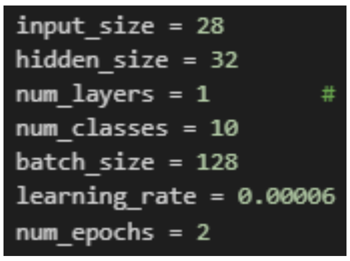
$$logits = W_{out} h_t + b_{out}$$

Lecture: Prof. Hsien-I Lin
TA: Satrio Sanjaya and Muhammad Ahsan

You must implement the full forward computation manually for each time step. In addition to completing the forward pass and classification:

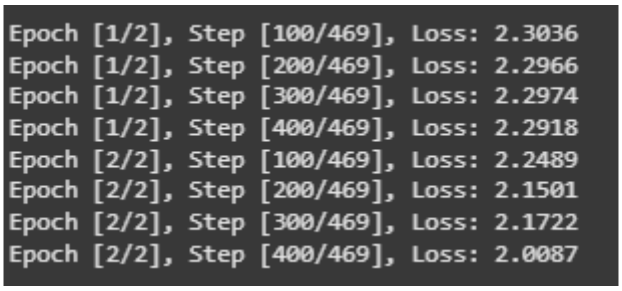
- Hyperparameter Tuning: You are required to adjust the hyperparameters (e.g., learning rate, batch size, number of hidden units, number of epochs, optimizer) to improve the final test accuracy as much as possible.
- Testing Loop: You must fill in the testing loop to calculate and print the overall accuracy on the MNIST test dataset (10,000 images).
- Visualization: You must visualize 10 example images from the test set (ideally showing digits 0–9 if possible).

In your pdf report, you must display:



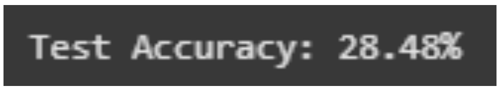
```
input_size = 28
hidden_size = 32
num_layers = 1          #
num_classes = 10
batch_size = 128
learning_rate = 0.00006
num_epochs = 2
```

(a) Hyperparameter



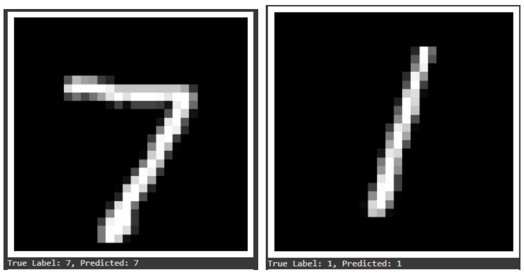
```
Epoch [1/2], Step [100/469], Loss: 2.3036
Epoch [1/2], Step [200/469], Loss: 2.2966
Epoch [1/2], Step [300/469], Loss: 2.2974
Epoch [1/2], Step [400/469], Loss: 2.2918
Epoch [2/2], Step [100/469], Loss: 2.2489
Epoch [2/2], Step [200/469], Loss: 2.1501
Epoch [2/2], Step [300/469], Loss: 2.1722
Epoch [2/2], Step [400/469], Loss: 2.0087
```

(b) Training Loss record



```
Test Accuracy: 28.48%
```

(c) Test Accuracy



(d) Prediction results

Part 2. Code Template

Step	Procedure

1	# ==
	# Assignment: Manual LSTM Cell for MNIST Digit Classification
	# ==
	
	import torch
	import torch.nn as nn
	import torch.optim as optim
	import torchvision
	import torchvision.transforms as transforms
	import matplotlib.pyplot as plt
	import numpy as np
	import os
	
	# ==
	# Hyperparameters - you may change the parameter to get the better accuracy
	# ==
	
	input_size = 28

```
hidden_size = 32
num_layers = 1
num_classes = 10
batch_size = 128
learning_rate = 0.00006
num_epochs = 2
```

2

```

# ============================================================
# Load the MNIST Dataset
# ============================================================

train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    transform=transforms.ToTensor(),
    download=True
)

test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    transform=transforms.ToTensor(),
    download=True
)

train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False
)
```

3

```
# ============================================================
# TODO 1 : Build Manual LSTM Cell
# ============================================================

class ManualLSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(ManualLSTMCell, self).__init__()
        # TODO: Define weight matrices for
        # - Forget gate (Weight f)
        # - Input gate (Weight i)
        # - Output gate (Weight o)
        # - Candidate cell (Weight c)

    def forward(self, x, h_prev, c_prev):
        # TODO:
        # 1. Concatenate input x and previous hidden state h_prev
        # 2. Calculate forget gate f_t
        # 3. Calculate input gate i_t
        # 4. Calculate candidate cell state c_tilde
        # 5. Update cell state c_t
        # 6. Calculate output gate o_t
        # 7. Update hidden state h_t
        # HINT: use torch.sigmoid and torch.tanh

        return h_t, c_t

# Full LSTM network
class ManualLSTMClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(ManualLSTMClassifier, self).__init__()
        # TODO: Create ManualLSTMCell
        # TODO: Create fully connected layer

    def forward(self, x):
        # TODO:
        # 1. Initialize h_t and c_t to zeros
        # 2. Unroll through the sequence (for each time step)
        # 3. Update h_t and c_t at each time step
        # 4. Pass last h_t into fully connected layer

        return out
```

4

```
# ============================================================
# Training and Testing - #You are allowed to change the optimizer
# ============================================================
#Define model, criterion, optimizer

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = ManualLSTMClassifier(input_size, hidden_size, num_classes).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, 28, 28).to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')
```

5

```
# TODO 2: Testing loop to print the accuracy

      print(f'Test Accuracy: {100 * correct / total:.2f}%')

# ============================================================
# TODO 3: Visualization prediction
# Print the accuracy from test_data
# Show 10 example images including true label and prediction
# ==================================================================

# (imshow)
```

Grading Assignment & Submission (70% Max)

Implementation:

1. (30%) Manual LSTM Cell: Correctly build a manual LSTM cell based on the provided LSTM equations.
2. (15%) Training and Hyperparameter Tuning: Successfully train the model and fine-tune hyperparameters to improve the final test accuracy; **the achieved test accuracy will determine the points awarded in this section.**
3. (5%) Testing Loop: Correctly implement the testing loop to calculate and print the test accuracy over the full 10,000 test images.
4. (5%) Visualization: Display 10 example test images, clearly showing both the true labels and the predicted labels.

Question:

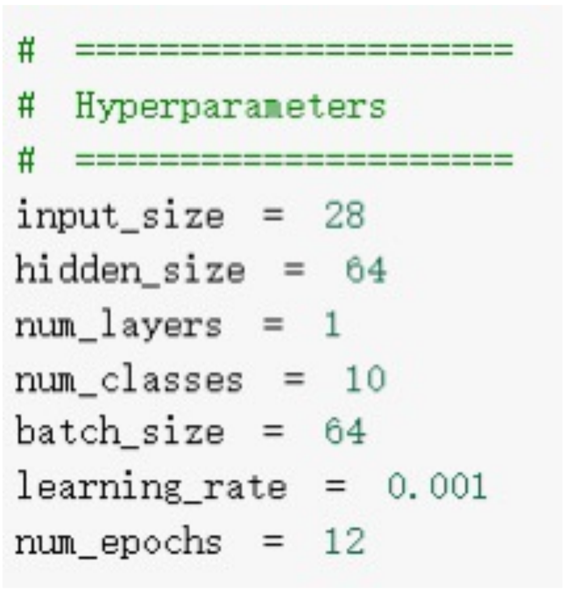
5. (5%) Explain briefly the role of the forget gate, input gate, output gate, and candidate cell in an LSTM.
6. (5%) Describe what hyperparameters you tuned and how they affected your model's final accuracy.
7. (5%) Between a simple RNN and an LSTM, which one is better for sequence learning tasks? Explain your reasoning, and discuss in which situations LSTM is more useful and in which situations a simple RNN might still be sufficient.

Submission :

1. Report: Provide your screenshots of your results in the last pages of this PDF File.
2. Code: Submit your complete Python script in either .py or .ipynb format.
3. Upload both your report and code to the E3 system (**Labs7 Homework**). Name your files correctly:
   a. Report: StudentID_Lab7_Homework.pdf
   b. Code: StudentID_Lab7_Homework.py or StudentID_Lab7_Homework.ipynb
4. Deadline: Sunday 21:00 PM
5. Plagiarism is **strictly prohibited**. Submitting copied work from other students will result in penalties.

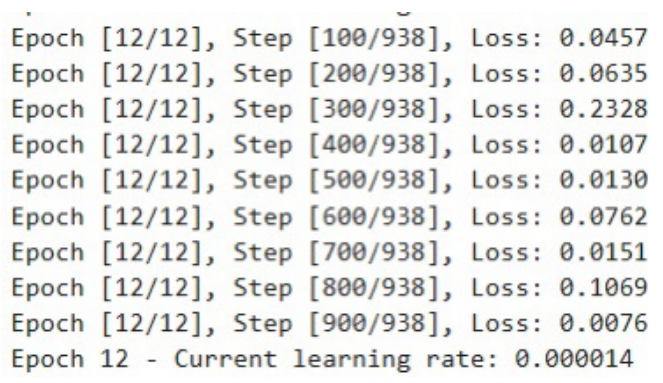
Results and Discussion:

(a) Hyperparameter

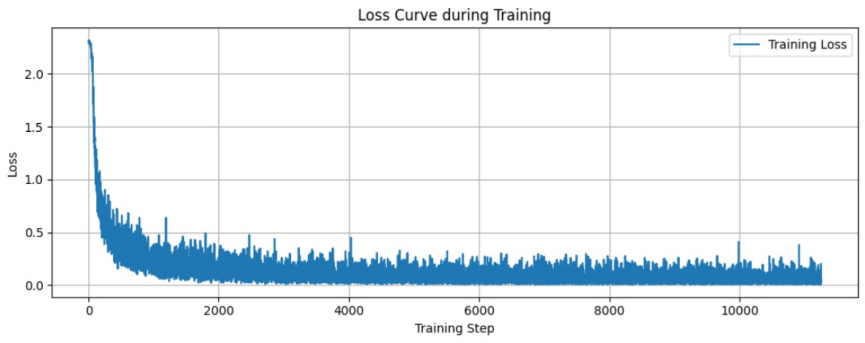


```
# =====================
# Hyperparameters
# =====================
input_size = 28
hidden_size = 64
num_layers = 1
num_classes = 10
batch_size = 64
learning_rate = 0.001
num_epochs = 12
```

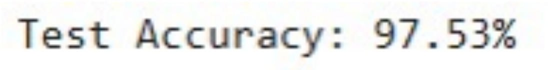
(b) Training loss record



```
Epoch [12/12], Step [100/938], Loss: 0.0457
Epoch [12/12], Step [200/938], Loss: 0.0635
Epoch [12/12], Step [300/938], Loss: 0.2328
Epoch [12/12], Step [400/938], Loss: 0.0107
Epoch [12/12], Step [500/938], Loss: 0.0130
Epoch [12/12], Step [600/938], Loss: 0.0762
Epoch [12/12], Step [700/938], Loss: 0.0151
Epoch [12/12], Step [800/938], Loss: 0.1069
Epoch [12/12], Step [900/938], Loss: 0.0076
Epoch 12 - Current learning rate: 0.000014
```

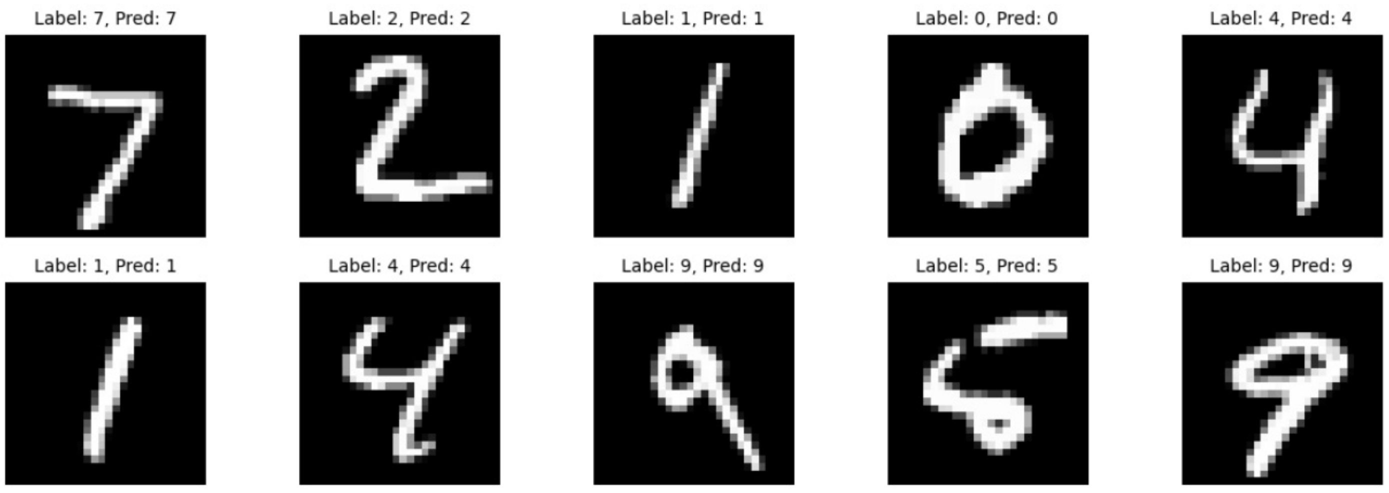


(c) Test Accruacy



```
Test Accuracy: 97.53%
```

(d) Prediction results.



Assignment Questions (Q5–Q7)

Q5: Explain briefly the role of the forget gate, input gate, output gate, and candidate cell in an LSTM.

- Forget Gate (f_t): Decides which parts of the previous cell state (c_{t-1}) to discard.
- Input Gate (i_t): Controls how much of the new candidate memory (c~_t) is added to the current cell state.
- Candidate Cell (c~_t): Represents new information to be potentially stored, calculated via tanh.
- Output Gate (o_t): Determines how much of the updated cell state (c_t) becomes the output hidden state (h_t).

These gates work together to manage memory in LSTM, enabling it to retain useful information and handle long-term dependencies better than a simple RNN.

Q6: Describe what hyperparameters you tuned and how they affected your model's final accuracy.

I adjusted the following:
- Hidden size: Increased to 64 for stronger sequence representation.
- Batch size: Reduced to 64 to improve generalization.
- Learning rate: Set to 0.001 and decayed using StepLR for stable training.
- Number of epochs: Increased to 12 to ensure full convergence.

These changes led to a test accuracy of 97.57% on the MNIST dataset.

Q7: Between a simple RNN and an LSTM, which one is better for sequence learning tasks?

LSTM is generally better for sequence learning, especially when long-term memory is required. It uses gating mechanisms to overcome vanishing gradients.

Simple RNNs can be sufficient for short sequences with limited dependencies and lower resource requirements.

Use LSTM for tasks like handwriting recognition, speech recognition, and language modeling. Use RNN for lightweight or simple time-series tasks.