Machine Learning

LABORATORY: LSTM-RNN In Class

NAME:	STUDENT ID#:

Objectives:

- Students will implement a Recurrent Neural Network (RNN) from scratch, without using built-in RNN APIs.
- Students will manually implement:
  - Hidden state updates over a sequence, a simple linear output layer for sentence classification and manual Stochastic Gradient Descent (SGD) updates.
- Students will understand how RNN hidden states summarize past information.
- Students will visualize the loss curve during training and observe model predictions for simple sentences.

Part 1. Instruction

In this assignment, you will implement a basic Vanilla Recurrent Neural Network (RNN) for simple sequence classification using PyTorch, but without using any high-level RNN modules (no *nn.RNN,* no *optim.SGD* , etc.).
You will manually implement:

- A step-by-step hidden state update based on the RNN equation.
- A manual simple output layer to predict whether a sentence is "good" or "bad".
- Manual parameter updates using basic gradient descent.

The general RNN computation is as follows:

$$s_t = \varphi\left(Ws_{t-1} + Ux_t + b\right)$$

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

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

where:

- $W_{out} \in R^{2\times n}$ = output weight matrix
- $b_{out} \in R^{2\times 1}$ = output bias
- The prediction is obtained by applying argmax over the logits.

Part 2. Code Template

Step	Procedure
1	import torch import matplotlib.pyplot as plt # 1. Vocabulary vocab = { "The": 0, "movie": 1, "is": 2, "good": 3, "bad": 4} def word_to_onehot(word): vec = torch.zeros(len(vocab), 1)

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

```
    vec[vocab[word]] = 1.0
    return vec
# 2. Training Samples
train_data = [
    (["The", "movie", "is", "good"], 1),
    (["The", "movie", "is", "bad"], 0)
]
# 3. Define RNN Parameters
n = 2   # hidden size
m = len(vocab)
T = 4   # number of words

torch.manual_seed(42)
```

2

```
# Initialize Weights (n x n), (n x m), (n x 1), (2 x n), (2 x 1)
W = None
U = None
b = None

W_out = None
b_out = None

# 4. Training Setup
learning_rate = 0.1
num_epochs = 300
loss_fn = torch.nn.CrossEntropyLoss()

loss_history = []
```

3

```
# 5. Training Loop
for epoch in range(num_epochs):
    total_loss = 0.0

    for sentence, label in train_data:
        inputs = [word_to_onehot(word) for word in sentence]
        s_prev = torch.zeros(n, 1)

        for x_t in inputs:
            # TODO: Compute s_t
            s_t = None  # TODO

            s_prev = s_t

        # Output layer
        logits = None  # TODO
        logits = logits.view(1, -1)

        # Loss
        target = torch.tensor([label])
        loss = loss_fn(logits, target)
        total_loss += loss.item()

        # Backward
        loss.backward()
```

```
        # Manual update (SGD)
        with torch.no_grad():
            # TODO: Update W, U, b, W_out, b_out
            pass

            # Zero gradients after updating
            W.grad.zero_()
            U.grad.zero_()
            b.grad.zero_()
            W_out.grad.zero_()
            b_out.grad.zero_()

    loss_history.append(total_loss)

    if (epoch + 1) % 50 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {total_loss:.4f}")
```

```
# 6. Plot Loss
plt.plot(loss_history)
plt.xlabel("Epoch")
plt.ylabel("Total Loss")
plt.title("Training Loss Curve")
plt.grid(True)
plt.show()
```

```
# 7. Test After Training
print("\n=== Testing ===")
test_sentences = [
    ["The", "movie", "is", "bad"],
    ["The", "movie", "is", "good"]
]

for test_sentence in test_sentences:
    inputs = [word_to_onehot(word) for word in test_sentence]
    s_prev = torch.zeros(n, 1)

    for x_t in inputs:
        #Forward pass again (same as above)
        s_t = None  # TODO
        s_prev = s_t

    logits = None  # TODO
    prediction = torch.argmax(logits)

    print(f"Sentence: {test_sentence} ➔ Prediction: {prediction.item()}")
```

Grading Assignment & Submission (30% Max)

Implementation:

1. (5%) Correctly initialize all weights.
2. (5%) Correctly compute the hidden state update and the output layer (logits).
3. (5%) Correctly update all parameters manually using gradients (manual SGD).
4. (5%) The model correctly predicts "good" = 1 and "bad" = 0 in the test sentences.

Question:

5. (5%) What does the hidden state s_t represent at each time step when processing a sequence of words?
6. (5%) Why are RNNs hard to train?

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 In Class Assignment**). Name your files correctly:
    a. Report: StudentID_Lab7_InClass.pdf
    b. Code: StudentID_Lab7_InClass.py or StudentID_Lab7_InClass.ipynb
4. Deadline: 16:20 PM
5. Plagiarism is **strictly prohibited**. Submitting copied work from other students will result in penalties.

Results and Discussion:

Epoch [50/300] - Loss: 0.0848

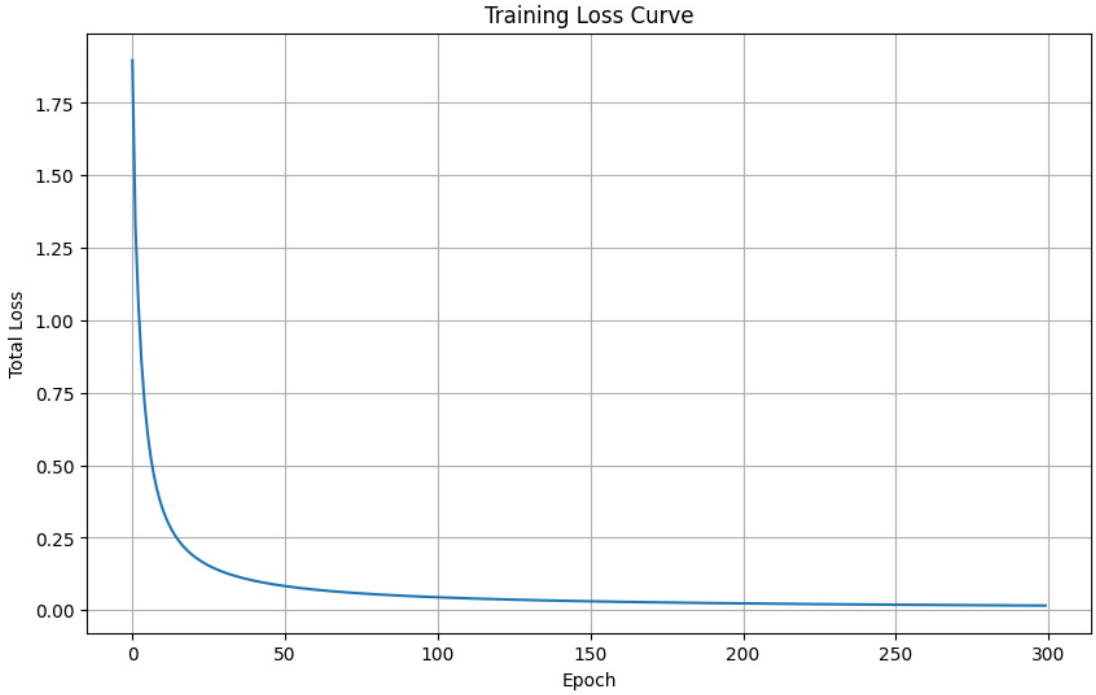
Epoch [100/300] - Loss: 0.0451

Epoch [150/300] - Loss: 0.0309

Epoch [200/300] - Loss: 0.0236

Epoch [250/300] - Loss: 0.0191

Epoch [300/300] - Loss: 0.0160



=== Testing ===

Sentence: ['The', 'movie', 'is', 'bad'] ➔ Prediction: 0 (bad)

Sentence: ['The', 'movie', 'is', 'good'] ➔ Prediction: 1 (good)

Question:

5. What does the hidden state s_t represent at each time step when processing a sequence of

words?

The hidden state s_t in a Recurrent Neural Network (RNN) serves as the network's memory, representing a compressed encoding of the sequence history up to the current time step. Let me explain this concept thoroughly:

Fundamental Role of the Hidden State

In an RNN, the hidden state at time step t (denoted as s_t) encodes information about all the words that have been processed in positions 1 through t. It acts as the network's internal memory mechanism, allowing information to flow from earlier parts of the sequence to later parts.

Mathematical Representation

Looking at the RNN equation from my lab:

$$s_t = tanh(W \cdot s_{t-1} + U \cdot x_t + b)$$

This equation reveals how the hidden state works:

- s_{t-1} is the previous hidden state (the memory up to that point)
- x_t is the current input (the word at position t)
- The new hidden state s_t combines both pieces of information

What Information Does It Capture?

The hidden state captures several types of information:

1. **Semantic Content**: It encodes the meaning of words seen so far. In my sentiment analysis example, words like "good" or "bad" contribute emotional valence to the hidden state.
2. **Contextual Information**: It represents how words relate to each other. The same word might contribute differently to the hidden state depending on what came before it.
3. **Positional Information**: Implicitly, it encodes where in the sequence I currently am, as this information is embedded through the recurrent updates.

4. **Relevant Features**: Through training, the network learns to preserve information that's useful for the classification task while discarding irrelevant details.

Concrete Example in My Assignment

In my movie review classification task:

- At t=1: After seeing "The", the hidden state contains minimal task-relevant information
- At t=2: After "The movie", it might encode that we're discussing a film
- At t=3: After "The movie is", it's prepared for a sentiment indicator
- At t=4: After the full sentence "The movie is good/bad", the hidden state contains the complete representation needed to make a sentiment prediction

The final hidden state s_T is particularly important because it serves as the summary of the entire sequence, which is why it's used for the final classification decision through the output layer: $logits = W_{out} \cdot s_T + b_{out}$.

Visualizing the Hidden State

I could think of the hidden state as a point moving through an n-dimensional space (where n is the hidden size, 2 in my assignment). As each word is processed, this point shifts to a new position that encodes the accumulated meaning. By the end of the sentence, the location of this point in the n-dimensional space should clearly separate "good" reviews from "bad" ones.

Limitations Worth Noting

While powerful, the hidden state in vanilla RNNs has limitations:

- Earlier information tends to fade as the sequence lengthens (the vanishing gradient problem)
- It has fixed capacity regardless of sequence length
- It must balance retaining old information with incorporating new information

This is why more sophisticated architectures like LSTM and GRU were developed,

providing more controlled mechanisms for updating and preserving information in the hidden state across longer sequences.

6. Why are RNNs hard to train?

Recurrent Neural Networks (RNNs) pose several unique training challenges that make them notoriously difficult to train effectively. Let me walk through the main obstacles:

The Vanishing Gradient Problem

The most fundamental issue with RNNs is the vanishing gradient problem. When we backpropagate through time (BPTT), gradients must flow backward through many time steps. During this process, the gradients are repeatedly multiplied by the recurrent weight matrix W:

For a basic RNN with activation function tanh, the gradient at time step t depends on products of terms like:

$$\partial L/\partial s_1 = \partial L/\partial s_T \cdot (\partial s_T/\partial s_{T-1}) \cdot (\partial s_{T-1}/\partial s_{T-2}) \cdot \ldots \cdot (\partial s_2/\partial s_1)$$

Each of these partial derivatives involves multiplying by W and the derivative of tanh, which has values between 0 and 1. When these values are repeatedly multiplied over many time steps, the result approaches zero exponentially fast.

This means that gradients for early time steps become vanishingly small, effectively preventing the network from learning long-range dependencies. In practical terms, my RNN might effectively "forget" the beginning of a sentence by the time it reaches the end.

The Exploding Gradient Problem

Conversely, under certain conditions (particularly with recurrent weight matrices whose eigenvalues exceed 1), gradients can grow exponentially large when backpropagated through time. This leads to:

- Enormous parameter updates that destabilize learning
- Numerical overflow issues
- Erratic training behavior with wildly oscillating loss values

Techniques like gradient clipping were developed specifically to address this problem by enforcing an upper bound on gradient magnitudes.

Difficulty Capturing Long-Range Dependencies

Because of the vanishing gradient problem, standard RNNs struggle to capture dependencies between elements that are far apart in a sequence. For example, in the sentence "The cat, who was sitting on the cozy mat that had been placed near the window earlier this morning by the housekeeper, purrs," an RNN might fail to maintain the connection between "cat" and "purrs" due to the long intervening phrase.

Limited Memory Capacity

The fixed-size hidden state must encode all relevant information from the past, regardless of sequence length. This creates an information bottleneck - as sequences get longer, more information must be compressed into the same fixed-dimensional vector.

Inefficient Computation

The sequential nature of RNNs makes them inherently difficult to parallelize:

- Each time step depends on the computation of the previous step
- This sequential processing is inefficient on modern parallel hardware like GPUs
- Training on long sequences becomes prohibitively slow

Unstable Dynamics

RNNs can exhibit chaotic behavior due to their recurrent connections. Small changes in input or parameters can lead to dramatically different outputs, creating:

- Difficulty in converging to stable solutions
- High sensitivity to weight initialization
- Training instability even with careful hyperparameter tuning

Architectural Limitations

Basic RNNs lack mechanisms to control information flow. They can neither

selectively forget irrelevant information nor preserve important information over many time steps. Each new input necessarily modifies the hidden state, potentially overwriting crucial earlier information.

Limited Context Window in Practice

Due to the vanishing gradient problem, vanilla RNNs effectively have a limited "context window" - they can only reliably use information from a certain number of steps back. This creates a disconnect between the theoretical ability to use unlimited context and practical limitations.

Solutions and Alternatives

These challenges led to the development of:

1. **LSTMs and GRUs**: These architectures introduce gating mechanisms that help control information flow, allowing the network to selectively remember or forget information.
2. **Gradient clipping**: Limiting gradient magnitudes to prevent explosion.
3. **Specialized initialization schemes**: Methods like orthogonal initialization help maintain gradient stability.
4. **Attention mechanisms**: These allow direct connections between arbitrary positions in a sequence, bypassing the need for storing all information in a hidden state.
5. **Transformer models**: These architectures eschew recurrence entirely in favor of self-attention, enabling parallel processing and capturing long-range dependencies more effectively.

Understanding these difficulties helps explain why, despite their conceptual elegance for sequence modeling, vanilla RNNs have been largely superseded by these more sophisticated architectures for most practical applications.