Assignment 7: Recurrent Neural Networks & Sequence Modeling

1 Instructions

In this assignment, you will explore sequential data modeling by implementing Recurrent Neural Networks (RNNs) from scratch and leveraging PyTorch's optimized RNN layers. Moving beyond fixed-window approaches, you will handle variable-length sequences natively.

You will construct Vanilla RNNs, LSTMs, and GRUs to understand their internal mechanics and how gating mechanisms solve the vanishing gradient problem. Finally, you will apply PyTorch's `nn.LSTM` across four distinct sequence modeling paradigms (One-to-One, Many-to-One, One-to-Many, Many-to-Many) to understand how data formatting and loss computation change based on the task.

1.1 Submission Requirements

You must submit two components to Gradescope:

1. **The Report:** A plaintext Markdown document containing the 5 sections described in Part 2.
    - **No Images:** Describe your findings with text and data tables.
    - **Character Limit:** 7,500 characters.
2. **Supplemental Material:** Your `.ipynb` notebook containing all code, plots, and raw results.

---

2 Part 1: Implementation (The Notebook)

You will author a Jupyter Notebook (`.ipynb`). Using **Google Colab** with a GPU runtime is highly recommended to keep training times for all models under 15 minutes.

Global Setup & Data Acquisition:

Fix the random seeds for reproducibility (e.g., `torch.manual_seed(42)`). You will use classic literary texts for this assignment. Run the following commands in your first Colab cell to download the raw text files directly from Project Gutenberg:

```
# Shakespeare's Sonnets (For Language Modeling)
!wget -q -O sonnets.txt https://www.gutenberg.org/cache/epub/1041/pg1041.txt
# Authors (For Author Identification Classification)
!wget -q -O austen.txt https://www.gutenberg.org/cache/epub/1342/pg1342.txt
!wget -q -O poe.txt https://www.gutenberg.org/cache/epub/2147/pg2147.txt
!wget -q -O shakespeare_play.txt https://www.gutenberg.org/cache/epub/1112/pg1112.txt
```

Sequence Encoding & Embeddings:

Neural networks cannot read raw strings. You must encode these sequences into numerical tensors.

1. **Vocabulary Mapping:** Create a dictionary that maps every unique character in your datasets, plus special tokens (`<PAD>`, `<SOS>`, `<EOS>`), to a unique integer index (e.g., `<PAD>` $\rightarrow 0$, `a` $\rightarrow 1$, `b` $\rightarrow 2$). The string `"cab"` becomes the sequence `[3, 1, 2]`.
2. **Embeddings:** Integers represent categories, not magnitudes. To give the network a continuous space to learn character relationships, use PyTorch's `nn.Embedding(num_embeddings=vocab_size, embedding_dim=hidden_size)`. This layer acts as a lookup table, transforming your sequence of integers into a sequence of dense, floating-point vectors of size `hidden_size`.

Datasets & DataLoaders (Handling Variable Lengths):

Because sentences are different lengths, they cannot naturally be stacked into a single rectangular tensor for batched processing. You must pad shorter sequences in a batch with your `<PAD>` token index so they match the longest sequence in that specific batch.

To achieve this in PyTorch, you will use a custom `collate_fn` inside your `DataLoader`. The `collate_fn` intercepts a list of data samples before they are batched and allows you to apply `torch.nn.utils.rnn.pad_sequence`.

2.1 Experiment 1: Architectural Mechanics (From Scratch)

Concept: In this experiment, we are focused on the task of **Author Identification**. Given a sentence, your model must classify whether it was written by Jane Austen, Edgar Allan Poe, or William Shakespeare. To understand how sequential memory works, you will build multiple RNN variants completely from scratch and compare their architectures for this specific classification task.

A Vanilla RNN consists of a hidden state h that updates sequentially by combining the previous hidden state h_{l-1} and the current input x_l. While elegant, vanilla RNNs suffer from vanishing gradients over long sequences because the gradient exponentially decreases with respect to the sequence length. LSTMs and GRUs introduce learned "gates" to control information flow, preserving long-term dependencies.

2.1.1 Task 1.1: Building the Cells

You will implement three recurrent cells for a single timestep using standard `nn.Linear` layers, `torch.tanh`, and `torch.sigmoid`. **Do not use `nn.RNN`, `nn.LSTM`, or `nn.GRU` for this task.**

- **Vanilla RNN Cell:** Implement the forward pass for a single timestep:

$$h_l = \tanh(W_h h_{l-1} + W_x x_l + b_h)$$

- **LSTM Cell:** Implement the forget f_t, input i_t, and output o_t gates. Let $h'_{t-1} = [h_{t-1}, x_t]$ represent the concatenated previous hidden state and current input. Calculate the new cell state C_t and hidden state h_t:

$$\tilde{C_t} = \tanh(W_c h'_{t-1} + b_c)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C_t}$$

$$h_t = o_t \odot \tanh(C_t)$$

- **GRU Cell:** Implement the GRU, which merges the cell and hidden states for a simpler architecture. Use the update gate z_t and reset gate r_t:

$$\tilde{h}_t = \tanh(W[r_t \odot h_{t-1}, x_t])$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

2.1.2 Task 1.2: The RNN Loop

Wrap your cells in a custom `nn.Module` that loops over the sequence dimension.

- **The Task:** Predict the author (3 classes) of a given sentence.
- **Implementation:** Your custom module must accept an input tensor `x_seq` of shape `(Batch, Seq_Len, Embed_Dim)` and a `lengths` tensor containing the true unpadded length of each sequence in the batch.
  - Depending on the RNN type, you will need to initialize the hidden state (and cell state for LSTM).
  - It should loop through the sequence dimension, applying your custom cell at each timestep, and storing the hidden states.
- **Extracting the "Last" Output:** Because your batch is padded with `<PAD>` tokens, simply taking the hidden state at the final timestep (`hiddens[:, -1, :]`) will often result in a hidden state that has processed multiple useless padding tokens. For author classification, you want the hidden state generated immediately after the *last valid character* (typically the `<EOS>` token) but *before* the padding begins. You can extract this in PyTorch using advanced indexing and the `lengths` tensor:

```
# Assuming 'all_hiddens' contains hidden states for all timesteps
# Shape: (Batch, Seq_Len, Hidden_Dim)
batch_size = all_hiddens.size(0)

# We subtract 1 because lengths are 1-indexed, but tensors are 0-indexed
last_valid_hiddens = all_hiddens[torch.arange(batch_size), lengths - 1, :]

# Pass last_valid_hiddens to your final nn.Linear layer...
```

2.1.3 Task 1.3: Deep (Stacked) RNNs

RNNs can be stacked into Deep RNNs to capture higher-level abstractions.

- **The Task:** Modify your custom module to support `num_layers=2`.
- **Implementation:** The sequence of hidden states produced by Layer 1 across all timesteps becomes the input `x_seq` for Layer 2. You will need to store the hidden state at each timestep from Layer 1 into a list, stack them into a tensor, and feed that tensor into Layer 2.

Notebook Output for Exp 1:

- Parameter count for your 1-Layer Vanilla RNN, 1-Layer LSTM, 1-Layer GRU, and 2-Layer LSTM.
- A single plot overlaying the Training Loss curves for all four models.
- Final Test Accuracy for all four models.

2.2 Experiment 2: Sequence Modeling Paradigms

Concept: RNNs are incredibly flexible. By changing how we construct the inputs and evaluate the outputs, the exact same core architecture can solve vastly different sequence problems.

For this experiment, use PyTorch's optimized `nn.LSTM` (or `nn.GRU`) and an `nn.Embedding` layer. Ensure your loss function ignores `<PAD>` tokens (e.g., `ignore_index=0` in `CrossEntropyLoss`).

2.2.1 Task 2.1: One-to-One (The Baseline)

Narrative: Before testing sequential memory, we need a baseline. Can the network guess the author just by looking at the very first character? This paradigm treats the data as a standard, non-sequential classification task using a sequence length of exactly one.

Implementation Details:

- **Dataset:** Author Identification.
- **Input Construction:** Slice your dataset to extract only the first character index of each sentence. The input tensor shape must be `(Batch, 1)`.
- **Forward Pass:** Pass the input through your Embedding layer, then the `nn.LSTM`. The LSTM output will have shape `(Batch, 1, Hidden_Dim)`. Squeeze the sequence dimension and pass it to your classification head.

- **Output Construction:** The final output shape is `(Batch, 3)` (logits for the 3 authors). Compute standard Cross Entropy Loss against the true author label.

2.2.2 Task 2.2: Many-to-One (Sequence Classification)

Narrative: This is the standard sequence classification setup (identical to Experiment 1). The network reads the entire sentence character-by-character, accumulating context. We only extract the network's "final thought" after the last character is processed to make our prediction.

Implementation Details:

- **Dataset:** Author Identification.
- **Input Construction:** The input is the full, padded sentence. Shape: `(Batch, Seq_Len)`. Pass it through the Embedding layer.
- **Forward Pass:** Pass the embedded sequence into `nn.LSTM`.
- **Output Construction:** As in Experiment 1, you must extract the hidden state corresponding to the *last valid character* before the `<PAD>` tokens begin using advanced indexing with sequence lengths, or by using PyTorch's `torch.nn.utils.rnn.pack_padded_sequence` utility. Pass this final valid hidden state to a Linear layer to get `(Batch, 3)` logits. Compute Cross Entropy Loss.

2.2.3 Task 2.3: One-to-Many (Sequence Generation)

Narrative: In this generative task, we want the network to hallucinate a novel sentence given a specific prompt. We provide the author class as the single initial input, and the network generates characters one by one. The output at timestep t becomes the input for timestep $t + 1$, continuing until the network outputs an End of Sequence (`<EOS>`) token.

Implementation Details:

- **Dataset:** Author Identification.
- **Input Construction:** During training, the input is the author class index (0, 1, or 2). Pass this through a distinct *author embedding layer* to get a vector. Use this vector to initialize the starting hidden state h_0 of the LSTM.
- **Forward Pass:** Feed a `<SOS>` (Start of Sequence) token as the first character input x_1. The LSTM outputs a hidden state h_1. Pass h_1 to a Linear layer mapped to the character vocabulary size to get character logits.
- **Output Construction & Loop:** During training, use "Teacher Forcing" (feed the true next character from the dataset as the next input). During inference/testing, take the `argmax` (or sample from the distribution) of the logits to get the predicted character, embed it, and feed it in as the next input. Stop when `<EOS>` is generated.

2.2.4 Task 2.4: Many-to-Many (Language Modeling)

Narrative: We want to train the model to anticipate the future. Instead of waiting until the end of the text to compute loss, the network must guess the *next* letter at *every single timestep*. The

input and target sequences are identical, except the target sequence is shifted one timestep into the future.

Implementation Details:

- **Dataset:** Shakespeare's Sonnets. Chunk the continuous text file into sequences of length 40.
- **Input Construction:** Prepend the `<SOS>` token to your string chunk. The chunk `"Shall I comp"` becomes the input sequence `[<SOS>, 'S', 'h', 'a', 'l', 'l', ...]`. Shape is `(Batch, Seq_Len)`.
- **Target Construction:** Append the `<EOS>` token to your string. The target sequence is `['S', 'h', 'a', 'l', 'l', ..., <EOS>]`.
- **Forward Pass:** Pass the embedded input sequence through the `nn.LSTM`. The output tensor contains a hidden state for every timestep, shaped `(Batch, Seq_Len, Hidden_Dim)`.
- **Output Construction:** Pass the entire output tensor through a Linear layer to get `(Batch, Seq_Len, Vocab_Size)`.
- **Loss Computation:** To compute Cross Entropy Loss across all timesteps simultaneously, reshape your logits to `(Batch * Seq_Len, Vocab_Size)` and your targets to `(Batch * Seq_Len)`.

Notebook Output for Exp 2:

- Final Accuracies for Tasks 2.1 and 2.2.
- Print 2 generated sentences styled as Shakespeare and 2 styled as Poe from Task 2.3.
- Print a 150-character generated sequence from your language model in Task 2.4 (seeded with the string `"The summer"`).

---

3 Part 2: Content Requirements (The Report)

Your report must be organized into **exactly five sections** with Markdown headers.

3.1 Section 1: Executive Summary

- **One-Sentence Takeaway:** Summarize the core finding contrasting the capabilities of the Vanilla RNN vs. LSTM/GRU.
- **Summary Paragraph:** Outline the progression of the assignment, highlighting the transition from building cells from scratch to orchestrating complex input/output paradigms.

3.2 Section 2: Architectural Mechanics (From Scratch)

- **Results Table:** A Markdown table comparing the Parameter Count, Final Training Loss, and Test Accuracy of your Custom Vanilla RNN, LSTM, GRU, and Deep LSTM.
- **Analysis:** Based on your loss curves, explain *why* the LSTM/GRU converged better or faster than the Vanilla RNN. Mechanically, how do the f_t, i_t, and o_t gates mitigate the vanishing gradient problem you observed?

3.3 Section 3: Data Formatting & Embeddings

- **Dimensionality:** Clearly explain the tensor shapes required by `nn.LSTM` (e.g., how the batch dimension interacts with the sequence dimension).
- **Batching:** Explain the necessity of `pad_sequence`. Why do we need to mask or ignore `<PAD>` tokens when computing the Many-to-Many sequence loss?

3.4 Section 4: The Four Paradigms

- **Comparison:** Compare the performance of One-to-One (First Character) vs. Many-to-One (Full Sentence) classification. What does this gap mechanically prove about the utility of sequence models?
- **Loss Dynamics:** Contrast the loss calculation in Task 2.2 (Many-to-One) vs Task 2.4 (Many-to-Many). Why does Task 2.4 calculate loss at *every* step instead of just the last step?

3.5 Section 5: Reflection

- **Generative vs. Discriminative:** Reflect on the difference between identifying an author (Task 2.2) and generating text (Task 2.3/2.4). Which task felt more difficult to format and implement, and why?
- **General Thoughts:** Provide any other insightful observations on PyTorch's `nn.RNN` implementation versus your manual implementation.

---

4 Grading Rubric

Each of the five sections is weighted equally (**20% each**).

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 1: Summary	Takeaway is specific and quantitatively grounded. Summary captures the architectural progression.	Takeaway is clear. Summary covers the main assignment arc.	Takeaway is generic. Summary is vague.	Summary misses key elements of the analysis.	Missing.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 2: Mechanics	Table is complete. Analysis demonstrates a deep, mechanical understanding of gating and gradient flow.	Table complete. Good explanation of vanishing gradients and LSTM fixes.	States results but the mechanical "why" behind gating is vague.	Values incorrect or analysis is missing.	Missing.
Section 3: Formatting	Clear, accurate explanation of tensor shapes, batching, and `<PAD>` masking logic.	Explains shapes and padding well, but masking logic is unclear.	Generic explanation of PyTorch dataset objects.	Incorrect dimensional explanations.	Missing.
Section 4: Paradigms	Deep comparison of One vs Many performance. Excellent explanation of timestep-wise loss computation.	Good comparison. Mentions loss differences but lacks specific technical reasoning.	Compares tasks but ignores the loss implementation discussion.	Implausible comparisons due to failed code.	Missing.
Section 5: Reflection	Thoughtful reflection on Generative vs. Discriminative sequence tasks and library constraints.	Good reflection on debugging generative models.	Generic reflection (e.g., "RNNs are hard").	Minimal effort.	Missing.