STATS-413 HW-4

Author: Hochan Son

UID: 206547205

Date: @November 6, 2025

Problem 1 "Barack Obama" example. Consider a MLP with one hidden layer. Let x be the input (Barack), h be the hidden layer, and y be the output (Obama). Suppose both x and y are words in the same dictionary, where x is the current word, and y is the next word. Both x and y are on-hot vector.

$$
\begin{gathered}
Let\ h = W_{embed} x, \\
s = W_{unembed} h, \\
p = \text{softmax}(s), \\
y \sim p \\
p(y_c = 1 | s) = p_c
\end{gathered}
$$

where

- $x = 'Barack'$: input layer
- $h = W_{embed}$: hidden layer
- $s = W_{unembed}\ h$:
- $p = softmax(s)$: probability of the output distribution
- $y = 'Obama'$: predictor

(1)What are the dimensionalities of W_{embed} and $W_{unembed}$? Interpret the meaning of the columns of W_{embed}

Let's say:

- Word size = W(100k) and,
- Hidden layer size = H (100 dimensions)

The dimensionality of each layers

- $W_{embed} = shape(H \times W)$
  - W_k is a dimension of one-hot embedding on K-th elements
  - H is hidden layer representation
- $W_{unembed} = shape(W \times H)$
  - $H \text{ is a hidden states } \odot \text{W-dimension logits score(for each word)}$

$(2)\text{Let } J = log\ p(y|s),\ \text{show that } \partial J/\partial s = y - p = e.$ $\text{Calculate } \partial J/\partial h,\ \partial J/\partial W_{unembed},\ \text{and } \ \partial J/\partial W_{embed} \text{ with chain rule}$ In your calculation, you can first pretend all the vectors and matrices are scalars (one-dimensional numbers), and then guess the forms of the general results.

Part 1: Loss Function

starting with softmax cross-entropy:

$$J = log\ p(y|s)$$

$$\begin{aligned} J &= logp(y|s) \\ &= \sum_c y_c logp_c \end{aligned}$$

For one-hot encoded target y(only one correct class):

$$= \sum_c y_c logp_c$$

Since y is one-hot, only one term survives (where $y_c = 1$):

$$\begin{gathered} ; p(y|s) = \prod_c p_c^{y_c} \\ p_c = \frac{e_c^s}{z} \text{ where } \sum_c e^{s_c} \\ J = \sum_c y_c(s_c - logz) \\ = \sum_c y_c s_c - logz \sum_c y_c \\ = (\sum_c y_c s_c) - logz \end{gathered}$$

Since $\sum y_c = 1$:

$$\boxed{J = \sum_c y_c s_c - logz}$$

Part 2: Gradient w.r.t Logits $(\partial J/\partial s)$

$$\frac{\partial J}{\partial s_c} = y_c - \frac{1}{z}\frac{\partial z}{\partial s_c}$$

Since $z = \sum e^s$:

$$\frac{\partial z}{\partial s} = e^s_c$$

Therefore:

$$\boxed{\frac{\partial J}{\partial s} = y - p = error}$$

Part 3: Gradient w.r.t. Hidden State $(\partial J/\partial h)$:

using the chain rule:

$$\frac{\partial J}{\partial h_i} = \sum_k \frac{\partial J}{\partial s_k} \cdot \frac{\partial s_k}{\partial h_i}$$

Since $s_k = \sum_j W^u_{kj} h_j$:

$$\frac{\partial s_k}{\partial h_i} = W^u_{ki}$$

Therefore:

$$\frac{\partial J}{\partial h_i} = \sum_k \frac{\partial J}{\partial s_k} \cdot W^u_{ki}$$

In matrix form:

$$\boxed{\frac{\partial J}{\partial h} = W^T_{unembed} \cdot \frac{\partial J}{\partial s}}$$

Part 4: Gradient w.r.t. Unembed Weights $(\partial J/\partial W_{unembed})$:

$$\frac{\partial J}{\partial W^u_{jk}} = \frac{\partial J}{\partial s_k} \cdot \frac{\partial s_k}{\partial W^u_{kj}}$$

Since $s_k = \sum_j W^u_{kj} h_j$

$$\frac{\partial s_k}{\partial W^u_{kj}} = h_j$$

Therefore:

$$\boxed{\frac{\partial J}{\partial W_{unembed}} = \frac{\partial J}{\partial s} \cdot h^T}$$

Part 5: Gradient w.r.t Embed Weights $(\partial J / \partial W_{embed})$:

$$\frac{\partial J}{\partial W^e_{kj}} = \frac{\partial J}{\partial h_i} \cdot \frac{\partial h_i}{\partial W^e_{kj}}$$

Since $h_i = \sum_k W^e_{ik} x_k$:

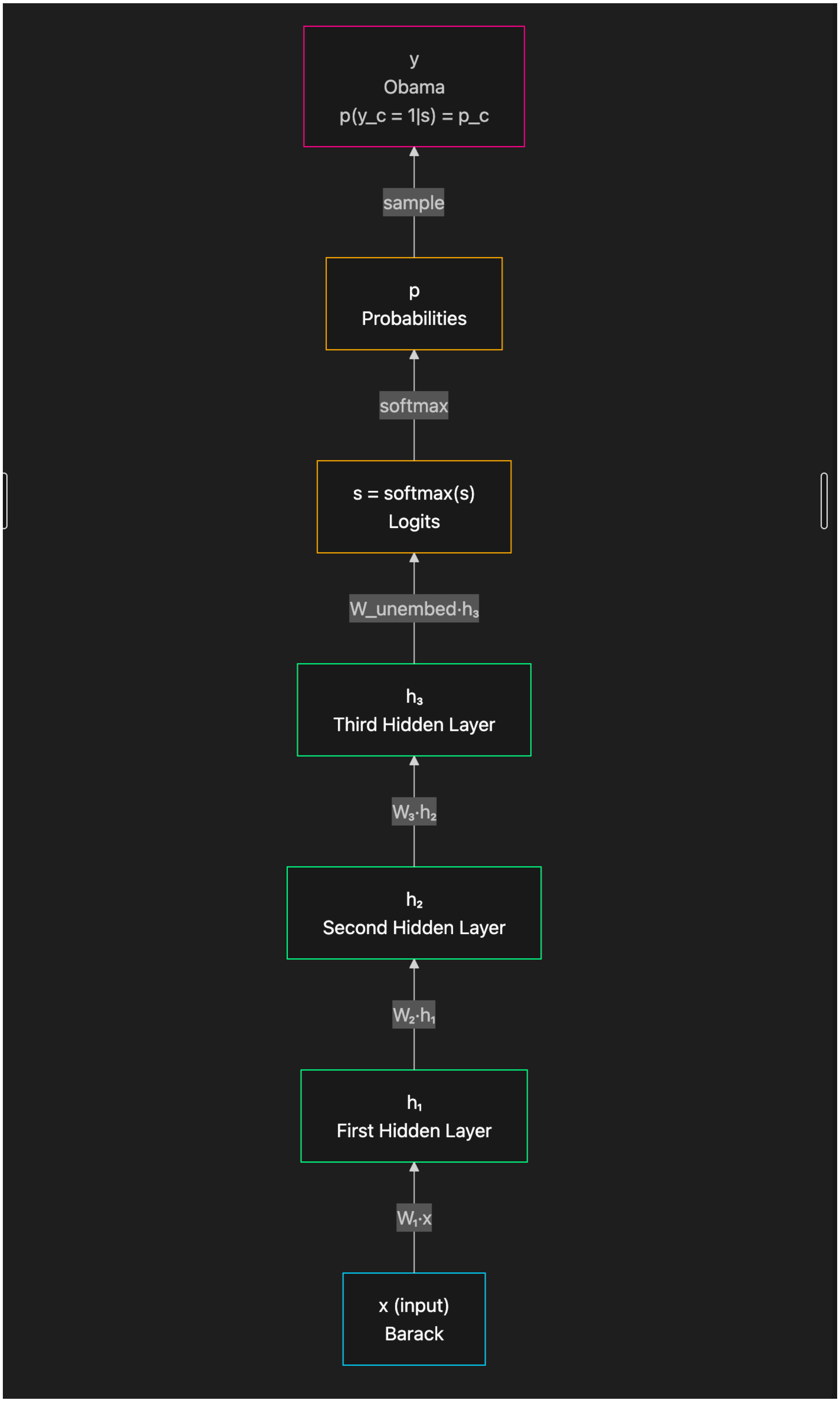
$$\frac{\partial h_i}{\partial W^e_{kj}} = x_j$$

Therefore:

$$\boxed{\frac{\partial J}{\partial W_{embed}} = \frac{\partial J}{\partial h} \cdot x^T}$$

(3) Draw a diagram of network with multiple layers of latent vectors

The multi layer perceptron:


y
Obama
p(y_c = 1|s) = p_c
sample
p
Probabilities
softmax
s = softmax(s)
Logits
W_unembed·h₃
h₃
Third Hidden Layer
W₃·h₂
h₂
Second Hidden Layer
W₂·h₁
h₁
First Hidden Layer
W₁·x
x (input)
Barack

```
graph BT
    x["x (input)<br/>Barack"] →|W₁·x| h1["h₁<br/>First Hidden Layer"]
    h1 →|W₂·h₁| h2["h₂<br/>Second Hidden Layer"]
    h2 →|W₃·h₂| h3["h₃<br/>Third Hidden Layer"]
    h3 →|W_unembed·h₃| s["s = softmax(s)<br/>Logits"]
    s →|softmax| p["p<br/>Probabilities"]
    p →|sample| y["y<br/>Obama<br/>p(y_c = 1|s) = p_c"]

    style x fill:#1a1a1a,stroke:#00d4ff,color:#fff
    style h1 fill:#1a1a1a,stroke:#00ff88,color:#fff
    style h2 fill:#1a1a1a,stroke:#00ff88,color:#fff
    style h3 fill:#1a1a1a,stroke:#00ff88,color:#fff
    style s fill:#1a1a1a,stroke:#ffaa00,color:#fff
    style p fill:#1a1a1a,stroke:#ffaa00,color:#fff
    style y fill:#1a1a1a,stroke:#ff0088,color:#fffRetry
```

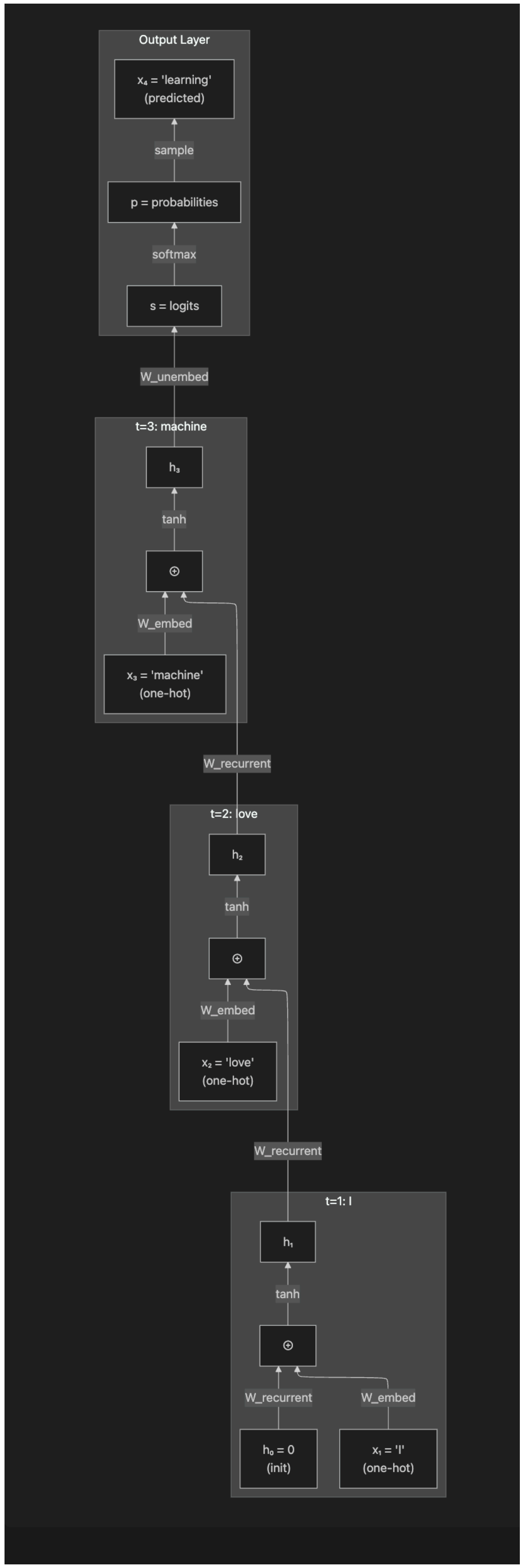
Where:

- Input (x): “Barack” as one-hot vector
- 3 hidden layers (h1,h2,h3): Each layer transforms the previous representation through learned weight matrices
- Logits (s): Final hidden layer projected back to vocabulary space
- Probability (p): Softmax applied to logits
- Output (y): Sampled next word “Obama”

Problem 2. “I love machine learning” example. Suppose we observe “I love machine” and we want to predict the next word.

$$
\begin{gathered}
Let\ x_t \text{ be one-hot vectors,} \\
x_1 = \text{“}I\text{”}, \\
x_2 = \text{“}love\text{”}, \\
x_3 = \text{“}machine\text{”}. \\
Let\ h_t \text{ be the hidden vectors,} \\
h_0 = 0, \\
h_1 = tanh(W_{embed}X_1 + W_{recurrent}h_0), \\
h_2 = tanh(W_{embed}X_2 + W_{recurrent}h_1), \\
h_3 = tanh(W_{embed}X_3 + W_{recurrent}h_2), and \\
s = W_{unembed}h_3, \\
p = softmax(s), \\
x_4 \text{ is sampled according to } p.
\end{gathered}
$$

(1) Draw a diagram to illustrate the model


Output Layer
x₄ = 'learning'
(predicted)
sample
p = probabilities
softmax
s = logits
W_unembed
t=3: machine
h₃
tanh
⊕
W_embed
x₃ = 'machine'
(one-hot)
W_recurrent
t=2: love
h₂
tanh
⊕
W_embed
x₂ = 'love'
(one-hot)
W_recurrent
t=1: I
h₁
tanh
⊕
W_recurrent
W_embed
h₀ = 0
(init)
x₁ = 'I'
(one-hot)

```
graph BT
    subgraph t1["t=1: I"]
        x1["x₁ = 'I'<br/>(one-hot)"]
        h0["h₀ = 0<br/>(init)"]
        x1 -->|W_embed| add1["⊕"]
        h0 -->|W_recurrent| add1
        add1 -->|tanh| h1["h₁"]
    end

    subgraph t2["t=2: love"]
        x2["x₂ = 'love'<br/>(one-hot)"]
        x2 -->|W_embed| add2["⊕"]
        h1 -->|W_recurrent| add2
        add2 -->|tanh| h2["h₂"]
    end

    subgraph t3["t=3: machine"]
        x3["x₃ = 'machine'<br/>(one-hot)"]
        x3 -->|W_embed| add3["⊕"]
        h2 -->|W_recurrent| add3
        add3 -->|tanh| h3["h₃"]
    end

    subgraph output["Output Layer"]
        h3 -->|W_unembed| s["s = logits"]
        s -->|softmax| p["p = probabilities"]
        p -->|sample| x4["x₄ = 'learning'<br/>(predicted)"]
    end
```

(2) Let $J = logp(x_4|s)$.

Calculate $\partial J/\partial W_{embed}, \partial J/\partial W_{unembed},$ and $\partial J/\partial\ W_{recurrent}$.***In your calculation, you can first pretend all the vectors and matrices are scalars (one-dimensional numbers), and then guess the forms of the general results.***

Let x_t be one-hot vectors.$x_1 = "I", x_2 = "love", x_3 = "machine"$
Let $h_0 = 0,\ h_t = \tanh(z_1)\ for\ t = 1, 2, 3$ where $z_1 = W_{embed}x_t + W_{recurrent}h_{t-1}$
Let $s = W_{unembed}h_3,\ p = softmax(s),\ x_4 \sim p$

Step 1. Forward pass:

$$
\begin{gathered}
h_0 = 0 \\
z_1 = W_{embed} x_1,\ h_1 = \tanh(z_1) \\
z_1 = W_{embed} x_2 + W_{recurrent} h_1,\ h_2 = \tanh(z_2) \\
z_1 = W_{embed} x_3 + W_{recurrent} h_2,\ h_3 = \tanh(z_3) \\
s = W_{unembed} h_3,\ p = softmax(s),\ J = \log p(x_4|s)
\end{gathered}
$$

Key Gradients (given earlier):

$$\frac{\partial J}{\partial s} = y - p = e \text{ (error signal)}$$

Define the backward errors:

$$\delta_t = \frac{\partial J}{\partial h_t}$$

Back propagate the error:

Recall: $s = W_{unembed} h_3, \text{ and } \partial J / \partial s = e = (y - p)$

$$
\begin{gathered}
\delta_3 = \frac{\partial J}{\partial h_3} = \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial h_3} = e \cdot W_{unembed}^T \\
\delta_2 = \frac{\partial J}{\partial h_2} = \frac{\partial J}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} = \delta_3 \cdot \sigma'(z_3) \cdot W_{recurrent}^T \\
\delta_1 = \frac{\partial J}{\partial h_1} = \frac{\partial J}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} = \delta_2 \cdot \sigma'(z_2) \cdot W_{recurrent}^T
\end{gathered}
$$

Let trace the error propagates all the way back to h_1:

$$\delta_1 = \delta_2 \cdot \sigma'(z_2) \cdot W_{recurrent}^T$$

Substitute δ_2:

$$\delta_1 = [\delta_3 \cdot \sigma'(z_3) \cdot W_{recurrent}^T] \cdot \sigma'(z_2) \cdot W_{recurrent}^T$$

Substitute δ_3:

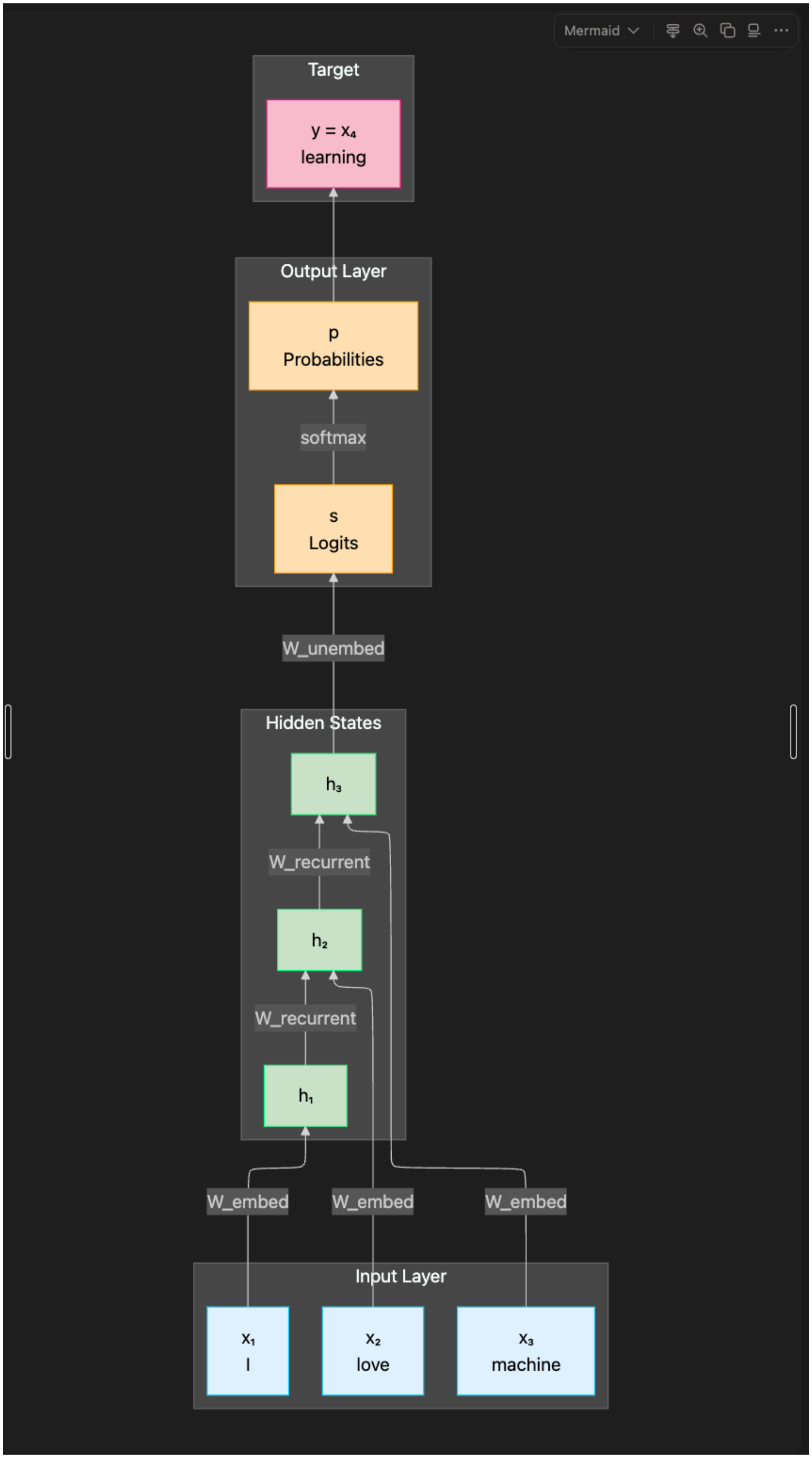
$$
\begin{gathered}
\delta_1 = [W_{recurrent}^T \cdot e] \cdot \sigma'(z_3) \cdot W_{recurrent}^T \cdot \sigma'(z_2) \cdot W_{recurrent}^T \\
\boxed{= W_{recurrent}^T \cdot e \cdot \sigma'(z_3) \cdot W_{recurrent}^T \cdot \sigma'(z_2) \cdot W_{recurrent}^T}
\end{gathered}
$$

where $\sigma'(z_t) = 1 - \tanh^2(z_t)$

- Calculate \partial J / \partial W_{unembed}

$$
\begin{gathered}
s = W_{unembed} h_3 \\
\frac{\partial J}{\partial W_{unembed}} = \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial W_{unembed}} \\
\boxed{\frac{\partial J}{\partial W_{unembed}} = e \cdot h_3}
\end{gathered}
$$

(3) Draw a diagram of network with multiple recurrent layers of latent vector


Mermaid
Target
$y = x_4$
learning
Output Layer
p
Probabilities
softmax
s
Logits
W_unembed
Hidden States
h_3
W_recurrent
h_2
W_recurrent
h_1
W_embed
W_embed
W_embed
Input Layer
x_1
I
x_2
love
x_3
machine

```
graph BT
    subgraph input["Input Layer"]
        x1["x₁<br/>I"]
        x2["x₂<br/>love"]
        x3["x₃<br/>machine"]
    end

    subgraph hidden["Hidden States"]
        h1["h₁"]
        h2["h₂"]
        h3["h₃"]
    end

    subgraph output["Output Layer"]
        s["s<br/>Logits"]
        p["p<br/>Probabilities"]
    end

    subgraph target["Target"]
        y["y = x₄<br/>learning"]
    end

    x1 -->|W_embed| h1
    x2 -->|W_embed| h2
    x3 -->|W_embed| h3

    h1 -->|W_recurrent| h2
    h2 -->|W_recurrent| h3

    h3 -->|W_unembed| s
    s -->|softmax| p
    p --> y

    style x1 fill:#e1f5ff,stroke:#00d4ff,color:#000
    style x2 fill:#e1f5ff,stroke:#00d4ff,color:#000
    style x3 fill:#e1f5ff,stroke:#00d4ff,color:#000
    style h1 fill:#c8e6c9,stroke:#00ff88,color:#000
    style h2 fill:#c8e6c9,stroke:#00ff88,color:#000
    style h3 fill:#c8e6c9,stroke:#00ff88,color:#000
    style s fill:#ffe0b2,stroke:#ffaa00,color:#000
    style p fill:#ffe0b2,stroke:#ffaa00,color:#000
    style y fill:#f8bbd0,stroke:#ff0088,color:#000
```

$$\boxed{h_0 \to h_1 \odot W_{embed}x_1 \to h_2 \odot W_{embed}x_2 \to h_3 \odot W_{embed}x_3 \to s : W_{unembed}h_3 \to p \to x_4}$$

Problem 3 Residual steam, For the “I love machine learning” example, consider the residual parameterization:

- $h_t = h_{t-1} + \tanh(W_{recurrent}\ h_{t-1} + W_{embed}x_t)$
- $Let\ J = logp(x_4|s),\ \text{where}\ s = W_{unembed}\ h_3,$

Starting from $\partial J/\partial h_3,$ calculate $\partial J/\partial h_1.$

Step 1: Forward Pass (computing hidden states)

At t=1:

$$z_1 = W_{recurrent}h_0 + W_{embed}x_1$$
$$h_1 = h_0 + \tanh(z_1) = 0 + \tanh(z_1) = \tanh(z_1)$$

At t=2:

$$z_2 = W_{recurrent}\ h_1 + W_{embed}x_2$$
$$h_2 = h_1 + \tanh(z_2)$$

At t=3:

$$z_3 = W_{recurrent}h_2 + W_{embed}x_3$$
$$h_3 = h_2 + \tanh(z_3)$$

Step 2: Backward pass(Gradient)

Given $\partial J/\partial h_3$

the gradient from the output layer:

$$\frac{\partial J}{\partial h_3} = (\text{Given from loss computation})$$

From earlier problems we know:

$$\frac{\partial J}{\partial s} = y - p = e(error)$$

where $s = W_{recurrent}h_3$ and $p = softmax(s)$

Therefore,

$$\boxed{\frac{\partial J}{\partial h_3} = \frac{\partial J}{\partial s} \cdot \frac{\partial s}{\partial h_3} = e \cdot W_{unembed}^T}$$

Back propagation h3→h2

Step 1: Set up the Jacobian:

Recall the residual update equation:

$$h_3 = h_2 + \tanh(z_3)$$

where $z_3 = W_{recurrent} h_2 + W_{embed} x_3$

Taking the derivative w.r.t. h_2:

$$\frac{\partial h_3}{\partial h_2} = \frac{\partial}{\partial h_2}(h_2 + \tanh(z_3))$$

split into two terms,

$$= \frac{\partial h_2}{\partial h_2} + \frac{\partial \tanh(z_3)}{\partial h_2}$$
$$= I + \frac{\partial \tanh(z_3)}{\partial h_2}$$

Step 2: Apply the chain rule to the tanh term

For the tanh derivative, use the chain rule:

$$\frac{\partial \tanh(z_3)}{\partial h_2} = \frac{\partial \tanh(z_3)}{\partial z_3} \times \frac{\partial z_3}{\partial h_2}$$

1. Compute tanh derivative rule:

$$\frac{d}{du}\tanh(u) = 1 - tanh^2(u)$$

so the tanh derivative is:

$$\frac{\partial \tanh(z_3)}{\partial z_3} = 1 - \tanh^2(z_3) = \sigma\text{'}(z_3)$$

2. Compute $\frac{\partial z_3}{\partial h_2}$

Recall:

$$z_3 = W_{recurrent} h_2 + W_{embed} x_3$$

Taking the derivative w.r.t. h_2:

$$\frac{\partial z_3}{\partial h_2} = \frac{\partial}{\partial h_2}(W_{recurrent} h_2 + W_{embed} x_3)$$

$\text{The } W_{embed} x_3 \text{ term doesn't depend on } h_2, \text{ so its derivative is } 0.$

$$= \frac{\partial}{\partial h_2}(W_{recurrent} h_2) = W_{recurrent} : \text{weight matrix}$$

3. Multiply the two derivatives (chain rule)

$$\frac{\partial \tanh(z_3)}{\partial h_2} = \frac{\partial \tanh(z_3)}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2}$$
$$= (1 - \tanh^2(z_3)) \cdot W_{recurrent}$$
$$= \sigma'(z_3) \cdot W_{recurrent}$$

Therefore:

$$\boxed{\frac{\partial h_3}{\partial h_2} = I + \sigma'(z_3) \cdot W_{recurrent}}$$

where sigma′(z_3) = 1-\tanh^2(z_3) is a scalar (or diagonal matrix when considering batch dimensions)

4. apply the chain rule for gradients

$$\frac{\partial J}{\partial h_2} = \frac{\partial J}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2}$$

Substitute the Jacobian:

$$\boxed{\frac{\partial J}{\partial h_2} = \frac{\partial J}{\partial h_3}(I + \sigma'(z_3)W_{recurrent})}$$

5. Expand element-wise distribute the multiplication

$$\frac{\partial J}{\partial h_2} = \frac{\partial J}{\partial h_3}I + \frac{\partial J}{\partial h_3}\sigma\text{'}(z_3)W_{recurrent}$$

Simplify (since $x \cdot I = x$):

$$\boxed{\frac{\partial J}{\partial h_2} = \underbrace{\frac{\partial J}{\partial h_3}}_{direct\ path} + \underbrace{\frac{\partial J}{\partial h_3}\sigma\text{'}(z_3)W_{recurrent}}_{recurrent\ path}}$$

Observations:

Two gradient paths:

1. Direct path (Identity term):
   - Gradient flows unchanged through I (identity)
   - No attenuation
2. Recurrent path (Tanh term):
   - Gradient multiplied by $\sigma\text{'}(z_3) \subset (0, 1)$ and $W_{recurrent}$
   - May vanish

Even if the recurrent path vanishes, the direct path keeps gradient flowing!

Problem 4 Please play with the PyTorch code provided by the following webpage:

https://machinelearningmastery.com/text-generation-with-lstm-in-pytorch/

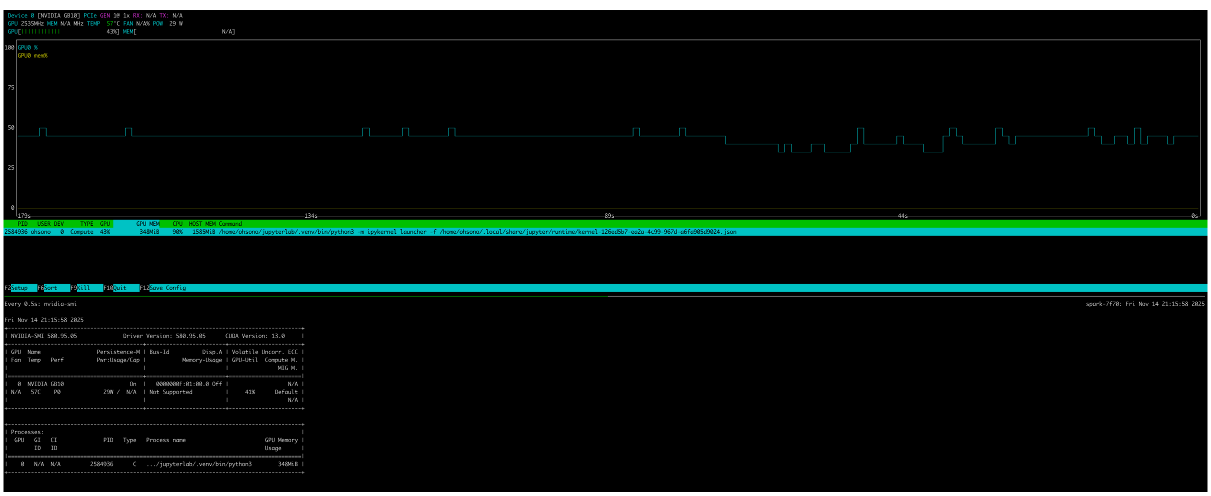
Please write a brief explanation of the code and show your results. You can explore the code by varying the design parameters.

What I used a text for training LSTM model

- The prince, Nicolo Machiavelli
- Source, https://www.gutenberg.org/files/1232/1232-h/1232-h.htm
- Parameters:

  ```
  Using device: cuda
  Loaded 282277 characters
  Vocab size: 55
  Dataset size: 282227 sequences
  Model parameters: 942775
  ```

- Machine Specs:
  - DGX-Spark, GB10, 128GB Memory
  - GPU utilization： 43% / Memory Utilization: 16GB



- CODE SAMPLE

```
"""
COMPLETE PyTorch LSTM Text Generation Implementation
Based on MachineLearningMastery approach
"""

import torch
import torch.nn as nn
```

```
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np

# ======================================================================
===
# STEP 1: DATA PREPARATION
# ======================================================================
===

class TextDataset(Dataset):
    """Convert text into character sequences for LSTM training"""

    def __init__(self, text, seq_length=50):
        """
        Args:
            text: Raw text string
            seq_length: Length of input sequences
        """
        self.seq_length = seq_length

        # Build character vocabulary
        self.chars = sorted(set(text))
        self.char_to_idx = {c: i for i, c in enumerate(self.chars)}
        self.idx_to_char = {i: c for i, c in enumerate(self.chars)}

        # Encode entire text to indices
        self.text_encoded = [self.char_to_idx[c] for c in text]

    def __len__(self):
        return len(self.text_encoded) - self.seq_length

    def __getitem__(self, idx):
        """Return (input_sequence, target_sequence) pair"""
        x = torch.tensor(
            self.text_encoded[idx:idx + self.seq_length],
            dtype=torch.long
        )
        y = torch.tensor(
            self.text_encoded[idx + 1:idx + self.seq_length + 1],
            dtype=torch.long
        )
        return x, y

    def decode(self, indices):
        """Convert indices back to text"""
        return ''.join([self.idx_to_char[i] for i in indices])
```

```
# ==========================================================================
===
# STEP 2: MODEL ARCHITECTURE
# ==========================================================================
===

class LSTMTextGenerator(nn.Module):
    """LSTM-based character-level text generation model"""

    def __init__(self, vocab_size, embedding_dim=128, hidden_dim=256,
                 num_layers=2, dropout=0.5):
        """
        Args:
            vocab_size: Number of unique characters
            embedding_dim: Dimension of character embeddings
            hidden_dim: Dimension of LSTM hidden state
            num_layers: Number of stacked LSTM layers
            dropout: Dropout rate between LSTM layers
        """
        super(LSTMTextGenerator, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            dropout=dropout if num_layers > 1 else 0.0,
            batch_first=True
        )

        self.fc = nn.Linear(hidden_dim, vocab_size)
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

    def forward(self, x, hidden=None):
        """
        Args:
            x: Input tensor of shape (batch_size, seq_length)
            hidden: Tuple of (hidden_state, cell_state) or None

        Returns:
            logits: Output logits of shape (batch_size, seq_length, vocab_size)
            hidden: Updated (hidden_state, cell_state)
        """
```

```
        # Embedding: (batch, seq_len) → (batch, seq_len, embed_dim)
        embedded = self.embedding(x)

        # LSTM: (batch, seq_len, embed_dim) → (batch, seq_len, hidden_dim)
        lstm_out, hidden = self.lstm(embedded, hidden)

        # Linear: (batch, seq_len, hidden_dim) → (batch, seq_len, vocab_size)
        logits = self.fc(lstm_out)

        return logits, hidden


# ============================================================================
===
# STEP 3: TRAINING LOOP
# ============================================================================
===

def train_epoch(model, train_loader, criterion, optimizer, device):
    """Train for one epoch"""
    model.train()
    total_loss = 0

    for batch_idx, (x, y) in enumerate(train_loader):
        x, y = x.to(device), y.to(device)

        # Forward pass
        logits, _ = model(x)  # (batch, seq_len, vocab_size)

        # Compute loss: reshape for CrossEntropyLoss
        # CrossEntropyLoss expects (N, C) where N = batch*seq_len, C = vocab_size
        loss = criterion(
            logits.view(-1, logits.size(-1)),  # (batch*seq_len, vocab_size)
            y.view(-1)                         # (batch*seq_len,)
        )

        # Backward pass
        optimizer.zero_grad()
        loss.backward()

        # Optional: gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
```

```
    return avg_loss


def train(model, train_loader, num_epochs, learning_rate, device='cpu'):
    """Train the LSTM model"""
    model.to(device)

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

    for epoch in range(num_epochs):
        loss = train_epoch(model, train_loader, criterion, optimizer, device)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss:.4f}")


# ==========================================================================
===
# STEP 4: TEXT GENERATION
# ==========================================================================
===

def generate_text(model, dataset, seed_text, length, temperature=1.0,
                  device='cpu'):
    """
    Generate text starting from seed_text

    Args:
        model: Trained LSTM model
        dataset: TextDataset instance (for encoding/decoding)
        seed_text: Starting text
        length: Number of characters to generate
        temperature: Controls randomness
                    - Low (0.5): More deterministic
                    - High (1.5): More random
        device: 'cpu' or 'cuda'

    Returns:
        Generated text string
    """
    model.eval()

    # Convert seed to indices
    indices = [dataset.char_to_idx[c] for c in seed_text]

    with torch.no_grad():
        for _ in range(length):
            # Use last seq_length characters as context
```

```
        if len(indices) >= dataset.seq_length:
            x = torch.tensor(
                indices[-dataset.seq_length:],
                dtype=torch.long
            ).unsqueeze(0).to(device)
        else:
            x = torch.tensor(
                indices,
                dtype=torch.long
            ).unsqueeze(0).to(device)

        # Get model prediction
        logits, _ = model(x)

        # Get logits for next character (last position in sequence)
        next_logits = logits[0, -1, :] / temperature

        # Apply softmax and sample
        probs = torch.softmax(next_logits, dim=0).cpu().numpy()
        next_idx = np.random.choice(len(probs), p=probs)

        indices.append(next_idx)

    return dataset.decode(indices)


# ==========================================================================
===
# STEP 5: COMPLETE USAGE EXAMPLE
# ==========================================================================
===

if __name__ == "__main__":
    # Configuration
    TEXT_FILE = "theprince.txt"  # Your text file
    SEQ_LENGTH = 50
    BATCH_SIZE = 32
    EMBEDDING_DIM = 128
    HIDDEN_DIM = 256
    NUM_LAYERS = 2
    DROPOUT = 0.3
    LEARNING_RATE = 0.001
    NUM_EPOCHS = 50
    DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    print(f"Using device: {DEVICE}")
```

```
# ===== Load and Prepare Data =====
# Load your text file
with open(TEXT_FILE, 'r', encoding='utf-8') as f:
    text = f.read().lower()  # Lowercase for consistency

print(f"Loaded {len(text)} characters")

# Create dataset and dataloader
dataset = TextDataset(text, seq_length=SEQ_LENGTH)
train_loader = DataLoader(
    dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    pin_memory=True if DEVICE.type == 'cuda' else False
)

print(f"Vocab size: {len(dataset.chars)}")
print(f"Dataset size: {len(dataset)} sequences")

# ===== Create Model =====
model = LSTMTextGenerator(
    vocab_size=len(dataset.chars),
    embedding_dim=EMBEDDING_DIM,
    hidden_dim=HIDDEN_DIM,
    num_layers=NUM_LAYERS,
    dropout=DROPOUT
)

print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

# ===== Train =====
train(
    model,
    train_loader,
    num_epochs=NUM_EPOCHS,
    learning_rate=LEARNING_RATE,
    device=DEVICE
)

# ===== Generate Text =====
seed = "the great"
print(f"\nGenerating text starting with: '{seed}'")

for temperature in [0.5, 1.0, 1.5]:
    print(f"\nTemperature: {temperature}")
    generated = generate_text(
        model,
```

```
        dataset,
        seed,
        length=200,
        temperature=temperature,
        device=DEVICE
    )
    print(generated)

# ===== Save Model =====
torch.save(model.state_dict(), 'lstm_model.pt')
print("\nModel saved to 'lstm_model.pt'")
```

- Evaluation

```
"""
LSTM TEXT GENERATION EVALUATION TOOLKIT
Compute metrics to analyze generated text quality
"""

import numpy as np
from collections import Counter
from typing import Dict, List, Tuple
import math


# ==========================================================================
===
# BASIC TEXT METRICS
# ==========================================================================
===

class TextMetrics:
    """Compute various metrics on text"""

    @staticmethod
    def distinctness(text: str) → float:
        """
        Compute distinctness (Type-Token Ratio)

        Range: 0-1
        Higher = more diverse vocabulary
        Lower = more repetitive

        Args:
            text: Generated text

        Returns:
```

```
        Distinctness score
    """
    words = text.lower().split()
    if len(words) == 0:
        return 0.0

    unique_words = len(set(words))
    total_words = len(words)

    return unique_words / total_words

@staticmethod
def repetition_ratio(text: str) → float:
    """
    Count adjacent repeated words

    Range: 0-1
    Higher = more repetitive (bad)
    Lower = more diverse (good)

    Example:
        "the the the and the" → repetition_ratio = 0.75
    """
    words = text.lower().split()
    if len(words) < 2:
        return 0.0

    repetitions = sum(1 for i in range(len(words) - 1) if words[i] == words[i + 1])
    return repetitions / (len(words) - 1)

@staticmethod
def average_word_length(text: str) → float:
    """Average length of words in characters"""
    words = text.split()
    if len(words) == 0:
        return 0.0
    return np.mean([len(w) for w in words])

@staticmethod
def sentence_count(text: str) → int:
    """Count sentences (periods, exclamation, question marks)"""
    return text.count('.') + text.count('!') + text.count('?')

@staticmethod
def average_sentence_length(text: str) → float:
    """Average words per sentence"""
    sentences = text.split('.')
```

```
        words_per_sentence = [len(s.split()) for s in sentences if s.strip()]

        if len(words_per_sentence) == 0:
            return 0.0

        return np.mean(words_per_sentence)

    @staticmethod
    def vocabulary_size(text: str) → int:
        """Number of unique words"""
        return len(set(text.lower().split()))


# ===================================================================================
===
# ADVANCED METRICS
# ===================================================================================
===

class AdvancedMetrics:
    """More sophisticated metrics for generation quality"""

    @staticmethod
    def perplexity_from_loss(loss: float) → float:
        """
        Convert cross-entropy loss to perplexity

        Perplexity = e^loss

        Args:
            loss: Cross-entropy loss value

        Returns:
            Perplexity score
        """
        return math.exp(loss)

    @staticmethod
    def self_bleu_score(generated_list: List[str], n_gram: int = 1) → float:
        """
        Compute SELF-BLEU (diversity among multiple generations)

        Measures: How different are multiple outputs from same seed?
        Lower = more diverse (good for creativity)
        Higher = less diverse (indicates overfitting)

        Args:
```

```
        generated_list: List of generated texts from same seed
        n_gram: N-gram size (1=unigrams, 2=bigrams)

    Returns:
        SELF-BLEU score (0-1, lower is better)
    """
    if len(generated_list) < 2:
        return 0.0

    def get_ngrams(text, n):
        words = text.lower().split()
        return set(tuple(words[i:i+n]) for i in range(len(words) - n + 1))

    # Compare each pair
    similarities = []
    for i in range(len(generated_list)):
        ngrams_i = get_ngrams(generated_list[i], n_gram)

        for j in range(i + 1, len(generated_list)):
            ngrams_j = get_ngrams(generated_list[j], n_gram)

            if len(ngrams_i) == 0 or len(ngrams_j) == 0:
                similarity = 0.0
            else:
                intersection = len(ngrams_i & ngrams_j)
                union = len(ngrams_i | ngrams_j)
                similarity = intersection / union if union > 0 else 0.0

            similarities.append(similarity)

    return np.mean(similarities) if similarities else 0.0

@staticmethod
def vocabulary_coverage(generated_text: str, reference_text: str) -> float:
    """
    Compute: What fraction of unique reference words appear in generated text?

    Range: 0-1
    Higher = model uses similar vocabulary

    Args:
        generated_text: Model output
        reference_text: Training data or reference

    Returns:
        Coverage score
    """
```

```
    gen_vocab = set(generated_text.lower().split())
    ref_vocab = set(reference_text.lower().split())

    if len(ref_vocab) == 0:
        return 0.0

    overlap = len(gen_vocab & ref_vocab)
    return overlap / len(ref_vocab)

@staticmethod
def entropy_score(text: str) → float:
    """
    Compute entropy of word distribution

    Range: 0 to infinity
    Higher = more diverse (good for generation)
    Lower = repetitive (bad)

    Args:
        text: Generated text

    Returns:
        Entropy score
    """
    words = text.lower().split()
    if len(words) == 0:
        return 0.0

    word_freq = Counter(words)
    probabilities = np.array(list(word_freq.values())) / len(words)

    entropy = -np.sum(probabilities * np.log(probabilities))
    return entropy

@staticmethod
def type_token_ratio_sliding(text: str, window: int = 50) → float:
    """
    Compute average TTR over sliding windows

    Measures: Vocabulary richness throughout text

    Args:
        text: Generated text
        window: Words per window

    Returns:
        Average TTR
```

```
    """
    words = text.lower().split()

    if len(words) < window:
        return len(set(words)) / len(words)

    ttrs = []
    for i in range(len(words) - window + 1):
        window_words = words[i:i+window]
        ttr = len(set(window_words)) / len(window_words)
        ttrs.append(ttr)

    return np.mean(ttrs)


# ==============================================================================
===
# COMPREHENSIVE EVALUATION PIPELINE
# ==============================================================================
===

def evaluate_single_generation(
    text: str,
    seed: str = None,
    reference: str = None,
    temperature: float = 1.0
) → Dict:
    """
    Comprehensive evaluation of a single generated text

    Args:
        text: Generated text
        seed: Original seed text (optional)
        reference: Reference/training text (optional)
        temperature: Temperature used for generation

    Returns:
        Dictionary of all metrics
    """

    results = {
        'temperature': temperature,

        # Basic metrics
        'length_chars': len(text),
        'length_words': len(text.split()),
        'sentences': TextMetrics.sentence_count(text),
```

```
        # Vocabulary metrics
        'distinctness': TextMetrics.distinctness(text),
        'vocab_size': TextMetrics.vocabulary_size(text),
        'avg_word_length': TextMetrics.average_word_length(text),
        'repetition_ratio': TextMetrics.repetition_ratio(text),

        # Advanced metrics
        'entropy': AdvancedMetrics.entropy_score(text),
        'avg_sentence_length': TextMetrics.average_sentence_length(text),
        'ttr_sliding': AdvancedMetrics.type_token_ratio_sliding(text),
    }

    # Optional metrics if reference provided
    if reference:
        results['vocabulary_coverage'] = AdvancedMetrics.vocabulary_coverage(text, reference)

    return results

# ==========================================================================
===
# EXAMPLE USAGE
# ==========================================================================
===

if __name__ == "__main__":

    # Example texts
    text_low_quality = "the greatest difficulty.the other to maintain themselves."
    text_medium_quality = "the greatest becoming poor at any private person. he kept having followed a
bandon of the army in susperition than recognio, having been seen also assist the venetians, and to rem
ain into italy,[1] this city o"
    text_high_quality = "the great country who we fuit divides him, so that i any easily eled by bind in the
beginning was cansterfor thy battle, he remains at your discourable."

    print("EXAMPLE EVALUATION:\n")

    # Evaluate each
    for text, label in [
        (text_low_quality, "Low Quality (T=0.5)"),
        (text_medium_quality, "Medium Quality (T=1.0)"),
        (text_high_quality, "High Quality (trained model)"),
    ]:
        print(f"\n{label}")
        print(f"Text: {text[:60]}...")
        print(f"—" * 70)
```

```
        metrics = evaluate_single_generation(text, temperature=[0.5, 1.0, 1.5][[0, 1, 2][0]])

        print(f"  Length: {metrics['length_words']} words")
        print(f"  Distinctness: {metrics['distinctness']:.3f} (higher=better)")
        print(f"  Repetition: {metrics['repetition_ratio']:.3f} (lower=better)")
        print(f"  Entropy: {metrics['entropy']:.3f} (higher=more diverse)")

    print("\n" + "="*70)
    print("Use these metrics to evaluate your LSTM outputs!")
    print("="*70)
```

- Evaluation output

```
EXAMPLE EVALUATION:


Low Quality (T=0.5)
Text: the greatest difficulty.the other to maintain themselves....
--------------------------------------------------------------------------------
—
  Length: 7 words
  Distinctness: 1.000 (higher=better)
  Repetition: 0.000 (lower=better)
  Entropy: 1.946 (higher=more diverse)

Medium Quality (T=1.0)
Text: the greatest becoming poor at any private person. he kept ha...
--------------------------------------------------------------------------------
—
  Length: 36 words
  Distinctness: 0.917 (higher=better)
  Repetition: 0.000 (lower=better)
  Entropy: 3.453 (higher=more diverse)

High Quality (trained model)
Text: the great country who we fuit divides him, so that i any eas...
--------------------------------------------------------------------------------
—
  Length: 28 words
  Distinctness: 0.964 (higher=better)
  Repetition: 0.000 (lower=better)
  Entropy: 3.283 (higher=more diverse)

======================================================================
```

```
Use these metrics to evaluate your LSTM outputs!
============================================================================
```