char_llm_from_scratch

February 27, 2026

1 Character-Level Language Model from Scratch

A step-by-step implementation of a simple neural language model trained on *Frankenstein* by Mary Shelley.

We build everything from scratch using only NumPy — no PyTorch, no TensorFlow.

1.1 Cell 1: Imports

Import the libraries we'll need throughout the notebook.

[1]:
```
# Standard libraries
import math
import random
import numpy as np
```

1.2 Cell 2: Load the Raw Text

Read the full text of *Frankenstein* from a local file. We print the total length and a preview of the first 500 characters to confirm it loaded correctly.

[2]:
```
# Load the raw text file (download from Project Gutenberg if you don't have it)
with open("frankenstein.txt", "r", encoding="utf-8") as f:
    text = f.read()

print("Length:", len(text))
print(text[:500])
```

```
Length: 438806
The Project Gutenberg eBook of Frankenstein; or, the modern prometheus

This ebook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this ebook or online
at www.gutenberg.org. If you are not located in the United States,
you will have to check the laws of the country where you are located
before
```

1.3 Cell 3: Clean the Text (Remove Gutenberg Boilerplate)

Project Gutenberg texts include legal/licensing text at the start and end. We find the `*** START OF` and `*** END OF` markers and slice out only the actual story.

[3]:
```
start_marker = "*** START OF"  # Marker indicating where the story begins
end_marker = "*** END OF"      # Marker indicating where the story ends

start = text.find(start_marker)  # Find start index of marker
end = text.find(end_marker)      # Find end index of marker

# If both markers exist, slice the text to only include the story
if start != -1 and end != -1:
    text = text[start + len(start_marker):end].strip()  # +len(...) skips the␣
 ↪marker itself

# Print cleaned text length and first 500 characters
print("Cleaned text length:", len(text))
print(text[:500])
```

```
Cleaned text length: 419409
THE PROJECT GUTENBERG EBOOK FRANKENSTEIN; OR, THE MODERN PROMETHEUS ***

Frankenstein;

or, the Modern Prometheus

by Mary Wollstonecraft (Godwin) Shelley


 CONTENTS

 Letter 1
 Letter 2
 Letter 3
 Letter 4
 Chapter 1
 Chapter 2
 Chapter 3
 Chapter 4
 Chapter 5
 Chapter 6
 Chapter 7
 Chapter 8
 Chapter 9
 Chapter 10
 Chapter 11
 Chapter 12
```

```
Chapter 13
Chapter 14
Chapter 15
Chapter 16
Chapter 17
Chapter 18
Chapter 19
Chapter 20
Chapter 21
Chapter 22
Chapter 23
Chapter 24



Letter 1
```

1.4 Cell 4: Build the Character Vocabulary

We identify all unique characters in the text and create two mappings: - `stoi` (string-to-integer): maps each character to a unique integer ID - `itos` (integer-to-string): maps each integer ID back to its character

These mappings are how we convert between human-readable text and numeric tokens the model can process.

[4]:
```python
# Identify all unique characters in the text
# 'set(text)' returns only unique characters
# 'sorted(...)' ensures a consistent, reproducible order
chars = sorted(list(set(text)))

# Total number of unique characters (this is our vocabulary size)
vocab_size = len(chars)

# Mapping from character -> integer index
stoi = {ch: i for i, ch in enumerate(chars)}

# Mapping from integer index -> character
itos = {i: ch for i, ch in enumerate(chars)}

# Inspect vocabulary
print("Vocab size:", vocab_size)
print("All unique characters:", chars)
```

```
Vocab size: 84
All unique characters: ['\n', ' ', '!', '(', ')', '*', ',', '-', '.', '0', '1',
'2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '?', 'A', 'B', 'C', 'D', 'E',
```

```
'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'R', 'S', 'T', 'U', 'V',
'W', 'Y', '[', ']', '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'æ',
'è', 'é', 'ê', 'ô', '-', '‘', '’', '"', '"']
```

1.5 Cell 5: Encode / Decode Helper Functions

Define two helper functions: - `encode(s)`: converts a string into a list of integer token IDs - `decode(l)`: converts a list of integer token IDs back into a string

These are the tokenizer for our character-level model.

[5]:
```python
def encode(s):
    """Convert a string into a list of integer token IDs"""
    return [stoi[c] for c in s]

def decode(l):
    """Convert a list of token IDs back into a string"""
    return ''.join([itos[i] for i in l])
```

1.6 Cell 6: Print the Character-to-ID Mappings

Visualize both mappings so we can see exactly how characters are encoded.

[6]:
```python
# Print stoi mapping (character -> ID)
print("stoi mapping:")
print(", ".join([f"{repr(ch)}:{i}" for ch, i in stoi.items()]))

print()

# Print itos mapping (ID -> character)
print("itos mapping:")
print(", ".join([f"{repr(ch)}:{i}" for ch, i in itos.items()]))
```

```
stoi mapping:
'\n':0, ' ':1, '!':2, '(':3, ')':4, '*':5, ',':6, '-':7, '.':8, '0':9, '1':10,
'2':11, '3':12, '4':13, '5':14, '6':15, '7':16, '8':17, '9':18, ':':19, ';':20,
'?':21, 'A':22, 'B':23, 'C':24, 'D':25, 'E':26, 'F':27, 'G':28, 'H':29, 'I':30,
'J':31, 'K':32, 'L':33, 'M':34, 'N':35, 'O':36, 'P':37, 'R':38, 'S':39, 'T':40,
'U':41, 'V':42, 'W':43, 'Y':44, '[':45, ']':46, '_':47, 'a':48, 'b':49, 'c':50,
'd':51, 'e':52, 'f':53, 'g':54, 'h':55, 'i':56, 'j':57, 'k':58, 'l':59, 'm':60,
'n':61, 'o':62, 'p':63, 'q':64, 'r':65, 's':66, 't':67, 'u':68, 'v':69, 'w':70,
'x':71, 'y':72, 'z':73, 'æ':74, 'è':75, 'é':76, 'ê':77, 'ô':78, '-':79, '‘':80,
'’':81, '"':82, '"':83

itos mapping:
0:
, 1: , 2:!, 3:(, 4:), 5:*, 6:,, 7:-, 8:., 9:0, 10:1, 11:2, 12:3, 13:4, 14:5,
15:6, 16:7, 17:8, 18:9, 19::, 20:;, 21:?, 22:A, 23:B, 24:C, 25:D, 26:E, 27:F,
```

```
28:G, 29:H, 30:I, 31:J, 32:K, 33:L, 34:M, 35:N, 36:O, 37:P, 38:R, 39:S, 40:T,
41:U, 42:V, 43:W, 44:Y, 45:[, 46:], 47:_, 48:a, 49:b, 50:c, 51:d, 52:e, 53:f,
54:g, 55:h, 56:i, 57:j, 58:k, 59:l, 60:m, 61:n, 62:o, 63:p, 64:q, 65:r, 66:s,
67:t, 68:u, 69:v, 70:w, 71:x, 72:y, 73:z, 74:æ, 75:è, 76:é, 77:ê, 78:ô, 79:-,
80:‘, 81:’, 82:", 83:"
```

1.7 Cell 7: Encode the Entire Text

Convert the full cleaned text into a list of integer token IDs. Then verify by decoding a small sample back to text.

[7]:
```python
# Encode the entire text into a list of integers
encoded_text = encode(text)

# Sanity check: print first 20 token IDs
print("First 20 token IDs:", encoded_text[:20])

# Decode them back to verify correctness
decoded_sample = decode(encoded_text[:20])
print("Decoded text:", repr(decoded_sample))
```

```
First 20 token IDs: [40, 29, 26, 1, 37, 38, 36, 31, 26, 24, 40, 1, 28, 41, 40,
26, 35, 23, 26, 38]
Decoded text: 'THE PROJECT GUTENBER'
```

1.8 Cell 8: Visualize Token ID Character Pairs

For the first 20 tokens, print each ID alongside its decoded character. This helps build intuition for how the encoding works.

[8]:
```python
# Create a list of tuples: (ID, character)
id_char_pairs = [(i, itos[i]) for i in encoded_text[:20]]

# Print neatly
for idx, char in id_char_pairs:
    print(f"ID {idx:2} -> {repr(char)}")

print()
# Also print as a compact inline view
print(" | ".join([f"{i}:{repr(itos[i])}" for i in encoded_text[:20]]))
```

```
ID 40 -> 'T'
ID 29 -> 'H'
ID 26 -> 'E'
ID  1 -> ' '
ID 37 -> 'P'
ID 38 -> 'R'
ID 36 -> 'O'
ID 31 -> 'J'
```

```
ID 26 -> 'E'
ID 24 -> 'C'
ID 40 -> 'T'
ID  1 -> ' '
ID 28 -> 'G'
ID 41 -> 'U'
ID 40 -> 'T'
ID 26 -> 'E'
ID 35 -> 'N'
ID 23 -> 'B'
ID 26 -> 'E'
ID 38 -> 'R'

40:'T' | 29:'H' | 26:'E' | 1:' ' | 37:'P' | 38:'R' | 36:'O' | 31:'J' | 26:'E' |
24:'C' | 40:'T' | 1:' ' | 28:'G' | 41:'U' | 40:'T' | 26:'E' | 35:'N' | 23:'B' |
26:'E' | 38:'R'
```

1.9 Cell 9: Create Input-Target Sequences

This is where we create training data for the model.

Concept: Given a window of `block_size` characters (the "context"), the model's job is to predict the very next character (the "target").

We slide this window across the entire text to produce thousands of (input, target) pairs.

Example with block_size=8: - Input: `"Chapter "` → Target: `"1"` - Input: `"hapter 1"` → Target: `"\n"`

[9]:
```python
# block_size = number of previous characters the model sees (context window)
block_size = 8  # small context window for a toy model

# Prepare empty lists to store sequences
x_manual = []  # input sequences (lists of integers)
y_manual = []  # target characters (next character after input)

# Loop over the encoded text to create sequences
# We stop at len(encoded_text) - block_size to ensure each input has exactly␣
↪block_size chars
for i in range(len(encoded_text) - block_size):
    # Slice the encoded text from i to i+block_size → the "context" the model␣
↪sees
    input_seq = encoded_text[i:i+block_size]
    x_manual.append(input_seq)

    # The target is the very next character (token ID) after the input sequence
    target = encoded_text[i + block_size]
    y_manual.append(target)
```

```
print(f"Total training examples: {len(x_manual)}")
```

```
Total training examples: 419401
```

1.10 Cell 10: Inspect Training Examples

Print the first 5 input-target pairs to verify our data preparation is correct.

[10]:
```
for i in range(5):
    print(f"Example {i+1}")
    print("Input IDs:     ", x_manual[i])          # token IDs of input
    print("Decoded Input: ", decode(x_manual[i]))  # convert IDs back to␣
 ↪characters
    print("Target ID:     ", y_manual[i])          # token ID of target
    print("Decoded Target:", itos[y_manual[i]])    # convert ID to character
    print("---")
```

```
Example 1
Input IDs:      [40, 29, 26, 1, 37, 38, 36, 31]
Decoded Input:  THE PROJ
Target ID:      26
Decoded Target: E
---
Example 2
Input IDs:      [29, 26, 1, 37, 38, 36, 31, 26]
Decoded Input:  HE PROJE
Target ID:      24
Decoded Target: C
---
Example 3
Input IDs:      [26, 1, 37, 38, 36, 31, 26, 24]
Decoded Input:  E PROJEC
Target ID:      40
Decoded Target: T
---
Example 4
Input IDs:      [1, 37, 38, 36, 31, 26, 24, 40]
Decoded Input:   PROJECT
Target ID:      1
Decoded Target:
---
Example 5
Input IDs:      [37, 38, 36, 31, 26, 24, 40, 1]
Decoded Input:  PROJECT
Target ID:      28
Decoded Target: G
---
```

1.11 Cell 11: Initialize Model Parameters

We define a simple feedforward neural network with: 1. **Embedding layer** (`W_embed`): Converts each token ID into a dense vector of size `embedding_dim` 2. **Hidden layer** (`W1`, `b1`): Takes the flattened concatenation of all embeddings and maps to `hidden_dim` neurons with tanh activation 3. **Output layer** (`W2`, `b2`): Maps hidden state to logits over the vocabulary (one score per character)

All weights are initialized with small random values; biases start at zero.

[11]:
```
# Hyperparameters for our toy model
vocab_size = len(chars)     # number of unique characters
embedding_dim = 16          # size of each character embedding vector
hidden_dim = 32             # size of hidden layer
block_size = 8              # context length (number of previous characters)

# Embedding matrix: maps token IDs -> dense vectors
# Shape: vocab_size x embedding_dim
W_embed = np.random.randn(vocab_size, embedding_dim) * 0.01

# Hidden layer weights: flattened embeddings -> hidden_dim
# Input size = block_size * embedding_dim (all embeddings concatenated)
W1 = np.random.randn(block_size * embedding_dim, hidden_dim) * 0.01
b1 = np.zeros(hidden_dim)  # bias for hidden layer

# Output layer: hidden_dim -> vocab_size logits
W2 = np.random.randn(hidden_dim, vocab_size) * 0.01
b2 = np.zeros(vocab_size)  # bias for output layer

# Sanity check: print shapes
print("Embedding matrix shape:", W_embed.shape)
print("Hidden layer weight shape:", W1.shape)
print("Output layer weight shape:", W2.shape)
```

```
Embedding matrix shape: (84, 16)
Hidden layer weight shape: (128, 32)
Output layer weight shape: (32, 84)
```

1.12 Cell 12: Softmax and Forward Pass Functions

Softmax: Converts raw logits into a valid probability distribution (all positive, sums to 1). We subtract the max for numerical stability to prevent overflow in `exp()`.

Forward pass: The full pipeline from input token IDs to output probabilities: 1. Look up embeddings for each token 2. Flatten all embeddings into one long vector 3. Pass through hidden layer with tanh activation 4. Compute output logits 5. Apply softmax to get probabilities

[12]:
```
def softmax(x):
    """Compute softmax probabilities for a 1D array.
    Subtracting max(x) prevents numerical overflow."""
    e_x = np.exp(x - np.max(x))
```

```
    return e_x / e_x.sum()


def forward(x_seq):
    """
    Forward pass for a single input sequence.

    Args:
        x_seq: list of token IDs (length = block_size)

    Returns:
        probs: probability distribution over vocab for the next character
    """
    # Step 1: Lookup embeddings for each token ID
    embeds = W_embed[x_seq]         # shape: block_size x embedding_dim

    # Step 2: Flatten embeddings into a single vector
    h_input = embeds.flatten()      # shape: block_size * embedding_dim

    # Step 3: Hidden layer with tanh activation
    h = np.tanh(h_input @ W1 + b1) # shape: hidden_dim

    # Step 4: Output layer -> logits (one score per vocab character)
    logits = h @ W2 + b2            # shape: vocab_size

    # Step 5: Softmax -> probabilities
    probs = softmax(logits)

    return probs
```

1.13 Cell 13: Test the Forward Pass (Before Training)

Run the model on the first training example to see what it predicts *before* any training. Since weights are random, the prediction will be essentially random too.

[13]:
```
# Take first input sequence and its target
x0 = x_manual[0]
y0 = y_manual[0]

# Forward pass: get predicted probability distribution
probs = forward(x0)

# Predicted character = highest probability
pred_id = np.argmax(probs)

# Print input, target, and prediction
print("Input sequence (decoded):", decode(x0))
```

```
print("Target character:", itos[y0])
print("Predicted character (before training):", itos[pred_id])

# Show probabilities of top 5 characters
top5 = np.argsort(probs)[-5:][::-1]
print("\nTop 5 predictions:")
for i in top5:
    print(f"  {repr(itos[i])}: {probs[i]:.4f}")
```

```
Input sequence (decoded): THE PROJ
Target character: E
Predicted character (before training): 0

Top 5 predictions:
  '0': 0.0119
  'x': 0.0119
  '4': 0.0119
  '1': 0.0119
  'é': 0.0119
```

1.14 Cell 14: Cross-Entropy Loss Function

The loss function measures how far off the model's predictions are from the true answer.

Cross-entropy loss = `-log(predicted probability of the correct character)` - If the model assigns probability 1.0 to the correct character → loss = 0 (perfect) - If the model assigns probability 0.01 → loss 4.6 (very bad)

We add a tiny epsilon (1e-9) to prevent `log(0)` which would be -infinity.

[14]:
```
def cross_entropy_loss(probs, target_id):
    """
    Compute cross-entropy loss for a single prediction.

    Args:
        probs: probability distribution over vocab (output of forward pass)
        target_id: integer ID of the correct next character

    Returns:
        Scalar loss value
    """
    return -np.log(probs[target_id] + 1e-9)
```

1.15 Cell 15: Training Loop (First Pass — Small Model)

Train the model using **stochastic gradient descent (SGD)** with manual backpropagation.

For each training example: 1. **Forward pass**: Compute predictions 2. **Compute loss**: Cross-entropy between prediction and target 3. **Backpropagation**: Compute gradients of loss w.r.t. every weight 4. **Update weights**: Nudge each weight in the direction that reduces loss

We also generate sample text periodically to see the model's progress.

[15]:
```
# Hyperparameters
learning_rate = 0.1       # step size for gradient descent
num_epochs = 5            # number of passes over the dataset

# Training loop
for epoch in range(num_epochs):
    total_loss = 0

    # Loop over each input-target pair
    for i in range(len(x_manual)):
        x_seq = x_manual[i]    # input sequence of token IDs
        y_true = y_manual[i]   # target token ID

        # ---- Forward Pass ----
        embeds = W_embed[x_seq]                  # Lookup embeddings:␣
 ↪block_size x embedding_dim
        h_input = embeds.flatten()               # Flatten to single vector
        h = np.tanh(h_input @ W1 + b1)           # Hidden layer with tanh␣
 ↪activation
        logits = h @ W2 + b2                     # Output logits
        probs = softmax(logits)                  # Probabilities over vocab

        # ---- Compute Loss (Cross-Entropy) ----
        loss = cross_entropy_loss(probs, y_true)
        total_loss += loss

        # ---- Backpropagation ----
        # Gradient of loss w.r.t logits (softmax + cross-entropy combined)
        dlogits = probs.copy()
        dlogits[y_true] -= 1  # subtract 1 at the true class

        # Gradients for output layer
        dW2 = np.outer(h, dlogits)   # hidden_dim x vocab_size
        db2 = dlogits                # vocab_size

        # Gradients for hidden layer
        dh = dlogits @ W2.T                      # propagate gradient back
        dh_raw = dh * (1 - h**2)                 # tanh derivative: 1 - tanh(x)^2

        dW1 = np.outer(h_input, dh_raw)          # input_dim x hidden_dim
        db1 = dh_raw                             # hidden_dim

        # Gradients for embeddings
        dembed_flat = dh_raw @ W1.T              # block_size * embedding_dim
        dembed = dembed_flat.reshape(block_size, embedding_dim)
```

```
        dW_embed = np.zeros_like(W_embed)
        for j, idx in enumerate(x_seq):
            dW_embed[idx] += dembed[j]          # accumulate grads for repeated␣
↪tokens

        # ---- Update Weights (SGD) ----
        W2 -= learning_rate * dW2
        b2 -= learning_rate * db2
        W1 -= learning_rate * dW1
        b1 -= learning_rate * db1
        W_embed -= learning_rate * dW_embed

    # End of epoch: print average loss and generate sample text
    avg_loss = total_loss / len(x_manual)
    print(f"Epoch {epoch+1}/{num_epochs}, Average Loss: {avg_loss:.4f}")

    # Generate a sample of 50 characters
    start_idx = np.random.randint(0, len(x_manual))
    generated_seq = x_manual[start_idx].copy()
    print("Sample start:", decode(generated_seq))

    for _ in range(50):
        probs = forward(generated_seq[-block_size:])
        next_id = np.random.choice(len(probs), p=probs)
        generated_seq.append(next_id)

    print("Generated text:", decode(generated_seq))
    print("---")
```

```
Epoch 1/5, Average Loss: 3.5390
Sample start: ished he
Generated text: ished hed bhosm
FMongroskbree badtel.e ase andtwpos thosse
---
Epoch 2/5, Average Loss: 3.5576
Sample start: ecution
Generated text: ecution
an wyssalttheqn anexlinb Ht acdentannd. ind doses
---
Epoch 3/5, Average Loss: 3.5335
Sample start: vouring
Generated text: vouring lantandw! an wast th SH
ann.ensons ,syekd Bnstinkw
---
Epoch 4/5, Average Loss: 3.5533
Sample start: ith bitt
Generated text: ith bitterttind andtan
,awtan.
```

```
ans
lastancs an ly tantangt
---
Epoch 5/5, Average Loss: 3.5455
Sample start: t, then,
Generated text: t, then, oaes annaajs anerl ln cas I ankth in asrant lnyam
---
```

1.16 Cell 16: Scale Up — Larger Model with Longer Context

Now we increase the model capacity: - `block_size`: 8 → 16 (sees more context) - `embedding_dim`: 16 → 32 (richer character representations) - `hidden_dim`: 32 → 64 (more expressive hidden layer)

We also need to **rebuild the training data** with the new block_size, and reinitialize all weights to match the new dimensions.

[16]:
```
# New hyperparameters
block_size = 16
embedding_dim = 32
hidden_dim = 64
vocab_size = len(chars)

# Rebuild training data with new block_size
x_manual = []
y_manual = []
for i in range(len(encoded_text) - block_size):
    x_manual.append(encoded_text[i:i+block_size])
    y_manual.append(encoded_text[i + block_size])

print(f"Training examples: {len(x_manual)} (with block_size={block_size})")

# Reinitialize weights with proper dimensions
np.random.seed(42)
W_embed = np.random.randn(vocab_size, embedding_dim) * 0.01
W1 = np.random.randn(block_size * embedding_dim, hidden_dim) * 0.01  # 16*32 =
 ↪512 inputs
b1 = np.zeros(hidden_dim)
W2 = np.random.randn(hidden_dim, vocab_size) * 0.01
b2 = np.zeros(vocab_size)

print(f"W_embed: {W_embed.shape}, W1: {W1.shape}, W2: {W2.shape}")
```

```
Training examples: 419393 (with block_size=16)
W_embed: (84, 32), W1: (512, 64), W2: (64, 84)
```

1.17 Cell 17: Text Generation Function

A reusable function to generate text from the trained model.

Key generation controls: - **Temperature**: Controls randomness. Lower = more conservative (picks

common characters), Higher = more creative/risky - **Top-k filtering**: Only sample from the top k most likely characters, zeroing out the rest. Prevents very unlikely characters from being chosen.

[17]:
```python
def generate_text(start_seq, length=200, temperature=0.8, top_k=5):
    """Generate text from a starting sequence using the current model.

    Args:
        start_seq: list of token IDs to start from (at least block_size long)
        length: number of characters to generate
        temperature: controls randomness (lower = more deterministic)
        top_k: only sample from top k most likely characters

    Returns:
        Full generated sequence (start + generated) as list of token IDs
    """
    generated = start_seq.copy()

    for _ in range(length):
        # Take the last block_size tokens as context
        context = generated[-block_size:]

        # Forward pass
        embeds = W_embed[context].flatten()
        h = np.tanh(embeds @ W1 + b1)
        logits = h @ W2 + b2
        probs = np.exp(logits) / np.sum(np.exp(logits))  # softmax

        # Apply temperature scaling (raise probs to power 1/T, then renormalize)
        probs = probs ** (1 / temperature)
        probs /= probs.sum()

        # Apply top-k filtering: keep only top k probabilities
        if top_k is not None:
            top_idx = np.argsort(probs)[-top_k:]
            mask = np.zeros_like(probs)
            mask[top_idx] = probs[top_idx]
            probs = mask / mask.sum()

        # Sample the next token from the probability distribution
        next_id = np.random.choice(len(probs), p=probs)
        generated.append(next_id)

    return generated
```

1.18 Cell 18: Train the Larger Model (20 Epochs)

Train the scaled-up model for 20 epochs with the same SGD + backpropagation approach. Every 5 epochs, we generate a 200-character sample to monitor quality.

Note: This will take a while since we're training on the full text with pure NumPy (no GPU). The loss should decrease over epochs, and generated text should become more English-like.

[18]:
```
learning_rate = 0.1
num_epochs = 20

for epoch in range(num_epochs):
    total_loss = 0

    for i in range(len(x_manual)):
        x_seq = x_manual[i]
        y_true = y_manual[i]

        # ---- Forward pass ----
        embeds = W_embed[x_seq].flatten()
        h = np.tanh(embeds @ W1 + b1)
        logits = h @ W2 + b2
        probs = np.exp(logits) / np.sum(np.exp(logits))  # softmax

        # ---- Loss ----
        loss = -np.log(probs[y_true] + 1e-9)
        total_loss += loss

        # ---- Backpropagation ----
        dlogits = probs.copy()
        dlogits[y_true] -= 1

        dW2 = np.outer(h, dlogits)
        db2 = dlogits

        dh = dlogits @ W2.T
        dh_raw = dh * (1 - h**2)  # tanh derivative

        dW1 = np.outer(embeds, dh_raw)
        db1 = dh_raw

        dembed_flat = dh_raw @ W1.T
        dembed = dembed_flat.reshape(block_size, embedding_dim)
        dW_embed = np.zeros_like(W_embed)
        for j, idx in enumerate(x_seq):
            dW_embed[idx] += dembed[j]

        # ---- Update weights (SGD) ----
        W2 -= learning_rate * dW2
        b2 -= learning_rate * db2
        W1 -= learning_rate * dW1
        b1 -= learning_rate * db1
```

```
        W_embed -= learning_rate * dW_embed

    # Print average loss for this epoch
    avg_loss = total_loss / len(x_manual)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")

    # Generate sample text every 5 epochs
    if (epoch + 1) % 5 == 0:
        start_idx = np.random.randint(0, len(x_manual))
        start_seq = x_manual[start_idx]
        generated_seq = generate_text(start_seq, length=200, temperature=0.8,␣
↪top_k=5)
        print("Sample generation:")
        print(decode(generated_seq))
        print("---")
```

```
Epoch 1/20, Loss: 5.3722
Epoch 2/20, Loss: 5.4459
Epoch 3/20, Loss: 5.4606
Epoch 4/20, Loss: 5.4705
Epoch 5/20, Loss: 5.4701
Sample generation:
hero, whose extian thi leytay
 lT wao was ano ens ano adn bn erdtpan wn  bo  aystaosihhir
e wasem bn thb vrs ane lns bls ai waat an who te  pau ansthagiwn ahae
thoe;cnn.lgn tls aistut.t ahl ans toct.tsia. wtes bhtt.a
---
Epoch 6/20, Loss: 5.4809
Epoch 7/20, Loss: 5.4786
Epoch 8/20, Loss: 5.4636
Epoch 9/20, Loss: 5.4820
Epoch 10/20, Loss: 5.4908
Sample generation:
he productions oe thslinte.ne wass yc wh
 wa sooeth
 whstaycewe shalebhenhw.n  eonst
i weseiardnh bn
 w st nk Hre ta t st stno a sdin saio emisdectesce se m stes
 wne he sase yhsonn wd sestan tn eristiltedf
nhaoh
teo
---
Epoch 11/20, Loss: 5.4929
Epoch 12/20, Loss: 5.5137
Epoch 13/20, Loss: 5.5134
Epoch 14/20, Loss: 5.4951
Epoch 15/20, Loss: 5.4865
Sample generation:
```

```
yourself. And yesededn.ank. cokere td
cl ttstee s des tesn nsr srd uoteatlwa
r nardlnse lodnalhfy
thtnresetld He tkdt shrnselirrs
.e and. sotnse.na
 blalethbcdetoses

not dndese to
cvnt dhedaaldnwneHn sert Ihrh
noaso
---
Epoch 16/20, Loss: 5.4720
Epoch 17/20, Loss: 5.4608
Epoch 18/20, Loss: 5.4522
Epoch 19/20, Loss: 5.4582
Epoch 20/20, Loss: 5.4574
Sample generation:
ent of a nervousn
 ca t ot nhet; nn
 faitrithdt dne weota cas
 thetw nn
nc cas
n ntewe ofv
 n t d apvn,ls.nd the nestli.n tee t strnse s sna ces wlst thende nm
caitpess.ntn thit inxmhas.tne n.
e bnasli
et seot inn
u
---
```

1.19 Cell 19: Debug — Verify Tensor Shapes

A quick sanity check to confirm all intermediate tensors have the expected shapes. This is essential for catching dimension mismatches that cause cryptic errors.

[19]:
```python
# Verify shapes through the forward pass
x_seq = x_manual[0]
embeds = W_embed[x_seq].flatten()
print("embeds.shape:", embeds.shape)      # Expected: (block_size *␣
 ↪embedding_dim,) = (512,)
print("W1.shape:", W1.shape)              # Expected: (512, 64)

h = np.tanh(embeds @ W1 + b1)
print("h.shape:", h.shape)                # Expected: (64,)

logits = h @ W2 + b2
print("logits.shape:", logits.shape)       # Expected: (vocab_size,)
print("\nAll shapes look correct!")
```

```
embeds.shape: (512,)
W1.shape: (512, 64)
h.shape: (64,)
logits.shape: (84,)

All shapes look correct!
```

1.20 Cell 20: Generate Final Sample

Use the fully trained model to generate a longer passage of text. Try different temperature and top_k values to see how they affect output quality.

[20]:
```python
# Pick a random starting context from the training data
start_idx = np.random.randint(0, len(x_manual))
start_seq = x_manual[start_idx]

print("=" * 60)
print("FINAL GENERATION (temperature=0.8, top_k=5)")
print("=" * 60)
generated = generate_text(start_seq, length=300, temperature=0.8, top_k=5)
print(decode(generated))

print("\n" + "=" * 60)
print("LOWER TEMPERATURE (temperature=0.5, top_k=5) - more conservative")
print("=" * 60)
generated = generate_text(start_seq, length=300, temperature=0.5, top_k=5)
print(decode(generated))

print("\n" + "=" * 60)
print("HIGHER TEMPERATURE (temperature=1.2, top_k=10) - more creative")
print("=" * 60)
generated = generate_text(start_seq, length=300, temperature=1.2, top_k=10)
print(decode(generated))
```

```
============================================================
FINAL GENERATION (temperature=0.8, top_k=5)
============================================================
foe. If I were an che
de sais ileentrnt o de ekitnn
n
otr tist tce
n Unane st spidtrlirovtheest ct tai
adhcanmrna f s adsesw nn
fe masoees.a,na dae woswln d d a seawcocrn
 y se the
we nadt n.vtnse s s iedstn..ndne them;e nt chispl neot sy sle nesthi.fidae t
sisoses.so sne t stan
e.
```

```
elit theest,n s d e wesnroct s l

============================================================
LOWER TEMPERATURE (temperature=0.5, top_k=5) - more conservative
============================================================
foe. If I were an chisastn nnd cnot in,s; cfe nest nonesisn nnase hnrn boatlruns
nn t slnesecnt n
 t son nesnc cf deest snr ciete snst nct ha
nsdce.
no tee to
thitrs
icnt ncst nlthd s s sistuwcrsone e so nae.n nn
nc c
a
 aneew rn
pc teaween.rh r s wese."n
o ne saeswese
do n
 tnese.s.rive tee.w bna nastawcrn
r naata

============================================================
HIGHER TEMPERATURE (temperature=1.2, top_k=10) - more creative
============================================================
foe. If I were a." indttrsfn
ne sa sae
wustr .ao libtdn.e.vnanm yle chrt no whd
n nerwerot in darisesnw..nl nide th Tn dnesfrhnrd uee wnitaosib
dene wa a nkess nnrncv.hok rne wlat bkrsteserdnv,a saacetsibn bn tedt,bhys miee
ine
n nnancicafoctht
si nyeelsus
 r itest,.!p n  maiweswsbn bhv se ceiSf"splo rhv ,pdeebs!n
```