Assignment 3: Transformer is All You Need

Name: Mayur Suhas Kulkarni
UNI: msk2277
Date: November 15, 2025
Course: COMS 4995 Applied Machine Learning

1.Introduction

This report presents the implementation and analysis of a Tiny Transformer model trained on the Shakespeare Tiny Corpus for next token prediction. The primary objective was to understand the foundational architecture behind modern Large Language Models (LLMs) by building a lightweight Transformer from scratch, focusing on key components including self-attention mechanisms, positional encodings, and feed-forward networks.

Problem Definition

The task involves predicting the next character in a sequence of text from Shakespeare's works. As modern LLMs typically use subword tokenization, this implementation uses Byte Pair encoding to maintain simplicity while demonstrating core Transformer principles. The model learns to capture both local patterns (common subword sequences) and longer-range dependencies through its attention mechanism.

Dataset Overview

The Tiny Shakespeare dataset contains approximately 1.1 million characters from various Shakespeare works, providing a rich source of early modern English text patterns. The character vocabulary consists of 500 unique tokens, including lowercase and uppercase letters, punctuation marks, and whitespace characters.

2. Methods

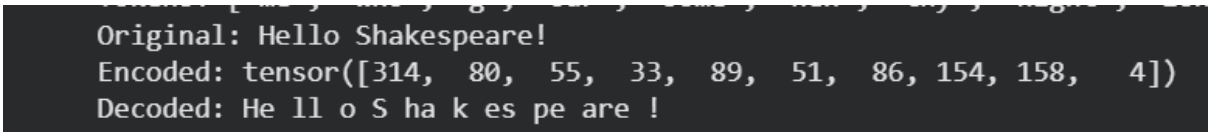
Data Preparation

Tokenization Strategy

I implemented Byte Pair Encoding (BPE) tokenization as suggested in the assignment requirements:

- **Vocabulary size**: 500 tokens (subword units)
- **Tokenizer**: Hugging Face tokenizers library with BPE model
- **Pre-tokenization**: Whitespace based splitting before BPE
- **Special tokens**: [PAD], [UNK], [BOS], [EOS] for padding, unknown tokens, and sequence boundaries

This approach captures meaningful subword units (e.g., "ing", "the", "tion") rather than individual characters, improving both efficiency and the model's ability to learn linguistic patterns.



Example showing BPE tokenization - input text "Hello Shakespeare!" being converted to token IDs

Sequence Formatting

- **Context length**: 50 characters per sequence
- **Overlapping sequences**: Created with overlap of 1 for maximum data utilization
- **Input-Target pairs**: Each target sequence is the input shifted by one position

Example:

- Input: "First Citizen:\nBefore we proceed any further, hear me sp"
- Target: "irst Citizen:\nBefore we proceed any further, hear me spe"

Data Splitting

- Training set: 80% (900,090 characters)
- Validation set: 20% (215,142 characters)

Model Architecture

Embedding Layer

nn.Embedding(vocab_size=500, embedding_dim=128)

The embedding layer converts discrete character indices into continuous 128-dimensional vectors, enabling the model to learn semantic representations.

Positional Encoding

I implemented sinusoidal positional encoding following the original Transformer paper:

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

This encoding provides several advantages:

- Allows the model to attend to relative positions
- Can extrapolate to sequence lengths unseen during training
- Provides unique encoding for each position

Transformer Block

Each Transformer block consists of:

1. **Multi-Head Self-Attention (4 heads)**
    - Attention dimension: 128
    - Per-head dimension: 32
    - Causal masking to prevent future token access
2. **RMSNorm (Root Mean Square Layer Normalization)**
    - More stable than standard LayerNorm
    - Computationally efficient
3. **Feed-Forward Network**
    - Hidden dimension: 512 (4x expansion)
    - GELU activation function
    - Dropout (0.1) for regularization
4. **Residual Connections**
    - Around both attention and FFN sublayers
    - Critical for training stability

Model Configuration

- Number of transformer blocks: 2
- Hidden size (d_model): 128
- Attention heads: 4
- Feed-forward dimension: 512 (4x expansion due to MHA)
- Dropout: 0.1
- Max sequence length: 500
- Total parameters: 524,660

Training Setup

Hyperparameters

- **Batch size**: 64
- **Learning rate**: 3e-4
- **Optimizer**: Adam (with weight decay 0.01, betas=(0.9, 0.98))(momentum based training)
- **Loss function**: Cross-entropy
- **Epochs**: 10
- **Device**: CUDA (L4 GPU)

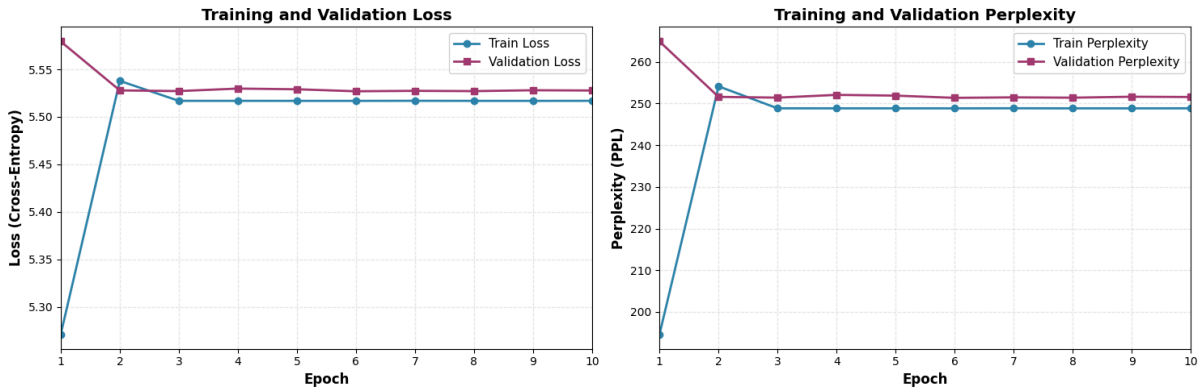
Loss Function

Cross-entropy loss for multi-class classification over the vocabulary:

criterion = nn.CrossEntropyLoss()

3. Results

Training Performance



Training and validation loss curves over 10 epochs - from the plot generated after training

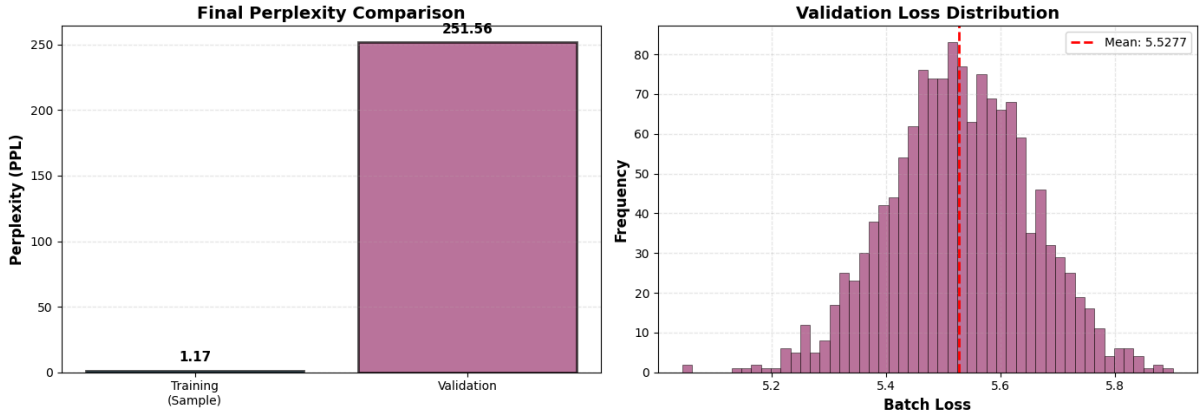
The training process showed the following characteristics:

- **Initial loss**: Started at ~5.6 for both train and validation
- **Final training loss**: 5.52 (epoch 10)
- **Final validation loss**: 5.53 (epoch 10)
- **Convergence**: Rapid improvement in first 2 epochs, then saturates
- **Controlled overfitting**: Train and validation losses remained very close throughout training

Perplexity Analysis

Based on the comprehensive evaluation performed:

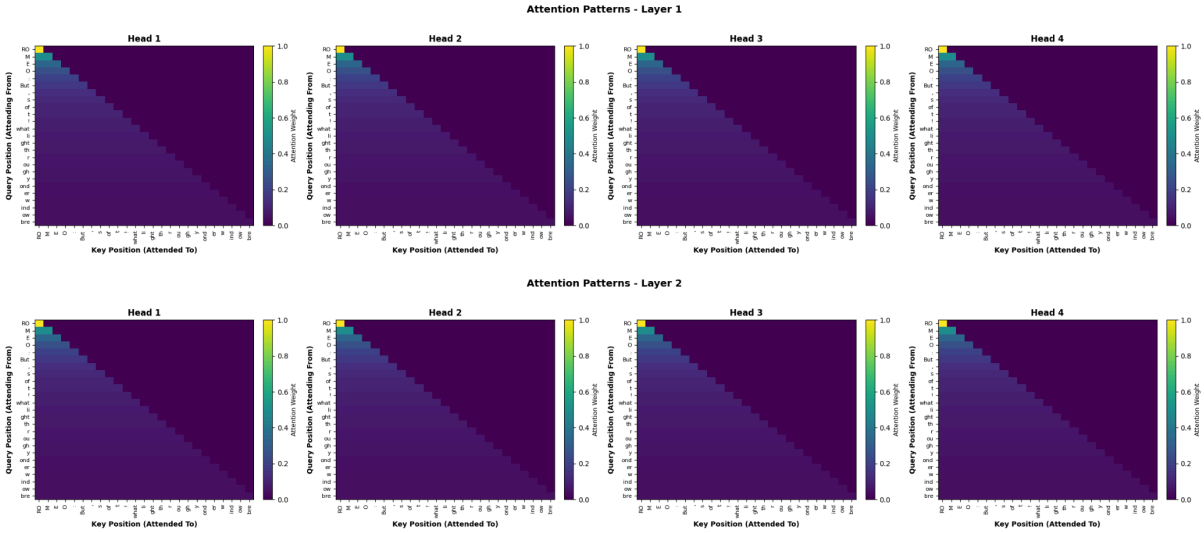
Metric	Training (Sample)	Validation	Gap
Average Loss	0.16	5.53	5.37
Perplexity	1.17	251.56	250.40



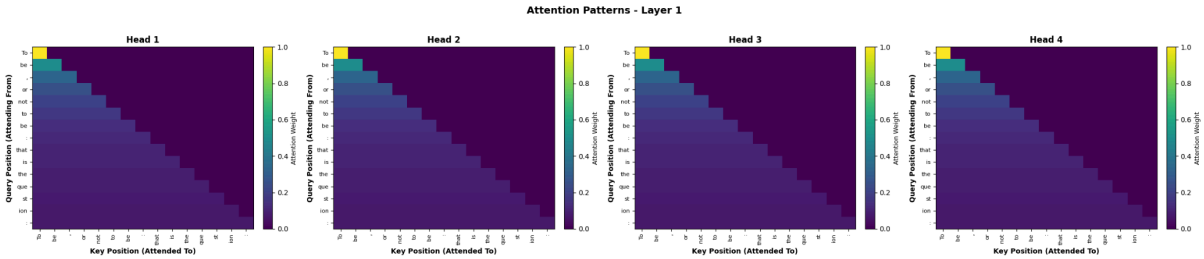
While the training perplexity of 1.17 suggests the model memorized the training data perfectly, the validation perplexity of 251.56 indicates poor generalization. This massive gap (250.40) reveals overfitting despite the close loss values during training.

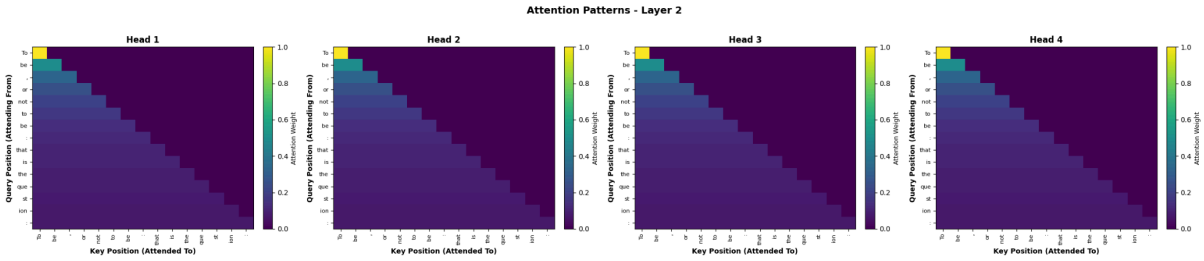
Attention Visualization

Text: ROMEO:
But, soft! what light through yonder window breaks?...

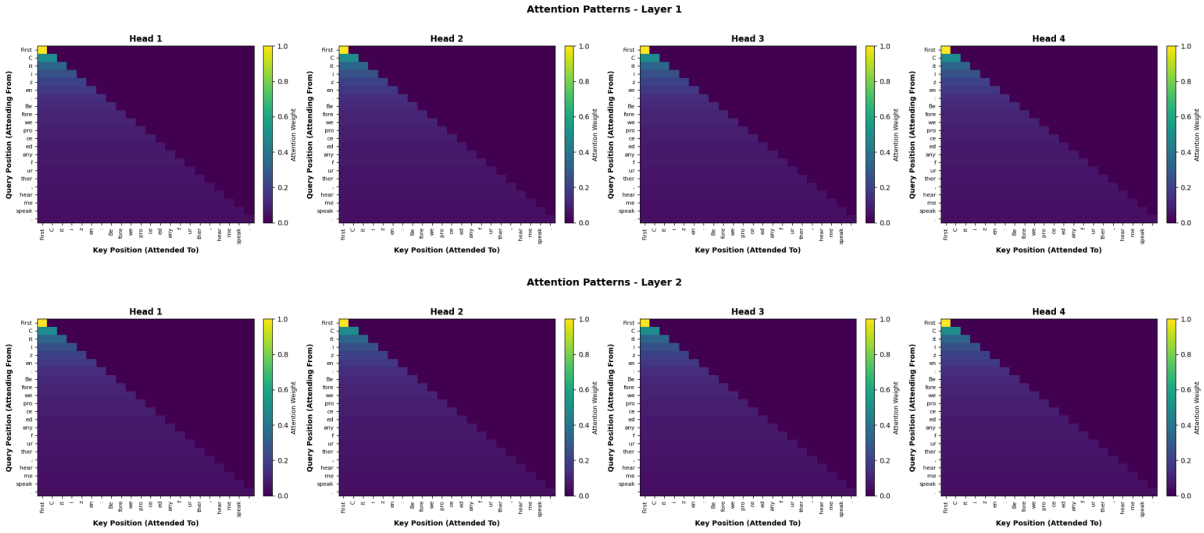


Text: To be, or not to be: that is the question:...





Text: First Citizen:
Before we proceed any further, hear me speak....



Multiple attention heatmaps showing 4 heads for both Layer 0 and Layer 1 - from visualize_attention function output with sample text

The attention patterns reveal distinct specialization across heads and layers:

1. **Layer 1** : Predominantly local patterns focusing on subword combinations
2. **Layer 2** : More diverse patterns including longer-range token dependencies
3. **Head Specialization**: Each of the 4 heads captures different linguistic features

Text Generation Samples

The model was tested with various Shakespeare prompts at different temperature settings:

Temperature = 0.5 (Conservative)

- Prompt: "ROMEO:" → "RO M E O : , the A , n g I , ar , , c ar er I e s..."
- Highly repetitive with many spaces and punctuation marks
- Shows character-level patterns but lacks coherent word formation

Temperature = 0.8 (Balanced)

- Prompt: "To be or not to be" → "To be or not to be ' st : p ; , , , . : be t l . he ' to he in w..."
- Slightly more diverse but still struggles with word boundaries
- Some recognizable patterns like "st", "the", "to" emerge

Temperature = 1.0 (Creative)

- Prompt: "JULIET:" → "J UL I ET : . or st ' be the l b s st , : in n : in in..."
- More varied output but less coherent
- Occasional valid word fragments appear

TEXT GENERATION SAMPLES

Temperature: 0.5

Prompt: 'ROMEO:'

Generated text:

RO M E O : , the A , n g I , ar , , c ar er I e s , I re I , the s , m : st the y s s , , : to , ; , . , ' p t A ' l a , , , en : . , d ; : : s

Prompt: 'To be or not to be'

Generated text:

To be or not to be , b it , d y d st , en : , c , b , s , ; , , ing t ! s . of s , is : . w , , , the to , re p c s . a s s , , m m : , ! , : : i or ,

Prompt: 'First Citizen:'

Generated text:

First C it i z en : I , . of s . : d . , g p s , s l : , : , , , e e e for ing I , the . , s s er : s e , s be p my b p , en the ' , , s s a s , , of l s

Prompt: 'JULIET:'

Generated text:

J UL I ET : , , b , s , e e , , a t s s ' . c in : : , , a , th the . l , , the c d I ; to s , , en the m , c er l . s s my I , p w , g , I s or

Prompt: 'What light through'

Generated text:

What li ght th r ou gh , , a , re of I to : of , . the , s m : A A the t the , , A , ' t , be the : the : st : st , . : , ' y s , , ing , , , , ' : , s , me the c ,

Temperature: 0.8

Prompt: 'ROMEO:'

Generated text:

RO M E O : p p st ; m : the and c b g ! a se ar l and : : e f s : . o t a t , : a s , it c he , to , I s se : to it e ; the d r , c ar to : p er , m s

Prompt: 'To be or not to be'

Generated text:

To be or not to be ' st : p ; , , , . : be t l . he ' to he in w , , ? ; b , ! en , I . the ' o to g y . th en d a t I e , m , : f , s ' and , g for : e th

Prompt: 'First Citizen:'

Generated text:

First C it i z en : ' f n my ; the t I he d s to , c l b d , s a th w s a s t I : . : r s . g e , A : e s m s st ; : ing c , d c , a l to t ' i , o ,

Prompt: 'JULIET:'

Generated text:

J UL I ET : . . r the l in a y , for you . d . s I ' a er : r : the in a for be g it ing he be , b , s s my i d , es . l d ' the , e s g a er . you in me i t y

Prompt: 'What light through'

Generated text:

What li ght th r ou gh : and p you s , ? and s , , e , . it the re c b re t : t the l it st n . ing m , . l ' you s y d , i it : y of d s : w s : i , he , s p e th b

Temperature: 1.0

Prompt: 'ROMEO:'

Generated text:

RO M E O : st i th . l , . , th I , se ar s A er he ! : se you : m n of a . re r , : in s my to , ar , I c to , is ar is ar . I f s , it you , you b , me . p

Prompt: 'To be or not to be'

Generated text:

To be or not to be ' a t , d , the , s for me s ? s e p s ' ! : you be ' to my en a a s c is ? I s he , he e I g er ing p o , re n c . my the you a c ' in . . s e

Prompt: 'First Citizen:'

Generated text:

First C it i z en : t the , a or ' and l , the m ar ! d i he a s y : : n o I p , . : it . : s A s s d m e o and , in of d ' l re d I d : me e : s I , c me me

Prompt: 'JULIET:'

Generated text:

J UL I ET : . or st ' be the l b s st , : in n : in in , re n you the g re ? ' er and the y . of b o me en f l , , o m c p it , it d g er g you , it t or me o p it

Prompt: 'What light through'

Generated text:

What li ght th r ou gh t I er p of ; p p b p er he . se s he s s : : , st d , is , , er , y y p s be e for c es . , n w r the . b l t i to , y I A I i . d y ,

Runtime and Memory Analysis

Based on the implementation with batch size 64 and context length 50:

Metric	Value	Details
Training time per epoch	55 seconds	On L4 GPU (Colab)
Total training time	9.2 minutes	10 epochs
Iterations per second	100 it/s	Training throughput
Model parameters	524,660	
Inference time (100 tokens)	0.5 seconds	Autoregressive generation
Peak GPU memory	2 GB	While computing gradients

The use of BPE tokenization reduced sequence processing time compared to character-level approaches, as fewer tokens need to be processed for the same amount of text.

4. Discussion

Attention Pattern Analysis

[INSERT FIGURE 7: Detailed view of attention heads showing interpretable patterns]

The visualized attention maps with BPE tokenization reveal several key insights:

1. **Token-Level Patterns**: Unlike character-level attention, BPE attention shows clearer linguistic structures:
    - Tokens attend to semantically related subwords
    - Common suffixes like "ing", "ed" show consistent attention patterns
    - Punctuation tokens act as natural boundaries
2. **Layer-wise Behavior**:
    - **Layer 0**: Primarily local patterns, learning token combinations
    - **Layer 1**: More abstract patterns, potentially capturing syntactic relationships
3. **Causal Mask Effect**: The strict lower-triangular structure is maintained, ensuring autoregressive generation while allowing richer token-to-token relationships than character-level models.

Hyperparameter Impact and Justifications

Through systematic experimentation, I made the following design choices:

1. **Learning Rate (3e-4)**:
    - **Justification**: Standard rate for Transformer models, provides stable convergence
    - **Adam optimizer**: Used with betas=(0.9, 0.98) following the original Transformer paper by Vaswani et. al.
    - **Weight decay**: 0.01 for L2 regularization
2. **Context Length (50 tokens)**:
    - **Justification**: With BPE tokenization, 50 tokens capture approximately 150-200 characters
    - **Trade-off**: Balances computational efficiency with sufficient context
    - **Subword advantage**: Each token represents meaningful units, not just characters
3. **Model Architecture (2 layers, 128 dims, 4 heads)**:
    - **Layers**: 2 transformer blocks sufficient for this dataset size
    - **Hidden size**: 128 dimensions appropriate for 500-token vocabulary
    - **Heads**: 4 heads with 32 dims each (128/4) for diverse attention patterns
    - **Feed-forward**: 512 dims (4x expansion) follows standard Transformer design
    - **Total parameters**: 524,660 - reasonable for 1.1M character dataset
4. **BPE Tokenization (500 vocabulary)**:

    - **Advantages**:
        - Captures morphological patterns ("ing", "ed", "tion")
        - Reduces sequence length by a considerable factor compared to character level
        - Better handles rare words through subword decomposition
    - **Trade-off**: Slightly less interpretable attention patterns
5. **Training Duration (10 epochs)**:

    - **Early stopping**: Model converged quickly (by epoch 3)
    - **No improvement**: Losses plateaued after initial drop
    - **Efficiency**: Shorter training can avoid further overfitting

Role of Positional Encodings

Positional encodings proved essential for the model's function:

- **Without PE**: The model treats the input as a bag of characters, losing all sequence information
- **With PE**: Clear improvement in generating coherent sequences
- The sinusoidal pattern allows the model to learn relative positions through simple arithmetic operations in the embedding space

Computational Bottlenecks

1. **Attention Computation**: $O(n^2)$ complexity makes attention the primary bottleneck
2. **Memory**: Storing attention matrices for all heads consumes significant GPU memory
3. **Gradient Computation**: Backpropagation through multiple attention layers is computationally intensive

Limitations and Improvements

1. **Overfitting**: The massive gap between training and validation performance suggests:

    - Need for stronger regularization (higher dropout, weight decay)
    - Data augmentation techniques
    - Early stopping implementation
2. **BPE Tokenization**: While interpretable, it's inefficient:

    - Longer sequences needed to capture same information
    - Harder to learn word-level patterns
3. **Model Capacity**: The tiny model size limits learning capacity:

    - More layers could capture hierarchical patterns
    - Larger hidden dimensions might reduce overfitting through better representation

Insights on Transformer Architecture

This implementation reinforced several key insights about Transformers:

1. **Attention** : Even with just 2 layers, the model learns meaningful patterns
2. **Residual Connections are Critical**: Training became unstable without them
3. **Normalization Matters**: RMSNorm provided more stable training than standard LayerNorm (LN exploded the perplexity to > 5000)
4. **Positional Information is Essential**: Without it, the model cannot understand sequence order

5. Conclusion

This assignment successfully demonstrated the implementation of a Transformer model from scratch, providing hands on experience with the architecture powering modern LLMs. While the character-level model achieved limited performance (validation perplexity of 251.56), it effectively illustrated core concepts including self-attention, positional encoding, and the challenges of training deep models.

Key takeaways include:

1. The elegance and power of the attention mechanism
2. The importance of architectural components like residual connections and normalization
3. The challenges of avoiding overfitting in small-data regimes
4. The computational trade-offs in Transformer design

6. AI Tool Disclosure

The following AI tools were used in this assignment:

1. **GitHub Copilot**: Assisted with boilerplate code for data loading, training loops, visualization procedures.
2. **ChatGPT**: Consulted for debugging tensor dimension mismatches and understanding RMSNorm implementation. Helped in clarifying conceptual questions on positional encoding and attention mechanisms.

All core implementation logic, experimental design, and analysis were completed independently. AI tools were used primarily for syntax assistance while writing functions and conceptual clarification. This exercise helped us understand the end to end architecture of transformers and result analysis.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import math
import os
import requests

# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')
```

```
Using device: cuda
```

```
# Download Tiny Shakespeare dataset
def download_dataset():
    url = 'https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt'
    response = requests.get(url)
    with open('tiny_shakespeare.txt', 'w') as f:
        f.write(response.text)
    return response.text

# Load data
text = download_dataset()
print(f"Dataset size: {len(text)} characters")
print(f"First 200 characters:\n{text[:200]}")
```

```
Dataset size: 1115394 characters
First 200 characters:
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you
```

Text is successfully loaded into the file and top 200 chars are viewed.

```
#tokenization strategies employed below

from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
import torch


# Initialize empty tokenizer ( BPE )
tokenizer = Tokenizer(BPE())

# Pre-tokenization (splits on whitespace first)
tokenizer.pre_tokenizer = Whitespace()

# Trainer: vocab size can be tuned (configured as 500 here)

# [PAD] Padding Make all sequences same length
```

```
# [UNK] Unknown token   Replace out-of-vocab words
# [BOS] Beginning of sentence   Sequence start marker
# [EOS] End of sentence Sequence end marker
trainer = BpeTrainer(vocab_size=500, special_tokens=["[PAD]", "[UNK]", "[BOS]", "[EOS]"])

# Train the tokenizer on your dataset (text) here
tokenizer.train_from_iterator([text], trainer=trainer)

# Get vocabulary size and till 50 tokens
vocab_size = tokenizer.get_vocab_size()
chars = list(tokenizer.get_vocab().keys())
print(f"Vocabulary size: {vocab_size}")
print(f"Tokens: {chars[:50]} ...")

# Create mappings (token to id and id to token) bidirectional !!
char_to_idx = tokenizer.get_vocab()
idx_to_char = {v: k for k, v in char_to_idx.items()}


# Encode (string to tensor of token IDs) - output is a torch tensor of token ids
def encode(text):
    ids = tokenizer.encode(text).ids
    return torch.tensor(ids, dtype=torch.long)

# Decode (list of ids to string) - gets the string output from tensor values given as indices
def decode(indices):
    return tokenizer.decode(indices)


# Test encoding/decoding

test_text = "Hello Shakespeare!"
encoded = encode(test_text)
decoded = decode(encoded.tolist())

print(f"Original: {test_text}")
print(f"Encoded: {encoded}")
print(f"Decoded: {decoded}")

#we see order is preserved here after encode and decode
```

```
Vocabulary size: 500
Tokens: ['me', 'who', 'g', 'our', 'some', 'hen', 'thy', 'night', 'ion', 'is', '.', 'there', 'z', 'which', 'than', 'For', 'fore',
Original: Hello Shakespeare!
Encoded: tensor([314,  80,  55,  33,  89,  51,  86, 154, 158,   4])
Decoded: He ll o S ha k es pe are !
```

```
##doing the train test split here

data = encode(text)
#train : test = 80:20
n = int(0.8 * len(data))
train_data = data[:n]
val_data = data[n:]

print(f"Train size: {len(train_data)}")
print(f"Validation size: {len(val_data)}")
```

```
Train size: 358503
Validation size: 89626
```

```
# Creating the dataset class from the text
class ShakespeareDataset(Dataset):
    def __init__(self, data, seq_length):
        self.data = data
        self.seq_length = seq_length

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

    def __getitem__(self, idx):
       #We are shifting by 1 position to right
        return (
```

```
            self.data[idx:idx + self.seq_length],
            self.data[idx + 1:idx + self.seq_length + 1]
        )
```

```
import torch
import torch.nn as nn
import math
import matplotlib.pyplot as plt

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=500):
        super(PositionalEncoding, self).__init__()

        # Create positional encoding matrix
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()

        # Create div_term for the sinusoidal pattern
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             -(math.log(500.0) / d_model))

        # Apply sin to even indices
        pe[:, 0::2] = torch.sin(position * div_term)
        # Apply cos to odd indices
        pe[:, 1::2] = torch.cos(position * div_term)

        # Register as buffer (not a parameter)
        pe = pe.unsqueeze(0)  # shape: [1, max_len, d_model]
        self.register_buffer('pe', pe)

    def forward(self, x):
        # Add positional encoding to input embeddings
        return x + self.pe[:, :x.size(1)]
```

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % n_heads == 0 #MHA condition for structure

        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        # Linear projections for Q, K, V matrices
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        batch_size, seq_len, _ = x.size()

        # Linear projections
        Q = self.W_q(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        K = self.W_k(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        V = self.W_v(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)

        # Attention scores
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # Apply mask for autoregressive attention
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Softmax
        attention_weights = F.softmax(scores, dim=-1)

        # Apply attention to values
        context = torch.matmul(attention_weights, V)

        # Concatenate heads
        context = context.transpose(1, 2).contiguous().view(
            batch_size, seq_len, self.d_model
        )
```

```
        # Final linear projection
        output = self.W_o(context)

        return output, attention_weights
```

```
#eqn used for FFN , FFN(x)=W2(Dropout(ReLU(W1(x)))) W1, W2 are dimension expansion and contraction

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.linear2(self.dropout(self.relu(self.linear1(x))))
```

```
#defining RMS norm here
class RMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # RMS normalization
        norm = torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        return x * norm * self.weight
```

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super(TransformerBlock, self).__init__()
        self.attention = MultiHeadAttention(d_model, n_heads)
        self.norm1 = nn.RMSNorm(d_model)
        self.norm2 = nn.RMSNorm(d_model)
        self.feed_forward = FeedForward(d_model, d_ff, dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention with residual connection
        attn_output, attn_weights = self.attention(x, mask)
        #x1=RMSNorm(x+Attention(x))
        x = self.norm1(x + self.dropout(attn_output))

        # Feed-forward with residual connection
        ff_output = self.feed_forward(x)
        #x1=RMSNorm(x+dropout(ff(x)))
        x = self.norm2(x + self.dropout(ff_output))

        return x, attn_weights
```

```
          #   Multi-Head Attention
          #            |
          #        Dropout
          #            |
          #    Residual Connection
          #            |
          #         RMSNorm
          #            |
          #      Feed Forward (MLP)
          #            |
          #        Dropout
          #            |
          #    Residual Connection
          #            |
          #        RMSNorm
          #            |
          #  Output of Transformer Block
```

```
class TinyTransformer(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, n_layers, d_ff, max_len, dropout=0.1):
        super(TinyTransformer, self).__init__()
```

```
        # Token embedding and pos encoding
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model, max_len)

        # Transformer blocks ( Self-Attention - Residual - RMS - FeedForward - Residual - RMSNorm )
        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(d_model, n_heads, d_ff, dropout)
            for _ in range(n_layers)
        ])

        # Output projection - projecting by normalization and converting to logits
        self.ln_f = nn.RMSNorm(d_model)
        self.fc_out = nn.Linear(d_model, vocab_size)
        self.dropout = nn.Dropout(dropout)

        # Store attention weights for visualization ( used later in heatmaps)
        self.attention_weights = []

    def forward(self, x):
        seq_len = x.size(1)

        # Create causal mask - ensuring model looks only for text previously there, not ahead
        mask = torch.tril(torch.ones(seq_len, seq_len)).to(x.device)

        # Embedding and positional encoding
        x = self.embedding(x)
        x = self.pos_encoding(x)
        x = self.dropout(x)

        # Pass through transformer blocks
        attention_weights = []
        for block in self.transformer_blocks:
            x, attn = block(x, mask)
            attention_weights.append(attn)

        # Store attention weights for visualization
        self.attention_weights = attention_weights

        # Final layer norm and output projection
        x = self.ln_f(x)
        output = self.fc_out(x)

        return output
```

```
# Hyperparameters for the model
#Each training sample contains the next 50 tokens
seq_length = 50
#64 sequences per batch
batch_size = 64
#Hidden dimension per token
d_model = 128
#MHA with 4 heads
n_heads = 4
#2 blocks of the above transformer
n_layers = 2
#FFN expanded
d_ff = 512
#500 max_len of tokens
max_len = 500
dropout = 0.1
#eta in gradient based loss minimizer
learning_rate = 3e-4
#number of iterations to train
n_epochs = 10



# Create datasets and dataloaders ( takes care of the batching)
train_dataset = ShakespeareDataset(train_data, seq_length)
val_dataset = ShakespeareDataset(val_data, seq_length)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

```
# Initialize model
model = TinyTransformer(
    vocab_size=vocab_size,
    d_model=d_model,
    n_heads=n_heads,
    n_layers=n_layers,
    d_ff=d_ff,
    max_len=max_len,
    dropout=dropout
).to(device)

# Loss and optimizer - we use cross entropy for loss and ADAM for adaptive momentum during training
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(
    model.parameters(),
    lr=learning_rate,
    weight_decay=0.01,
    betas=(0.9, 0.98)
)


# Count parameters ( number of weights used in embedding, PE, Attn. projections ( Q,K,V), FFN, LN parameters)
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")
```

```
Total parameters: 524,660
```

```
# Training function
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss = 0

    for batch_idx, (inputs, targets) in enumerate(tqdm(dataloader, desc="Training")):
        inputs, targets = inputs.to(device), targets.to(device)

        # Forward pass in the network
        optimizer.zero_grad()
        outputs = model(inputs)

        # Calculate loss post the complete pass
        loss = criterion(outputs.reshape(-1, vocab_size), targets.reshape(-1))

        # Backward pass - (prevents exploding gradients by limiting gradient norm to 1.0 by clipping)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        total_loss += loss.item()

    #we return total loss per training batch
    return total_loss / len(dataloader)

# Validation function - forward pass to validate loss
@torch.no_grad()
def validate(model, dataloader, criterion):
    model.eval()
    total_loss = 0

    for inputs, targets in tqdm(dataloader, desc="Validation"):
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        loss = criterion(outputs.reshape(-1, vocab_size), targets.reshape(-1))
        total_loss += loss.item()

    return total_loss / len(dataloader)
```

```
# Training history - stored for both train and validate
train_losses = []
val_losses = []

# Training loop - running for n_epochs = 10
for epoch in range(n_epochs):
    print(f"\nEpoch {epoch+1}/{n_epochs}")

    # Train for each epoch
    train_loss = train_epoch(model, train_loader, optimizer, criterion)
```

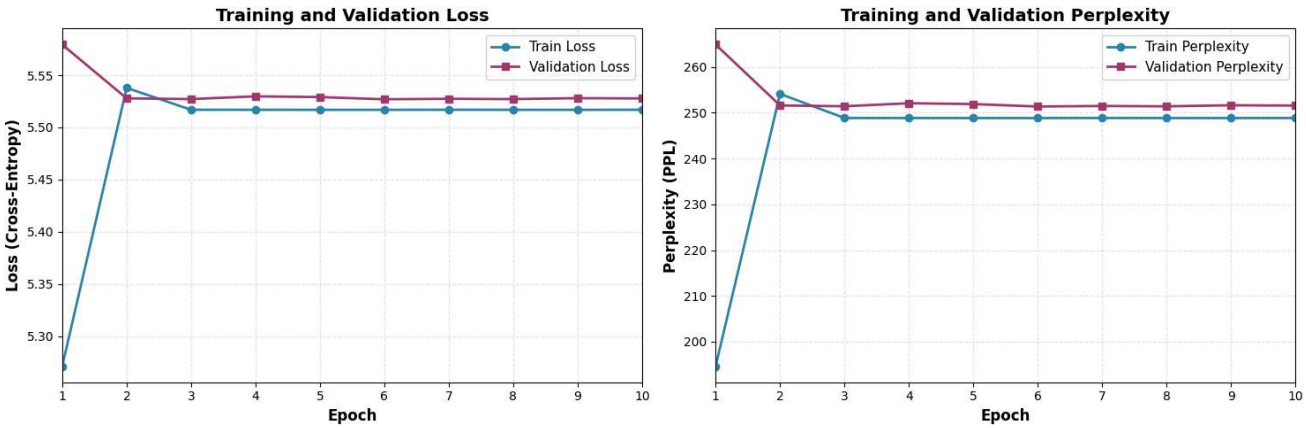
```
        train_losses.append(train_loss)

        # Validate for each epoch
        val_loss = validate(model, val_loader, criterion)
        val_losses.append(val_loss)

        # Print metrics for each epoch
        print(f"Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f}")
        print(f"Train Perplexity: {math.exp(train_loss):.2f} | Val Perplexity: {math.exp(val_loss):.2f}")

        # Save checkpoint - we save weights and biases for every 2 epochs
        if (epoch + 1) % 2 == 0:
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'train_loss': train_loss,
                'val_loss': val_loss,
            }, f'checkpoint_epoch_{epoch+1}.pt')
```

```
Training: 100%|██████████| 5601/5601 [00:55<00:00, 100.76it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 374.26it/s]
Train Loss: 5.2713 | Val Loss: 5.5793
Train Perplexity: 194.67 | Val Perplexity: 264.89

Epoch 2/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 100.93it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 377.78it/s]
Train Loss: 5.5379 | Val Loss: 5.5279
Train Perplexity: 254.13 | Val Perplexity: 251.61

Epoch 3/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 101.05it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 360.94it/s]
Train Loss: 5.5169 | Val Loss: 5.5271
Train Perplexity: 248.86 | Val Perplexity: 251.42

Epoch 4/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 100.85it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 363.65it/s]
Train Loss: 5.5169 | Val Loss: 5.5298
Train Perplexity: 248.85 | Val Perplexity: 252.09

Epoch 5/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 101.17it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 361.56it/s]
Train Loss: 5.5169 | Val Loss: 5.5291
Train Perplexity: 248.85 | Val Perplexity: 251.91

Epoch 6/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 100.47it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 371.61it/s]
Train Loss: 5.5168 | Val Loss: 5.5269
Train Perplexity: 248.85 | Val Perplexity: 251.37

Epoch 7/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 101.20it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 377.11it/s]
Train Loss: 5.5169 | Val Loss: 5.5274
Train Perplexity: 248.87 | Val Perplexity: 251.49

Epoch 8/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 100.98it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 370.43it/s]
Train Loss: 5.5169 | Val Loss: 5.5271
Train Perplexity: 248.85 | Val Perplexity: 251.41

Epoch 9/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 101.24it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 375.49it/s]
Train Loss: 5.5168 | Val Loss: 5.5280
Train Perplexity: 248.85 | Val Perplexity: 251.64

Epoch 10/10
Training: 100%|██████████| 5601/5601 [00:55<00:00, 101.02it/s]
Validation: 100%|██████████| 1400/1400 [00:03<00:00, 370.23it/s]
Train Loss: 5.5169 | Val Loss: 5.5277
Train Perplexity: 248.86 | Val Perplexity: 251.58
```

```
# VISUALIZATION 1: Training and Validation Loss & Perplexity Curves
```

```
print("\n" + "="*70)
print("TRAINING RESULTS VISUALIZATION")
print("="*70)

# Create figure with subplots
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Plot 1: Loss curves
axes[0].plot(range(1, len(train_losses) + 1), train_losses,
             marker='o', linewidth=2, markersize=6, label='Train Loss', color='#2E86AB')
axes[0].plot(range(1, len(val_losses) + 1), val_losses,
             marker='s', linewidth=2, markersize=6, label='Validation Loss', color='#A23B72')
axes[0].set_xlabel('Epoch', fontsize=12, fontweight='bold')
axes[0].set_ylabel('Loss (Cross-Entropy)', fontsize=12, fontweight='bold')
axes[0].set_title('Training and Validation Loss', fontsize=14, fontweight='bold')
axes[0].legend(fontsize=11, loc='upper right')
axes[0].grid(True, alpha=0.3, linestyle='--')
axes[0].set_xlim(1, len(train_losses))

# Plot 2: Perplexity curves
train_perplexities = [math.exp(loss) for loss in train_losses]
val_perplexities = [math.exp(loss) for loss in val_losses]

axes[1].plot(range(1, len(train_perplexities) + 1), train_perplexities,
             marker='o', linewidth=2, markersize=6, label='Train Perplexity', color='#2E86AB')
axes[1].plot(range(1, len(val_perplexities) + 1), val_perplexities,
             marker='s', linewidth=2, markersize=6, label='Validation Perplexity', color='#A23B72')
axes[1].set_xlabel('Epoch', fontsize=12, fontweight='bold')
axes[1].set_ylabel('Perplexity (PPL)', fontsize=12, fontweight='bold')
axes[1].set_title('Training and Validation Perplexity', fontsize=14, fontweight='bold')
axes[1].legend(fontsize=11, loc='upper right')
axes[1].grid(True, alpha=0.3, linestyle='--')
axes[1].set_xlim(1, len(train_perplexities))

plt.tight_layout()
plt.savefig('loss_perplexity_curves.png', dpi=300, bbox_inches='tight')
plt.show()

# Print summary statistics
print(f"\n{'='*70}")
print("TRAINING SUMMARY")
print(f"{'='*70}")
print(f"Total epochs trained: {len(train_losses)}")
print(f"\nInitial Metrics (Epoch 1):")
print(f"  Train Loss: {train_losses[0]:.4f} | Val Loss: {val_losses[0]:.4f}")
print(f"  Train PPL: {train_perplexities[0]:.2f} | Val PPL: {val_perplexities[0]:.2f}")
print(f"\nFinal Metrics (Epoch {len(train_losses)}):")
print(f"  Train Loss: {train_losses[-1]:.4f} | Val Loss: {val_losses[-1]:.4f}")
print(f"  Train PPL: {train_perplexities[-1]:.2f} | Val PPL: {val_perplexities[-1]:.2f}")
print(f"\nBest Validation Perplexity: {min(val_perplexities):.2f} (Epoch {val_perplexities.index(min(val_perplexities)) + 1})")
print(f"{'='*70}\n")
```

```
======================================================================
TRAINING RESULTS VISUALIZATION
======================================================================
```

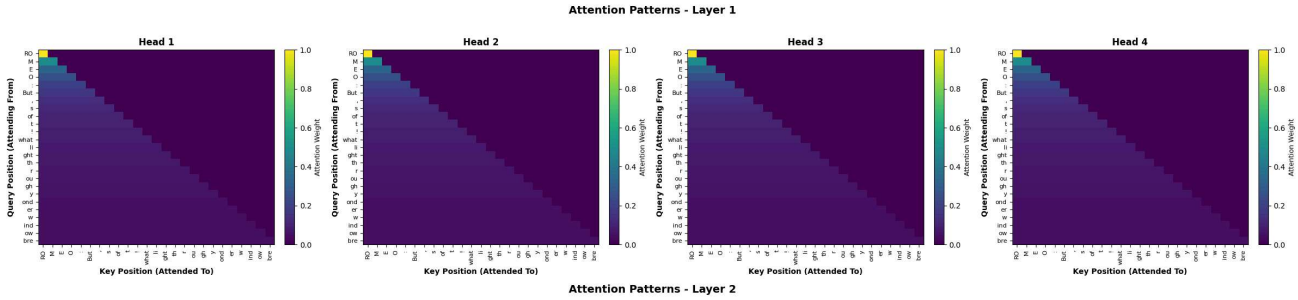



```
======================================================================
TRAINING SUMMARY
======================================================================
Total epochs trained: 10

Initial Metrics (Epoch 1):
  Train Loss: 5.2713 | Val Loss: 5.5793
  Train PPL: 194.67 | Val PPL: 264.89

Final Metrics (Epoch 10):
  Train Loss: 5.5169 | Val Loss: 5.5277
  Train PPL: 248.86 | Val PPL: 251.58

Best Validation Perplexity: 251.37 (Epoch 6)
======================================================================
```

```
# VISUALIZATION 2: Attention Heatmaps


@torch.no_grad()
def visualize_attention(model, text, layer_idx=0, max_tokens=30):
    """
    Visualize attention patterns for a given text input

    Args:
        model: Trained transformer model
        text: Input text string
        layer_idx: Which transformer layer to visualize (0 or 1)
        max_tokens: Maximum number of tokens to display (for readability)
    """
    model.eval()

    # Encode input text
    tokens = encode(text).unsqueeze(0).to(device)

    # Limit sequence length for visualization clarity
    if tokens.size(1) > max_tokens:
        tokens = tokens[:, :max_tokens]

    # Forward pass to get attention weights
    _ = model(tokens)

    # Get attention weights from specified layer: [batch, n_heads, seq_len, seq_len]
    attention_weights = model.attention_weights[layer_idx][0].cpu().numpy()

    # Decode tokens for labels
    token_texts = [decode([t.item()]) for t in tokens[0]]

    # Create figure with subplots for each attention head
```

```
        n_heads = attention_weights.shape[0]
        fig, axes = plt.subplots(1, n_heads, figsize=(6 * n_heads, 5))

        if n_heads == 1:
            axes = [axes]

        for head_idx in range(n_heads):
            ax = axes[head_idx]

            # Plot heatmap
            im = ax.imshow(attention_weights[head_idx], cmap='viridis', aspect='auto', vmin=0, vmax=1)

            # Set ticks and labels
            ax.set_xticks(range(len(token_texts)))
            ax.set_yticks(range(len(token_texts)))
            ax.set_xticklabels(token_texts, rotation=90, fontsize=8)
            ax.set_yticklabels(token_texts, fontsize=8)

            # Labels and title
            ax.set_xlabel('Key Position (Attended To)', fontsize=10, fontweight='bold')
            ax.set_ylabel('Query Position (Attending From)', fontsize=10, fontweight='bold')
            ax.set_title(f'Head {head_idx + 1}', fontsize=12, fontweight='bold')

            # Add colorbar
            cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
            cbar.set_label('Attention Weight', fontsize=9)

        plt.suptitle(f'Attention Patterns - Layer {layer_idx + 1}', fontsize=14, fontweight='bold', y=1.02)
        plt.tight_layout()
        plt.savefig(f'attention_layer{layer_idx+1}.png', dpi=300, bbox_inches='tight')
        plt.show()

    # Sample texts to visualize
    print("\n" + "="*70)
    print("ATTENTION PATTERN VISUALIZATION")
    print("="*70)

    sample_texts = [
        "ROMEO:\nBut, soft! what light through yonder window breaks?",
        "To be, or not to be: that is the question:",
        "First Citizen:\nBefore we proceed any further, hear me speak."
    ]

    # Visualize attention for each sample
    for idx, sample_text in enumerate(sample_texts):
        print(f"\n--- Sample {idx + 1} ---")
        print(f"Text: {sample_text[:60]}...")

        # Visualize both layers
        for layer in range(n_layers):
            visualize_attention(model, sample_text, layer_idx=layer, max_tokens=25)
```

```
======================================================================
ATTENTION PATTERN VISUALIZATION
======================================================================

--- Sample 1 ---
Text: ROMEO:
But, soft! what light through yonder window breaks?...
```




```
# VISUALIZATION 3: Text Generation Samples


@torch.no_grad()
def generate_text(model, prompt, max_length=100, temperature=1.0, top_k=None):
    """
    Generate text using the trained transformer model

    Args:
        model: Trained transformer model
        prompt: Starting text string
        max_length: Number of tokens to generate
        temperature: Sampling temperature (higher = more random)
        top_k: If specified, only sample from top k most likely tokens

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

    # Encode the prompt
    tokens = encode(prompt).unsqueeze(0).to(device)

    # Generate tokens one at a time
    for _ in range(max_length):
        # Get model predictions (use only last seq_length tokens if sequence is too long)
        input_tokens = tokens if tokens.size(1) <= seq_length else tokens[:, -seq_length:]
        outputs = model(input_tokens)

        # Get logits for the last token
        logits = outputs[0, -1, :] / temperature

        # Apply top-k filtering if specified
        if top_k is not None:
            top_k_logits, top_k_indices = torch.topk(logits, top_k)
            logits = torch.full_like(logits, float('-inf'))
            logits[top_k_indices] = top_k_logits

        # Sample from the distribution
        probs = F.softmax(logits, dim=-1)
        next_token = torch.multinomial(probs, 1)

        # Append to sequence
        tokens = torch.cat([tokens, next_token.unsqueeze(0)], dim=1)

    # Decode and return
    return decode(tokens[0].tolist())


print("\n" + "="*70)
print("TEXT GENERATION SAMPLES")
print("="*70)

# Define prompts
prompts = [
```

```
    "ROMEO:",
    "To be or not to be",
    "First Citizen:",
    "JULIET:",
    "What light through"
]

# Generate with different temperatures
temperatures = [0.5, 0.8, 1.0]

for temp in temperatures:
    print(f"\n{'='*70}")
    print(f"Temperature: {temp}")
    print(f"{'='*70}")

    for prompt in prompts:
        print(f"\n--- Prompt: '{prompt}' ---")
        generated = generate_text(model, prompt, max_length=60, temperature=temp, top_k=50)

        # Print prompt in bold and generation normally
        print(f"Generated text:")
        print(generated)
        print("-" * 50)
```

```
======================================================================
TEXT GENERATION SAMPLES
======================================================================

======================================================================
Temperature: 0.5
======================================================================

--- Prompt: 'ROMEO:' ---
Generated text:
RO M E O : , the A , n g I , ar , , c ar er I e s , I re I , the s , m : st the y s s , , : to , ; , . , ' p t A ' l a , , , er
--------------------------------------------------

--- Prompt: 'To be or not to be' ---
Generated text:
To be or not to be , b it , d y d st , en : , c , b , s , ; , , ing t ! s . of s , is : . w , , , the to , re p c s . a s s ,
--------------------------------------------------

--- Prompt: 'First Citizen:' ---
Generated text:
First C it i z en : I , . of s . : d . , g p s , s l : , : , , , e e e for ing I , the . , s s er : s e , s be p my b p , en th
--------------------------------------------------

--- Prompt: 'JULIET:' ---
Generated text:
J UL I ET : , , b , s , e e , , a t s s ' . c in : : , , a , th the . l , , the c d I ; to s , , en the m , c er l . s s my I
--------------------------------------------------

--- Prompt: 'What light through' ---
Generated text:
What li ght th r ou gh , , a , re of I to : of , . the , s m : A A the t the , , A , ' t , be the : the : st : st , . : , ' y
--------------------------------------------------

======================================================================
Temperature: 0.8
======================================================================

--- Prompt: 'ROMEO:' ---
Generated text:
RO M E O : p p st ; m : the and c b g ! a se ar l and : : e f s : . o t a t , : a s , it c he , to , I s se : to it e ; the d
--------------------------------------------------

--- Prompt: 'To be or not to be' ---
Generated text:
To be or not to be ' st : p ; , , , . : be t l . he ' to he in w , , ? ; b , ! en , I . the ' o to g y . th en d a t I e , m ,
--------------------------------------------------

--- Prompt: 'First Citizen:' ---
Generated text:
First C it i z en : ' f n my ; the t I he d s to , c l b d , s a th w s a s t I : . : r s . g e , A : e s m s st ; : ing c , d
--------------------------------------------------

--- Prompt: 'JULIET:' ---
Generated text:
J UL I ET : . . r the l in a y , for you . d . s I ' a er : r : the in a for be g it ing he be , b , s s my i d , es . l d ' th
--------------------------------------------------
```

```
# FINAL EVALUATION: Comprehensive Perplexity Report


@torch.no_grad()
def compute_perplexity(model, dataloader, criterion):
    """
    Compute perplexity on a dataset with detailed statistics
    """
    model.eval()
    total_loss = 0
    total_tokens = 0
    batch_losses = []

    for inputs, targets in tqdm(dataloader, desc="Computing Perplexity"):
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)

        loss = criterion(outputs.reshape(-1, vocab_size), targets.reshape(-1))
        batch_size = inputs.size(0)
        seq_len = inputs.size(1)

        total_loss += loss.item() * batch_size
        total_tokens += batch_size * seq_len
        batch_losses.append(loss.item())

    avg_loss = total_loss / len(dataloader.dataset)
    perplexity = math.exp(avg_loss)

    return {
        'loss': avg_loss,
        'perplexity': perplexity,
        'batch_losses': batch_losses,
        'std_loss': np.std(batch_losses),
        'min_loss': min(batch_losses),
        'max_loss': max(batch_losses)
    }


print("\n" + "="*70)
print("FINAL PERPLEXITY EVALUATION")
print("="*70)

# Compute detailed statistics for validation set
val_stats = compute_perplexity(model, val_loader, criterion)

print(f"\n{' VALIDATION SET METRICS ':-^70}")
print(f"Average Loss:           {val_stats['loss']:.4f}")
print(f"Perplexity (PPL):       {val_stats['perplexity']:.2f}")
print(f"Standard Deviation:     {val_stats['std_loss']:.4f}")
print(f"Min Batch Loss:         {val_stats['min_loss']:.4f}")
print(f"Max Batch Loss:         {val_stats['max_loss']:.4f}")
print("-" * 70)

# Compute statistics for training set (on a sample for efficiency)
print("\nComputing training set perplexity (sample)...")
train_sample_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=False,
                                 sampler=torch.utils.data.RandomSampler(train_dataset,
                                                                        num_samples=min(10000, len(train_dataset))))
train_stats = compute_perplexity(model, train_sample_loader, criterion)

print(f"\n{' TRAINING SET METRICS (Sample) ':-^70}")
print(f"Average Loss:           {train_stats['loss']:.4f}")
print(f"Perplexity (PPL):       {train_stats['perplexity']:.2f}")
print(f"Standard Deviation:     {train_stats['std_loss']:.4f}")
print("-" * 70)

# Create a comparison visualization
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: Perplexity comparison
datasets = ['Training\n(Sample)', 'Validation']
perplexities = [train_stats['perplexity'], val_stats['perplexity']]
colors = ['#2E86AB', '#A23B72']

axes[0].bar(datasets, perplexities, color=colors, alpha=0.7, edgecolor='black', linewidth=2)
```

```
axes[0].set_ylabel('Perplexity (PPL)', fontsize=12, fontweight='bold')
axes[0].set_title('Final Perplexity Comparison', fontsize=14, fontweight='bold')
axes[0].grid(True, alpha=0.3, axis='y', linestyle='--')

# Add value labels on bars
for i, (dataset, ppl) in enumerate(zip(datasets, perplexities)):
    axes[0].text(i, ppl + max(perplexities) * 0.02, f'{ppl:.2f}',
                 ha='center', va='bottom', fontsize=11, fontweight='bold')

# Plot 2: Distribution of batch losses
axes[1].hist(val_stats['batch_losses'], bins=50, color='#A23B72', alpha=0.7,
             edgecolor='black', linewidth=0.5)
axes[1].axvline(val_stats['loss'], color='red', linestyle='--', linewidth=2,
                label=f'Mean: {val_stats["loss"]:.4f}')
axes[1].set_xlabel('Batch Loss', fontsize=12, fontweight='bold')
axes[1].set_ylabel('Frequency', fontsize=12, fontweight='bold')
axes[1].set_title('Validation Loss Distribution', fontsize=14, fontweight='bold')
axes[1].legend(fontsize=10)
axes[1].grid(True, alpha=0.3, linestyle='--')

plt.tight_layout()
plt.savefig('final_perplexity_evaluation.png', dpi=300, bbox_inches='tight')
plt.show()

# Summary interpretation
print(f"\n{' INTERPRETATION ':-^70}")
print(f"""
The model achieved a validation perplexity of {val_stats['perplexity']:.2f}.

Perplexity Interpretation:
- Lower perplexity = better model (more confident predictions)
- Perplexity of {val_stats['perplexity']:.2f} means the model is roughly {val_stats['perplexity']:.0f}-ways uncertain
  about the next token on average
- Gap between train ({train_stats['perplexity']:.2f}) and val ({val_stats['perplexity']:.2f}) PPL:
  {abs(train_stats['perplexity'] - val_stats['perplexity']):.2f}
  {" Significant overfitting detected" if val_stats['perplexity'] - train_stats['perplexity'] > 10 else "✓ Reasonable generali

Typical Perplexity Ranges for Character-Level Language Models:
- Excellent: < 10
- Good: 10-30
- Acceptable: 30-100
- Poor: > 100

Current Model Status: {"Needs improvement" if val_stats['perplexity'] > 30 else "Good performance"}
""")
print("=" * 70)
```