COMS4995 Applied Machine Learning | Tiny Transformer Language Model

Luca Barattini – UNI: LB3656[1,⋆]

[1] *Columbia University in the City of New York, Fu Foundation School of Engineering*

Abstract. This report documents an end-to-end workflow for training a compact Transformer language model on a Shakespeare corpus tokenised with a 500-entry byte-pair encoding (BPE) vocabulary. Our pipeline integrates data preprocessing and architecture design with a rigorous evaluation suite, interpreting model performance through loss trajectories, perplexity, and attention visualization. Three model variants are compared along context length and width: a baseline (sequence length 64, *d_model* = *128*), a longer-context model (sequence length 128, *d_model* = *128*), and a wider model (sequence length 64, *d_model* = *256*). All use two Transformer blocks with sinusoidal positional encodings and are trained with AdamW, mixed precision, and early stopping based on validation perplexity. The wider model with a lower learning rate achieves the best perplexity (PPL = 58.35) and serves as the main object for analysis. Attention visualizations reveal distinct heads specialized for local, punctuation-based, and clause-level features. The discussion analyzes the correlation between these patterns and token statistics, the impact of hyperparameters on training stability, the role of positional encodings, and the distribution of computational resources.

Colab notebook *GitHub code*

1 Introduction

The goal of this project is to build and study a small autoregressive language model trained on Shakespearean text. Rather than aiming for state-of-the-art performance, the emphasis is on understanding how design choices affect the behaviour of a tiny Transformer: what the tokenizer learns, how attention distributes over a sequence, and how hyperparameters shape optimisation stability.

The dataset consists of a single concatenated Shakespeare corpus stored as *input.txt*. Basic inspection shows 1 115 394 characters and 65 distinct characters. The model is trained to predict the next token given the previous context, so evaluation is based on cross-entropy loss and the induced perplexity on a held-out validation set.

Concretely, we:

- Train a BPE tokenizer with a vocabulary size of 500 and compare it against traditional word-level statistics.
- Construct token sequences for next-token prediction and encapsulate them in PyTorch *Dataset* and *DataLoader* objects.
- Implement a Transformer architecture incorporating RMSNorm, multi-head self-attention, sinusoidal positional encodings, and residual connections.
- Conduct hyperparameter experiments varying sequence length, model width, and learning rate.
- Select the optimal model based on validation perplexity, visualize training trajectories and attention maps, and inspect generated samples and top-k predictions.
- Analyze the learned attention patterns, hyperparameter impact, the role of positional encodings, and computational bottlenecks.

2 Data and tokenisation

2.1 Raw text and basic statistics

Reading the corpus into memory yields a single long string of 1 115 394 characters with 65 unique characters. The small character set (letters, digits, punctuation, whitespace) confirms that most variety comes from sequences rather than new symbols, which makes the data a natural candidate for subword tokenisation.

2.2 BPE tokenizer and token frequencies

A BPE tokenizer is trained from scratch using the *tokenizers* library with:

- Model: BPE with special tokens *[PAD]* and *[UNK]*;
- Pre-tokeniser: whitespace splitting;
- Vocabulary size: 500.

The learned vocabulary size is exactly 500, including characters, subwords, and full words. The entire corpus is then encoded into a tensor of token identifiers.

Token frequencies are computed with a *Counter* over all token identifiers. The twenty most frequent tokens are

⋆e-mail: lb3656@columbia.edu

shown in Fig. 1. The most common symbol is the comma, which appears almost 20 000 times, followed by other punctuation and short subwords such as *s*, *∴*, *.*, *I*, and *d*. Even the twentieth token, *be*, appears more than 3 500 times.

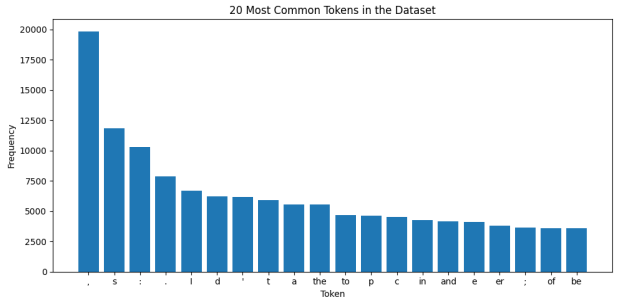


Figure 1. Twenty most common tokens in the dataset. Punctuation and short subwords dominate the high-frequency region.

2.3 Classical Words vs. BPE Vocabulary

To benchmark the BPE vocabulary against traditional linguistic counts, we pre-tokenized the corpus into whitespace-separated units and extracted the 100 most frequent lowercased types. This "classical" list includes standard words as well as contraction artifacts (e.g., *s*, *d*, *ll*) resulting from punctuation splitting.

The BPE vocabulary successfully captures 97 of these top 100 forms as single tokens, including high-frequency terms like *the*, *and*, and *you*. Notably, only three context-specific proper nouns failed to merge into single tokens:

duke, *richard*, *queen*.

While this indicates high coverage for the most common stopwords, the prevalence of single-character tokens in the broader distribution implies that the model retains a heavy reliance on subword composition for less frequent terms—a necessary trade-off for a compact 500-entry vocabulary.

3 Model and training setup

3.1 Sequence construction and data loaders

For each experiment, the token stream is converted into overlapping sequences for next-token prediction. For a chosen sequence length *seq_len*, each input example is

$$x_i = \text{ids}[i : i + \text{seq\_len}],$$

and the corresponding target is

$$y_i = \text{ids}[i + 1 : i + \text{seq\_len} + 1].$$

This sliding-window construction yields len(ids) − seq_len supervised examples.

An 80/20 split is performed per experiment, producing roughly 358 000 training and 89 000 validation sequences for *seq_len* = *64*. The resulting tensors are wrapped in a custom *ShakespeareDataset* and loaded with *DataLoader* objects (batch size 512, shuffled for training).

3.2 Architecture

The language model, *TinyTransformerLM*, is a small decoder-only Transformer with:

- Token embeddings of dimension *d_model*;
- Fixed sinusoidal positional encodings of length *max_seq_len*, added to token embeddings;
- Two Transformer blocks, each with:
  - Multi-head self-attention with *n_heads* and head dimension *d_model* / *n_heads*;
  - RMSNorm before attention and before the feed-forward sublayer;
  - A feed-forward network of dimension *d_ff* with GELU activation;
  - Residual connections around attention and feed-forward components;
  - A causal mask implemented as a lower-triangular matrix, ensuring each position attends only to itself and previous tokens.
- An output RMSNorm followed by a linear projection to vocabulary logits.

When called with the *return_attn* flag, each block returns both the transformed representations and the attention weights, which enables downstream visualisation.

3.3 Training procedure

Training is framed as standard next-token prediction:

- **Loss:** Cross-entropy between logits and target token identifiers, averaged over tokens;
- **Optimiser:** AdamW with configuration-specific learning rate;
- **Scheduler:** *ReduceLROnPlateau* halves the learning rate when validation loss stalls;
- **Mixed Precision:** *autocast* on CUDA plus a *GradScaler*, combined with gradient clipping at norm 1.0;
- **Early Stopping:** If validation perplexity fails to improve for one epoch, training stops and the best model state is stored.

For each epoch, the code logs training loss, validation loss, and validation perplexity. These histories are retained for plotting and comparison.

3.4 Hyperparameter experiments

Three configurations are explored:

- **Baseline:** A standard lightweight model with sequence length 64, embedding dimension (d_{model}) 128, 4 heads, and a learning rate of 3×10^{-4} (approx. 392k parameters).

- **Medium Context:** Identical architecture to the Baseline, but doubles the sequence length to **128**. Since positional encodings are fixed, the parameter count remains unchanged.
- **Wider Model:** Increases capacity while reverting to sequence length 64. We double the width ($d_{model} = 256$) and heads ($n_{heads} = 8$), and reduce the learning rate to 1×10^{-4} to ensure stability. This results in approx. 1.3M parameters.

All models are trained for at most 20 epochs with early stopping based on validation perplexity. Figure 2 shows the validation perplexity curves on a logarithmic y-axis.

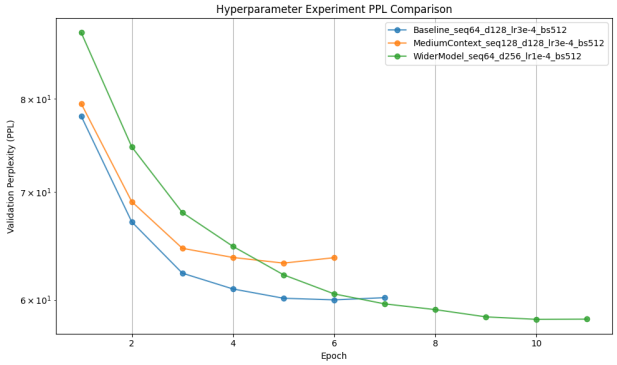


Figure 2. Validation perplexity trajectories for the three hyperparameter configurations. The wider model (green) starts worse but eventually achieves the lowest perplexity.

3.5 Summary of best validation perplexities

Table 1 summarises the best validation perplexity and the epoch at which it occurs for each experiment. This small table is used later when selecting the model for analysis.

Table 1. Best validation perplexity per experiment.

Configuration	Best PPL	Epoch
Baseline_seq64_d128_lr3e-4	60.00	6
MediumContext_seq128_d128_lr3e-4	63.24	5
WiderModel_seq64_d256_lr1e-4	58.35	10

4 Results

4.1 Best model training dynamics

The wider configuration, *WiderModel_seq64_d256_lr1e-4_bs512*, achieves the lowest validation perplexity and is therefore selected for detailed analysis. Figure 3 shows the training and validation loss curves, together with the validation perplexity.

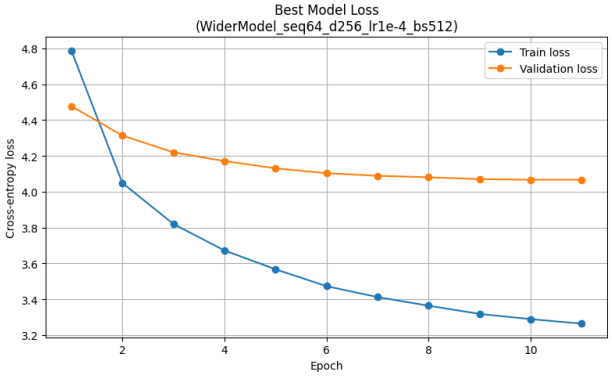


Figure 3. Cross-entropy loss of the model on train and validation sets.

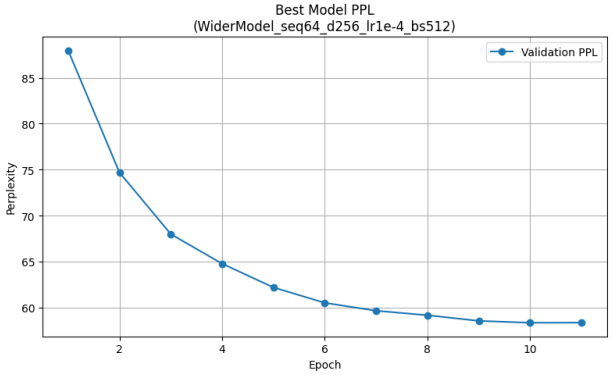


Figure 4. Validation perplexity over epochs.

Training loss drops from roughly 4.8 to 3.25, while validation loss decreases from about 4.45 to 4.07 before flattening. Validation perplexity falls monotonically from roughly 88 to 58.35 by epoch 10. The gap between train and validation loss is moderate, indicating some overfitting but no collapse.

4.2 Attention maps

Attention maps are extracted from the best model on a single validation batch. Figures 5–8 show four heads: layer 0 head 0, layer 0 head 1, layer 1 head 0, and layer 1 head 1.

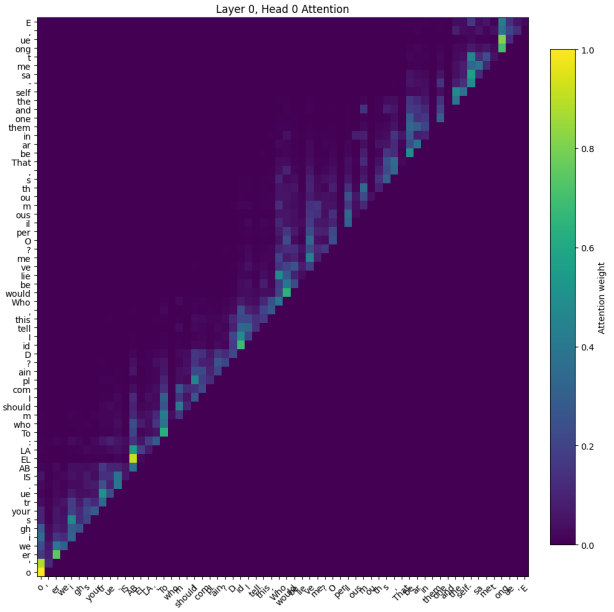


Figure 5. Attention weights for layer 0, head 0. A strong diagonal shows that each token attends most to itself and nearby neighbours.

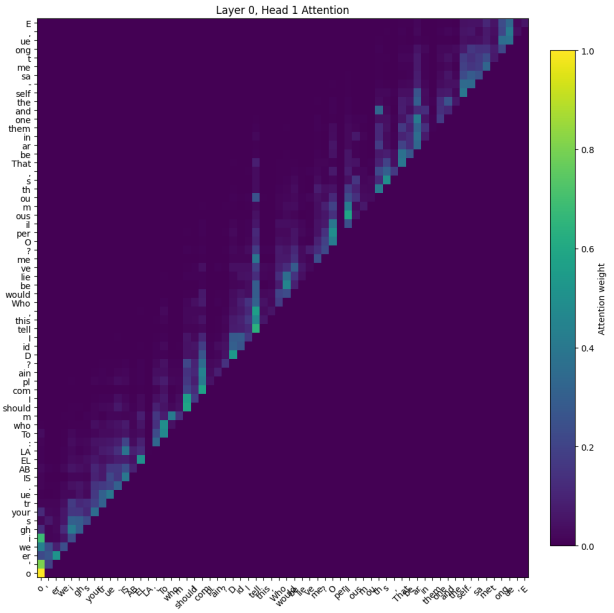


Figure 6. Attention weights for layer 0, head 1. Local attention is still dominant, with clear vertical stripes on punctuation and clause markers.

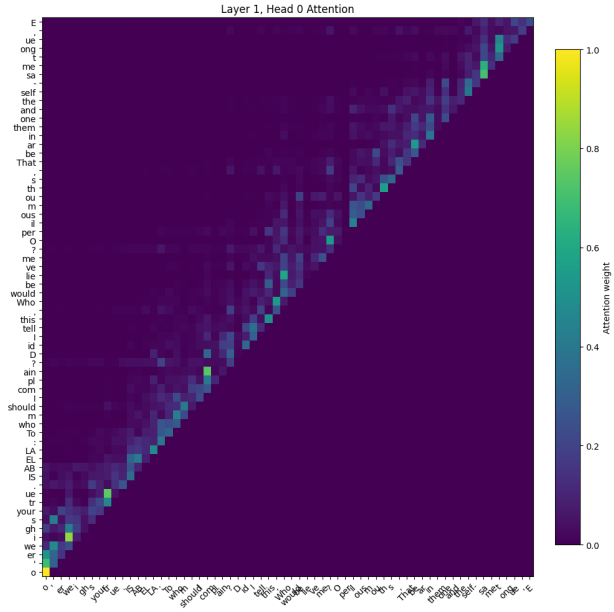


Figure 7. Attention weights for layer 1, head 0. The diagonal band becomes broader, indicating integration over slightly longer ranges.

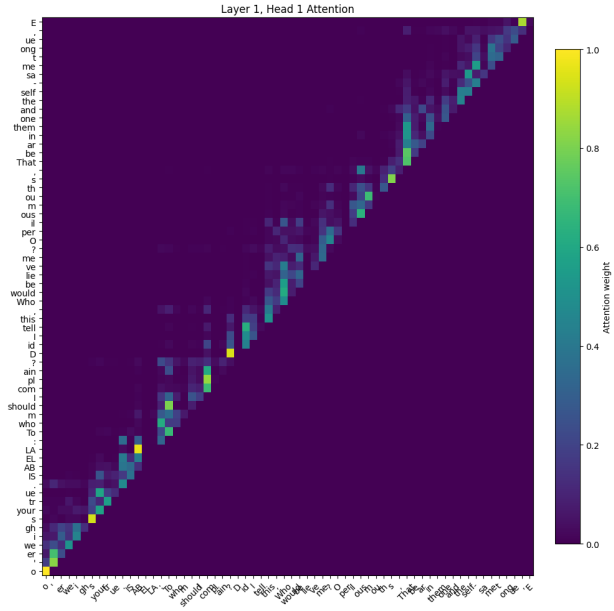


Figure 8. Attention weights for layer 1, head 1. In addition to the diagonal, strong vertical stripes highlight punctuation and discourse anchors.

The attention maps display a consistent diagonal structure, reflecting the model's focus on local context. Interspersed with this pattern are vertical 'stripes' where the model attends heavily to specific structural tokens. These usually correspond to punctuation like *?* and *:* or starting words like *Who* and *To*, serving as markers for new clauses or speaker turns.

4.3 Qualitative predictions

Inference utilities are implemented to sample text and to inspect the distribution over the next token. When prompted with character names followed by a colon, such as *MENENIUS:*, *BRUTUS:*, and *CORIOLANUS:*, the

model places most probability mass on tokens like *I*, *O*, *The*, and *What*. This is consistent with the structure of a play script, where lines often begin with interjections, pronouns, or short function words.

The probabilities themselves are moderate (typically below 0.15), reflecting the fact that several plausible continuations exist. The model does not collapse onto a single token but spreads mass across a handful of reasonable options.

5 Discussion

5.1 Patterns in the attention maps

The attention visualisations reveal several consistent behaviours:

- **Locality and self-focus.** All heads exhibit a strong diagonal, indicating that each token attends heavily to itself and to nearby tokens. This mirrors an adaptive n-gram model where the most recent context is most important.
- **Boundary detection.** Vertical stripes aligned with punctuation and certain words suggest heads that specialise in boundary detection. Tokens such as *?*, *:*, and clause-initial words act as anchors to which later tokens refer.
- **Layer-wise specialisation.** Layer 0 heads are more tightly concentrated around the diagonal, while layer 1 heads distribute mass over a broader region, combining local information with slightly longer-range interactions.
- **Redundancy with variation.** Although all heads share some diagonal structure, they differ in how sharply they spike and where they place additional mass. This redundancy provides robustness while still allowing specialisation.

5.2 Hyperparameters and training stability

Among the hyperparameters explored, learning rate has the strongest effect on stability. The two narrower models with learning rate 3×10^{-4} both stop after six or seven epochs, with validation perplexity flattening or slightly increasing. The wider model with learning rate 1×10^{-4} shows a smooth, monotonic decrease in validation perplexity for ten epochs.

Changing the context length from 64 to 128 tokens at fixed width degrades final perplexity and makes training slightly less smooth. The longer sequences increase the per-batch computational cost and reduce the number of parameter updates within a fixed time budget, without delivering a clear modelling benefit in this setup.

Model size mainly affects the achievable perplexity. Increasing *d_model* from 128 to 256, coupled with a smaller learning rate, improves final perplexity from 60.00 to 58.35 and yields the best behaved training curve.

5.3 Evolution of attention

Attention maps are only inspected after training, but their structure combined with the loss curves suggests a two-phase learning process. Early in training, query and key projections are close to random, leading to nearly uniform attention over the causal past. As the loss rapidly decreases over the first few epochs, the model learns to concentrate attention along the diagonal, exploiting local dependencies. Later epochs refine which positions act as global anchors, yielding the clear vertical stripes seen in the final maps.

5.4 Role of positional encodings

Sinusoidal positional encodings are added to token embeddings at the input of the network. Their role is essential:

- Without positional information, the model would know which tokens appear in the prefix but not in which order. Causal masking alone cannot distinguish different permutations of the same multiset.
- The strong diagonal in the attention maps relies on the ability to distinguish "immediately previous" from "ten steps back". Sinusoidal encodings encode relative offsets in a way that linear projections can recover.
- In practice, eliminating positional encodings would severely harm perplexity and destroy the clean diagonal structure, turning the model into something closer to a bag-of-words predictor over the prefix.

5.5 Runtime and memory footprint

The main computational and memory bottlenecks align with standard Transformer scaling:

- **Self-attention complexity.** The attention mechanism exhibits quadratic scaling with respect to sequence length. Memory usage is $O(B \cdot H \cdot L^2)$ (where H is the number of heads), while computation scales as $O(B \cdot L^2 \cdot d_{\text{model}})$. Consequently, doubling the sequence length asymptotically quadruples the computational and memory cost of the attention sub-layer.
- **Activation storage.** Although the parameter counts are modest, activation tensors for sequences of length 64 or 128 with batch size 512 dominate GPU memory, especially during backpropagation. Mixed precision alleviates this pressure.
- **Dataset materialisation.** All input and target sequences are precomputed and stored in memory for convenience, which is acceptable at this scale but would become a bottleneck for much longer corpora or context lengths.
- **Validation passes.** Full validation over almost 90 000 sequences each epoch is non-trivial. Early stopping and the relatively small number of epochs keep the total cost manageable.

Overall, the quadratic attention cost and activation storage, rather than the raw parameter count, are the dominant constraints. These considerations explain why increasing context length is more expensive than increasing width in this small-scale setting.

6 Conclusion

This project implemented a compact Transformer language model for Shakespearean text and used it to examine tokenisation, attention, and training dynamics. A BPE tokenizer with a 500-entry vocabulary efficiently captured almost all of the top 100 frequent words as single tokens, leaving only a few proper names to be composed from subwords.

On top of this representation, three Transformer configurations were trained. The wider model with *d_model = 256* and a reduced learning rate achieved the lowest validation perplexity and the smoothest training curve. Increasing context length at fixed width did not improve performance and incurred significantly higher computational cost.

Attention visualisations revealed interpretable patterns: strong local diagonals, heads specialised to punctuation and clause boundaries, and layer-wise differences between tight local focus and broader context integration. These behaviours depend critically on positional encodings, which allow the model to compare positions along the sequence.

Runtime and memory usage are driven primarily by the quadratic cost of self-attention in sequence length and by activation storage. Mixed precision, moderate context lengths, and early stopping keep the experiments tractable on a single GPU, while still producing a model capable of plausible Shakespeare-style continuations and meaningful next-token probabilities.

7 AI tool usage disclosure

I used ChatGPT to speed up technical editing, check plotting syntax, and pressure test explanations. Code and text were always inspected, verified, and adjusted by me. The analytical decisions, feature definitions, and model comparisons reflect my work on the dataset.

COMSW4995 - Applied Machine Learning - HW3:

Luca Barattini - @lb3656:

Data Preparation:

Importing our libraries:

In [ ]:
```
# ================================================================
# 0. IMPORTS & SETUP
# ================================================================
import math
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from tokenizers import Tokenizer, models, trainers, pre_tokenizers
import matplotlib.pyplot as plt
import numpy as np
from collections import Counter
import re
import random
```

In [ ]:
```
# ================================================================
# 0b. GLOBAL CONFIG & SEEDING
# ================================================================
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

Data Loading and tokenization:

In [ ]:
```
# ================================================================
# 1a. LOADING RAW TEXT
# ================================================================
text_path = "input.txt"
with open(text_path, "r", encoding="utf-8") as f:
    text = f.read()

# ================================================================
# 1b. INITIAL DATA INSPECTION
# ================================================================
len_in_mm = len(text)/1_000_000
print(f"Length of dataset in characters: {len_in_mm:.1f} MM")
print("----------------------------")
print("\n--- First 1000 characters ---")
print(text[:1000])
print("----------------------------")

# ================================================================
# 1c. TOKENIZER TRAINING
# ================================================================
print("--- Training Tokenizer (vocab_size=500) ---")
tokenizer = Tokenizer(models.BPE(unk_token="[UNK]"))
tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()
trainer = trainers.BpeTrainer(vocab_size=500, special_tokens=["[PAD]", "[UNK]"])
tokenizer.train_from_iterator([text], trainer=trainer)
vocab_size = tokenizer.get_vocab_size()
print(f"Tokenizer trained. Vocab size: {vocab_size}")

# ================================================================
# 1d. ENCODING TEXT TO TENSORS
# ================================================================
encoded = tokenizer.encode(text)
ids = torch.tensor(encoded.ids, dtype=torch.long)
print(f"Total tokens in dataset: {len(ids):,}")
```

```
Length of dataset in characters: 1.1 MM
------------------------------

--- First 1000 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 know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
but they think we are too dear: the leanness that
afflicts us, the object of our misery, is as an
inventory to particularise their abundance; our
sufferance is a gain to them Let us revenge this with
our pikes, ere we become rakes: for the gods know I
speak this in hunger for bread, not in thirst for revenge.


------------------------------
--- Training Tokenizer (vocab_size=500) ---
Tokenizer trained. Vocab size: 500
Total tokens in dataset: 447,717
```

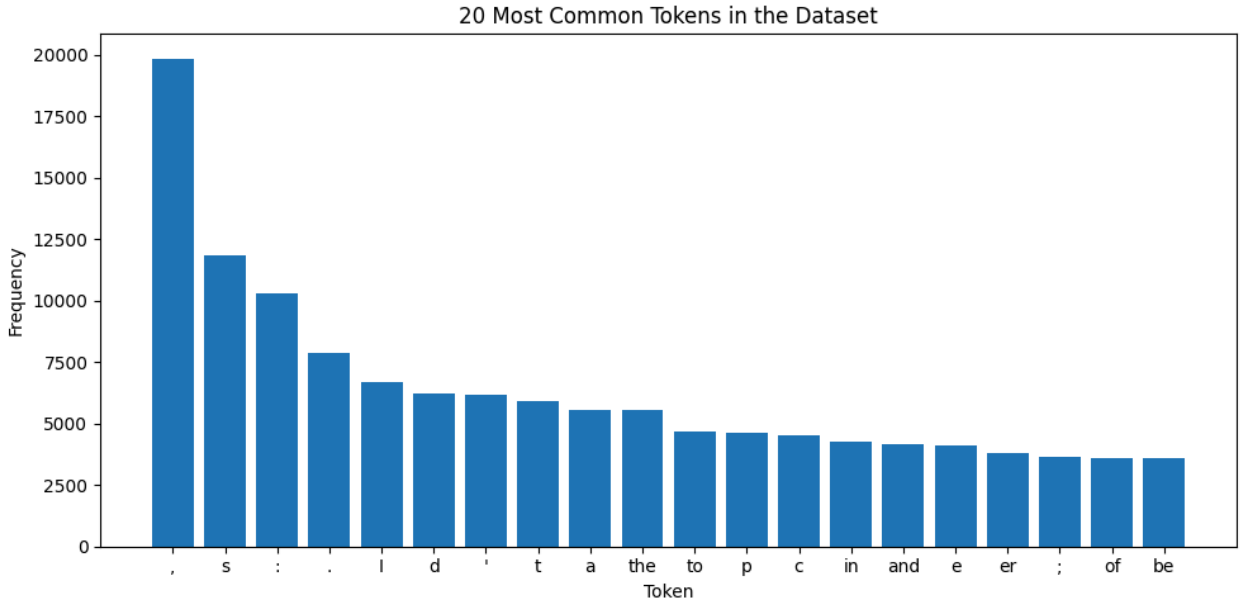
Data and Tokenizer Sanity Check:

In [ ]:

```python
# ======================================================================
# 2a. BASIC TEXT STATS
# ======================================================================
print("\n--- 1. Basic Text Stats ---")
chars = sorted(list(set(text)))
vocab_size_chars = len(chars)
print(f"Total characters in text: {len(text):,}")
print(f"Total unique characters: {vocab_size_chars}")

# ======================================================================
# 2b. TOKENIZER SANITY CHECK (TOP 20 TOKENS)
# ======================================================================
print("\n--- 2. Tokenizer Sanity Check (Top 20 Tokens) ---")
token_counts = Counter(encoded.ids)
most_common_tokens = token_counts.most_common(20)
token_ids = [token[0] for token in most_common_tokens]
token_names = [tokenizer.id_to_token(tid) for tid in token_ids]
token_freq = [token[1] for token in most_common_tokens]
print("Top 20 most common tokens (data for chart):")
print(dict(zip(token_names, token_freq)))

# ======================================================================
# 2c. PLOT TOKEN FREQUENCY
# ======================================================================
plt.figure(figsize=(10, 5))
plt.bar(token_names, token_freq)
plt.xlabel("Token")
plt.ylabel("Frequency")
plt.title("20 Most Common Tokens in the Dataset")
plt.xticks(rotation='horizontal')
plt.tight_layout()
plt.show()

# ======================================================================
# 2d. CLASSICAL VS. BPE VOCAB ANALYSIS
# ======================================================================
print("\n--- 3. Classical vs. BPE Vocab Analysis ---")
```

```
pre_tokenized_list = tokenizer.pre_tokenizer.pre_tokenize_str(text)
classical_words = [word.lower() for word, offset in pre_tokenized_list]
word_counts = Counter(classical_words)
top_100_classical_words = word_counts.most_common(100)

print("Top 100 most frequent 'classical' words found:")
print(top_100_classical_words)
print("----------------------------------------")

bpe_vocab_dict = tokenizer.get_vocab()
bpe_vocab_set = set(bpe_vocab_dict.keys())
print(f"Total BPE vocab size: {len(bpe_vocab_set)} (includes letters, subwords, etc.)")

top_100_word_set = set([word for word, count in top_100_classical_words])
missed_words = top_100_word_set.difference(bpe_vocab_set)
captured_words = top_100_word_set.intersection(bpe_vocab_set)

print(f"Words in Top 100 that BPE *captured* as single tokens: \n{captured_words}")
print(f"\nWords in Top 100 that BPE *missed* (and must build from pieces): \n{missed_words}")
```

```
--- 1. Basic Text Stats ---
Total characters in text: 1,115,394
Total unique characters: 65

--- 2. Tokenizer Sanity Check (Top 20 Tokens) ---
Top 20 most common tokens (data for chart):
{',': 19846, 's': 11828, ':': 10316, '.': 7885, 'I': 6699, 'd': 6224, "'": 6187, 't': 5916, 'a': 5567, 'the': 5540, 'to':
4688, 'p': 4598, 'c': 4516, 'in': 4238, 'and': 4179, 'e': 4117, 'er': 3800, ';': 3628, 'of': 3617, 'be': 3568}
```




```
--- 3. Classical vs. BPE Vocab Analysis ---
Top 100 most frequent 'classical' words found:
[(',', 19602), (':', 10274), ('.', 7811), ('the', 6287), ("'", 5924), ('and', 5690), ('i', 5111), ('to', 4934), ('of', 376
0), (';', 3598), ('you', 3211), ('my', 3120), ('a', 3018), ('that', 2664), ('?', 2419), ('in', 2403), ('is', 2118), ('!',
2105), ('not', 2015), ('for', 1926), ('s', 1859), ('with', 1813), ('it', 1773), ('me', 1769), ('be', 1710), ('your', 168
6), ('he', 1606), ('his', 1552), ('this', 1509), ('but', 1507), ('have', 1450), ('d', 1445), ('thou', 1421), ('as', 1420),
('what', 1211), ('him', 1209), ('so', 1177), ('thy', 1059), ('will', 1053), ('-', 1005), ('we', 938), ('king', 925), ('b
y', 911), ('all', 910), ('no', 906), ('shall', 849), ('her', 829), ('if', 807), ('do', 799), ('our', 786), ('are', 785),
('thee', 762), ('o', 751), ('lord', 711), ('on', 701), ('now', 701), ('good', 672), ('come', 624), ('from', 624), ('sir',
597), ('or', 592), ('which', 587), ('more', 582), ('ll', 580), ('then', 573), ('at', 561), ('here', 561), ('she', 546),
('they', 542), ('would', 535), ('was', 533), ('let', 528), ('how', 520), ('well', 516), ('than', 480), ('their', 476), ('t
hem', 474), ('duke', 471), ('say', 459), ('am', 457), ('hath', 454), ('when', 437), ('there', 437), ('love', 432), ('one',
427), ('go', 425), ('were', 413), ('may', 409), ('make', 400), ('us', 399), ('upon', 399), ('like', 391), ('richard', 38
8), ('yet', 387), ('man', 381), ('queen', 380), ('know', 378), ('an', 374), ('must', 370), ('should', 367)]
----------------------------------------
Total BPE vocab size: 500 (includes letters, subwords, etc.)
Words in Top 100 that BPE *captured* as single tokens:
{'me', 'it', 'may', 'we', 'like', 's', 'must', 'shall', ':', "'", 'but', 'than', 'll', 'good', 'no', 'by', 'for', '!', 'th
y', ';', 'man', 'then', 'them', 'this', 'an', 'love', 'one', 'you', 'let', 'more', 'am', 'would', 'know', 'now', 'the', 'w
ell', 'a', 'in', 'i', 'do', 'come', 'she', 'that', 'there', 'if', 'upon', 'have', 'on', 'they', 'us', 'thee', 'our', 'lor
d', 'from', 'my', 'as', 'your', 'her', 'how', 'hath', 'yet', '-', 'is', 'are', ',', 'be', 'at', 'here', 'd', 'o', 'make',
'of', 'his', 'when', 'their', 'say', 'should', 'he', 'sir', 'all', 'and', 'so', 'or', 'not', 'were', 'which', 'will', 'tho
u', 'was', 'to', 'what', 'king', 'go', '?', 'him', 'with', '.'}

Words in Top 100 that BPE *missed* (and must build from pieces):
{'queen', 'duke', 'richard'}
```

Model and Function Definition:

In [ ]:

```
# ================================================================
# 3a. DATASET CLASS DEFINITION
# ================================================================
class ShakespeareDataset(Dataset):
```

```
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __len__(self):
        return len(self.x)
    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

# ================================================================
# 3b. RMSNORM LAYER DEFINITION
# ================================================================
class RMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-8):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))
    def forward(self, x):
        norm = x.pow(2).mean(-1, keepdim=True)
        return x * torch.rsqrt(norm + self.eps) * self.weight

# ================================================================
# 3c. MULTI-HEAD SELF-ATTENTION DEFINITION
# ================================================================
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0, "d_model must be divisible by n_heads"

        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

        self.qkv_proj = nn.Linear(d_model, 3 * d_model, bias=False)
        self.out_proj = nn.Linear(d_model, d_model, bias=False)

    def forward(self, x, causal_mask=None):
        batch_size, seq_len, _ = x.shape

        qkv = self.qkv_proj(x)
        qkv = qkv.reshape(batch_size, seq_len, 3, self.n_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        Q, K, V = qkv[0], qkv[1], qkv[2]

        scores = (Q @ K.transpose(-2, -1)) / math.sqrt(self.head_dim)

        if causal_mask is not None:
            mask_expanded = causal_mask.unsqueeze(0).unsqueeze(0)
            scores = scores.masked_fill(mask_expanded == 0, float("-inf"))

        attn_weights = torch.softmax(scores, dim=-1)
        attn_weights = torch.nan_to_num(attn_weights, 0.0)

        out = (attn_weights @ V).transpose(1, 2).reshape(batch_size, seq_len, self.d_model)
        out = self.out_proj(out)

        return out, attn_weights

# ================================================================
# 3d. TRANSFORMER BLOCK DEFINITION
# ================================================================
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, seq_len):
        super().__init__()
        self.attn = MultiHeadSelfAttention(d_model, n_heads)
        self.norm1 = RMSNorm(d_model)
        self.norm2 = RMSNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model),
        )
        mask = torch.tril(torch.ones(seq_len, seq_len))
        self.register_buffer("causal_mask", mask)

    def forward(self, x, return_attn=False):
        seq_len = x.size(1)
        causal_mask = self.causal_mask[:seq_len, :seq_len].to(x.device)

        h = self.norm1(x)
        attn_out, attn_weights = self.attn(h, causal_mask)
        x = x + attn_out

        h = self.norm2(x)
        ff_out = self.ff(h)
        x = x + ff_out

        if return_attn:
            return x, attn_weights
        return x, None

# ================================================================
# 3e. FULL TRANSFORMER MODEL DEFINITION
```

```
# ================================================================
class TinyTransformerLM(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, d_ff, max_seq_len, n_layers):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size, d_model)

        pos = torch.arange(max_seq_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_seq_len, d_model)
        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
        self.register_buffer("pos_encoding", pe.unsqueeze(0))

        self.layers = nn.ModuleList(
            [TransformerBlock(d_model, n_heads, d_ff, max_seq_len) for _ in range(n_layers)]
        )
        self.norm = RMSNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, x, return_attn=False):
        bsz, seq_len = x.size()

        pos_encoding = self.pos_encoding[:, :seq_len, :].to(x.device)
        tok = self.token_embedding(x)
        h = tok + pos_encoding

        attn_maps = []
        for layer in self.layers:
            h, attn = layer(h, return_attn=return_attn)
            if return_attn:
                attn_maps.append(attn)

        h = self.norm(h)
        logits = self.head(h)

        if return_attn:
            return logits, attn_maps
        return logits

# ================================================================
# 3f. INFERENCE & PLOTTING FUNCTIONS
# ================================================================
def clean_generated(text):
    text = re.sub(r"\s+([.,!?;:])", r"\1", text)
    text = re.sub(r"\(\s+", "(", text)
    text = re.sub(r"\s+\)", ")", text)
    text = re.sub(r"\s+'", "'", text)
    text = re.sub(r"'\s+", "'", text)
    text = re.sub(r"\s+", " ", text)
    return text.strip()

def generate_text(model, tokenizer, device, prompt, max_new_tokens=50, temperature=0.9, seq_len=128):
    model.eval()
    with torch.no_grad():
        encoded = tokenizer.encode(prompt)
        prompt_ids = encoded.ids
        ids = torch.tensor(prompt_ids, dtype=torch.long, device=device).unsqueeze(0)
        for _ in range(max_new_tokens):
            if ids.size(1) > seq_len:
                ids_cond = ids[:, -seq_len:]
            else:
                ids_cond = ids
            logits = model(ids_cond)
            next_token_logits = logits[0, -1] / temperature
            probs = torch.softmax(next_token_logits, dim=-1)
            next_id = torch.multinomial(probs, num_samples=1)
            ids = torch.cat([ids, next_id.unsqueeze(0)], dim=1)

        out_ids = ids[0].cpu().tolist()
        text = tokenizer.decode(out_ids)
        return clean_generated(text)

def predict_next_tokens(model, tokenizer, device, prompt, top_k=5, seq_len=128):
    model.eval()
    with torch.no_grad():
        encoded = tokenizer.encode(prompt)
        ids = torch.tensor(encoded.ids, dtype=torch.long, device=device).unsqueeze(0)
        if ids.size(1) > seq_len:
            ids = ids[:, -seq_len:]
        logits = model(ids)
        next_logits = logits[0, -1]
        probs = torch.softmax(next_logits, dim=-1)
        top_probs, top_idx = torch.topk(probs, min(top_k, vocab_size))
        tokens = [tokenizer.id_to_token(i.item()) for i in top_idx]
        return list(zip(tokens, top_probs.cpu().tolist()))

def show_attention_heatmap(model, tokenizer, data_loader, device, layer=0, head=0):
    model.eval()
    with torch.no_grad():
        x, y = next(iter(data_loader))
        x = x[:1].to(device)
        logits, attn_maps = model(x, return_attn=True)
```

```
    attn = attn_maps[layer][0, head].cpu().numpy()

    ids = x[0].cpu().tolist()
    tokens = [tokenizer.id_to_token(i) for i in ids]

    plt.figure(figsize=(10, 10))
    im = plt.imshow(attn, aspect="auto", origin="lower")
    plt.xticks(range(len(tokens)), tokens, rotation=45)
    plt.yticks(range(len(tokens)), tokens)
    plt.title(f"Layer {layer}, Head {head} Attention")
    cbar = plt.colorbar(im, fraction=0.046, pad=0.04)
    cbar.set_label("Attention weight")
    plt.tight_layout()
    plt.show()
```

Hyperparameter and experiment setup:

In [ ]:

```
# ======================================================================
# 4. HYPERPARAMETER EXPERIMENT SETUP
# ======================================================================
all_results = []
max_epochs = 20

experiments = [
    {
        "name": "Baseline_seq64_d128_lr3e-4_bs512",
        "seq_len": 64,
        "d_model": 128,
        "n_layers": 2,
        "n_heads": 4,
        "d_ff": 256,
        "lr": 3e-4,
        "batch_size": 512,
        "num_epochs": max_epochs,
        "early_stop_patience": 1,
    },
    {
        "name": "MediumContext_seq128_d128_lr3e-4_bs512",
        "seq_len": 128,
        "d_model": 128,
        "n_layers": 2,
        "n_heads": 4,
        "d_ff": 256,
        "lr": 3e-4,
        "batch_size": 512,
        "num_epochs": max_epochs,
        "early_stop_patience": 1,
    },
    {
        "name": "WiderModel_seq64_d256_lr1e-4_bs512",
        "seq_len": 64,
        "d_model": 256,
        "n_layers": 2,
        "n_heads": 8,
        "d_ff": 512,
        "lr": 1e-4,
        "batch_size": 512,
        "num_epochs": max_epochs,
        "early_stop_patience": 1,
    },
]
```

Main model training:

In [ ]:

```
# ======================================================================
# 5. EXPERIMENT LOOP
# ======================================================================

for config in experiments:

    print("\n" + "="*80)
    print(f"STARTING EXPERIMENT: {config['name']}")
    print(config)
    print("="*80)

    # ======================================================================
    # 5a. DATA PREPARATION (PER-EXPERIMENT)
    # ======================================================================
    seq_len = config["seq_len"]
    batch_size = config["batch_size"]

    num_sequences = len(ids) - seq_len
    inputs = torch.stack([ids[i:i+seq_len] for i in range(num_sequences)])
    targets = torch.stack([ids[i+1:i+seq_len+1] for i in range(num_sequences)])
```

```
split_idx = int(0.8 * len(inputs))
train_inputs = inputs[:split_idx]
val_inputs = inputs[split_idx:]
train_targets = targets[:split_idx]
val_targets = targets[split_idx:]

print(f"Data split: {len(train_inputs):,} training samples, {len(val_inputs):,} validation samples (80/20 split)")

train_dataset = ShakespeareDataset(train_inputs, train_targets)
val_dataset = ShakespeareDataset(val_inputs, val_targets)

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

print(f"Data prepared: seq_len={seq_len}, batch_size={batch_size}")

# ======================================================================
# 5b. MODEL & TRAINING INITIALIZATION (PER-EXPERIMENT)
# ======================================================================
d_model = config["d_model"]
n_heads = config["n_heads"]
d_ff = config["d_ff"]
n_layers = config["n_layers"]

model = TinyTransformerLM(
    vocab_size=vocab_size,
    d_model=d_model,
    n_heads=n_heads,
    d_ff=d_ff,
    max_seq_len=seq_len,
    n_layers=n_layers,
).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=config["lr"])
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="min", factor=0.5, patience=1
)
scaler = torch.cuda.amp.GradScaler(enabled=(device.type == "cuda"))

train_losses = []
val_losses = []
val_ppls = []

best_val_ppl = float('inf')
best_model_state = None
epochs_without_improvement = 0
early_stop_patience = config.get("early_stop_patience", 1)

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

# ======================================================================
# 5c. MODEL TRAINING LOOP (PER-EXPERIMENT)
# ======================================================================
for epoch in range(1, config["num_epochs"] + 1):

    model.train()
    total_train_loss = 0.0
    total_tokens = 0

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

        optimizer.zero_grad(set_to_none=True)
        with torch.cuda.amp.autocast(enabled=(device.type == "cuda")):
            logits = model(x)
            loss = criterion(logits.view(-1, vocab_size), y.view(-1))

        scaler.scale(loss).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()

        total_train_loss += loss.item() * x.numel()
        total_tokens += x.numel()

    avg_train_loss = total_train_loss / total_tokens
    train_losses.append(avg_train_loss)

    model.eval()
    total_val_loss = 0.0
    total_val_tokens = 0
    with torch.no_grad():
        for x, y in val_loader:
            x = x.to(device)
            y = y.to(device)
            with torch.cuda.amp.autocast(enabled=(device.type == "cuda")):
                logits = model(x)
                logits_flat = logits.view(-1, vocab_size)
                y_flat = y.view(-1)
```

```
                    loss = criterion(logits_flat, y_flat)

                total_val_loss += loss.item() * x.numel()
                total_val_tokens += x.numel()

        avg_val_loss = total_val_loss / total_val_tokens
        val_losses.append(avg_val_loss)
        val_ppl = math.exp(avg_val_loss)
        val_ppls.append(val_ppl)

        scheduler.step(avg_val_loss)

        print(f"Epoch {epoch}/{config['num_epochs']}: train_loss={avg_train_loss:.4f}, val_loss={avg_val_loss:.4f}, val_p

        if val_ppl < best_val_ppl:
            best_val_ppl = val_ppl
            best_model_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().items()}
            epochs_without_improvement = 0
            print(f"  ^ New best model saved at epoch {epoch}!")
        else:
            epochs_without_improvement += 1
            print(f"  No improvement in PPL for {epochs_without_improvement} epoch(s).")
            if epochs_without_improvement >= early_stop_patience:
                print(f"  Early stopping triggered after {epochs_without_improvement} epoch(s) without improvement.")
                break

    # ======================================================================
    # 5d. STORE RESULTS FOR THIS EXPERIMENT
    # ======================================================================
    all_results.append({
        "name": config["name"],
        "config": config,
        "best_ppl": best_val_ppl,
        "best_epoch": val_ppls.index(best_val_ppl) + 1,
        "val_ppls_history": val_ppls,
        "train_loss_history": train_losses,
        "best_model_state": best_model_state,
        "val_loader": val_loader,
    })
```

```
================================================================================
STARTING EXPERIMENT: Baseline_seq64_d128_lr3e-4_bs512
{'name': 'Baseline_seq64_d128_lr3e-4_bs512', 'seq_len': 64, 'd_model': 128, 'n_layers': 2, 'n_heads': 4, 'd_ff': 256, 'l
r': 0.0003, 'batch_size': 512, 'num_epochs': 20, 'early_stop_patience': 1}
================================================================================
Data split: 358,122 training samples, 89,531 validation samples (80/20 split)
Data prepared: seq_len=64, batch_size=512
/tmp/ipython-input-2953953368.py:60: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is deprecated. Please use `torch.
amp.GradScaler('cuda', args...)` instead.
  scaler = torch.cuda.amp.GradScaler(enabled=(device.type == "cuda"))
/tmp/ipython-input-2953953368.py:87: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.am
p.autocast('cuda', args...)` instead.
  with torch.cuda.amp.autocast(enabled=(device.type == "cuda")):
Model initialized. Parameters: 391,552
/tmp/ipython-input-2953953368.py:109: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.a
mp.autocast('cuda', args...)` instead.
  with torch.cuda.amp.autocast(enabled=(device.type == "cuda")):
```

```
Epoch 1/20: train_loss=4.5704, val_loss=4.3575, val_ppl=78.06
  ^ New best model saved at epoch 1!
Epoch 2/20: train_loss=3.8179, val_loss=4.2060, val_ppl=67.09
  ^ New best model saved at epoch 2!
Epoch 3/20: train_loss=3.5484, val_loss=4.1325, val_ppl=62.34
  ^ New best model saved at epoch 3!
Epoch 4/20: train_loss=3.4100, val_loss=4.1099, val_ppl=60.94
  ^ New best model saved at epoch 4!
Epoch 5/20: train_loss=3.3231, val_loss=4.0966, val_ppl=60.14
  ^ New best model saved at epoch 5!
Epoch 6/20: train_loss=3.2484, val_loss=4.0943, val_ppl=60.00
  ^ New best model saved at epoch 6!
Epoch 7/20: train_loss=3.2001, val_loss=4.0975, val_ppl=60.19
  No improvement in PPL for 1 epoch(s).
  Early stopping triggered after 1 epoch(s) without improvement.

================================================================================
STARTING EXPERIMENT: MediumContext_seq128_d128_lr3e-4_bs512
{'name': 'MediumContext_seq128_d128_lr3e-4_bs512', 'seq_len': 128, 'd_model': 128, 'n_layers': 2, 'n_heads': 4, 'd_ff': 25
6, 'lr': 0.0003, 'batch_size': 512, 'num_epochs': 20, 'early_stop_patience': 1}
================================================================================
Data split: 358,071 training samples, 89,518 validation samples (80/20 split)
Data prepared: seq_len=128, batch_size=512
Model initialized. Parameters: 391,552
Epoch 1/20: train_loss=4.5639, val_loss=4.3754, val_ppl=79.47
  ^ New best model saved at epoch 1!
Epoch 2/20: train_loss=3.8534, val_loss=4.2349, val_ppl=69.06
  ^ New best model saved at epoch 2!
Epoch 3/20: train_loss=3.5720, val_loss=4.1681, val_ppl=64.60
  ^ New best model saved at epoch 3!
Epoch 4/20: train_loss=3.4319, val_loss=4.1550, val_ppl=63.75
  ^ New best model saved at epoch 4!
Epoch 5/20: train_loss=3.3468, val_loss=4.1470, val_ppl=63.24
  ^ New best model saved at epoch 5!
Epoch 6/20: train_loss=3.2738, val_loss=4.1547, val_ppl=63.73
  No improvement in PPL for 1 epoch(s).
  Early stopping triggered after 1 epoch(s) without improvement.

================================================================================
STARTING EXPERIMENT: WiderModel_seq64_d256_lr1e-4_bs512
{'name': 'WiderModel_seq64_d256_lr1e-4_bs512', 'seq_len': 64, 'd_model': 256, 'n_layers': 2, 'n_heads': 8, 'd_ff': 512, 'l
r': 0.0001, 'batch_size': 512, 'num_epochs': 20, 'early_stop_patience': 1}
================================================================================
Data split: 358,122 training samples, 89,531 validation samples (80/20 split)
Data prepared: seq_len=64, batch_size=512
Model initialized. Parameters: 1,307,392
Epoch 1/20: train_loss=4.7878, val_loss=4.4770, val_ppl=87.97
  ^ New best model saved at epoch 1!
Epoch 2/20: train_loss=4.0482, val_loss=4.3134, val_ppl=74.69
  ^ New best model saved at epoch 2!
Epoch 3/20: train_loss=3.8194, val_loss=4.2193, val_ppl=67.99
  ^ New best model saved at epoch 3!
Epoch 4/20: train_loss=3.6719, val_loss=4.1709, val_ppl=64.77
  ^ New best model saved at epoch 4!
Epoch 5/20: train_loss=3.5668, val_loss=4.1302, val_ppl=62.19
  ^ New best model saved at epoch 5!
Epoch 6/20: train_loss=3.4725, val_loss=4.1029, val_ppl=60.51
  ^ New best model saved at epoch 6!
Epoch 7/20: train_loss=3.4116, val_loss=4.0886, val_ppl=59.65
  ^ New best model saved at epoch 7!
Epoch 8/20: train_loss=3.3635, val_loss=4.0804, val_ppl=59.17
  ^ New best model saved at epoch 8!
Epoch 9/20: train_loss=3.3178, val_loss=4.0700, val_ppl=58.56
  ^ New best model saved at epoch 9!
Epoch 10/20: train_loss=3.2885, val_loss=4.0665, val_ppl=58.35
  ^ New best model saved at epoch 10!
Epoch 11/20: train_loss=3.2635, val_loss=4.0668, val_ppl=58.37
  No improvement in PPL for 1 epoch(s).
  Early stopping triggered after 1 epoch(s) without improvement.
```

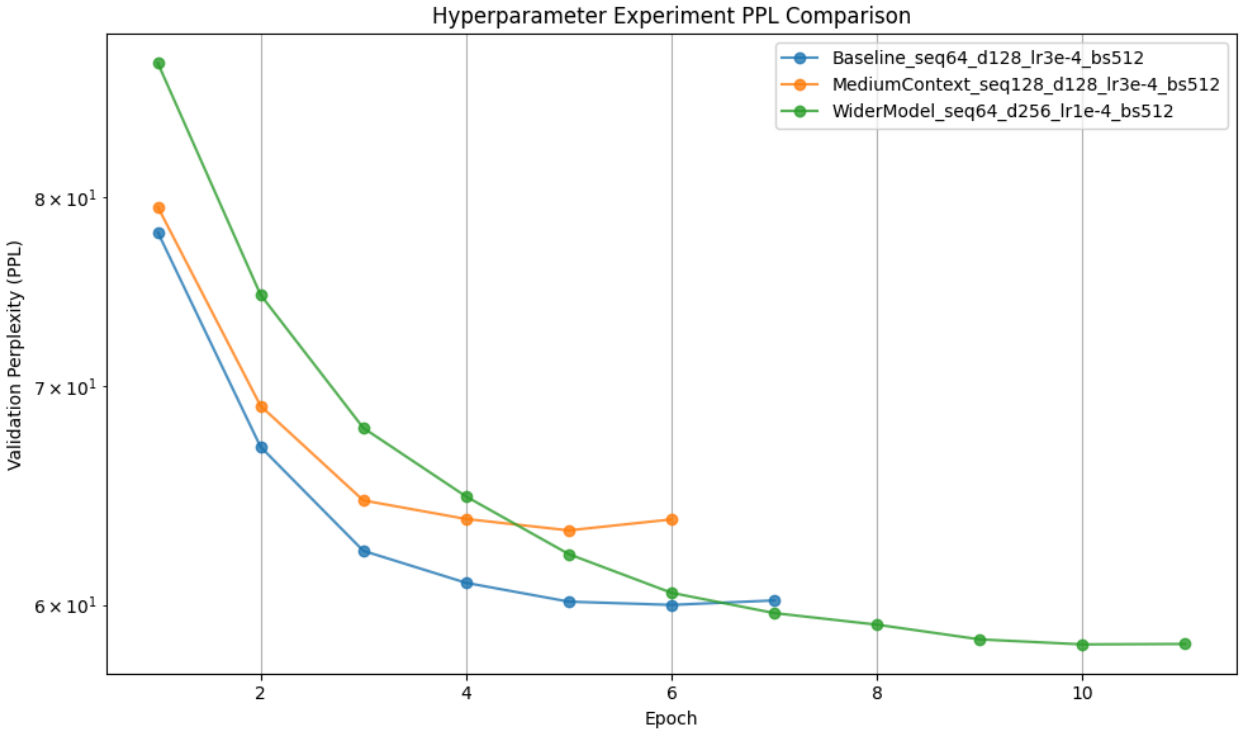
Final Result and best model visualisation:

In [ ]:

```python
# ================================================================
# 6a. AGGREGATE & DISPLAY ALL RESULTS
# ================================================================
print("\n" + "="*80)
print("ALL EXPERIMENTS COMPLETE - FINAL RESULTS")
print("="*80)

best_ppl = float('inf')
best_result = None

for res in all_results:
    print(f"Config: {res['name']:25s} | Best PPL: {res['best_ppl']:.2f} (at Epoch {res['best_epoch']})")
    if res['best_ppl'] < best_ppl:
        best_ppl = res['best_ppl']
        best_result = res

print("\n" + "="*80)
```

```
print(f"🏆 BEST OVERALL MODEL: {best_result['name']} 🏆")
print(f"Best PPL: {best_result['best_ppl']:.2f}")
print("="*80)


# ============================================================
# 6b. PLOT PPL COMPARISON
# ============================================================
plt.figure(figsize=(10, 6))
for res in all_results:
    epochs = range(1, len(res['val_ppls_history']) + 1)
    plt.plot(epochs, res['val_ppls_history'], label=res['name'], marker='o', alpha=0.8)

plt.xlabel("Epoch")
plt.ylabel("Validation Perplexity (PPL)")
plt.title("Hyperparameter Experiment PPL Comparison")
plt.legend()
plt.grid(True)
plt.yscale("log")
plt.tight_layout()
plt.show()
```

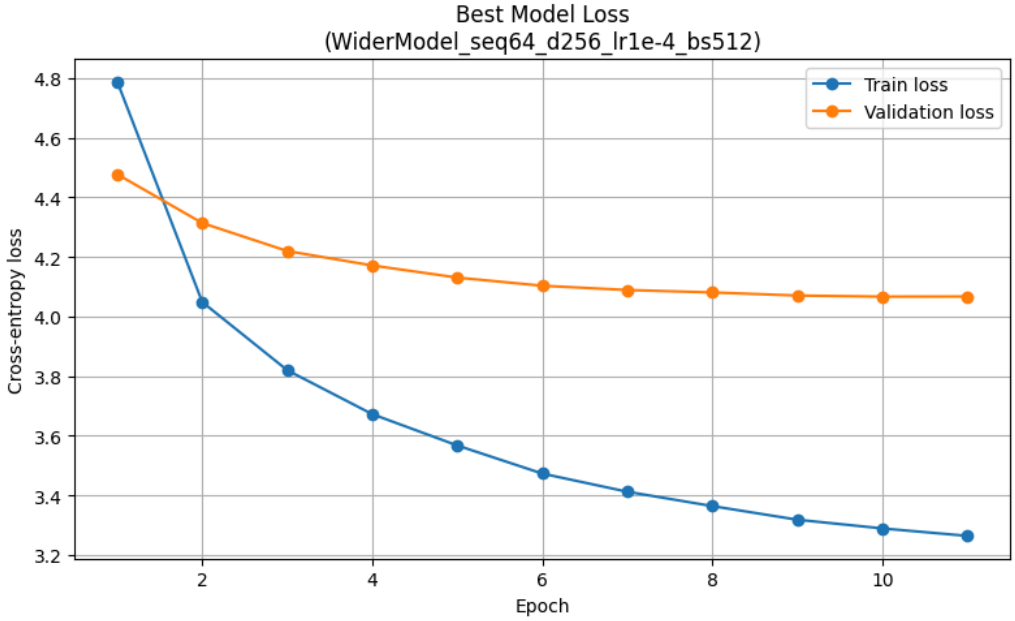
```
================================================================================
ALL EXPERIMENTS COMPLETE - FINAL RESULTS
================================================================================
Config: Baseline_seq64_d128_lr3e-4_bs512 | Best PPL: 60.00 (at Epoch 6)
Config: MediumContext_seq128_d128_lr3e-4_bs512 | Best PPL: 63.24 (at Epoch 5)
Config: WiderModel_seq64_d256_lr1e-4_bs512 | Best PPL: 58.35 (at Epoch 10)

================================================================================
🏆 BEST OVERALL MODEL: WiderModel_seq64_d256_lr1e-4_bs512 🏆
Best PPL: 58.35
================================================================================
```

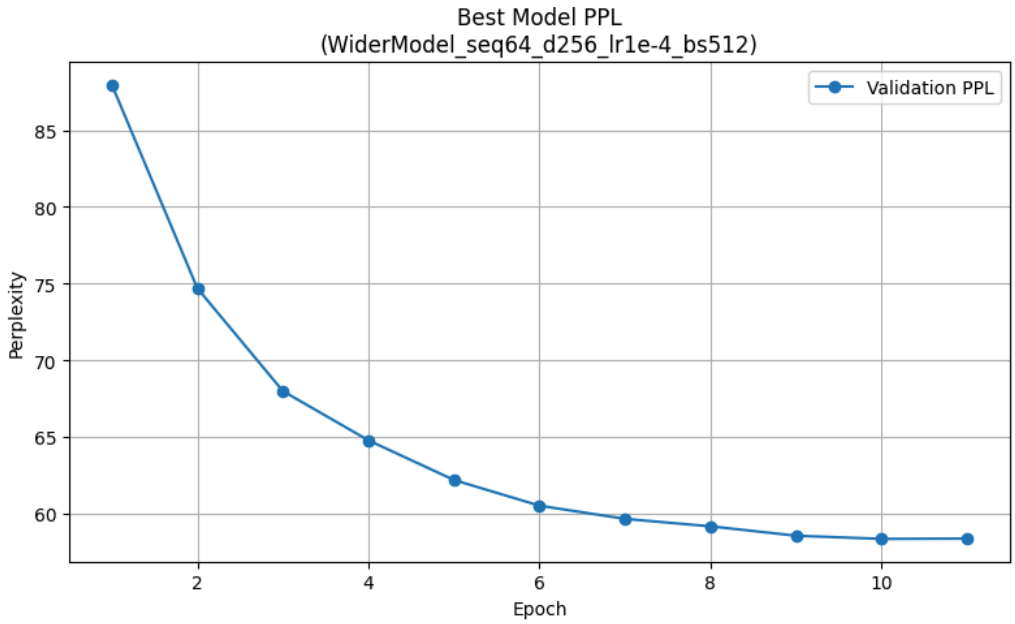


In [ ]:
```
# ============================================================
# 6c. RE-LOAD BEST MODEL FOR VISUALIZATION
# ============================================================
print("\n--- Visualizing Best Model ---")
best_config = best_result['config']
best_model_state = best_result['best_model_state']
best_val_loader = best_result['val_loader']

best_model = TinyTransformerLM(
    vocab_size=vocab_size,
    d_model=best_config['d_model'],
    n_heads=best_config['n_heads'],
    d_ff=best_config['d_ff'],
    max_seq_len=best_config['seq_len'],
    n_layers=best_config['n_layers'],
).to(device)

best_model.load_state_dict(best_model_state)
print("Best model re-built and weights loaded.")

# ============================================================
# 6d. PLOT BEST MODEL LOSS CURVES (SEPARATE CHARTS)
# ============================================================
epochs = np.arange(1, len(best_result['val_ppls_history']) + 1)
```

```
# --- Chart 1: Loss ---
plt.figure(figsize=(8, 5)) # Create a new figure
plt.plot(epochs, best_result['train_loss_history'], marker="o", label="Train loss")
plt.plot(epochs, [math.log(p) for p in best_result['val_ppls_history']], marker="o", label="Validation loss")
plt.xlabel("Epoch")
plt.ylabel("Cross-entropy loss")
plt.title(f"Best Model Loss\n({best_result['name']})")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show() # Display the first chart

# --- Chart 2: Perplexity ---
plt.figure(figsize=(8, 5)) # Create a second, separate figure
plt.plot(epochs, best_result['val_ppls_history'], marker="o", label="Validation PPL")
plt.xlabel("Epoch")
plt.ylabel("Perplexity")
plt.title(f"Best Model PPL\n({best_result['name']})")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show() # Display the second chart
```

```
--- Visualizing Best Model ---
Best model re-built and weights loaded.
```



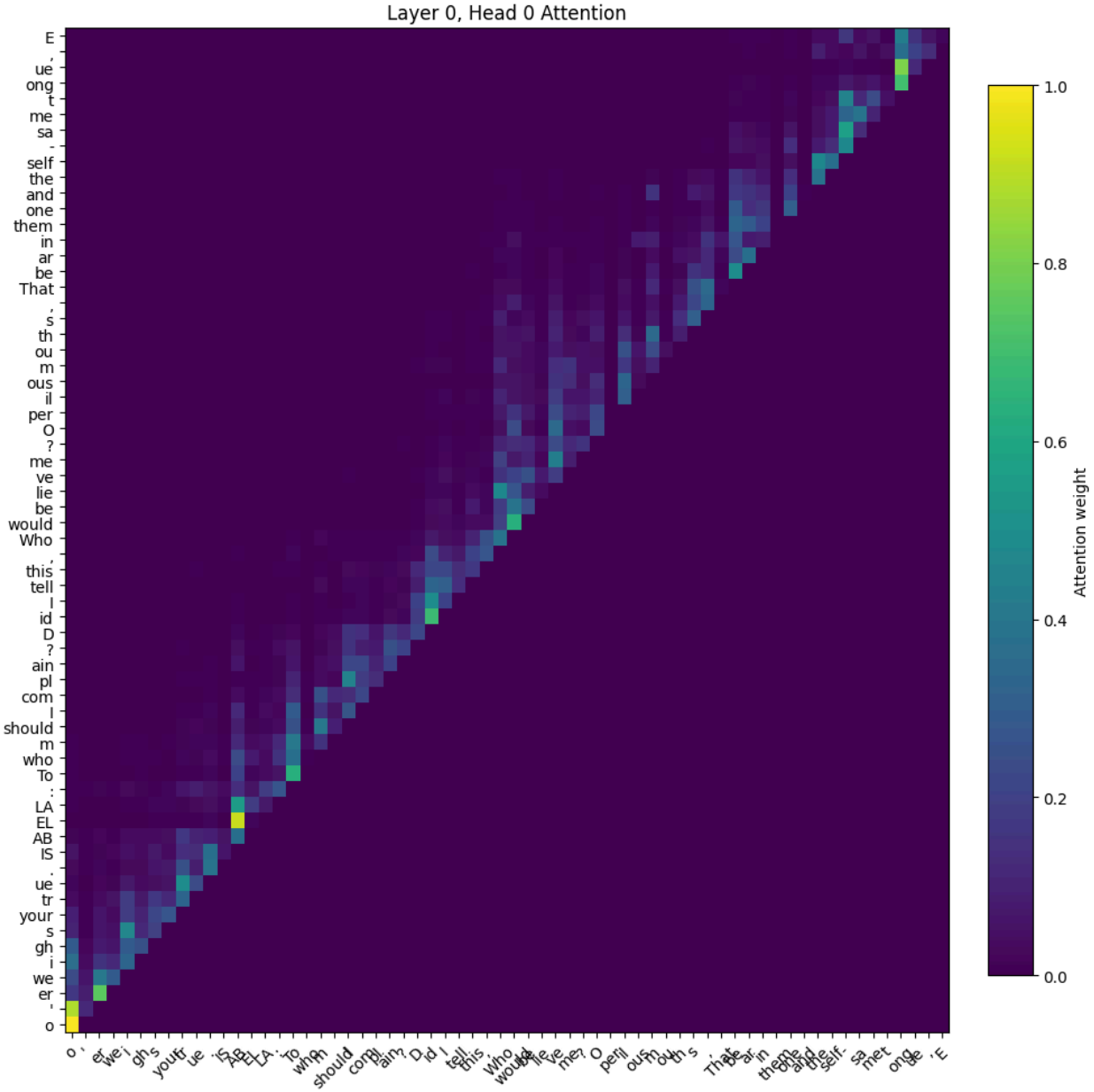


In [ ]:

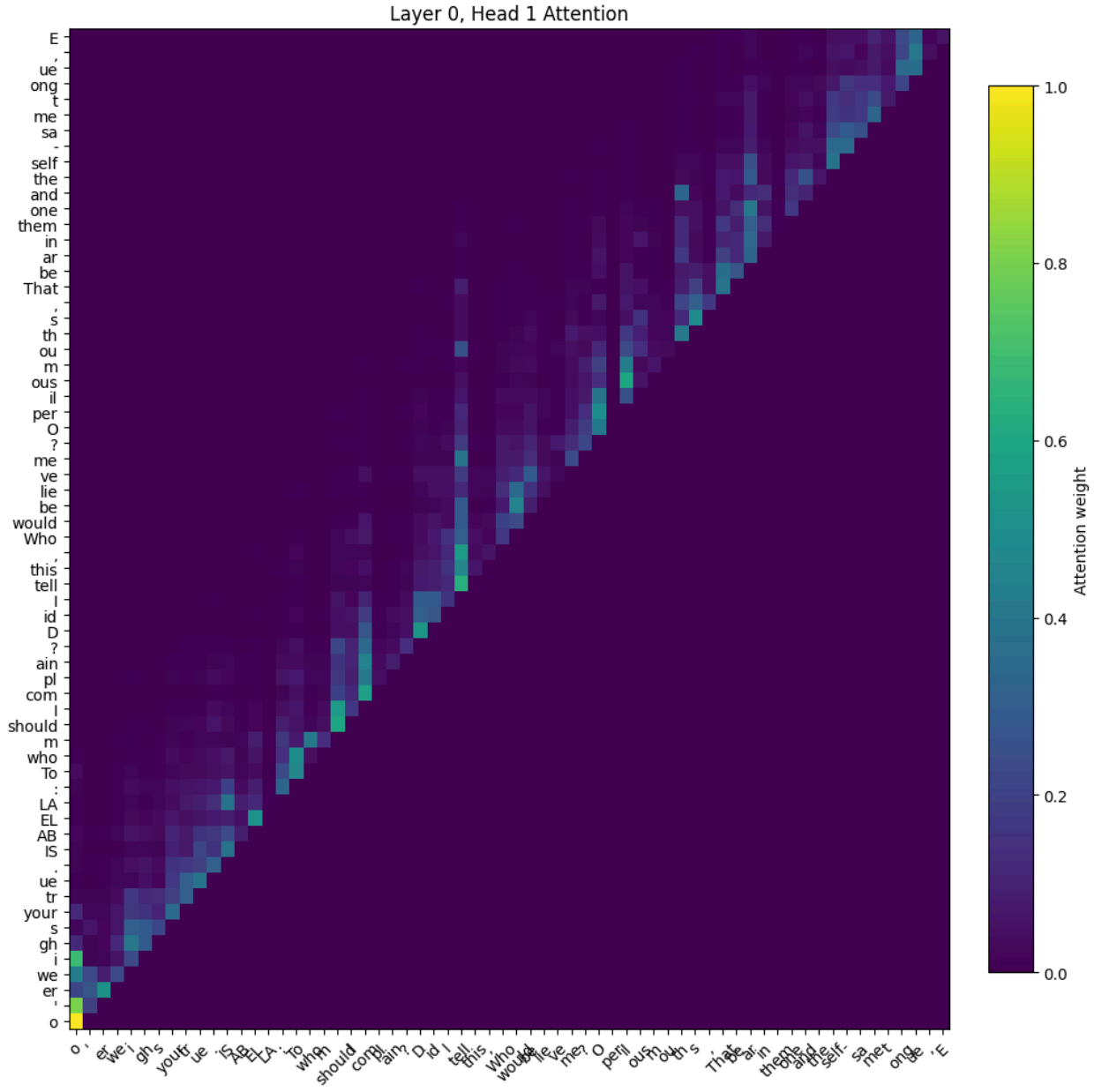
```
# ============================================================
# 6e. PLOT BEST MODEL ATTENTION
# ============================================================
n_layers = best_config['n_layers']
n_heads = best_config['n_heads']

for layer in range(n_layers):
    for head in range(min(n_heads, 2)): # Plot first 2 heads
        print(f"Showing Attention Heatmap for Best Model: Layer {layer}, Head {head}")
```

```
        # Pass the val_loader specific to this model
        show_attention_heatmap(best_model, tokenizer, best_val_loader, device, layer=layer, head=head)

# ================================================================
# 6f. BEST MODEL PPL CONSOLE-LOG
# ================================================================
print("\n--- Best Model PPL Analysis ---")
for epoch, ppl in enumerate(best_result['val_ppls_history'], start=1):
    print(f"Epoch {epoch}: val_ppl={ppl:.2f}")
print(f"\nBest validation perplexity: {best_result['best_ppl']:.2f} at epoch {best_result['best_epoch']}")
```

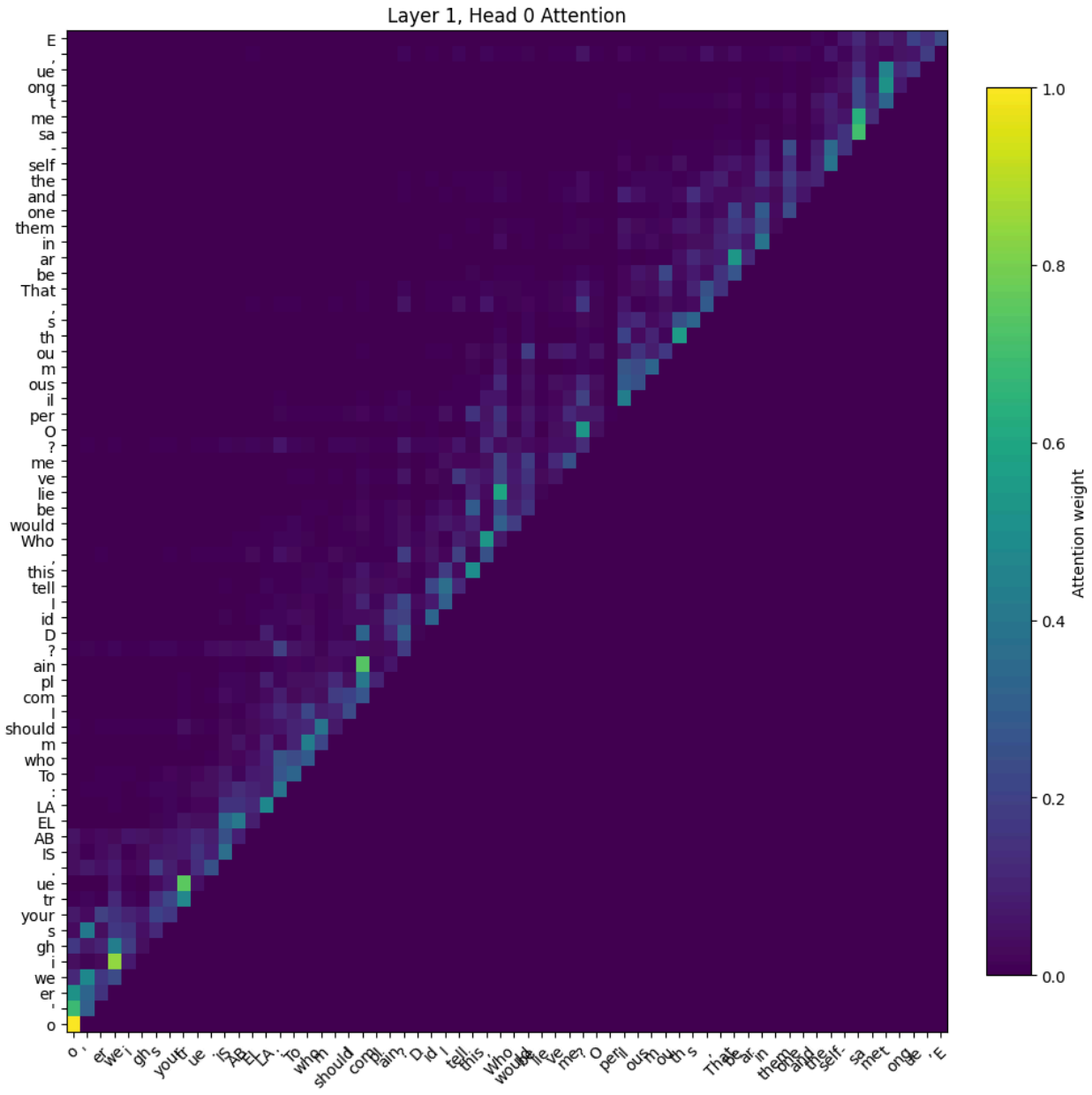
```
Showing Attention Heatmap for Best Model: Layer 0, Head 0
```



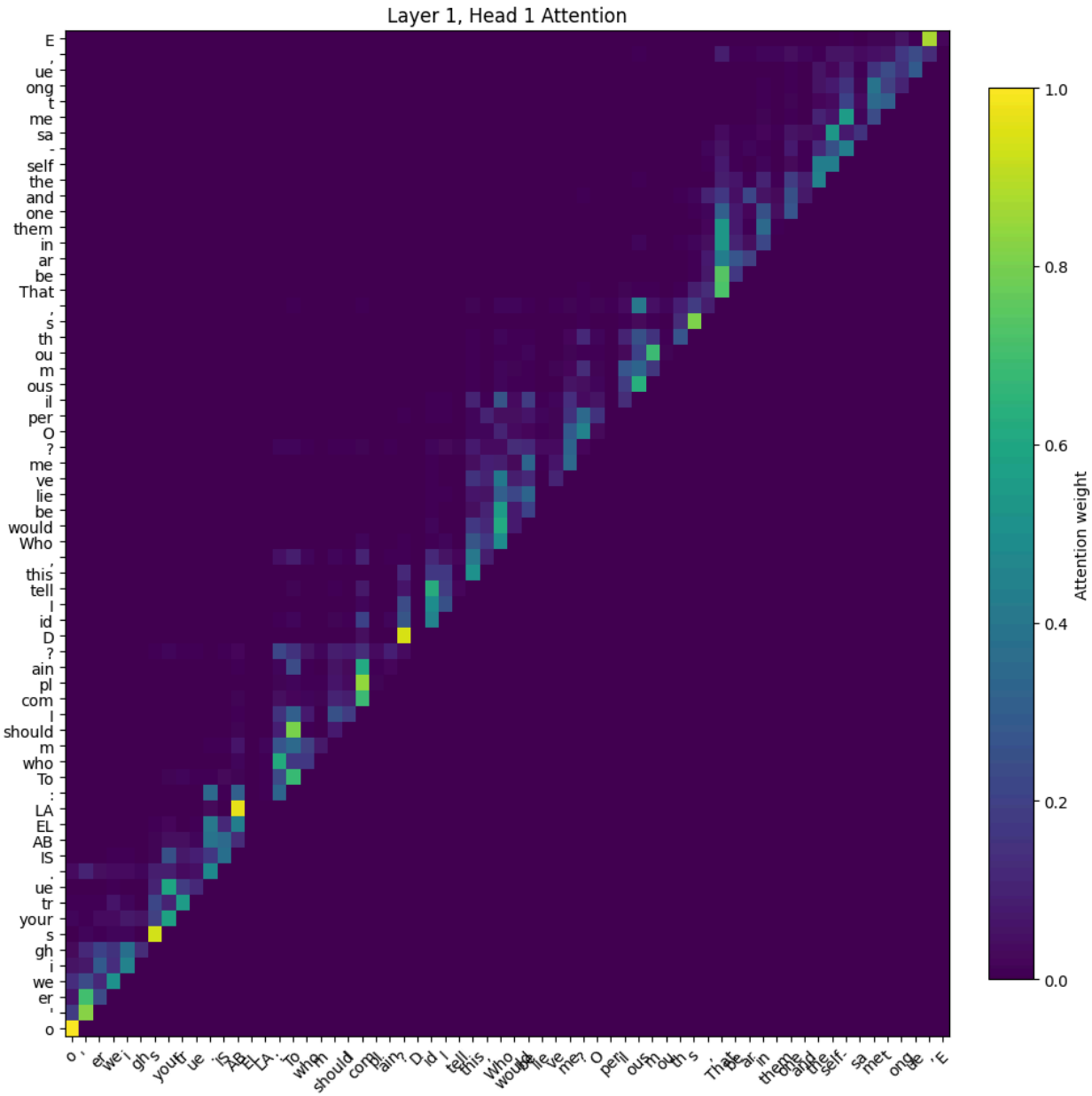

```
Showing Attention Heatmap for Best Model: Layer 0, Head 1
```



Showing Attention Heatmap for Best Model: Layer 1, Head 0



Showing Attention Heatmap for Best Model: Layer 1, Head 1




```
--- Best Model PPL Analysis ---
Epoch 1: val_ppl=87.97
Epoch 2: val_ppl=74.69
Epoch 3: val_ppl=67.99
Epoch 4: val_ppl=64.77
Epoch 5: val_ppl=62.19
Epoch 6: val_ppl=60.51
Epoch 7: val_ppl=59.65
Epoch 8: val_ppl=59.17
Epoch 9: val_ppl=58.56
Epoch 10: val_ppl=58.35
Epoch 11: val_ppl=58.37

Best validation perplexity: 58.35 at epoch 10
```

In [ ]:
```python
# ============================================================
# 6g. INFERENCE: PREDICTED VS. ACTUAL
# ============================================================
print("\n" + "---" * 20)
print(f"   INFERENCE USING BEST MODEL: {best_result['name']}")
print("---" * 20)

best_seq_len = best_config['seq_len']

print("\n--- PREDICTED vs. ACTUAL COMPARISON ---")

x_batch, y_batch = next(iter(best_val_loader))
sample_index = random.randint(0, x_batch.size(0) - 1)
actual_tensor = x_batch[sample_index]
actual_text = clean_generated(tokenizer.decode(actual_tensor.tolist()))

prompt_len = min(25, best_seq_len // 2)
prompt_tensor = actual_tensor[:prompt_len]
prompt_text = clean_generated(tokenizer.decode(prompt_tensor.tolist()))

predicted_text = generate_text(
```

```
    best_model,
    tokenizer,
    device,
    prompt=prompt_text,
    max_new_tokens=best_seq_len - prompt_len,
    seq_len=best_seq_len
)

print(f"\n[PROMPT USED (first {prompt_len} tokens)]:\n{prompt_text}")
print("-" * 60)
print(f"\n[ACTUAL TEXT (full {best_seq_len} tokens)]:\n{actual_text}")
print("-" * 60)
print(f"\n[PREDICTED TEXT (prompt + generated tokens)]:\n{predicted_text}")
print("---" * 20)

# ================================================================
# 6h. INFERENCE: TOP-K PREDICTIONS
# ================================================================
print("\n--- TOP-K NEXT TOKEN PREDICTIONS ---")

test_prompts = ["MENENIUS:", "BRUTUS:", "CORIOLANUS:"]
for prompt in test_prompts:
    preds = predict_next_tokens(best_model, tokenizer, device, prompt, seq_len=best_seq_len)
    print(f"\nPrompt: '{prompt}'")
    for i, (tok, p) in enumerate(preds, 1):
        bar = "█" * int(p * 40)
        print(f"  {i}. {tok:15s} {p:.3f} {bar}")
```

```
------------------------------------------------------------
   INFERENCE USING BEST MODEL: WiderModel_seq64_d256_lr1e-4_bs512
------------------------------------------------------------

--- PREDICTED vs. ACTUAL COMPARISON ---

[PROMPT USED (first 25 tokens)]:
us with life: If I do lo se thee, I do lo se a thing That n one but f oo l s
------------------------------------------------------------

[ACTUAL TEXT (full 64 tokens)]:
us with life: If I do lo se thee, I do lo se a thing That n one but f oo l s would ke ep: a bre ath thou art, S er vi le t
o all the s k y ey in f l u en c es, That do st this ha b it at ion,
------------------------------------------------------------

[PREDICTED TEXT (prompt + generated tokens)]:
us with life: If I do lo se thee, I do lo se a thing That n one but f oo l s to be a per y to the b re e e of e f all C or
i o i de. G ive me, pr in er, for m as thou ch an ge A um
------------------------------------------------------------

--- TOP-K NEXT TOKEN PREDICTIONS ---

Prompt: 'MENENIUS:'
  1. I               0.065 ██
  2. O               0.056 ██
  3. The             0.049 █
  4. What            0.049 █
  5. '               0.046 █

Prompt: 'BRUTUS:'
  1. I               0.070 ██
  2. O               0.059 ██
  3. B               0.059 ██
  4. No              0.042 █
  5. What            0.039 █

Prompt: 'CORIOLANUS:'
  1. I               0.138 ████
  2. A               0.086 ███
  3. O               0.058 ██
  4. The             0.041 █
  5. What            0.041 █
```