COMS 4995 Applied Machine Learning Assignment 3: Transformer is All You Need

Federico Giorgi - fg2617

1 Introduction

Transformer architectures have become the foundation of modern Large Language Models (LLMs). By relying entirely on self-attention rather than recurrence or convolution, Transformers model long-range dependencies and complex contextual interactions in a highly parallelisable way. This assignment explores these ideas in a lightweight setting by building, training, and interpreting a Tiny Transformer for next-token prediction on the *Tiny Shakespeare* corpus, a continuous text file totalling roughly one million characters. The task is to model the corpus as a sequence of discrete tokens and, given a context of preceding tokens, predict the next one. Although this is a small toy problem compared to modern LLM training regimes, it already highlights key challenges: designing an effective tokenisation scheme, constructing a suitable architecture, training it stably, and interpreting learned behaviour. The full notebook is available at the following link.

Goal of the project The aim is to build an end-to-end Transformer-based language model, from raw text to evaluation. The project consists of three main components:

1. preprocessing the *Tiny Shakespeare* corpus into fixed-length input-target sequences and training a subword Byte Pair Encoding (BPE) tokenizer;
2. implementing and training a compact Transformer architecture with embeddings, positional encodings, causal self-attention, feed-forward layers, and residual connections;
3. visualising and interpreting the model's behaviour through loss and perplexity curves, attention maps, and qualitative sample generations.

2 Data Preparation

2.1 Corpus loading

The *Tiny Shakespeare* corpus is downloaded directly from Karpathy's public repository using `urllib`. The text is loaded into memory as a single Unicode string.

2.2 Subword tokenization

Tokenizer choice and configuration A subword-level Byte Pair Encoding (BPE) tokenizer is used, implemented with the `tokenizers` library from HuggingFace. To preserve spacing, punctuation, and newlines exactly as in the raw corpus, the tokenizer adopts a byte-level pre-tokenization strategy. The tokenizer is configured as follows:

- special tokens: `[PAD]`, `[UNK]`, `[BOS]`, `[EOS]`;
- model: `BPE` with `[UNK]` as the unknown token;
- pre-tokenizer: byte-level splitting to preserve the exact text layout;
- decoder: byte-level decoder to reconstruct text faithfully;
- vocabulary size: at most 500 tokens, with a minimum frequency of 2.

After training on the full corpus, the complete text is encoded into integer token ids, which serve as the numerical input for sequence formatting.

Qualitative checks Two simple sanity checks confirm that the tokenizer behaves correctly: encoding and decoding short text snippets reconstructs the original string, and an encode–decode–encode round trip returns identical token id sequences, ensuring consistency and lossless operation.

2.3 Sequence formatting

For next-token prediction, the model must predict token $t+1$ given tokens up to t. To create training examples, a fixed-length window of size $T = 50$ is slid across the entire sequence of token ids with stride 1. For each starting index i, a chunk of length $T+1$ is extracted: the input is defined as the first T tokens,

$$\text{input} = [x_i, \ldots, x_{i+T-1}],$$

and the target is the same sequence shifted by one position,

$$\text{target} = [x_{i+1}, \ldots, x_{i+T}].$$

This procedure yields a large number of overlapping input, target pairs, each encoding a short local context and its next-token labels.

Qualitative check A simple sanity check confirms that the first few input sequences, when decoded, match the corresponding decoded target sequences shifted by one token.

2.4 Train-validation split

The list of windowed sequences is wrapped in a custom `NextTokenDataset` class, which stores precomputed input and target tensors of shape (T). The full dataset is then split deterministically into 80% training and 20% validation using `random_split` with a fixed random seed.
Since all sequences have a fixed length of 50 tokens, the `collate_batch` function simply stacks them along a new batch dimension to form input and target tensors of shape (B, T). A batch size of $B = 128$ is used for both training and validation, with shuffling and `drop_last = True` enabled for the training loader to maintain consistent batch shapes.

2.5 Token embedding

Token embedding Each token id is mapped to a dense vector in $\mathbb{R}^{d_{\text{model}}}$ using an embedding layer:

$$\text{Embed} : \{0, \ldots, V-1\} \to \mathbb{R}^{d_{\text{model}}},$$

where V is the vocabulary size and d_{model} is the model's hidden dimension. In the preliminary embedding module, a `TokenEmbedding` class wraps `nn.Embedding`. Given a batch of token ids $X \in \mathbb{N}^{B \times T}$, the module outputs an embedding tensor $H \in \mathbb{R}^{B \times T \times d_{\text{model}}}$.

Sinusoidal positional encoding Transformers are permutation-invariant by design: self-attention treats all positions symmetrically. To inject information about token order, positional encodings are added to the token embeddings. Following Vaswani et al., fixed sinusoidal encodings are implemented using

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

for positions $pos \in \{0, \ldots, L-1\}$ and dimension index i. The `SinusoidalEmbeddings` module precomputes a matrix $\text{PE} \in \mathbb{R}^{L \times d_{\text{model}}}$ for a maximum length L and registers it as a non-trainable buffer so it moves correctly between devices but is not updated by the optimizer. In the forward pass, the module receives a tensor $H \in \mathbb{R}^{B \times T \times d_{\text{model}}}$ and returns

$$z_{b,t,:} = H_{b,t,:} + \text{PE}_{t,:},$$

This combined representation encodes both token identity and absolute position and is then fed into the Transformer blocks. It forms the input to all subsequent self-attention layers, enabling the model to compute position-aware contextual interactions.

3 Tiny Transformer Implementation

3.1 Transformer Structure

Self-attention with causal mask The core building block of the model is a single-head self-attention module implemented in the `SelfAttention` class. It allows each token in a sequence to refine its representation by attending to other tokens. Given input embeddings $X \in \mathbb{R}^{B\times T\times d_{\text{model}}}$, the model computes three linear projections:

$$Q = X \cdot W_Q, \quad K = X \cdot W_K, \quad V = X \cdot W_V$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}}\times d_{\text{model}}}$ are learned parameter matrices. The Query encode what each token seeks, the Keys encode how each token should be matched, and the Value contain the information to be aggregated. Similarity scores between tokens are computed through scaled dot-products:

$$\text{Scores} = \frac{QK^\top}{\sqrt{d_k}} \quad \in \quad \mathbb{R}^{B\times T\times T},$$

where each entry $S_{i,j}$ measures how strongly token i relates to token j. In next-token prediction, future tokens must not influence the current prediction. To enforce this autoregressive constraint, an upper-triangular causal mask is applied:

$$\text{Scores}'_{b,i,j} = \begin{cases} \text{Scores}_{b,i,j}, & j \le i, \\ -\infty, & j > i. \end{cases}$$

ensuring that positions $j > i$, i.e., future tokens, receive zero probability after the softmax. The attention weights are obtained by normalizing the masked scores:

$$\text{Weights} = \text{softmax}\left(\text{Scores}'\right)$$

so that each row forms a probability distribution over past and current tokens. Finally, each output representation is a weighted sum of Value:

$$\text{Output} = \text{Weights} \cdot V$$

Through this mechanism, the model constructs contextualized embeddings that integrate both local and long-range dependencies. The implementation returns both the attended representations and the attention matrices, which are later used for visualization and analysis.

Feed-forward network In addition to self-attention, each Transformer layer includes a position-wise `FeedForward` network (FFN). The FFN consists of two linear transformations with a non-linearity in between:

$$\text{FFN}(x) = W_2\ \text{ReLU}(W_1 x + b_1) + b_2,$$

where $W_1 \in \mathbb{R}^{d_{\text{model}}\times d_{\text{ff}}}$ projects the representation into a higher-dimensional hidden space, and $W_2 \in \mathbb{R}^{d_{\text{ff}}\times d_{\text{model}}}$ maps it back to the model dimension. This component enables the model to apply non-linear transformations that complement the information-mixing performed by the attention mechanism.

Transformer block The `TransformerBlock` integrates self-attention and the feed-forward network, each wrapped in pre-normalization using RMSNorm and combined with residual connections. This design follows the modern "pre-norm" formulation, which improves training stability compared to the original post-LayerNorm architecture. Formally, for an input sequence X:

$$X' = X + \text{SelfAttn}\big(\text{RMSNorm}(X)\big),$$

$$X^* = X' + \text{FFN}\big(\text{RMSNorm}(X')\big).$$

The first residual pathway allows each token to incorporate contextual information via attention, while the second refines the representation through non-linear transformations. Together, these components form a compact yet expressive Transformer layer suitable for the Tiny Shakespeare modeling task.

Complete Tiny Transformer architecture The `TinyTransformer` class composes all components into a complete language model:

- token embedding: `TokenEmbedding` class with dimension $d_{\text{model}} = 128$;
- positional encoding: `SinusoidalEmbeddings` added element-wise to token embeddings;
- Transformer stack: two `TransformerBlock` layers with feed-forward hidden size $d_{\text{ff}} = 512$;
- output head: a linear projection mapping $\mathbb{R}^{d_{\text{model}}}$ back to the vocabulary space $\mathbb{R}^{V}$.

3.2 Training procedure

Objective and optimization The model is trained to minimise the token-level cross-entropy loss between the predicted logits and the target token ids:

$$\mathcal{L} = -\frac{1}{BT} \sum_{b=1}^{B} \sum_{t=1}^{T} \log p_\theta(y_{b,t} \mid x_{b,1:t}),$$

where p_θ denotes the softmax distribution over the vocabulary. In practice, the logits are reshaped to (BT, V) and the targets to (BT) before being passed to `nn.CrossEntropyLoss`. Training uses the Adam optimizer (learning rate 10^{-3}, weight decay 10^{-5}), together with mixed-precision computation and gradient clipping (max-norm 1.0) to enhance numerical stability and prevent exploding gradients. A summary of all hyperparameters used during training is provided in Table 1.

Epoch loop and validation The model is trained for 50 epochs. For each epoch the training loop iterates over all mini-batches, performing forward and backward passes and parameter updates, while accumulating the running training loss. After each epoch, the model is switched to evaluation mode and the average validation loss is computed over all validation batches without gradient tracking. To make this quantity easier to interpret in the language modelling setting, the validation loss is converted into validation perplexity,

$$\text{PPL} = \exp(\text{validation loss}),$$

a standard metric for next-token prediction. Intuitively, perplexity corresponds to the effective average number of choices the model considers plausible at each timestep: lower values indicate sharper and more confident predictions.
Monitoring validation perplexity across epochs, together with the corresponding training and validation loss curves, provides a concise indicator of whether the model is learning increasingly informative token distributions.

Hyperparameter	**Value**
Embedding dimension d_{model}	128
Number of layers	2
Feed-forward dimension d_{ff}	512
Batch size	128
Context length T	50
Vocabulary size V	500
Learning rate	10^{-3}
Weight decay	10^{-5}
Epochs	50
Optimizer	Adam

Table 1: *Summary of main hyperparameters used for training the Tiny Transformer.*

4 Results: Visualization & Interpretation

4.1 Training and validation loss

Figure 1 shows the evolution of the training and validation losses across epochs. Both curves decrease rapidly in the initial phase and then continue to decline more gradually, indicating stable convergence of the optimisation process. The validation loss closely tracks the training loss throughout training and does not exhibit upward divergence, suggesting that the model does not suffer from substantial overfitting despite the small dataset and the absence of explicit regularisation mechanisms such as dropout. The use of RMSNorm and residual connections appears sufficient to maintain stable gradients and prevent training instabilities. With 50 epochs, the model continues to refine its predictions rather than plateau prematurely, which is consistent with the moderate capacity of this two-layer Transformer and the limited size of the Tiny Shakespeare corpus. Overall, the loss curves indicate a well-behaved training dynamic in which the model steadily improves without signs of instability or memorisation.

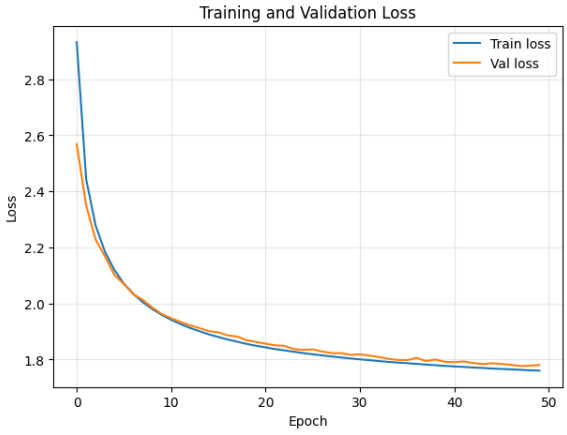

Figure 1: *Training and validation loss over epochs for the Tiny Transformer.*

4.2 Attention heatmaps

To inspect how the model distributes attention across a sequence, attention matrices from both Transformer layers are visualised for a sample taken from the validation set. After a forward pass in evaluation mode, the stored attention weights $A^{(\ell)} \in \mathbb{R}^{T \times T}$ for layer ℓ and for the first example in the batch are extracted and plotted as heatmaps.

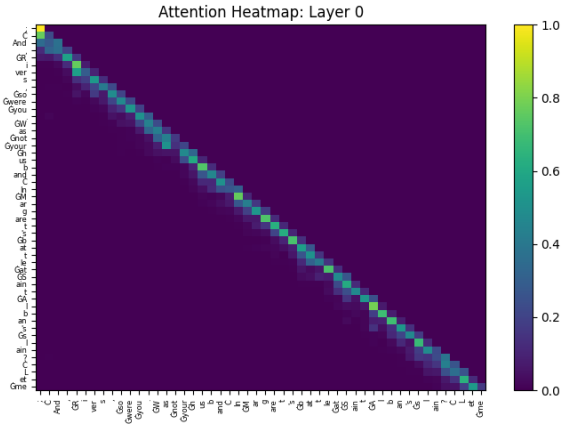

Figure 2: *Attention heatmap for layer 0 on a sample sequence from the validation set (epoch 50). Brighter cells indicate higher attention weights.*

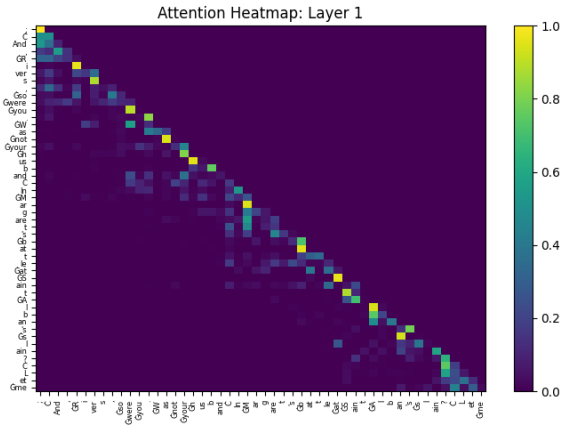

Figure 3: *Attention heatmap for layer 1 on the same sequence (epoch 50). Compared to layer 0, attention becomes less strictly local and shows more structured off-diagonal behaviour.*

Layer-wise comparison Across the two layers, several qualitative patterns emerge. **Layer 0** exhibits a strong diagonal dominance, meaning that each token attends primarily to itself and to its immediate predecessors. This behaviour is typical of early layers in small Transformer models: attention acts as a locality-preserving mechanism, refining token embeddings without yet relying on longer-range context and providing a stable foundation for deeper layers. **Layer 1**, in contrast, displays a more diffuse and structured pattern. While the diagonal remains visible, multiple off-diagonal regions appear, indicating that the model has begun to integrate information from earlier textual elements. These structures suggest a transition from local processing to a global form of contextual mixing. In both layers, the upper-triangular region remains strictly zero, confirming that the causal mask successfully prevents tokens from attending to future positions.

Evolution over training A comparison between the attention patterns at epoch 5 and epoch 50 shows a clear but modest evolution. At epoch 5 (Figure 4), layer 1 exhibits an almost purely diagonal structure with small off-diagonal activations, reflecting a strong reliance on local context and relatively unrefined representations. By epoch 50 (Figure 3), these off-diagonal regions become more coherent and clusters appear more consistently, indicating that the model has learned to incorporate slightly broader contextual cues. This change parallels the quantitative improvement in validation perplexity, which drops only from roughly 2.2 at epoch 5 to about 1.8 after full training. The limited shift in both attention structure and perplexity reflects the constrained capacity of this two-layer Transformer, whose ability to model long-range dependencies remains intrinsically modest.

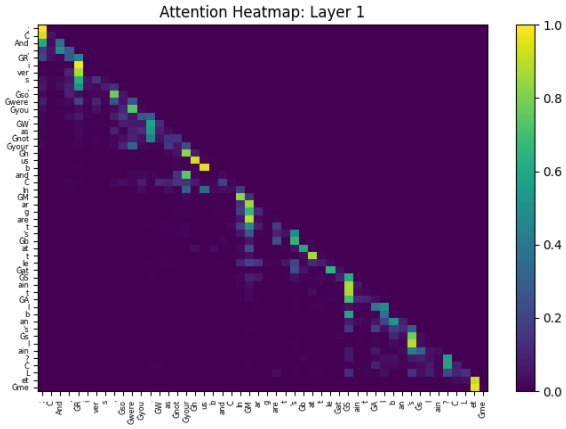

Figure 4: *Attention heatmap for layer 1 on the same sequence at epoch 5. Attention is almost entirely local, with a narrow diagonal and scattered, unstable off-diagonal activations.*

4.3 Validation perplexity

Figure 5 reports the validation perplexity computed at the end of each epoch. Because perplexity is simply the exponential of the validation cross-entropy loss, its trajectory closely mirrors the shape of the validation-loss curve. While the final perplexity is far from that of large-scale state-of-the-art models, it is reasonable for a two-layer transformer with a small hidden size trained on a toy corpus.

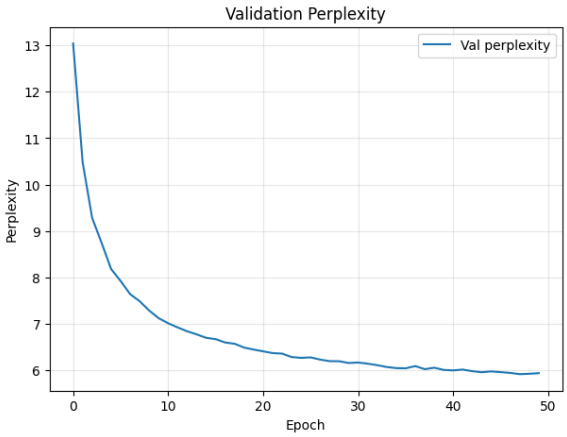

Figure 5: *Validation perplexity over epochs. Perplexity is computed as the exponential of the validation cross-entropy loss.*

4.4 Sample generations

Generation procedure The model is also evaluated qualitatively by generating text conditioned on short prompts. Generation is implemented in a `generate_text` helper function that temporarily switches the model to evaluation mode, performs autoregressive sampling, and then restores the original training/evaluation state.
Given a specified prompt string, the function first encodes it into token IDs using the trained BPE tokenizer. The IDs are then converted into a tensor of shape $(1, T)$ and moved to the same device as the model parameters. Inside a `torch.no_grad()` block, the function iteratively generates up to `max_new_tokens` additional tokens. At each step, the current context tensor x is truncated again to the last `SEQ_LEN` positions (maintaining a sliding window), and passed through the Transformer to obtain logits over the vocabulary for every time step. Only the logits at the final position are retained, divided by a temperature parameter τ (here typically $\tau = 0.8$), and converted to probabilities with a softmax. The next token ID is then sampled via `torch.multinomial` from this categorical distribution, rather than selecting the argmax, to encourage diverse and non-deterministic continuations. The sampled ID is concatenated to the context and the loop continues. After generation, the full sequence of IDs is moved back to CPU, and only the IDs corresponding to the continuation (i.e. excluding the original prompt) are decoded back to text. Finally, if the model was originally in training mode, `generate_text` restores this state to avoid side effects on subsequent training steps. Several prompts were chosen to probe whether the model can adapt to different stylistic cues, ranging from Shakespearean openings to more modern inputs.

Shakespeare-style prompts As a first qualitative test, prompting the model with a classical Shakespearean cue

```
ROMEO:\n
```

and generating with `max_new_tokens = 80` and temperature $\tau = 0.8$ yields the following continuation:

```
their bones
kindle in your follower
And blORK:
For comes the very late; who,
they say, or take
to des
```

The output clearly departs from grammatical English, yet it displays several hallmarks of the training corpus: short verse-like lines, archaic turns of phrase (*kindle in your follower*), and invented proper nouns or interjections (*blORK*) in a position where a character name might appear.
Conditioning on another character name,

```
CLAUDIO:\n
```

yields

```
utinous kindred.
Second Murderer:
If the events be.
First Watchman:
I should might be stone
```

Here the model spontaneously introduces new speaker labels (*Second Murderer*, *First Watchman*) and short declarative lines, closely resembling the structure of stage directions and dialogue in Shakespearean plays. The invented phrase *utinous kindred* is syntactically plausible and stylistically archaic, even if semantically opaque.

Out-of-distribution prompts Interestingly, non-Shakespearean prompts are quickly pulled into the learned style. For instance, a modern exclamation

```
Help!:\n
```

produces

```
To hear my want-like farewell.
DUKE VINCENTIO:
He shall.
ISABELLA:
And is as good as mother
parghing
```

while a more generic romantic seed

```
I love:\n
```

gives

```
ad to see your
son the guests; and so may my
person
To addarers, and I will make a
brieve,
And all please your
```

In both cases, the model quickly shifts into a register resembling early modern English, introduces character names from other plays (*DUKE VINCENTIO*, *ISABELLA*), and respects a line-based structure. At the same time, it often generates non-words such as *parghing* or malformed constructions (*I should might be stone*, *ad to see your*), reflecting its limited capacity and the modest size of the training corpus.

Interpretation Overall, these samples show that even a very small two-layer Transformer can internalise non-trivial stylistic and structural regularities of the Tiny Shakespeare corpus: it learns that colons frequently precede character names, that dialogue is organised into short lines, and that certain lexical patterns (titles, archaic verb forms) are more likely than others. However, the lack of global coherence and the frequent syntactic errors confirm that substantially larger models and datasets are required to achieve fluent and fully coherent generations.

4.5 Positional encodings

A key observation is that adding positional information is essential for meaningful learning: without an explicit notion of order, the self-attention mechanism would treat the input as an unordered set, making it impossible to distinguish sequences such as "KING HENRY" from "HENRY KING" or to preserve the line-based structure seen in both the attention maps and the generated samples. The faster convergence and the emergence of coherent diagonal patterns in the heatmaps directly reflect the contribution of positional encodings.

4.6 Limitations and extensions

The current implementation is intentionally minimal and omits several components that are standard in larger Transformer architectures, such as dropout, multi-head attention, and deeper stacks of layers. Adding dropout or increasing model depth would likely improve robustness and generalisation, while multi-head attention could enable different heads to specialise in complementary linguistic cues. Beyond architectural changes, a systematic sweep over context length, learning rate, and batch size would help identify which factors most strongly affect perplexity and convergence, and would clarify how far a model of this scale can be pushed before capacity becomes the dominant bottleneck.

5 AI tool usage disclosure

Artificial intelligence tools assisted various stages of this project, particularly in code development and report preparation. ChatGPT was used to recall specific PyTorch and LaTeX syntax (e.g., details of the `DataLoader` API, mixed-precision utilities, and figure formatting), to refine the presentation of the Transformer architecture, and to explore alternative phrasings for technical explanations. Part of the implementation was adapted from starter code provided in class, which served as a foundation for building the full Transformer model used here. The tool also supported high-level sanity checks of conceptual understanding, including the behaviour of attention mechanisms, positional encodings, and perplexity.

Only the most relevant points from the "Discussion & Reflection" guidelines were addressed explicitly; the analysis focused on attention patterns and positional encodings, which were the most informative aspects for a model of this scale. Finally, experimenting with prompt-based generation proved particularly enjoyable, effectively simulating interactions with a miniature language model and observing how training shaped its stylistic tendencies.

All AI-assisted content was carefully reviewed, edited, and integrated with independent reasoning. Formulas, code snippets, and claims about model behaviour were manually verified, and all results, plots, attention heatmaps, and generated samples, were produced directly from the implementation described in this report.