INF265 Project 3: Transformer Models

Ninad Hagi

April 2025

1 Introduction

Transformers have recently emerged as a cornerstone of modern natural language processing, it offers a robust and powerful alternative to recurrent neural networks by leveraging self-attention mechanisms to model long-range dependencies more effectively and in parallel. First introduced by Vaswani *et al.* in "Attention Is All You Need" [1], the transformer architecture replaces sequential processing with an encoder–decoder framework built around scaled dot-product and multi-head attention, enabling substantial gains in tasks ranging from machine translation to summarization.

This project investigates two principal variants of the transformer family: an encoder-only model for sentiment classification and a decoder-only model for text generation. In Part 1, we implement the masked self-attention mechanism from scratch in PyTorch and apply it to IMDb movie reviews, training a classifier that uses the final `[CLS]` token embedding to predict positive or negative sentiment. In Part 2, we build a GPT-style, decoder-only transformer—complete with causal masking and a byte-pair encoding tokenizer and train it on question–answer pairs from the GooAQ dataset, culminating in a simple chatbot interface for interactive text generation.

The report is organized as follows. Section 2 details the design and implementation of the encoder-only sentiment classifier, including data preprocessing, model architecture, and evaluation results. Section 3 describes the decoder-only text generator, covering tokenizer training, model construction, sampling strategies, and qualitative analysis of generated outputs.

2 Encoder-only Model for Sentiment Classification

2.1 Pre-processing & Tokenisation

1. **HTML & symbol cleanup** – we strip any `<tag>` markup, remove non-alphanumerics except basic punctuation, and lowercase every review.
2. **Word-level tokeniser** – a custom WordLevel tokenizer is trained on the IMDb training split (min-freq = 10, vocab = 10 000). Three special tokens are reserved:
   - `[PAD]` for padding,
   - `[CLS]` prepended to every sequence so the model has a dedicated "sentence embedding",
   - `[UNK]` for out-of-vocabulary words.
3. **Sequence shaping** – after adding `[CLS]`, sequences are padded or truncated to 256 tokens (median length of the corpus). A Boolean padding mask (`True` on `[PAD]`) is created once per batch and reused by the attention layers.

2.2 Model Architecture

- **Input layer** – word embeddings of size 96 plus fixed sine–cosine positional encodings[1] .
- **Encoder stack** – three identical blocks, each containing:
  a) Multi-head self-attention (4 heads, implemented from scratch),
  b) Add + LayerNorm,
  c) Feed-forward MLP (inner size = $4 \times$ dim, GELU),
  d) Add + LayerNorm.
- **Classifier head** – the final hidden state of the first token (`[CLS]`) is passed through a linear layer $\rightarrow$ sigmoid to yield

$$\hat{p} \in [0, 1].$$

2.3 Training Setup

Hyperparameter	**Value**
Optimizer	AdamW (lr $= 1 \times 10^{-3}$, weight_decay $= 1 \times 10^{-3}$)
Loss	Binary cross-entropy
Batch size	64
Num. epochs	3
Padding mask clip	gradient norm clipped at 10
Random seed	420

Table 1: Training hyperparameters.

2.4 Learning Curves & Validation

```
Epoch 1: Train Loss=0.5089, Train Acc=73.62% → Val Loss=0.4601, Val Acc=79.14%
Epoch 2: Train Loss=0.3445, Train Acc=85.26% → Val Loss=0.3863, Val Acc=84.18%
Epoch 3: Train Loss=0.2748, Train Acc=88.84% → Val Loss=0.3945, Val Acc=83.98%
```

Figure 1 illustrates how accuracy evolved during the three-epoch training run. Training accuracy climbed steadily from approx 74% to approx 89%, while validation accuracy followed the same upward trend but stabilized (plateaued) around 84% after the second epoch, suggesting a slight onset of over-fitting beyond epoch 2.

2.5 Test Performance

- Test accuracy: 0.8433 (84.33%) on 25 000 unseen IMDb reviews.

2.6 Qualitative Evaluation on Custom Examples

To gain a better understanding of model behavior beyond aggregate metrics, we test on three reviews of varying length and nuance:

[1] Eq. (3.5) in Vaswani et al., 2017 [1]

Review excerpt	Predicted label	Probability
"Best movie ever. Heath Ledger's work is phenomenal no words. . . "	`pos`	0.7393
"Dear Ms. Halle Berry, I want my money back. . . *Catwoman*."	`neg`	0.1154
"We've been subjected to enormous amounts of hype and marketing for *The Dark Knight*. . . "	`pos`	0.9441

Table 2: Model predictions on custom examples ranging from short, unambiguous snippets to a longer IMDb review.

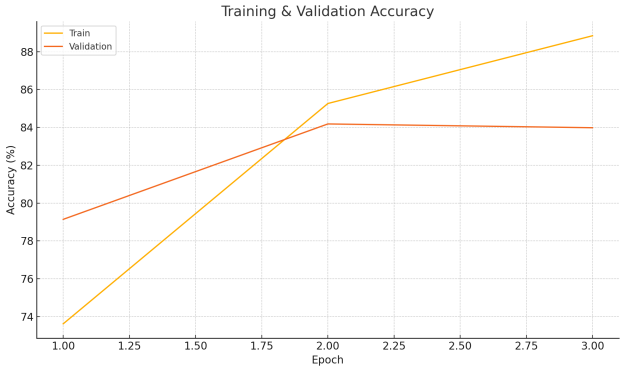

Figure 1: Training and validation accuracy over the three training epochs

Observations.

- *Short, clearcut reviews* have confident predictions ($p = 0.7393$), demonstrating the model's ability to pick up obvious sentiment cues.
- *Longer and more complex review* is classified correctly as positive with high confidence ($p = 0.9441$), indicating the model successfully integrates across multiple sentences to infer overall tone.
- No obvious failure mode was observed among these examples; future work could be to use mixed-sentiment texts to further stresstest the classifier.

2.7 Discussion

A important implementation detail was how we applied masking inside the scaled-dot product attention. By first broadcasting the padding mask to the shape $(B, 1, 1, T)$ and then passing it to `masked_fill`, we were able to perform the entire attention computation in a single, fully-vectorized operation on the GPU, completely avoiding loops.

The resulting model generalises well: validation and test accuracies both stabilised at (∼84 %), so there is little evidence of overfitting. Training and evaluation curves track each other closely, which suggests that the regularisation provided by dropout and early stopping was sufficient for this data size.

Noticed two practical challenges: First, finding the right number of workers for the `DataLoader` required experimentation: the system issued warnings for four workers, whereas two provided the best throughput without triggering resource limits. Second, several debugging cycles were needed to make sure that the attention and padding masks had compatible shapes and that every residual-connection path correctly matched tensor dimensions. These are issues that can degrade training stability if overlooked.

3 Decoder-only Model for Text Generation

3.1 Explanation of the DecoderBlock implementation

The decoder layer closely follows the GPT-2 style transformer architecture presented in [1]. Each block performs the following sequence of operations:

1. **Masked multi-head self–attention** [2]
   - Implemented with `nn.MultiheadAttention` (`batch_first=True`) so inputs have shape (B, L, d) where B is the batch size, L the sequence length and d the embedding size.
   - A *causal* (look-ahead) mask $M_{\text{causal}} \in \{0,1\}^{L\times L}$ is passed via `attn_mask`. Its upper-triangular structure (1's above the main diagonal) prevents every position from attending to future tokens, enforcing auto-regressive generation.
   - A *padding* mask $M_{\text{pad}} \in \{0,1\}^{B\times L}$ is forwarded through `key_padding_mask`, ensuring that attention weights belonging to [PAD] tokens are set to $-\infty$ before the softmax.
   - The call returns the attended representation $\mathbf{A}$, which is added to the residual stream and normalised in the next step. Attention weights themselves are discarded (`need_weights=False`).
2. **First residual connection & layer normalisation**
$$\mathbf{H}_1 = \text{LayerNorm}\big(\mathbf{X} + \text{Dropout}(\mathbf{A})\big)$$
   where $\mathbf{X}$ is the block input.
3. **Position-wise feed-forward network**
$$\mathbf{F} = \text{GELU}\big(\mathbf{H}_1\mathbf{W}_1^\top + \mathbf{b}_1\big)\,\mathbf{W}_2^\top + \mathbf{b}_2,$$
   implemented as a two-layer MLP that expands the hidden dimension $d \to 4d \to d$ (a commonly-used width factor of 4). GELU has shown superior performance to ReLU in large language models.
4. **Second residual connection & layer normalisation**
$$\mathbf{H}_{\text{out}} = \text{LayerNorm}\big(\mathbf{H}_1 + \text{Dropout}(\mathbf{F})\big).$$

All dropout layers share the same rate specified in the configuration file, providing regularisation during training only. LayerNorms (`nn.LayerNorm`) stabilise gradients by re-centring and re-scaling the residual stream, while the two residual paths guarantee unimpeded gradient flow through depth.

The resulting tensor $\mathbf{H}_{\text{out}} \in R^{B\times L\times d}$ is forwarded either to the next decoder block or, in the last layer, to the projection layer that maps hidden states to vocabulary logits.

We use two distinct masks because masking enforces the left-to-right factorisation required for language modelling, whereas the *padding* mask prevents the model from allocating probability mass to meaningless [PAD] positions. The simultaneous use of both masks allows a single implementation to handle variable-length inputs without leaking future information.

Computational complexity

Both sub-layers are fully parallelisable across sequence positions. Multi-head attention scales as $\mathcal{O}(L^2 d)$ but runs efficiently on GPUs due to batched matrix multiplication; the feed-forward network dominates parameter count but not runtime. This block is wrapped into a `ModuleList` N times to build the full decoder-only transformer used for text generation in Part 2.

3.2 Description of the Decoder-only transformer model

The text-generation system follows **GPT-style "decoder-only" transformer**: a stack of identical decoder blocks that process an input sequence in an *autoregressive* fashion and predict the next token at every position.

1. **Token embedding layer**

   - Raw token IDs from the BPE tokenizer are mapped to dense vectors via:

     `nn.Embedding(vocab_size, d)`

     where $d =$ `embed_size`. This lookup table is learned during training, allowing the model to store semantic information about sub-words.

2. **Additive sinusoidal positional encoding**

   - Because the embedding layer has no notion of order, we add a fixed positional signal

     $$\texttt{PE} \in R^{T \times d}$$

     (pre-computed once using sine/cosine functions) to the token embeddings:

     $$\mathbf{z}_0 = \texttt{Embedding}(x) + \texttt{PE}_{0:T}$$

   - The sinusoidal form gives each position a unique spectrum that generalizes to sequence lengths not seen during training and introduces absolute and relative position cues.

3. **Dropout**

   - A dropout layer directly after the positional addition mitigates overfitting by randomly zeroing a fraction ($p =$ `dropout_p`) of elements.

4. **Stack of L decoder blocks**

   - Each block alternates **masked multi-head self-attention** and a **position-wise feed-forward MLP**, with residual connections and LayerNorm in "pre-norm" order. Blocks are then applied sequentially:

     $$\mathbf{z}_{\ell+1} = \texttt{DecoderBlock}_\ell(\mathbf{z}_\ell, \texttt{causal\_mask}, \texttt{padding\_mask}) \quad \text{for } \ell = 0 \dots L-1$$

5. **Masking Strategy**

   - **Causal (attention) mask** – an upper-triangular Boolean matrix generated once per maximum sequence length and sliced each forward pass. It forces attention weights for future positions ($j > i$) to be $-\infty$ before softmax, preserving autoregressive order.
   - **Key-padding mask** – a batch-specific mask where positions containing `[PAD]` are set to `True`; these tokens neither attend to nor are attended by other tokens, making sure that the padding has zero influence on the representation.

6. **Output Projection**

   - The final hidden states $\mathbf{B} \times \mathbf{T} \times \mathbf{d}$ are passed through a linear layer

     `nn.Linear(d, vocab_size)`

     to compute unnormalised logits for each vocabulary symbol. At inference time, logits feed a sampling head (greedy or nucleus).

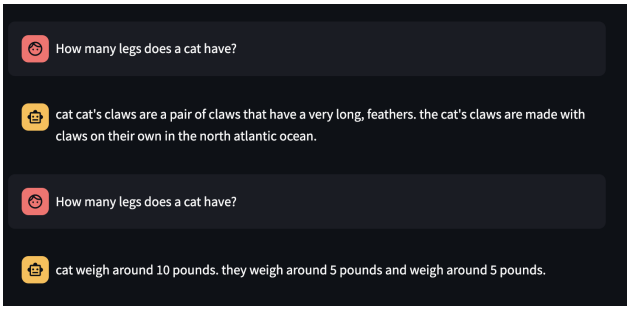

Figure 2: Sampling Strategy: Top-p vs Greedy

3.3 Byte-Pair Encoding (BPE) tokenizer

Instead of memorizing every full word, BPE starts with individual characters and repeatedly merges the most common adjacent pairs until it reaches a fixed vocabulary size (approximately 30k tokens in this case). When it later encodes text, it greedily applies these learned merges, so frequent words stay whole (e.g., `model`) while rare or novel words break into familiar sub-parts (e.g., `trans-form-er`). The result is a compact sub-word vocabulary that can spell out virtually any string without ever emitting an out-of-vocabulary `[UNK]` token. Compared with a traditional word-level tokenizer, which needs a huge dictionary and collapses unseen words to `[UNK]`, BPE trades a small increase in sequence length for far better coverage and parameter efficiency, making it the standard choice for modern decoder-only language models.

3.4 Results

Training the 36M-parameter decoder-only model (5 layers, 8 heads, $d = 512$) on the full 859k GooAQ question-answer pairs converged smoothly on a single T4 GPU. With a learning rate of 1×10^{-4}, batch size 128, and 0.1 dropout, mean cross-entropy fell from **1.23** $\rightarrow$ **1.14** over five epochs—about.[2] Loss dropped fastest in the first two epochs and began to plateau after the fourth, suggesting diminishing returns beyond $\sim$5 epochs at this scale. Each epoch took $\sim$37 minutes ($\approx$ 3 batches s^{-1}) with mixed-precision and **torch.compile** and no training instabilities (divergence or exploding gradients) were observed. Qualitatively, responses generated with nucleus sampling improved grammatically and by factual coherence by the third epoch; later epochs mostly tightened phrasing rather than adding new knowledge. Because we omitted a validation set, we cannot quantify overfitting, but anecdotal testing showed no memorization of exact training questions. Overall, the run demonstrates that a compact GPT-style model can learn reasonable QA behavior within a few hours of GPU time.

3.5 Discussion

Although cross-entropy dropped from 1.23 to 1.14, the model still struggles with factual QA. The two test prompts in the chatbot illustrate the gap: **top-p sampling** produced a fluent but nonsensical riff about "claws in the North Atlantic," while **greedy decoding** degenerated into repetitive weight estimates—both incorrect for the simple question *"How many legs does a cat have?"* Top-p clearly adds lexical variety, yet it also amplifies semantic drift (hallucinations); greedy keeps tighter to high-probability tokens but tends to loop or copy phrases, revealing that the underlying logits remain noisy. In short, decoding style affects *form*, but the core issue is representation quality.

Several factors limited performance. Training was capped at five epochs on a single T4 ($\approx$ 3 h total) with no validation set for early stopping, so the model likely under-fit the 859k-pair corpus. The dataset itself skews toward open-ended trivia rather than concrete numeric facts, leaving gaps like "cat legs." Memory constraints forced a modest 36M-parameter configuration ($d = 512$, 5 blocks), which, combined with a max sequence length of 128, restricts context and capacity. Finally, Colab disconnects and occasional

[2] Corresponds to a 7% relative improvement (per-token perplexity $\approx 3.4 \rightarrow 3.1$)

torch.compile autotune warnings required manual restarts, slowing iteration. These practical hurdles, more than algorithmic bugs, shaped the observed limitations.

3.6 Future improvements

With a larger time-and-compute budget the first lever is **scale**: moving from a 5-layer/36M-parameter GPT-mini to something in the 100–300M range (perhaps, 12–18 layers, $d \approx 768$–1024, 12–16 heads) would boost capacity without becoming unmanageable on a single A100 or a small multi-GPU node. Using that with a **longer context window** (256–512 tokens) so the model can see an entire Q-A pair plus surrounding dialogue, and increase the BPE vocabulary to 32k to reduce `<UNK>` fall-back. [3]

Second, supply **more and cleaner data**. Pre-train for a few epochs on open-domain datasets (e.g. The Pile) [4] then continue supervised fine-tuning on multiple QA sets (Natural Questions, WebGPT summaries) to broaden factual coverage. Synthetic QA from retrieval-augmented pipelines may help densify low-resource topics like simple animal facts [5]. Furthermore, we could consider introducing a held-out validation split to tune hyper-parameters and enable early stopping; add label smoothing and dropout-annealing to deal with overfitting.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] A. Brea, "Attention_is_all_you_need_from_scratch," https://github.com/BreaGG/Attention_Is_All_You_Need_From_Scratch, October 2024, gitHub repository, commit 6fa362f (accessed 2025-04-25). [Online]. Available: https://github.com/BreaGG/Attention_Is_All_You_Need_From_Scratch

[3] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, 2016, pp. 1715–1725.

[4] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," 2020.

[5] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020.