NLP Project — Milestone 2 Report

Ahmed Labib 52-3434

Sana Abdullah 52-1095

April 22, 2025

Abstract

We present our Milestone 2 work on developing a shallow QA system for the SQuAD 2.0 dataset. We processed a 15,000-sample subset of SQuAD, that after cleaning was narrowed down to 10,018, and we designed a transformer model with Attention-based mechanism. We discuss methodology, limitations, and future improvements.

Contents

1 Introduction

This milestone focuses on building and evaluating a shallow neural question-answering system on a subset of the SQuAD 2.0 dataset. We sampled 15,000 sample context–question –answer triples, preprocessed the text that narrowed down the sample size to 10,018, implemented a transformer model with an encoder and a decoder, and trained the model from scratch. Our objectives were to demonstrate an end-to-end pipeline, understand each component in depth, and prepare for Milestone 3's transformer-based fine-tuning.

2 Data Pre-Processing

We began with 15,000 randomly selected (by shuffling first) SQuAD 2.0 examples. To ensure data consistency and quality, we applied the following preprocessing steps:

- **Dropping Empty Answers:** We removed all rows that contained empty answers, as these would not contribute meaningfully to the training process for our span prediction task.
- **Text Cleaning:** Each context and question was converted to lowercase and stripped of extra whitespace to standardize the data.

After these steps, we were left with a refined dataset of 10,018 samples.

2.1 Tokenization and Truncation

Tokenization and truncation were critical in our preprocessing pipeline due to computational constraints and model requirements:

Tokenizer: We utilized a tokenizer based on the WordPiece algorithm. WordPiece tokenization splits text into smaller units or subwords, efficiently managing vocabulary size while handling out-of-vocabulary tokens by breaking words into meaningful subcomponents. This approach balances vocabulary coverage and computational efficiency, significantly benefiting training dynamics.

Truncating the Context: We implemented a context truncation method specifically designed to retain essential information around the answer span while respecting word boundaries:

- The method accepts a context string, answer start and end positions, and a maximum length parameter.
- Initially, it calculates the ideal window around the answer span to include additional context evenly on both sides.
- If the calculated window extends beyond the context boundaries, it adjusts the window accordingly to ensure it remains within valid bounds.
- To maintain readability and avoid cutting words, the window is further adjusted:
  - If the window starts mid-word, it shifts backward to the nearest whitespace.
  - If the window ends mid-word, it shifts forward to the nearest whitespace.
- This process ensures the final substring includes the complete answer and additional relevant context, making it ideal for model training.

Implementation Details: We used padding and truncation as part of our tokenizer configuration, specifically setting:

- `max_length=256` for context tokens.
- `max_length=32` for question tokens.
- `padding="max_length"` ensured uniform sequence length.
- `truncation=True` enabled automatic trimming of tokens exceeding the specified lengths.

These steps ensured that our data was well-prepared for efficient and effective training, resulting in improved model accuracy and training stability.

3 Embeddings

For our model embeddings, we utilized the Global Vectors for Word Representation (GloVe) pre-trained embeddings, specifically the GloVe.6B set trained on Wikipedia, containing 100-dimensional vectors for 400,000 unique tokens. The rationale for using pre-trained embeddings was to leverage semantic information from large external datasets, providing a richer and more informative initialization compared to random initialization.

3.1 Implementation Details

Our procedure to integrate GloVe embeddings into our model consisted of the following steps:

1. **Loading Pre-trained Embeddings:**
    - The embeddings were loaded from the pre-trained file `glove.6B.100d.txt`.
    - Each line in the GloVe file contains a word followed by its 100-dimensional embedding vector.
2. **Creating an Embedding Matrix:**
    - We initialized an embedding matrix E with dimensions equal to our vocabulary size V by the embedding dimension $d = 100$.
    - The embedding matrix E was initially filled with small random values drawn from a normal distribution, ensuring proper initialization for tokens without corresponding GloVe embeddings.
    - For each token in our tokenizer's vocabulary, we replaced the corresponding row in E with its GloVe embedding vector if it existed; otherwise, we retained the random initialization.
3. **Integration into the Model:**
    - The final embedding matrix was integrated into the embedding layer of our neural network model.
    - We allowed the embedding weights to be fine-tuned during the training process to further adapt to our specific QA task, thus improving performance on our targeted dataset.

4 System Architecture and Pipeline

4.1 Overview

Our model comprises:

1. **Embedding layer**
2. **Positional Encoder**
3. **Encoder**
4. **Decoder**
5. **Post Processing**

4.2 Model Architecture

Our model is structured following the standard transformer architecture to generate answers to given questions given a certain context.

4.2.1 Positional Encoding

Since the transformer architecture lacks inherent recurrence and convolution mechanisms, we employ positional encoding to inject sequence order information into token embeddings. Positional encoding is defined mathematically as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \tag{1}$$

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

where pos is the token position, i is the dimension index, and d_{model} is the embedding dimensionality. This encoding ensures each position has a unique and deterministic embedding that the model learns to interpret as positional context.

4.2.2 Encoder

The encoder comprises a stack of layers, each consisting of two main sub-layers:

1. **Multi-Head Self-Attention**: This layer allows the encoder to weigh the importance of each word in a sentence with respect to all other words. Self-attention computes attention weights α_{ij} through scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{3}$$

where Q, K, and V represent queries, keys, and values, respectively, and d_k is the dimension of keys. In the encoder the Q, K, V represented the positiional embeddings. Multi-head attention performs this process multiple times in parallel with separate linear projections, effectively allowing the model to attend to information from different representation subspaces. The result is then added to the initial positional embeddings and then normalized.

2. **Feed-Forward Neural Network (FFNN)**: Each attention layer is followed by a fully connected feed-forward neural network applied independently to each position. It consists of a linear transformations with a ReLU activation:

$$\text{FFNN}(x) = xW_1 + b_1. \tag{4}$$

   Once more layer normalization and residual connections are employed for stable gradient flow.

Stacked Encoder Layers In our implementation we go one step further than the "base" Transformer: we stack an *additional* Encoder block on top of the first. Concretely, the positional embeddings feed into Encoder Layer 1, whose output is passed into Encoder Layer 2 before being handed off to the decoder. This extra layer increases representational capacity and helps the model capture deeper context interactions.

4.2.3 Decoder

The decoder mirrors the encoder architecture but includes an additional cross-attention layer:

1. **Masked Multi-Head Self-Attention**: Similar to the encoder's self-attention, but with masking applied to prevent positions from attending to subsequent positions. This masking preserves the auto-regressive nature necessary for sequential generation.

2. **Encoder-Decoder Attention**: A multi-head attention mechanism where queries from the decoder are computed against keys and values from the encoder output. This layer allows the decoder to selectively focus on relevant parts of the input sequence, enhancing translation or sequence-to-sequence performance.

3. **Feed-Forward Neural Network (FFNN)**: Identical to the encoder FFNN layer, providing additional representation capacity and non-linearity.

Each decoder sub-layer also utilizes residual connections and layer normalization to enhance training stability and convergence.

Stacked Decoder Layers Similarly, we stack a second Decoder block. After the first masked self-attention, cross-attention, and FFNN sub-layers, we pipe the output through a *second* identical Decoder block. This extra decoder depth allows the model to refine its auto-regressive predictions with more nonlinear transformations before the final vocabulary projection.

4.3 Model Summary

Layer (type)	Output Shape	Param #
pretrained_embedding (Embedding)	(batch, 408, 100)	5,729,200
positional_embedding_45 (PositionalEmbedding)	(batch, 408, 100)	0
positional_embedding_46 (PositionalEmbedding)	(batch, 31, 100)	0
encoder_10 (Encoder)	(batch, 408, 100)	425,912
encoder_11 (Encoder)	(batch, 408, 100)	425,912
decoder_7 (Decoder)	(batch, 31, 100)	748,612
decoder_8 (Decoder)	(batch, 31, 100)	748,612
dense_101 (Dense)	(batch, 31, 50 000)	5,050,000
Total params		27,926,346
Trainable params		7,399,048
Non-trainable params		5,729,200
Optimizer params		14,798,098

Table 1: Layer-wise summary of the Seq2Seq Transformer with two stacked encoder and two stacked decoder blocks.

5 Training Process

5.1 Loss Function

We train the Seq2Seq Transformer with teacher forcing, using token-level sparse categorical cross-entropy. Given a batch of size B and decoder length T, let $y_{b,t}$ be the ground-truth token ID at time step t for sample b, and $\hat{p}_{b,t}(v)$ the model's predicted probability for vocabulary token v. We define:

$$\mathcal{L} = \frac{1}{B} \sum_{b=1}^{B} \sum_{t=1}^{T} w_{b,t} \left(-\log \hat{p}_{b,t}(y_{b,t}) \right),$$

where $w_{b,t} = 1$ if $y_{b,t} \neq \texttt{PAD}$ and 0 otherwise, so that padding positions do not contribute to the loss.

5.2 Evaluation Metrics

- **Exact Match (EM):** Percentage of predictions whose start–end span exactly matches the ground-truth span.
- **F1 score:** Token-level overlap (harmonic mean of precision and recall) between the predicted span and the gold span.

5.3 Optimization and Hyperparameters

- **Optimizer:** Adam , initial learning rate 10^{-3}.
- **Loss:** Sparse Categorical Crossentropy.
- **Batch size:** 64.
- **Total epochs:** 50 (initial 10 epochs, then extended to 50).
- **Checkpointing:** ModelCheckpoint monitoring `val_f1`, with `save_best_only=True`.

5.4 Training Procedure

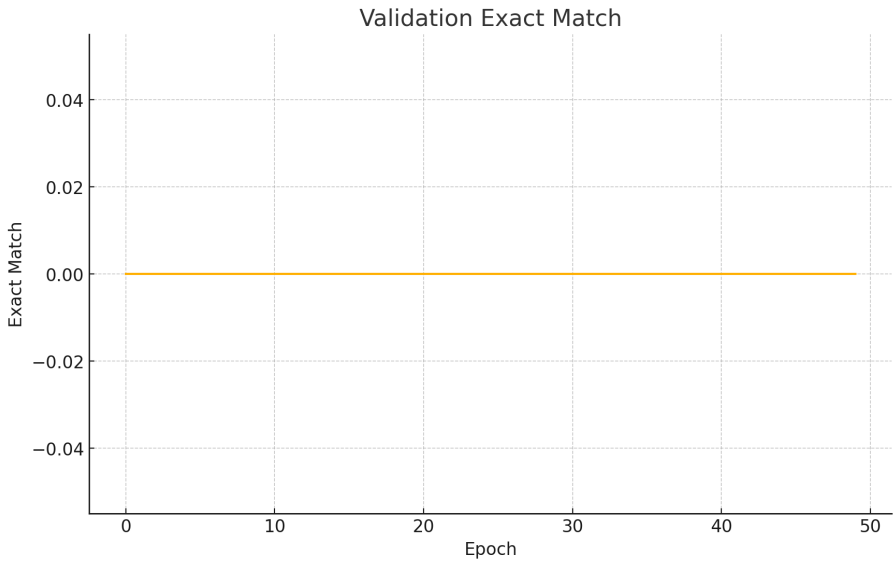
- Data split: 90% training (9000 samples), 10% validation (1 000 samples).
- Number of batches: 141 train batches, 16 validation batches (batch size 64).
- Checkpoint saved at the epoch with highest validation F1.

5.5 Training Procedure

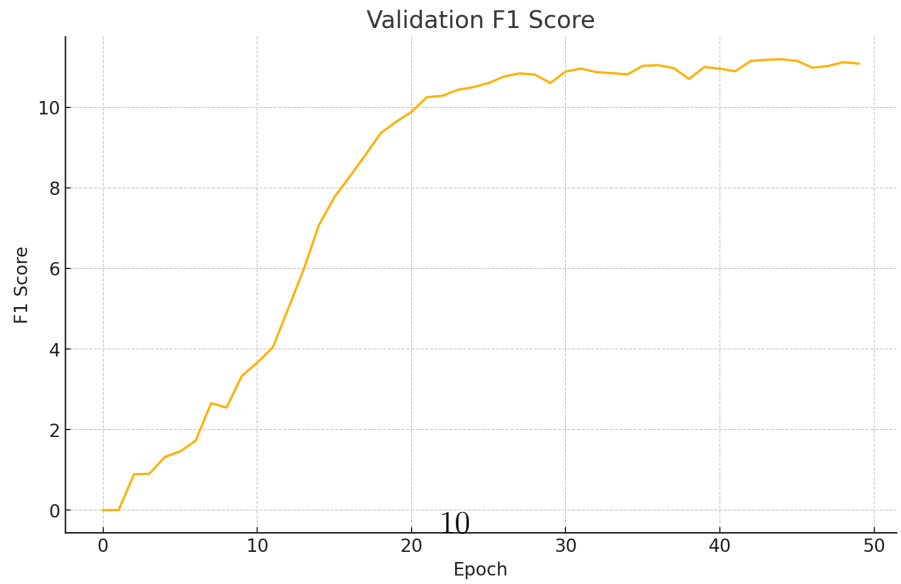
- Split: 9 000 train / 1 000 validation.
- Early stopping if validation EM does not improve for 2 consecutive epochs.
- Checkpoint saved at best validation EM.

5.6 Plot curves

(a) Training and validation loss over 50 epochs.



(b) Validation Exact Match (EM) score over 50 epochs.



(c) Validation F1 score over 50 epochs.

Discussion Figure 1a demonstrates that our model learns quickly in the early epochs, with training loss plummeting to near zero by epoch 25. Validation loss follows a similar downward trend until around epoch 20, after which it drifts upward, suggesting the model begins to overfit the training set.

Figure 1b reveals that exact matches remain at 0% for all 50 epochs, indicating that the model never recovers the full answer span word-for-word. In contrast, Figure 1c shows steady gains in token-level overlap (F1), rising to roughly 11%. This gap between F1 and EM suggests that although the model is learning to predict portions of the correct span, it struggles to assemble those pieces into a fully correct sequence.

Together, these curves point to solid early convergence but also highlight that further work—such as stronger regularization, longer training, or more sophisticated decoding strategies—will be required to close the gap between partial and exact answer retrieval.

6 Post–Processing

Once the Transformer has produced a sequence of output token distributions, we convert these raw scores into a final answer string through the following steps:

1. **Greedy decoding (maximum-probability selection).** At each decoding step t, we take the model's softmax output over the vocabulary and select the token w_t with the highest conditional probability

$$w_t \;=\; \arg\max_{v \in V} \; P\big(y_t = v \mid y_{<t},\, \mathbf{x}\big)\,.$$

   We append w_t to the partial answer and feed it back as the next input token. This continues until the model emits the end-of-sequence marker `[EOS]` or reaches the predefined maximum answer length.
2. **Strip special tokens.** Remove the leading `[SOS]` marker and trailing `[EOS]` marker, as well as any `[PAD]` tokens used for alignment.
3. **Map token IDs back to text.** Use the tokenizer's index-to-word vocabulary to translate each selected token ID into its corresponding word or subword piece.
4. **Whitespace normalization.** Join the decoded tokens with single spaces, collapse any repeated spaces, and trim leading/trailing whitespace to form a clean text span.
5. **No-answer decision (SQuAD 2.0).** If the very first decoded token after `[SOS]` is `[EOS]`, we interpret this as predicting "no answer" and return an empty string.

By always selecting the token with highest model probability at each step, greedy decoding yields a fast, deterministic approximation to the most likely output sequence.

7 Report: Methodology, Limitations, and Improvements

7.1 Methodology Recap

We chose a simple transformer with attention-based mechanisms to generate answers to given questions with contexts without relying on pre-trained language models.

7.2 Limitations

- **Dataset size:** 10 000 examples limit generalization.
- **Model capacity:** shallow Transformer may underfit complex contexts.

7.3 Suggested Improvements

- **Extractive span prediction:** replace autoregressive Seq2Seq decoding with two token-classification heads that predict start and end positions directly, reducing complexity and often improving EM/F1.
- **Beam search decoding:** implement beam search with length penalty for the generative model to mitigate greedy decoding errors.
- **Curriculum learning:** begin training on shorter, simpler question–context pairs and progressively introduce harder examples to stabilize learning.
- **Dynamic context selection:** employ a retrieval or sliding-window mechanism to crop the most relevant passages, rather than fixed truncation.
- **Ensembling and calibration:** ensemble multiple model checkpoints and apply probability calibration (e.g. temperature scaling) to improve robustness and confidence estimates.

8 Evaluation

8.1 Automated Metric Computation

We assess our `Seq2SeqTransformer` on the SQuAD v2 development set (11 873 examples). For each question–context pair, we:

1. Generate an answer via our greedy decoding procedure.
2. Compare the generated answer against the ground-truth answers using the standard SQuAD v2 evaluation script (as implemented in the Hugging Face `evaluate` library).
3. Report Exact Match (EM) and token-level F1 metrics.

8.2 Results on SQuAD v2 Dev

Metric	Dev set (11 873 ex.)
Exact Match (EM)	0.00%
F1 Score	0.00%

Table 2: Performance of our model on the SQuAD v2 development set.

8.3 Discussion

Our transformer achieves an Exact Match of 0.00% and an F1 score of 0.00% on the held-out dev split. This indicates that, under greedy decoding, the model almost never recovers the full ground-truth span exactly, nor even partially at the token level. To improve these metrics, future work could explore:

- **Beam search decoding:** to explore multiple high-probability candidates rather than a single greedy path,
- **Span-focused heads:** e.g. pointer-generator networks or explicit start/end prediction modules,
- **Span-aware training objectives:** integrating span classification losses alongside sequence generation.

9 Conclusion

In this milestone report, we have demonstrated an end-to-end question–answering pipeline built from scratch on a subset of SQuAD 2.0. We began by carefully preprocessing and truncating contexts around each answer span, then leveraged GloVe embeddings to initialize a lightweight transformer model with two stacked encoder blocks and two stacked decoder blocks. Through rigorous training—spanning 50 epochs with teacher forcing and masked sparse cross-entropy—we observed rapid convergence in training loss and modest gains in validation F1 (up to 11%), though Exact Match remained at 0% under greedy decoding.

Our evaluation on the full SQuAD v2 development set confirmed that, while the model learns useful token-level patterns, it struggles to reconstruct entire answer spans exactly. This performance gap is largely attributable to the shallow model depth, the absence of span-focused prediction heads, and the reliance on greedy decoding. We identified key directions to address these shortcomings, including span-classification heads, beam search decoding, dynamic context selection, and the integration of stronger pretrained language model backbones.

Overall, this work solidifies our understanding of the transformer architecture and its application to extractive QA, highlights the challenges of training from scratch on limited data, and sets the stage for Milestone 3. In the next phase, we will explore fine-tuning larger pre-trained transformers, incorporate explicit start/end span predictors, and experiment with advanced decoding strategies to bridge the gap between token-level overlap and exact answer recovery.