NLP Project — Milestone 3 (Span QA) Report

Ahmed Labib `52-3434`
Sana Abdullah `52-1095`

May 18, 2025

Contents

1 Introduction **2**
1.1 Motivation 2

2 Dataset **2**
2.1 Source Corpus and Sub-Sampling 2
2.2 Tokenisation 2
2.3 Pre-processing Pipeline 2

3 Methodology **3**
3.1 Base Model 3
3.2 Experiment 1: Full Fine-Tuning 3
3.3 Experiment 2 – Feature-Based Fine-Tuning (Encoder Frozen) 3
3.4 RAG Pipeline Setup 4
3.5 Experiment 3 – Zero-Shot Prompting 5
3.6 Experiment 4 – Chain-of-Thought Prompting 6
3.7 ChatBot with Memory 6

4 Results **7**
4.1 Fine-Tuning Comparison (Experiments 1 vs. 2) 7
4.2 Prompting Comparison (Experiments 3 vs. 4) 8

5 Limitations **9**

6 Conclusion **10**

1 Introduction

1.1 Motivation

Why a lightweight model (DistilBERT) and what we hope to show by contrasting full vs. partial fine-tuning.

2 Dataset

2.1 Source Corpus and Sub-Sampling

We begin with the original **SQuAD 1.1** corpus (88 599 (context, question, answer) triples). To keep training time manageable on the course hardware we randomly sample **20 000** examples (≈23 % of the full set) with a fixed seed. This subset is then split 80 / 20:

- **Train** – 16 000 examples
- **Dev** – 4 000 examples (for hyper-parameter search)

For final, comparable reporting we still evaluate on the *official* SQuAD validation partition (10 570 examples) and quote Exact-Match and F1.

2.2 Tokenisation

All experiments share the same input encoding:

- **Tokenizer** – `AutoTokenizer` linked to `distilbert-base-uncased`.
- **Max sequence length** – 256 tokens.
- **Truncation policy** – *only_second* (never truncate the question, truncate the context if needed).
- **Sliding-window stride** – 128 tokens; long contexts are broken into overlapping windows so that an answer spanning a cut line appears intact in at least one chunk.
- **Padding** – *max_length* to enable efficient batching on GPU.

2.3 Pre-processing Pipeline

Every raw triple is transformed into one or more training windows by the following sequence:

1. **Tokenise** the concatenated sequence `[CLS]` q `[SEP]` c `[SEP]` with the hyper-parameters above. If the context exceeds 256 tokens it is split into overlapping windows (stride 128).
2. **Map windows to originals** The tokenizer returns an `overflow_to_sample_mapping` so we know which window came from which original example.

3. **Character–to–token alignment** For each window we consult the `offset_mapping` (char-level start–end for every token). If the gold answer span lies fully inside the current window we record its `start_positions` and `end_positions` (token indices). Otherwise both labels are set to the `[CLS]` position; such windows are ignored by the loss.

4. **Feature set** The final dataset contains `input_ids`, `attention_mask`, `start_positions`, and `end_positions`— ready for the `Trainer`.

We have 16 k training triples expand to ≈22 k fixed-length examples, each padded or truncated to 256 tokens.

3 Methodology

3.1 Base Model

We adopt **DistilBERT**, a 6-layer, 66 M-parameter Transformer encoder distilled from `BERT-base`. While the vanilla encoder outputs a contextual vector $\mathbf{h}_i$ for every sub-token, it does not natively solve question-answering. We therefore load `AutoModelForQuestionAnswering` which appends a lightweight, task-specific *span-prediction head* (two parallel linear projections) on top of the final hidden state. The head learns to assign a start and end score to each token, from which the answer span is extracted.

3.2 Experiment 1: Full Fine-Tuning

All weights—encoder *and* QA head—are updated during training. Hyper-parameters are shown in Table 1.

Parameter	**Value**
Batch size (train / eval)	128 / 128
Epochs	10
Learning rate	3×10^{-5}
Weight decay	0.01
Mixed precision (fp16)	on (GPU)
Optimizer	AdamW (default in `Trainer`)

Table 1: TrainingArguments used in Experiment 1.

Trainer setup The `Trainer` API receives the tokenised train / dev splits, the above arguments, and a custom `compute_metrics` callback that reports SQuAD Exact-Match and F1 each epoch.

3.3 Experiment 2 – Feature-Based Fine-Tuning (Encoder Frozen)

Freezing the encoder DistilBERT comprises a 6-layer Transformer encoder followed by a two-layer span-prediction head. By setting `param.requires_grad = False` for all parameters in `model.distilbert`, we lock the encoder weights at their pre-trained values. During training only the QA head's weights receive gradient updates:

$$\theta^{(t+1)}_{\text{encoder}} = \theta^{(t)}_{\text{encoder}}, \quad \theta^{(t+1)}_{\text{QA-head}} = \theta^{(t)}_{\text{QA-head}} - \alpha \nabla_{\text{QA-head}} \mathcal{L}.$$

Justification

- **Domain match.** DistilBERT was distilled from BERT trained on English Wikipedia and BookCorpus—the same sources as SQuAD contexts—so its embeddings already capture the necessary language patterns.
- **Preventing catastrophic forgetting.** With only 20 000 fine-tuning examples, updating all encoder layers risks erasing valuable general-purpose representations.
- **Efficiency.** Freezing $\sim$99.5% of parameters reduces VRAM usage by $\approx$40% and cuts training time by more than half compared to full fine-tuning.

Optimisation settings We reuse the same hyper-parameters as Experiment 1 (see Table 1) except:

- Learning rate increased to 5×10^{-4}
- Number of epochs reduced to 3

All other settings (batch size, weight decay, fp16, logging frequency, optimizer) remain identical to the full fine-tuning configuration. Training converges in three epochs, at which point validation loss plateaus.

3.4 RAG Pipeline Setup

Our end-to-end Retrieval-Augmented Generation (RAG) pipeline is constructed in five stages:

1. Corpus Assembly We extract every context paragraph from the SQuAD `train` and `validation` splits and remove exact duplicates:

- Load `raw["train"]["context"] + raw["validation"]["context"]`
- Deduplicate via `unique = list(dict.fromkeys(contexts))`
- Wrap each in `Document(page_content=...)`

This ensures our knowledge base contains only unique passages.

2. Chunking To fit within the model's token window and improve retrieval precision, each passage is split into overlapping chunks:

- `chunk_size=500`, `chunk_overlap=50` using `RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)`
- Overlap preserves boundary context, reducing edge-case information loss.

3. Embedding & Indexing Each 500-token chunk is encoded into a 384-dimensional vector:

- `embeddings = HuggingFaceEmbeddings()` (behind the scenes uses `all-MiniLM-L6-v2`)
- Build FAISS index via
  `db = FAISS.from_documents(split_docs, embeddings)`
- FAISS enables sub-millisecond nearest-neighbor lookup even on tens of thousands of vectors.

4. Retrieval For each incoming question, we retrieve the top-4 most semantically similar chunks:

```
retriever = db.as_retriever(search_kwargs={"k": 4})
```

Choosing $k = 4$ balances having enough context to answer without overwhelming the generator.

5. Generation For each question, we take the four retrieved chunks and stitch them into a single text-to-text prompt that asks the model to produce the answer. Our generator is Google's FLAN-T5-Large, a sequence-to-sequence Transformer with approximately 770 million parameters (24 encoder layers, 24 decoder layers, hidden size 1024, feed-forward size 4096). T5-Large was pretrained on the C4 corpus under the unified "text-to-text" objective, enabling tasks such as translation, summarization, and question answering to all be cast as conditional generation. The "FLAN" variant then receives additional instruction-tuning on a mixture of prompts, which improves zero-shot and few-shot generalization.

In practice, we truncate the concatenated context+question to 512 tokens, ask FLAN-T5-Large to generate up to 64 new tokens, and decode greedily (temperature 0.0, top_p 1.0) to maximize consistency. We load the model with automatic device mapping so that its 770 M parameters are split across GPU and CPU memory, ensuring it fits within our hardware constraints while still delivering fast, deterministic inference.

3.5 Experiment 3 – Zero-Shot Prompting

We wrap this RAG pipeline in LangChain's `RetrievalQA` with a minimal prompt:

```
{context}

Question: {question}
Answer:
```

This baseline measures how well the model can answer with no examples or reasoning cues.

3.6 Experiment 4 – Chain-of-Thought Prompting

All components (retriever, generator, decoding, chain type) are identical to Experiment 3. We only swap in a CoT prompt that includes:

- A *mini demonstration* (2+ 2 example) so the model learns the tagging pattern.
- A delimiter for unambiguous final-answer extraction.

```
Example:
Passages:
2 + 2 equals four.

Question: What is 2 + 2?
Step-by-step reasoning:
1. Two plus two equals four.
#### 4
----

Use the following passages to answer the question.
Think step-by-step. When  y o u re  done, write
#### <answer> on its own line.

Passages:
{context}

Question: {question}

Step-by-step reasoning:
1.
```

We then use `re.search(r"####\s*(.*)", raw_output)` to isolate the final answer for EM, F1, ROUGE-L and BLEU evaluation. This isolates the effect of CoT prompting on answer quality and interpretability.

3.7 ChatBot with Memory

To turn our RAG-powered QA pipeline into a multi-turn chatbot that retains context, we wrap the FLAN-T5-Large generator in LangChain's `ConversationChain` with a windowed memory buffer. This allows the assistant to remember the last k exchanges, improving coherence and follow-up understanding.

Model Pipeline We reuse the same sequence-to-sequence LLM as in Experiments 3–4:

- **Model:** `google/flan-t5-large` (770 M parameters), loaded with automatic device mapping.
- **Tokenization:** `AutoTokenizer.from_pretrained(MODEL_ID)`.

- **Generation Pipeline:** `pipeline("text2text-generation", max_length=512, max_new_tokens=64)` wrapped in a `HuggingFacePipeline`.

Memory Configuration We use a sliding-window memory of size $k = 6$, storing the last six user–assistant exchanges:

```
memory = ConversationBufferWindowMemory(
    k=6,
    ai_prefix="Assistant",
    human_prefix="User",
    return_messages=True,
)
```

This buffer retains both the `human_prefix` and `ai_prefix` on each line, so the model sees context like:

```
User: How do I fine-tune?
Assistant: You can freeze the encoder by ...
User: And what about generation?
...
```

Conversation Chain Finally, we construct the chat interface:

```
chatbot = ConversationChain(
    llm=llm,
    memory=memory,
    verbose=False,
)
```

Each time `chatbot.predict(input_text)` is called, the last six turns are prepended to the prompt, enabling the assistant to answer follow-up questions and maintain dialog coherence.

Usage Example

```
>>> chatbot.predict("Hi, can you remind me what model we're using?")
"Assistant: We're using google/flan-t5-large, a 24-layer encoder-decoder..."
```

By leveraging windowed memory, the chatbot can carry forward definitions, clarifications, and user preferences over several turns without retraining or manual context stitching.

4 Results

4.1 Fine-Tuning Comparison (Experiments 1 vs. 2)

In Experiments 1 and 2 we compare two strategies for adapting DistilBERT to SQuAD 1.0:

We evaluate both on the held-out validation set using four complementary metrics:

Exact Match (EM) Percentage of predictions that exactly match a ground-truth answer span. This is the gold standard for extractive QA.

F1 Score Token-level overlap between prediction and ground-truth (harmonic mean of precision and recall). Unlike EM, it gives partial credit for near-matches.

ROUGE-L Longest-common-subsequence recall between generated and reference answer. Captures generative fluency and coverage in case of paraphrasing.

BLEU N-gram precision measure, highlighting how closely the model's phrasing matches the reference.

Experiment	EM (%)	F1 (%)	ROUGE-L (%)	BLEU (%)
Full fine-tuning (Exp 1)	74.60	83.51	76.18	53.32
Feature-based (Exp 2)	77.75	85.90	78.11	56.54

Table 2: Performance of full vs. feature-based fine-tuning on SQuAD 1.0.

Results Feature-based fine-tuning (freezing the encoder) unexpectedly outperforms full fine-tuning on all four metrics. By preventing catastrophic forgetting in the pretrained encoder and focusing updates on the small QA head, we retain stronger general-purpose representations while still adapting to the SQuAD task, resulting in higher EM, F1, ROUGE-L, and BLEU scores.

4.2 Prompting Comparison (Experiments 3 vs. 4)

In Experiments 3 and 4 we freeze all model weights and compare two prompting strategies on our retrieval-augmented FLAN-T5-Large generator:

Metric Rationale Although our reader is now a fully generative model (FLAN-T5-Large), we retain the four standard QA metrics for consistency and comparability:

- *Exact Match (EM)*: Even when the model generates free-form text, EM checks whether the generated answer string exactly matches one of the reference spans. It is a strict correctness criterion and remains a clear signal of precise retrieval or generation.
- *F1 Score*: Token-level overlap gives partial credit when the model's phrasing differs slightly from the reference (e.g. "1852" vs. "the year 1852"). It complements EM by rewarding close but not identical answers.
- *ROUGE-L*: Measures the longest common subsequence between predicted and reference answers, capturing fluency and paraphrase ability. For a generative model that may rephrase, ROUGE-L better reflects how much of the essential content is preserved.
- *BLEU*: An n-gram precision metric borrowed from machine translation. While it can be brittle on very short answers, BLEU still provides a lens on how closely the model's wording aligns with the reference phrasing.

In purely extractive QA EM and F1 are paramount, but in a RAG setting with generative output, ROUGE-L and BLEU help us quantify not just correctness but also the quality and faithfulness of the generated text. If a metric seems less appropriate for very short answers, we note that its scores should be interpreted with caution.

Experiment	EM (%)	F1 (%)	ROUGE-L (%)	BLEU (%)
Zero-Shot (Exp 3)	63.40	70.23	64.23	49.21
Chain-of-Thought (Exp 4)	13.75	29.64	26.52	4.93

Table 3: Zero-shot vs. CoT prompting on RAG with FLAN-T5-Large.

Results Zero-shot prompting yields moderate performance on our generative QA task , but introducing explicit chain-of-thought reasoning causes a dramatic drop across all metrics. Because FLAN-T5-Large is a free-form generator, the CoT template drives it to produce longer reasoning chains, which appears to come at the expense of concise, precise answer text. In other words, while chain-of-thought improves interpretability, it reduces the model's ability to focus generation on the minimal answer span in this generative RAG setting.

5 Limitations

Although our study provides insights into fine-tuning strategies and prompting methods for RAG and extractive QA, several limitations should be noted:

- **Dataset scope.** We evaluate exclusively on SQuAD1.0/2.0 contexts and a 20000-example subset of the training split. This narrow domain and reduced training size may limit the generality of our conclusions to other QA benchmarks or to truly open-domain settings.
- **Retrieval oracle.** We build the FAISS index directly over SQuAD contexts and do not report retrieval accuracy (Recall@K). Any retrieval failures are absorbed into the end-to-end metrics, making it hard to disentangle retriever vs. generator performance.
- **Metric suitability.** Exact Match and token-level F1 are designed for extractive span-selection. When applied to generative outputs (Experiments 3–4), they can under-represent correctness if the model paraphrases or includes reasoning. Although we include ROUGE-L and BLEU to capture fluency, these metrics also have known brittleness on very short answers.
- **Prompting strategies.** We compare only zero-shot vs. a single CoT template. We do not explore few-shot exemplars beyond our one-shot demo, self-consistency sampling, or alternative CoT variants (e.g. "refine" or "map-reduce" prompting).
- **Model capacity.** For generative experiments we use a single model (FLAN-T5-Large). Results may differ with smaller (e.g. base) or larger (e.g. XL, XXL) checkpoints, or with non-T5 architectures.
- **Hardware constraints.** Chunk size, overlap, and $k = 4$ retrieval were chosen to fit GPU memory and prompt-length limits. Different settings could yield different trade-offs between context coverage and generation quality.
- **Qualitative analysis.** We focus on quantitative metrics and do not perform human evaluation of generated reasoning chains or answer faithfulness. Thus, the

interpretability benefits of CoT prompting remain assumed rather than empirically verified.

- **Comparison across paradigms.** Experiments 1–2 use an extractive QA head on DistilBERT, while Experiments 3–4 use a generative T5 model. Applying the same metrics (EM/F1/ROUGE-L/BLEU) to both paradigms can be misleading, since generative outputs may be correct yet penalized for verbosity or paraphrasing.
- **Verbosity penalty.** Chain-of-thought prompting often surrounds the correct answer with multi-step reasoning. EM and F1 will mark those as incorrect if the answer string is not isolated perfectly, even though the model demonstrated correct reasoning.
- **Evaluation granularity.** Our metrics capture only surface-level text overlap, not the factual correctness or logical soundness of the intermediate reasoning steps. A model may arrive at the right answer via flawed logic, or vice versa, without affecting EM/F1 scores.

6 Conclusion

In this report we explored two axes of span-QA design: how to adapt a pretrained encoder to SQuAD via full vs. feature-based fine-tuning (Experiments 1–2), and how to prompt a frozen generative model in a retrieval-augmented setting via zero-shot vs. chain-of-thought (Experiments 3–4).

Our key findings are:

- **Feature-based fine-tuning (Exp 2) outperforms full fine-tuning (Exp 1)** on Exact Match (77.75 % vs. 74.60 %) and F1 (85.90 % vs. 83.51 %), as well as in ROUGE-L and BLEU. Freezing the transformer encoder preserves its general-purpose language representations, avoids catastrophic forgetting, and requires far fewer compute resources.
- **Zero-shot prompting (Exp 3) vastly outperforms CoT prompting (Exp 4)** in our generative RAG setup: zero-shot yields 63.40 % EM versus 13.75 % with chain-of-thought, and similarly large gaps in F1, ROUGE-L and BLEU. While CoT improves interpretability, it distracts the model from concise span recovery.

Taken together, these results suggest the following deployment recipe for a resource-constrained chatbot:

1. Use **feature-based fine-tuning** of a lightweight encoder (DistilBERT) to build a fast, high-accuracy extractive QA backend when ground-truth spans are available.
2. For open-domain, generative question answering via RAG, prefer **zero-shot** prompts over chain-of-thought to maximize answer precision and throughput.
3. Reserve CoT prompting for debugging or cases where step-by-step explanations are explicitly required, rather than as the default production strategy.

This combination delivers strong accuracy, low latency, and minimal overhead—key properties for practical, multi-turn conversational systems operating under hardware and time constraints. Future work may revisit CoT in conjunction with self-consistency or few-shot exemplars to recover some of its interpretability benefits without sacrificing precision.