Milestone 2

Abstract

In this project, we build a question answering (QA) system using a deep learning model. The system takes a question and a related paragraph (context) as input and tries to find the correct answer from the paragraph. We use a filtered version of the SQuAD 2.0 dataset that includes only 20,000 answerable question-context-answer examples. The goal is to train a model that can predict the start and end of the answer span in the context. We are limited to using only three layers in our model (excluding input and output layers), and we experiment with different types of models such as RNNs, LSTMs, GRUs, and attention-based networks. This report explains how we prepared the data, designed the model, trained it, and evaluated the results. We also discuss what worked well, what was challenging, and how the system could be improved in the future.

Keywords: Question Answering, SQuAD 2.0, Deep Learning, LSTM, Attention.

1. Introduction

Question Answering (QA) is an important task in Natural Language Processing (NLP), where a system tries to find the answer to a question based on a given text (called the context). QA models are used in many real-world applications like search engines, virtual assistants, and chatbots. In this project, we focus on extractive question answering, where the answer must be a continuous span of text taken directly from the context.

We use a subset of the SQuAD 2.0 dataset, which is one of the most widely used benchmarks for QA tasks. It was one of several datasets suggested for this project, and we chose it because of its popularity and relevance to real-world QA problems. We filtered the dataset to include only 20,000 answerable question-context-answer examples.

As a starting point, we experimented with a simple setup where the model tries to answer questions without using any context. As expected, the model struggled to perform well, which confirmed the importance of including context information in extractive QA systems.

We implemented multiple models using PyTorch, including LSTM-based, GRU-based, and attention-based architectures. We used the HuggingFace `tokenizers` library for efficient text preprocessing. Our model designs were restricted to a maximum of three layers (excluding input and output), and we were instructed to use either RNNs or attention-based models—but not a combination of both. However, through a brief literature review, we found that combining RNNs with attention mechanisms often gives better results, especially in small-scale models like ours.

Another challenge was the limited computational resources available, which made it difficult to experiment with larger models or perform many training iterations. Despite these constraints, we aimed to build a functional and reasonably accurate QA system.

This report describes how we processed the data, built and trained the models, evaluated their performance, and what we learned from the process. We also discuss the limitations of our approach and propose directions for future improvements.

2. Data Preprocessing

Effective data preprocessing was a crucial part of our pipeline to ensure the model could handle the input format correctly and learn from the SQuAD 2.0 dataset under our project constraints.

2.1 Dataset Preparation

We used the SQuAD 2.0 dataset, which includes both answerable and unanswerable questions. For our task, we focused only on answerable questions. The dataset was downloaded in JSON format from the official website and parsed using custom Python scripts.

To reduce training time and memory usage, we sorted all answerable examples by context length and selected the 20,000 shortest ones. Each example included a question, a context paragraph, and the ground truth answer span (given as character-level start and end positions within the context).

2.2 Tokenization

We used **Byte Pair Encoding (BPE)** to tokenize the text. A vocabulary of 10,000 subword tokens was generated from the dataset using HuggingFace's `tokenizers` library. This was appropriate given that the dataset had approximately 90,000 unique raw words. BPE allowed us to break rare or unknown words into more frequent subword units, reducing the effective vocabulary size while still capturing meaningful tokens.

All text was lowercased as part of preprocessing. We avoided removing punctuation or other characters, since the ground truth answer spans in SQuAD are given as character-level indices. Any modification of the text could invalidate these indices and cause incorrect training labels.

2.3 Input Construction

Each input sequence was constructed by concatenating the tokenized question and context with special tokens in the following format:

[SOS] question tokens [SEP] context tokens [SEP]

We used the following special tokens throughout preprocessing:

- **[UNK]** — Represents unknown tokens not in the vocabulary.
- **[PAD]** — Used to pad shorter sequences to match a fixed length. These tokens are masked out during training.
- **[MASK]** — Included in the vocabulary for completeness, though not used in this task.
- **[SOS]** — Marks the beginning of a sequence.
- **[EOS]** — Marks the end of a sequence. Not strictly needed for extractive QA, but included for consistency.
- **[SEP]** — Separates the question and the context in the input sequence.

For example, a sample input could look like:

[SOS] What is the capital of France? [SEP] France is a country in Europe. Its capital is Paris. [SEP]

2.4 Truncation and Padding

Inputs longer than the model's maximum allowed sequence length were **truncated** from the end of the context. This was necessary due to GPU memory constraints and to maintain a consistent input size. Shorter sequences were **padded** with the `[PAD]` token.

To avoid the model attending to these padded positions during training and evaluation, we applied **attention masks**. These masks assign a weight of 0 to `[PAD]` tokens and 1 to all valid tokens, ensuring the model only learns from meaningful parts of the input.

2.5 Label Processing

Since the original dataset provides **character-level start and end indices** of the answer in the context, we had to convert these into **token-level indices** after tokenization.

This was done by:

1. Tokenizing the context separately.
2. Mapping the character-based start and end positions of the answer to the corresponding tokens.
3. Adjusting the indices to align with the combined `[SOS] question [SEP] context` input.

During training, the model learns to predict these token indices as the start and end positions of the answer.

2.6 Dataset Splits and Batching

We used the original **training and development (dev)** splits provided in SQuAD 2.0 and applied our filtering strategy to both sets to keep only answerable examples. These filtered sets were then used for training and validation respectively.

We trained using a **batch size of 32**, and all sequences in a batch were padded to the same length to enable efficient parallel computation. Padding masks were used to ensure correct attention and loss computation.

To efficiently handle the data during training, we implemented a **custom PyTorch Dataset class** that inherits from `torch.utils.data.Dataset`. This class handles preprocessing on-the-fly and returns the model inputs and labels in a consistent format. We then used **PyTorch DataLoaders** to create batches, shuffle the training data, and manage parallel data loading during training and evaluation.

2.7 Libraries and Tools

We used the following tools in our preprocessing pipeline:

- **HuggingFace Tokenizers** — For BPE-based tokenization and handling special tokens.
- **Custom Python scripts** — For loading and filtering the dataset, mapping character indices to token indices, and constructing input sequences with proper masks.
- **PyTorch Dataset and DataLoader** — For batching, shuffling, and parallelized data loading.