Question answering with DistilBERT

Introduction - Extractive question answering

Extractive question answering (QA) is a task where a model identifies and extracts a specific segment of text from a given context to answer a question. Unlike generative models that create new text, extractive QA focuses on pinpointing the exact phrase or sentence from a predefined passage. This approach is particularly effective for tasks like retrieving facts, summarizing key points, or navigating dense documents. While the method has its limitations, one of its biggest advantages is that it requires much less computational power than generative tasks.

Model and technologies

As a model we use DistilBERT, a lightweight version of the BERT model, fine-tuned for the task of question answering. DistilBERT is designed to retain 97% of BERT's performance while being 40% smaller and 60% faster. It has a similar achitecture to BERT, although with significantly less parameters. It is trained via knowledge distillation to mimic the behaviour of its larger counterprart. It works by training the smaller model the larger model's non-target outputs of the final classification layer with a softmax activation function. These are called soft target probabilities and are quite representative of a model's generalization capabilites. During the training, a so-called distillation loss is applied, which is calculated with the following formula:

$$L_{ce} = \sum_i t_i \cdot log(S_i)$$

Where t_i is the propability estimated by the teacher and s_i is the probability estimated by the student. A softmax-temperature is also introduced to control the smoothness of the output distribution of the softmax activation function:

$$p_i = \frac{exp(\frac{Z_i}{T})}{\sum(\frac{Z_j}{T})}$$

Where T is the temperature and Z_i is the model score for the i th class. For tasks like knowledge distillation, a smoother output distribution, hence a higher temperature value is more beneficial. [1] A model pretrained with this method can later be finetuned for certain downstream tasks, such as sentiment analysis, or, as in our case, extractive question-answering.

Base Model

The exact model that we used for our project was a checkpoint of the uncased DistilBERT model (which doesn't differentiate between lower- and uppercase characters) finetuned for question-answering on the SQuAD v1.1 Dataset (distilbert-base-uncased-distilled-squad) [2].

Tokenization

The model uses the AutoTokenizer from Hugging Face's Transformers library to preprocess questions and contexts into input tokens. We used the tokenizer finetuned with the model (checkpoint distilbert-base-uncased-distilled-squad) but we did not finetune the tokenizer further.

Parameter Efficiency

In order to reduce the training time and the require computational power, we decided to train our model using low-rank adaptation. This freezes the weights of the layers of the original model and, instead of updating them, approximates their changes as the product of two lower-rank matrices:

$$W' = W + \Delta W = W + A \cdot B$$

Where W is the original weight matrix, ΔW is the weight update and A and B are two lower-rank matrices. Finetuning with this method reduces the number of trainable parameters, hence the training costs significantly without any major compromises on the accuracy. Furthermore, it also prevents catastrophic forgetting, which can happen when finetuning pretrained neural networks. [3]

Quantization

Another method to reduce computational costs without any significant effects on model performance is quantization. This method consists of representing the model during inference with lower precision data types (such as 4-bit floats or integers). This is usually done only for inference, not for training, but in our case, since we don't modify the weights of the original model, it can be represented in lower precision during the entire finetuning (except for the lora matrices, which we are training). [4]

Dataset

The MRQA (Machine Reading for Question Answering) dataset was developed as part of the MRQA 2019 Shared Task to evaluate the generalization capabilities of reading comprehension systems. This dataset unifies 18 question-answering (QA) datasets into a standardized extractive QA format, allowing models to test their ability to answer questions by identifying text spans in a given context. The dataset consists of three splits, each containing six of the original datasets. The training split contains datasets usually made up of Wikipedia articles and general trivia questions. The two remaining datasets are for model development and evaluation and contain questions from different domains than the training split, mostly from biology textbooks in order to evaluate the generalizing performance of the trained models. [5]

Training and evaluation

Preprocessing

The MRQA dataset is a well-curated dataset with the questions, contexts and answer start- and end indices all provided in the same format. Even their tokenized versions are present in the dataset, however, since we wanted to use the tokenizer finetuned with the model (distilbert/distilbert-base-uncased-distilled-squad), so using them was not an option. The preprocessing was quite straightforward, all we had to do was to tokenize the questions and contexts with the pretrained tokenizer and map the start and end indices from the text to the tokens. Since some of the evaluation metrics rely on the actual text format of the answer, the evaluation set also had to include the ids of the examples, so that the text format of the answer could be retrieved.

Training

One major constraint for the training was that we did not have unlimited access to hardware powerful enough for the task. This meant that we had to rely on parameter-efficient training methods to reduce computational time. We implemented both quantization and low-rank adaption. We opted for a 4-bit quantization, which improved training speed by a lot. We have experimented with multiple lora ranks and ended up using a rank of 4, with which striked a good balance between efficiency and accuracy. The limited resources also meant that using the entire dataset was not an option. We started with training on 20% of the mrqa dataset, but that did not lead to an improvement in the evaluation metrics and also led to training curves that plateaued after 3-4 epochs (as seen on Figure 1), so we gradually increased it to 40%, with which we managed to improve model performance. We used the AdamW optimizer for all of our experiments, and, since the output layer of the model is softmax activated, the loss we used was cross-entropy. Since the intended use of the model was to answer general trivia questions, we only used the training split of the mrqa dataset, and created our training, testing and evaluation sets from that.

Evaluation

Quantifying a model's performance on extractive question answering is not a straightforward task. We chose three metrics to evaluate the model's performance, the exact match, F_1-score and the bleu score. The exact match is a strict evaluation metric used primarily for question answering. It checks whether the predicted output matches the ground truth exactly, including punctuation and word order. It is a binary metric, awarding a score of 1 for an exact match and 0 otherwise. The F_1 Score is the harmonic mean of precision (the proportion of correctly predicted tokens to all predicted tokens) and recall (the proportion of correctly predicted tokens to all tokens in the ground truth) [6]. A good F_1 score for question answering tasks is above 85. For reference, the model checkpoint we finetuned further performed 86.9 on the dev set of the SQuAD v1.1 dataset (the dataset it was finetuned on).

$$F_1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 \cdot \frac{precision \cdot recall}{precision + recall} = \frac{2TP}{2TP + FP + FN}$$

BLEU (Bilingual Evaluation Understudy) is a metric designed for evaluating machine translation but can also be applied to question answering tasks, especially if the expected answers are short. It measures the similarity between the predicted text and one or more reference texts based on overlapping n-grams (e.g., unigrams, bigrams). BLEU uses a weighted precision approach, penalizing overly short outputs through a brevity penalty. The BLEU score is calculated the following way:

$$BLEU = BrecityPenality \cdot GeometricAveragePrecisionScore$$

$$BrecityPenality = \begin{cases} 1, & if c > r \\ e^{(1-r/c)}, & if c \leq r \end{cases}$$

Where c is the length of the extracted answer and r is the length of the target answer and

$$GeometricAveragePreision(N) = exp(\sum_{n=1}^{N} w_n \log p_n) = \prod_{n-1}^{N} p_n^{w_n} = (p_1)^{\frac{1}{4}} \cdot ((p_2)^{\frac{1}{4}}) \cdot (p_3)^{\frac{1}{4}} \cdot (p_4)^{\frac{1}{4}}$$

Where p_n is an n-gram and w_n is its respective weight. As BLEU scores on different datasets, languges and tasks are not really comparable, we used it mostly to measure the change in the model performance before and after the finetuning. We calculated the BLEU-score with a maximum n-gram order of 4. [7, 8]

Results

Since we were working with an already finetuned model, we did not expect a huge increase in any of the above metrics upon finetuning. We calculated the metrics on the evaluation dataset created by us both before and after the training, and most of our experiments did not improve the model performance or even led to somewhat worse metrics than before the training. Our most successful experiment (Figure 1, bottom picture) managed to improve the Exact Match score from 55.57 to 57.47, the F_1 Score from 68.04 to 70.45, and the BLEU score from 0.4375 to 0.4550. These are slight improvements, which could be improved with further optimization of the hyperparameters (we experimented with a hyperparameter optimization algorithm but had to abandon it due to resource constraints).

Inference pipeline, ML as a service

We built an inference pipeline for the model that automatically fetches context for the questions through a google search and the Wikipedia API. It works by creating a search term from the question through part-of-speech tagging each word in the original question using the nltk library and removing words and punctuation marks that fall into categories deemed irrelevant for a google search term. The final search term

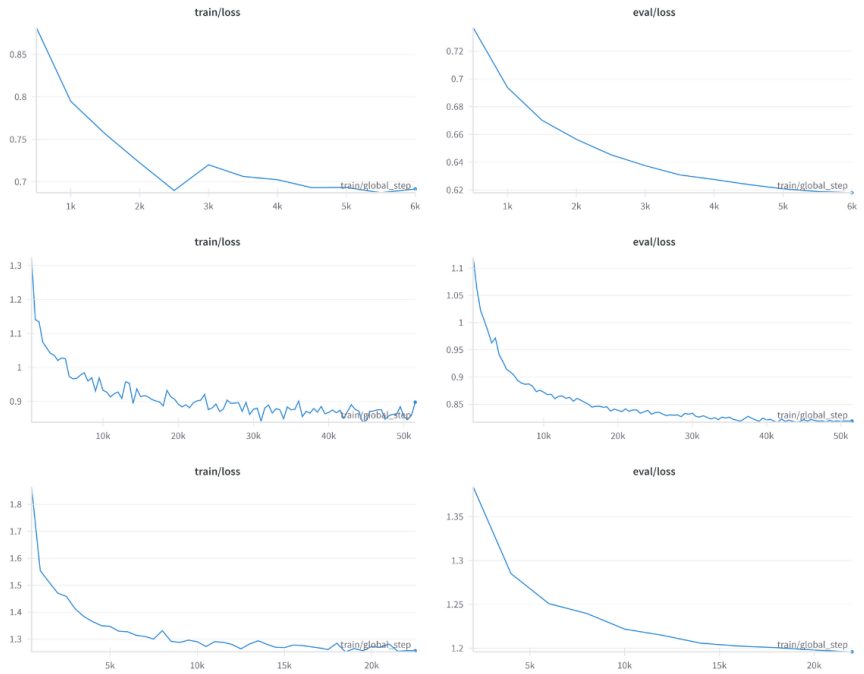

Figure 1: Hyperparameter optimization

is then created by appending the remaining words with 'wikipedia' so that we retrieve the Wikipedia articles in the top results.

Following this, the titles of the top 10 results are checked with the Wikipedia API and the summaries are returned for the valid Wikipedia articles. These summaries will be used as contexts for the question answering task. The summaries are then tokenized along with the questions. Since the model has a maximum capacity for taking in 512 token long contexts, but was finetuned with contexts no longer than 384 tokens, contexts that exceed that length are broken down into smaller chunks and fed to the model individually. A softmax function is then applied to the outputs, and the softmax score for the start and end indices is added together, effectively creating a confidence score (ranging from 0 to 2). Before applying the softmax function, however, another step is necessary. In most transformer-based models a [CLS] token is added at the beginning of every sequence by the tokenizer, which does not correspond to any of the input tokens explicitly, but is rather supposed to be an aggregate representation of the following sequence. This is mainly used for sequence-level downstream tasks, such as sentiment analysis or spam detection []. When the answer is relatively hard to find, our finetuned model tends to return the [CLS] token as the answer. Therefore, before applying the softmax function, I replace the values at the respective place of the [CLS] token with negative infinity in the output logits for both the start and end indices. The answer with the highest confidence score is returned as the final answer. This is done for all the retrieved valid Wikipedia articles, and the final answers are sorted based on their confidence scores and returned in a descending order.

The project includes a basic reference implementation of Machine Learning as a Service. The service enables users to input questions and receive answers. The answers can be rated, and the ratings can be used for further tuning of the model. The ratings are saved along with the question, answer, and the full context, and saved individually in JSON files. In a real-life scenario, the ratings could be saved in a database and used for tuning the model.

The whole solution runs in a docker container. The API was written in Python using FastAPI. The front-end was written in Svelte.

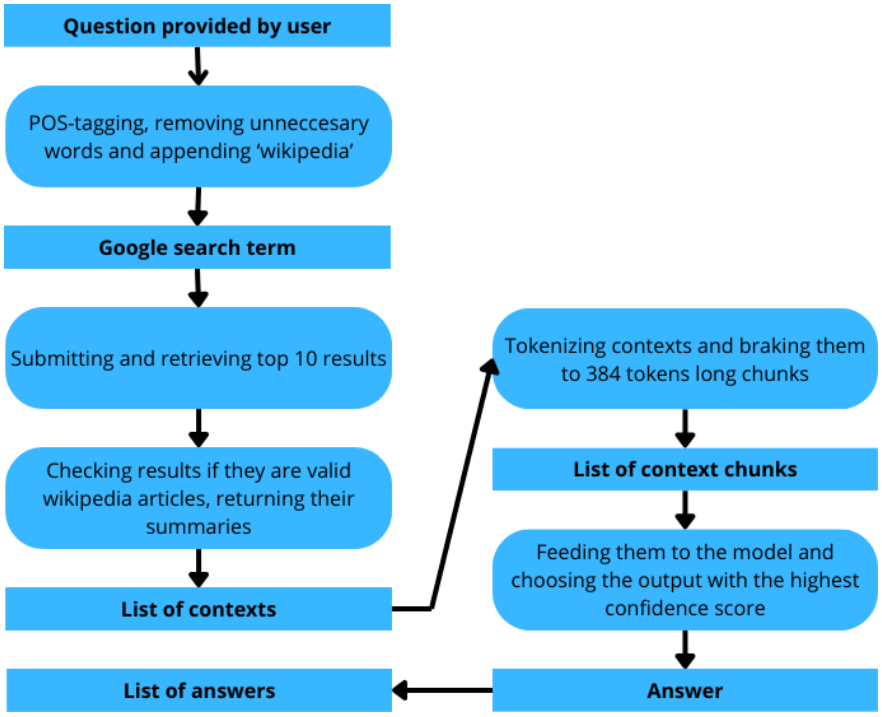

Figure 2: Inference pipeline

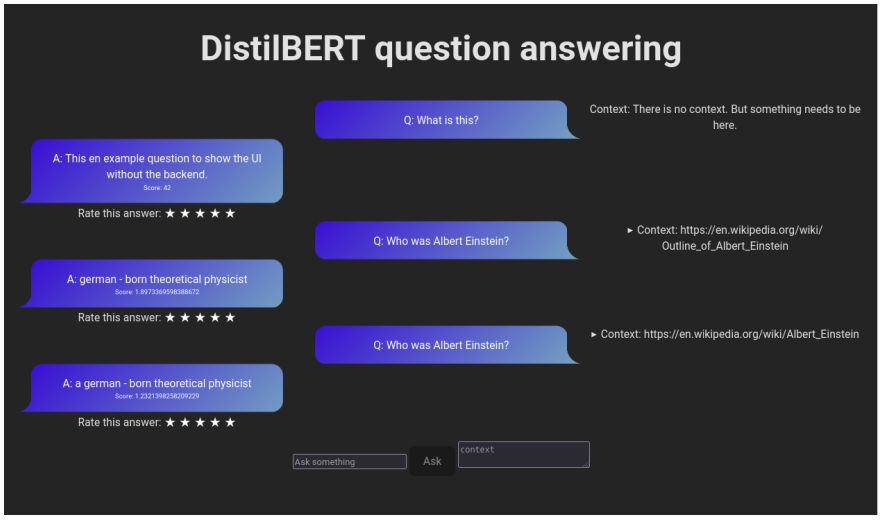

Figure 3: Front-end

Use cases

- Customer support
  - Automated response based on FAQ
- Knowledge bases
  - Trained on documents that are not publicly available a similar solution can provide answers on confidential material

Resources

[1] https://arxiv.org/abs/1910.01108
[2] https://huggingface.co/distilbert/distilbert-base-uncased-distilled-squad
[3] https://arxiv.org/abs/2106.09685
[4] https://huggingface.co/docs/optimum/concept_guides/quantization
[5] https://paperswithcode.com/paper/mrqa-2019-shared-task-evaluating
[6] https://en.wikipedia.org/wiki/F-score
[7] https://towardsdatascience.com/foundations-of-nlp-explained-bleu-score-and-wer-metrics-1a5ba06d812b
[8] https://huggingface.co/spaces/evaluate-metric/bleu