Quick review of the last Presentation about Chatbots, NLPs and Machine Learning

Question-Answering (QA) models have been around for a while now, but there is still room for improvement. One of the biggest challenges is creating models that can understand big context - that is, they can take into account a large amount of information when answering a question.

One of the main limitations of traditional QA models is that they are designed to work with a limited amount of context. For example, a model might be trained to answer questions based on a single paragraph or even just a sentence.

However, real-world questions often require more extensive context. For instance, if someone asks 'What is the capital of France?', a good answer would take into account not only the name of the city (Paris), but also its location, history, culture, and other relevant factors. Creating models that can handle this kind of complexity is a major goal of current research

In our last presentation we talked about NLP and ML techniques to make our chatbots make work.Machine learning algorithgms make the Natural Language Processing possible,Natural Language Processing models will make QA models possible and NLP QA models will make chatbots that are capable of understanding deep contextual data

Haystack, Huggingface and Roberta QA

Haystack is an open-source natural language processing (NLP) framework that allows users to build and customize question-answering systems. It is commonly used for information retrieval and text mining tasks.

Hugging Face is a company and an open-source community that focuses on developing state-of-the-art natural language processing (NLP) technologies. They provide a wide range of NLP tools and libraries, including pre-trained models, datasets, and transformers, which are widely used by researchers and developers in the NLP community.

Robert QA refers to a fine-tuned version of the RoBERTa language model that is specifically trained for the task of question answering. It is based on the SQuAD2.0 dataset and is commonly used for extractive QA tasks. Roberta QA is available as a pre-trained model on Hugging Face and can be fine-tuned on custom datasets for specific use cases.

Challenges in implementation of Roberta

While using RoBERTa for natural language processing tasks can yield impressive results, there are several challenges that can arise during its implementation. Some of the key challenges include:

1. Model size and computational requirements: RoBERTa models are quite large in size, consisting of hundreds of millions or even billions of parameters. This poses challenges in terms of storage, memory, and computational resources required to train and deploy the models effectively. Implementing RoBERTa may require high-performance hardware and infrastructure to handle the computational demands.
2. Training data and fine-tuning: RoBERTa models typically require a substantial amount of labeled training data to achieve optimal performance. Acquiring and preprocessing large-scale datasets for fine-tuning can be time-consuming and resource-intensive. Additionally, fine-tuning RoBERTa models often involves careful experimentation with hyperparameters and training setups to obtain the best results.
3. Data preprocessing and tokenization: Properly preprocessing and tokenizing the input data to match the RoBERTa model's requirements can be challenging. RoBERTa models have specific tokenization rules, and the input data needs to be encoded in a compatible format. Ensuring consistency between the tokenization process during pre-training and fine-tuning is crucial to obtain accurate results.

Addressing these challenges often involves a combination of technical expertise, computational resources, and careful experimentation. It is essential to have a thorough understanding of the specific requirements of your task, along with the capabilities and limitations of RoBERTa models, to successfully implement and leverage them for your NLP applications.

Code review of Roberta QA in Python

Learn how to build a question answering system using Haystack's DocumentStore, Retriever, and Reader. Your system will use Game of Thrones files and will be able to answer questions like "Who is the father of Arya Stark?". But you can use it to run on any other set of documents, such as your company's internal wikis or a collection of financial reports.
To help you get started quicker, we simplified certain steps in this tutorial. For example, Document preparation and pipeline initialization are handled by ready-made classes that replace lines of initialization code. But don't worry! This doesn't affect how well the question answering system performs.

1) Initializing the DocumentStore

We'll start creating our question answering system by initializing a DocumentStore. A DocumentStore stores the Documents that the question answering system uses to find answers to your questions. In this tutorial, we're using the InMemoryDocumentStore, which is the simplest DocumentStore to get started with. It requires no external dependencies and it's a good option for smaller projects and debugging. But it doesn't scale up so well to larger Document collections, so it's not a good choice for production systems. To learn more about the DocumentStore and the different types of external databases that we support, see DocumentStore.
Let's initialize the the DocumentStore. The DocumentStore is now ready. Now it's time to fill it with some Documents.

2) Preparing Documents

Download 517 articles from the Game of Thrones Wikipedia. You can find them in *data/build_your_first_question_answering_system* as a set of *.txt* files.

Use TextIndexingPipeline to convert the files you just downloaded into Haystack Document objects and write them into the DocumentStore:

The code in this tutorial uses the Game of Thrones data, but you can also supply your own *.txt* files and index them in the same way.
As an alternative, you can cast you text data into Document objects and write them into the DocumentStore using

DocumentStore.write_documents().

3) Initializing the Retriever

Our search system will use a Retriever, so we need to initialize it. A Retriever sifts through all the Documents and returns only the ones relevant to the question. This tutorial uses the BM25 algorithm. For more Retriever options, see Retriever. Let's initialize a BM25Retriever and make it use the InMemoryDocumentStore we initialized earlier in this tutorial:
The Retriever is ready but we still need to initialize the Reader.

4) Initializing the Reader

A Reader scans the texts it received from the Retriever and extracts the top answer candidates. Readers are based on powerful deep learning models but are much slower than Retrievers at processing the same amount of text. In this tutorial, we're using a FARMReader with a base-sized RoBERTa question answering model called deepset/roberta-base-squad2. It's a strong all-round model that's good as a starting point. To find the best model for your use case, see Models. Let's initialize the Reader:
We've initalized all the components for our pipeline. We're now ready to create the pipeline.

5) Creating the Retriever-Reader Pipeline

In this tutorial, we're using a ready-made pipeline called ExtractiveQAPipeline. It connects the Reader and the Retriever. The combination of the two speeds up processing because the Reader only processes the Documents that the Retriever has passed on. To learn more about pipelines, see Pipelines. To create the pipeline, run: The pipeline's ready, you can now go ahead and ask a question!

6) Asking a Question

1. Use the pipeline run() method to ask a question. The query argument is where you type your question. Additionally, you can set the number of documents you want the Reader and Retriever to return using the top-k parameter. To learn more about setting arguments, see Arguments. To understand the importance of the top-k parameter, see Choosing the Right top-k Values.

Future expectations from Question-Answering models

Large Language Models (LLMs) are Deep Learning models trained to produce text. With this impressive ability, LLMs have become the backbone of modern Natural Language Processing (NLP). Traditionally, they are pre-trained by academic institutions and big tech companies such as OpenAI, Microsoft and NVIDIA. Most of them are then made available for public use. This plug-and-play approach is an important step towards large-scale AI adoption — instead of spending huge resources on the training of models with general linguistic knowledge, businesses can now focus on fine-tuning existing LLMs for specific use cases.

Most teams and NLP practitioners will not be involved in the pre-training of LLMs, but rather in their fine-tuning and deployment. However, to successfully pick and use a model, it is important to understand what is going on "under the hood". In this section, we will look at the basic ingredients of an LLM:

- Training data
- Input representation
- Pre-training objective
- Model architecture (encoder-decoder)

Each of these will affect not only the choice, but also the fine-tuning and deployment of your LLM.

Language modelling is a powerful upstream task — if you have a model that successfully generates language, congratulations — it is an intelligent model. However, the business value of having a model bubbling with random text is limited. Instead, NLP is mostly used for more targeted **downstream tasks** such as sentiment analysis, question answering and information extraction. This is the time to apply **transfer learning** and reuse the existing linguistic knowledge for more specific challenges. During fine-tuning, a portion of the model is "freezed" and the rest is further trained with domain- or task-specific data.

Let's summarise some general guidelines for the selection and deployment of LLMs 1. When evaluating potential models, be clear about where you are in your AI journey: At the beginning, it might be a good idea to experiment with LLMs deployed via cloud APIs.Once you have found product-market fit, consider hosting and maintaining your model on your side to have more control and further sharpen model performance to your application.
2. To align with your downstream task, your AI team should create a short-list of models based on the following criteria: Benchmarking results in the academic literature, with a focus on your downstream taskAlignment between the pre-training objective and downstream task: consider auto-encoding for NLU and autoregression for NLGPrevious experience reported for this model-task combination. 4. The short-listed models should be then tested against your real-world task and dataset to get a first feeling for the performance.
5. In most cases, you are likely to achieve a better quality with dedicated fine-tuning. However, consider few-/zero-shot-learning if you don't have the internal tech skills or budget for fine-tuning, or if you need to cover a large number of tasks.
6. LLM innovations and trends are short-lived. When using language models, keep an eye on their lifecycle and the overall activity in the LLM landscape and watch out for opportunities to step up your game.

Finally, be aware of the limitations of LLMs. While they have the amazing, human-like capacity to produce language, their overall cognitive power is galaxies away from us humans. The world knowledge and reasoning capacity of these models are strictly limited to the information they find at the surface of language. They also can't situate facts in time and might provide you with outdated information without blinking an eye. If you are building an application that relies on generating up-to-date or even original knowledge, consider combining your LLM with additional multimodal, structured or dynamic knowledge sources.

Semantic search

Take the leap from using keyword search on your own documents to semantic search with Haystack.

- Store your documents in the database of your choice (Elasticsearch, SQL, in memory, FAISS)
- Perform question driven queries.

Expect to see results that highlight the very sentence that contains the answer to your question. Thanks to the power of Transformer based language models, results are chosen based on compatibility in meaning rather than lexical overlap.

Information Extractor

Automate the extraction of relevant information from a set of documents that pertain to the same topics but for different entities.
Haystack can:

- Apply a set of standard questions to each document in a store
- Return a NO_ANSWER if a given document does not contain the answer to a question

Say you have the financial reports for different companies over different years. You can gather a set of standard questions which are applicable to

each financial report, like *what is the revenue forecast for 2020*? or *what are the main sources of income*?. Haystack will try to find an answer for each question within each document!
We've seen this style of application be particularly effective in the sphere of finance and patent law but we see a lot of potential in using this to gain a better overview of academic papers and internal business documents.

FAQ Style Question Answering

Leverage existing FAQ documents and semantic similarity search to answer new incoming questions. The workflow is as follows:

- Store a set of FAQ documents in Haystack
- The user presents a new question
- Haystack will find the closest match to the new question in the FAQ documents
- The user will be presented with the most similar Question Answer pair

Haystack's flexibility allows you to give new users more dynamic access to your existing documentation.

Risk related to Machine learning models