Today I would like to give a demo of a bot that I built some weeks ago.

OpenAI recently released its chatGPT model, and has attracted a lot of attention from people in various industries.

It also released the GPT3.5 API last week, so now we can call the ChatCompletion model after upgrading the OpenAI package to version 0.27. Before this, the most recent model is GPT3.

The bot uses GPT3 and I haven’t tried out the new model yet. It was a "learning by doing" project, it is simple and small, easy to implement (thanks to openAI API). I will walk you through the steps of building it.

Also during my exploration in Language Model Model, I found some interesting aspects of LLM, I think it’s worth sharing,

So I would start with the key aspects in LLM, and after that demo how I implemented the bot.

When Deep Learning was introduced to NLP field around 2013, the popular deep learning technology in NLP was mainly relied on RNN model.

It uses RNN as typical feature extractor, and encoder-decoder as typical architecture for various specific tasks.

e.g. Text understanding (text classification, sentiment analysis etc.), Text generation (machine translation, dialogue system, text summarization etc.)

Encoder is used for language modelling, which usually uses RNN model (e.g. LSTM) to encode given words, then predicts the statistical likelihood of next word.

Decoder is used for text generation. There’re some algorithms, they provides different solution in how to select next output according to the probability distribution generated by the encoder.

For example, top-n sampling algorithm is a sampling based approach, it randomly picks next output from top-n most likely candidates. If n=1, it picks the most likely word, the n can control the variation in the output. Can be used in story-telling such kind of creation tasks.

But using RNN as encoder-decoder doesn’t seem to have advantage compared to non-deep learning approach in terms of the performance.

One reason could be the representations of features extracted by RNN are not expressive enough, it didn’t effectively absorb the knowledge in the data.

For example, same word maps to same embedding, however, the word in different context may carry different meaning.

e.g. bank: a nice walk by the river bank, another sentence: walk to the bank and get cash.

This is overcome by introducing transformer.

Another reason could be the training dataset is limited. Because by that time before the LLM occurs, there’s no universal model to handle all NLP tasks. Different tasks are handled specifically, and it only uses the dataset for that specific task. The limited dataset limits the potential of scaling model parameters.

NLP 2.0 introduces large language model which significantly changes the NLP field.

It uses transformer as pre-trained model.

Transformer uses multi-head attention, which is more expressive than RNN.

Because Transformer will output contextualised embedding that carries the context information of the word in that sentence, and it can be different than initial input embedding.

In first example, the contextualised embedding for bank may be more related to river, the bank in second example may be more related to cash.

This information will be used as weight to create a new context embedding for the bank. In math, it’s about doing dot product between bank and other words, then softmax the value to normalise the similarity score between 0 and 1, then use the score as weight to do the linear combination of its context word to generate a new word embedding for the bank, the new embedding of bank in two different sentence has different embedding. This is basically how self-attention works.

When doing the context embedding, Bert and GPT does it slightly differently.

in Bert, the output of each token has the entire context in the sentence.

Bi-directional: Make prediction using information from two directions.

e.g. I went to \_\_ for study.

→ guess: (high probability) library, classroom

→ won’t guess: (low probability) shopping mall, pub

Like we do English cloze, combining information at both ends of the space.

In GPT, the output of each token in the sentence only has the context of previous words.

Autoregressive: Make prediction from left to right.

Like we do in writing, we must be thinking while writing.

Autoregression simply means **regression on self.** It mean given the previously observed outcomes of that sequence, predicting a future outcome of a sequence.

The difference among them led to Bert being better at natural language understanding tasks, GPT better at natural language generation tasks.

What happens as the model scales increases?

An interesting finding is there’re two different performance patterns while the number of model parameter increases.

X axis: the number of model parameter, increasing exponentially.

Y axis: the performance

Some model abilities can be improved by scaling up the number of parameters.

Those abilities occur in both small model and large model. But as the number of model parameter increases exponentially, the performance gets better.

Some model abilities are suddenly unlocked only when the number of parameters reached to certain threshold. And LLMs are not directly trained to have these abilities. This is called emergent ability.

Researchers found these abilities in both small and large model shares some similarities. For example, they tend to be knowledge-intensive task. Language understanding tasks are knowledge-intensive tasks, such as text classification, sentiment analysis etc.

Those abilities only occur in large model tend to be tasks that requires multi-step reasoning. For example, train the model to be able to write summery for news, and train the model to be able to write the code, and surprisingly the LLM can write summery for the code.

Despite scaling up the parameters improves the performance of the model, some abilities can also be improved by using prompting.

Fine-tuning vs. prompting is another major difference between Bert and GPT.

Bert and GPT both have their pre-trained model, to use the model in domain specific task, (let’s say, identify if the article contains illegal information or not), Bert and GPT does it differently.

Bert uses fine-tuning. Fine-tuning would require some data to be collected in that specific domain, then the data will be used to fine-tune the model, usually the parameters of the layers’ that are close to the output layer will be modified. It’s like 2nd-time learning.

Prompting is different. Prompting only shows the model some examples or guidance in that domain, then the model’s performance can be improved.

Providing some examples is called few-shot prompting.

Providing reasoning examples is called chain-of-thought prompting.

Providing zero examples is called zero-shot prompting or instruct. This is what chatGPT uses.

Prompting’s not fine tuning, because the parameters of the model won’t be modified. it’s like saying to the model: “hey bro, take a look into it, use it as reference, but don’t take it personally”. It’s really amazing, and till now the reason is unknown, “we don’t know why, but it works”.

* few-shot prompting:

GPT 3.0 uses few-shot prompting. It is effective because it is found prompt is more effective to tell the model how to do the task rather than describing the task using human language to the model.

The example of Xn, Yn, {prompt: Xn, ideal generated text: Yn} doesn’t have to be correctly mapped. If swap the value of Yi and Yj, it doesn’t affect the performance of the model, which means the LLM doesn’t learn a mapping function between Xn and Yn, otherwise the wrong labelling would cause a serious issue.

But if change Yi to another random answer that is beyond the scope, the performance of the model will be affected. Turns out what is matters to the model is the distribution of Xn and Yn. But the real reason behind it is still a mystery.

* Chain-of-thought:

The pictures at right shows an example of chain-of-thought prompting. The standard prompting shows that the model’s output answer to the second question is wrong.

But after adding reasoning steps to the first question, and then use this as an example, ask the second question again, the model’s output answer to the second question is correct.

Assumption: it may remind the model of the reasoning knowledge in its training materials, and activate its dead memory, so it began to imitate those reasoning process and deduce them step by step. Still, a mystery.

* Zero-shot prompting:

ChatGPT is an example of zero-shot prompt. User can directly give prompts to ChatGPT, say “translate this to English”, “write me a business proposal” without giving examples. It is an improvement, the model is trained to understand the human instruction much better.

But You can improve its performance by giving more setting: “imagine you are a consultant, and now you are responsible for …, you want to achieve a goal …, now write me a business proposal” The quality of answer can be improved.

Next part comes to my code demo of the Q&A bot.

This is very straight forward app. It only answers questions related to its extracted knowledge base.

The knowledge base used here is two KPMG financial report for year 2020 and 2021.

It’s very simple. The main logic is, before feeding the question to the openai completion model API, I constructed some context in the prompting, and feed the context with the question to the API, then constrain the answer only in the provided context, in this way, it becomes a domain-specific Q&A bot.

These are the steps of building the bot.

I will walk you through the overview of each step.

1st step is: text preprocessing,

then 2nd step: text embedding,

The 3rd step is the main logic of the bot: construct prompting & call openai GPT3 API.

Then wrap all logic in a model class as a backend service, expose api using FastAPI, serve it using Uvicorn.

For frontend: serve the webpage using HTML + Node.JS.

1st step: text preprocessing

Segment report by parapraph

Get the token length for each paragraph

Construct a dataframe to contain all paragraphs (used for create embedding)

2nd step: do Text embedding by Calling openai embedding API

embedding shape:

(row\_number, 4095)

Do two embeddings,

One is docs embedding, doc here is the segmented paragraph, this content is static, this needs to be pre-calculated offline.

Another one is query embedding, query is the question asked by end user, this embedding needs to be calculated on demand.

These two embeddings will be used for calculating vector similarities. The similarity is using dot product. Higher the value means higher the similarity between the queries and docs.

The similarity score is then sorted, only top n most similar docs will be selected, and they will be added to the prompt as given context.

3rd step: Construct prompting & call API of GPT3

Temperature: # Higher values means the model will take more risks. 0.9 for more creative applications, and 0 (argmax sampling) for ones with a well-defined answer.

Max\_token: # The maximum number of tokens to generate in the completion/answer. The token count of prompt + max\_tokens <= model's context length (usually 2048)

COMPLETIONS\_MODEL = "text-davinci-002"

Davinci Most capable GPT-3 model. Can do any task the other models can do, often with higher quality, max\_requests: 2,049 tokens

=====

* Wrap up similarity computation, prompting engineering in qa\_model
* Expose API: <http://localhost:8000/ask>

FastAPI is currently the most popular and high-performance framework to build API with pythons.

Initialize a fastapi instance, then use decorator to pass the self-defined callback: the ask function to the fastAPI post function. Now every time when a post request is fired, it will pass query to the self-defined ask function.

* Serving using uvicorn

Uvicorn is a web server implementation for python. Once it set up, it listens to the http request from the specified port. If a request is sent to that port, and the request is pre-defined, it triggers the associated post function. Otherwise, it response {"detail": "Not Found"}

That brings to the end of my demo.

I personally feel excited to know how powerful the GPT is. Especially amazed by its in-context learning. It gives non-NLP expert such as me some room for imagination. Because it means building specific service using LLM is much easier in terms of the cost, data preparation kind of thing.

In the in-context learning or the few-shot prompting, it’s not about how many examples the model sees, it’s about how variety in the examples the model can see. When the examples are various enough (some researchers say 20 Examples), the model can automatically generalise to unseen prompts.

OpenAI released GPT3.5 API last week, after upgrading the Openai package to 0.27, then can call ChatCompletion model.

Myself haven't tried it out for GPT3.5, but I think just a little bit modification in the code, it can work with new model.

So if you're interested, maybe this code can be a quick starter code.