

Fine-Tuning and Deployment of Large Language Models

CHARACTERISTICS OF LLM FINE-TUNING

- **News**
- **Creating your own ChatGPT: Supervised fine-tuning (SFT)**
- **Pydantic and Instructor**
- **Project Discussions**
- **Tasks until next week**

NEWS

CREATING YOUR OWN CHATGPT: SUPERVISED FINE-TUNING (SFT)

The screenshot shows a GitHub repository page for 'Transformers-Tutorials' by user 'NielsRogge'. The repository has 247 issues, 8 pull requests, and 67.3 KB of code. The selected file is 'Supervised_fine_tuning_(SFT)_of_an_LLM_using_Hugging_Face_tooling.ipynb'. The notebook content is displayed in a preview window, showing the title 'Supervised fine-tuning (SFT) of an LLM' and a list of three steps for creating a ChatGPT at home. The steps are: 1. pre-training a large language model (LLM) to predict the next token on internet-scale data, on clusters of thousands of GPUs. One calls the result a "base model"; 2. supervised fine-tuning (SFT) to turn the base model into a useful assistant; 3. human preference fine-tuning which increases the assistant's friendliness, helpfulness and safety. The notebook also includes a paragraph about supervised fine-tuning taking in a "base model" from step 1, and a paragraph about requiring human annotators to collect useful completions, with links to OpenAI's hiring of human contractors and a collection of open SFT datasets.

Transformers-Tutorials / Mistral / Supervised_fine_tuning_(SFT)_of_an_LLM_using_Hugging_Face_tooling.ipynb

NielsRogge Gemaakt met Colaboratory 96edeb2 · 3 months ago History

1275 lines (1275 loc) · 67.3 KB

Open in Colab

Supervised fine-tuning (SFT) of an LLM

Recall that creating a ChatGPT at home involves 3 steps:

1. pre-training a large language model (LLM) to predict the next token on internet-scale data, on clusters of thousands of GPUs. One calls the result a "base model"
2. supervised fine-tuning (SFT) to turn the base model into a useful assistant
3. human preference fine-tuning which increases the assistant's friendliness, helpfulness and safety.

In this notebook, we're going to illustrate step 2. This involves supervised fine-tuning (SFT for short), also called instruction tuning.

Supervised fine-tuning takes in a "base model" from step 1, i.e. a model that has been pre-trained on predicting the next token on internet text, and turns it into a "chatbot"/"assistant". This is done by fine-tuning the model on human instruction data, using the cross-entropy loss. This means that the model is still trained to predict the next token, although we now want the model to generate useful completions given an instruction like "what are 10 things to do in London?", "How can I make pancakes?" or "Write me a poem about elephants".

To do this, one requires human annotators to collect useful completions, on which we can train the model. OpenAI for instance [hired human contractors for this](#), which were asked to generate useful completions given instructions, like "In London, you can visit the Big Ben and (...)". A nice collection of openly available SFT datasets can be found [here](#).

PYDANTIC

```
1 from datetime import datetime
2
3 from pydantic import BaseModel, PositiveInt
4
5
6 class User(BaseModel):
7     id: int
8     name: str = 'John Doe'
9     signup_ts: datetime | None
10    tastes: dict[str, PositiveInt]
11
12
13 external_data = {
14     'id': 123,
15     'signup_ts': '2019-06-01 12:22',
16     'tastes': {
17         'wine': 9,
18         'cheese': 7,
19         'cabbage': '1',
20     },
21 }
```

```
23 user = User(**external_data)
24
25 print(user.id)
26 #> 123
27 print(user.model_dump())
28 """
29 {
30     'id': 123,
31     'name': 'John Doe',
32     'signup_ts': datetime.datetime(2019, 6, 1, 12, 22),
33     'tastes': {'wine': 9, 'cheese': 7, 'cabbage': 1},
34 }
35 """
36
```

PYDANTIC'S BASEMODEL CLASS

- **Data Validation**

When you create an instance of a BaseModel, Pydantic validates the input data based on the field types declared in the model. This process ensures that the data conforms to specified formats and constraints.

- **Data Parsing**

Automatically converts or parses incoming data to the declared types. For example, if a field is declared as int, but the input is a string that can be parsed into an integer, Pydantic will handle this conversion.

- **Data Serialization**

Models can easily be converted to dictionaries, JSON, and other formats, supporting clean serialization of complex structures.

- **Rich Type Annotations**

Supports advanced type hints, including List, Dict, Optional, and even custom types, making it powerful for use in both simple and complex data structures.

- **Immutability Option**

You can make models immutable so that their attributes can't be changed once they are set.

- **Default Values & Validation**

You can define default values for fields and add additional validation through the use of custom validator methods.

PYDANTIC VALIDATION FAIL

```
1 # continuing the above example...
2
3 from pydantic import ValidationError
4
5
6 class User(BaseModel):
7     id: int
8     name: str = 'John Doe'
9     signup_ts: datetime | None
10     tastes: dict[str, PositiveInt]
11
12
13 external_data = {'id': 'not an int', 'tastes': {}}
14
```

```
15 try:
16     User(**external_data)
17 except ValidationError as e:
18     print(e.errors())
19     """
20     [
21         {
22             'type': 'int_parsing',
23             'loc': ('id',),
24             'msg': 'Input should be a valid integer, unable to parse',
25             'input': 'not an int',
26             'url': 'https://errors.pydantic.dev/2/v/int_parsing',
27         },
28         {
29             'type': 'missing',
30             'loc': ('signup_ts',),
31             'msg': 'Field required',
32             'input': {'id': 'not an int', 'tastes': {}},
33             'url': 'https://errors.pydantic.dev/2/v/missing',
34         },
35     ]
36     """
37
```

Instructor: Structured LLM Outputs

Instructor is a Python library that makes it a breeze to work with structured outputs from large language models (LLMs). Built on top of Pydantic, it provides a simple, transparent, and user-friendly API to manage validation, retries, and streaming responses. Get ready to supercharge your LLM workflows!

 Follow @jxnico discord 113 online downloads 143k/month

Key Features

- **Response Models:** Specify Pydantic models to define the structure of your LLM outputs
- **Retry Management:** Easily configure the number of retry attempts for your requests
- **Validation:** Ensure LLM responses conform to your expectations with Pydantic validation
- **Streaming Support:** Work with Lists and Partial responses effortlessly
- **Flexible Backends:** Seamlessly integrate with various LLM providers beyond OpenAI



```
import instructor
from pydantic import BaseModel
from openai import OpenAI

# Define your desired output structure
class UserInfo(BaseModel):
    name: str
    age: int

# Patch the OpenAI client
client = instructor.from_openai(OpenAI())

# Extract structured data from natural language
user_info = client.chat.completions.create(
    model="gpt-3.5-turbo",
    response_model=UserInfo,
    messages=[{"role": "user", "content": "John Doe is 30 years old."}],
)

print(user_info.name)
#> John Doe
print(user_info.age)
#> 30
```

PROJECT DISCUSSIONS

TASKS UNTIL NEXT WEEK

- **Who is doing the news section?**

TASKS UNTIL NEXT WEEK

- Do a “literature review” for your project.
- Watch the first two chapters of [LLM Engineering: Structured Outputs](#) (“Asking LLMs for Structured Data” and “Prompting LLMs”)
- Note a least one question on the videos above.