Text generation

11.09.2023

—

Gaurav

Your Company - Parentics Solution

Your City- Delhi

Introduction

This documentation outlines the process of using the LLM-2 (Large Language Model 2) to generate smart job descriptions from basic job descriptions . LLM-2 is a powerful language model capable of generating content based on input prompts.

Goals

> Our goal is to generate a smart job description from a normal/basic job description for compare the candidate they are eligible or not for the job .

Prerequisites

Before you begin using LLM-2 for job description generation, ensure you have the following prerequisites in place:

- Access to LLM-2 or a fine-tuned version of the model for job description generation.
- Relevant job description data, including job titles, responsibilities, and qualifications.
- Access to a programming environment for model initialization (e.g., Python).
- Knowledge of data preprocessing techniques.
- We need model from hugging face ,model name = “meta-llama/llama-2-7n-chat-h”

Code for train the model

- **This can be helpful when you want to reduce noise in output or temporarily ignore non-critical warnings during development or testing.**

```
# ignoring the warnings

import warnings

warnings.filterwarnings("ignore")
```

- This code imports necessary libraries and modules for natural language processing tasks, including training Transformers models and using pre-trained models for various tasks like text generation.

```
import os

import torch

from datasets import load_dataset

from transformers import (

AutoModelForCausalLM,

AutoTokenizer,

BitsAndBytesConfig,

HfArgumentParser,

TrainingArguments,

pipeline,

logging,

)
```

```
from peft import LoraConfig, PeftModel

from trl import SFTTrainer
```

- In this code we convert the excel dataset into pandas dataframe.

```
# covert the dataset to pandas dataframe for instruction finetuning dataset preparatio

import pandas as pd

ds =pd.read_excel('/content/drive/MyDrive/llama2 dataset/Jds_discriptions.xlsx')

ds
```

- Resets the index of 'test_df' for a clean data representation.

```
test_df=pd.read_excel('/content/drive/MyDrive/llama2 dataset/Jds_discriptions.xlsx')

test_df = test_df.tail(5)

test_df.reset_index(drop=True,inplace=True)
```

- This function is useful for preparing a dataset for fine-tuning a language model for a specific task, such as generating smart job descriptions. It ensures that the input data is properly formatted with the required instruction and structure for the fine-tuning process.

```
# instruction finetuning data preparation function

def prepare_dataset(df,split='train'):

text_col = []

instruction = """Write a smart jd""" # change instuction according to the task

if split == 'train':

for _ , row in df.iterrows():

input_q = row["entities"]

output = row["smartjd"]

text = ("### Instruction: \n" + instruction +

"\n### Input: \n" + input_q +

"\n### Response :\n" + output) # keeping output column in training dataset

text_col.append(text)

df.loc[:,'text'] = text_col
```

```
else:

for _ , row in df.iterrows():

input_q = row["entities"]

text = ("### Instruction: \n" + instruction +

"\n### Input: \n" + input_q +

"\n### Response :\n" ) # not keeping output column in test dataset

text_col.append(text)

df.loc[:,'text'] = text_col

return df
```

These lines of code prepare two DataFrames, `train_df` and `test_df`, for training and testing a language model. They format the data with instructions and input, with `train_df` including output (for training) and `test_df` excluding output (for testing), making them ready for a task like generating smart job descriptions.

```
<train_df = prepare_dataset(train_df,'train')

test_df = prepare_dataset(test_df,'test')

# looking at one of the train text column format

print(train_df['text'][0])
```

> The code you've provided converts a Pandas DataFrame, `train_df`, into a dataset format that's compatible with Hugging Face's Transformers library.

```
# coverting the dataframe to huggingface dataset for easy finetuning

from datasets import Dataset

dataset = Dataset.from_pandas(train_df)>
```

- This code sets up all the necessary hyperparameters and configurations for fine-tuning a language model, including specifying the model architecture, training settings, and options for efficient memory usage. The specific values for these parameters can significantly impact the training process and the performance of the fine-tuned model for generating smart job descriptions

```
.# The model that you want to train from the Hugging Face hub

model_name = "meta-llama/Llama-2-7b-chat-hf"

# Fine-tuned model name
```

```
new_model = "/content/drive/MyDrive/smartjd_python/model_2_1"

################################################################################

# QLoRA parameters

################################################################################

# LoRA attention dimension

lora_r = 64

# Alpha parameter for LoRA scaling

lora_alpha = 16

# Dropout probability for LoRA layers

lora_dropout = 0.1

################################################################################

# bitsandbytes parameters

################################################################################

# Activate 4-bit precision base model loading

use_4bit = True

# Compute dtype for 4-bit base models

bnb_4bit_compute_dtype = "float16"

# Quantization type (fp4 or nf4)

bnb_4bit_quant_type = "nf4"

# Activate nested quantization for 4-bit base models (double quantization)

use_nested_quant = False

################################################################################

# TrainingArguments parameters

################################################################################

# Output directory where the model predictions and checkpoints will be stored

output_dir = "./results"

# Number of training epochs
```

```
num_train_epochs = 5

# Enable fp16/bf16 training (set bf16 to True with an A100)

fp16 = False

bf16 = False

# Batch size per GPU for training

per_device_train_batch_size = 4

# Batch size per GPU for evaluation

per_device_eval_batch_size = 4

# Number of update steps to accumulate the gradients for

gradient_accumulation_steps = 1

# Enable gradient checkpointing

gradient_checkpointing = True

# Maximum gradient normal (gradient clipping)

max_grad_norm = 0.3

# Initial learning rate (AdamW optimizer)

learning_rate = 2e-4

# Weight decay to apply to all layers except bias/LayerNorm weights

weight_decay = 0.001

# Optimizer to use

optim = "paged_adamw_32bit"

# Learning rate schedule (constant a bit better than cosine)

lr_scheduler_type = "constant"

# Number of training steps (overrides num_train_epochs)

max_steps = -1

# Ratio of steps for a linear warmup (from 0 to learning rate)

warmup_ratio = 0.03

# Group sequences into batches with same length

# Saves memory and speeds up training considerably
```

```
group_by_length = True

# Save checkpoint every X updates steps

save_steps = 25

# Log every X updates steps

logging_steps = 25

################################################################################

# SFT parameters

################################################################################

# Maximum sequence length to use

max_seq_length = None

# Pack multiple short examples in the same input sequence to increase efficiency

packing = False

# Load the entire model on the GPU 0

device_map = {"": 0}
This code snippet authenticates the user or application using the provided access toke
access_token = "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"

from huggingface_hub import login

login(token = access_token)
```

- BitsAndBytes Configuration: The code initializes a configuration for BitsAndBytes quantization (`bnb_config`). It specifies whether to use 4-bit precision, the quantization type (fp4 or nf4), and the compute data type.
- GPU Compatibility Check: It checks whether the GPU supports bfloat16 (bf16) if 4-bit precision is used. If the GPU supports bf16, it prints a message suggesting that training can be accelerated with bf16.
- Load Base Model: It loads a base language model (e.g., Llama-2) with the specified quantization configuration and device mapping. The `model_name` variable determines which pre-trained model to load.
- Load Tokenizer: The code loads the tokenizer associated with the base model. It also adds special tokens and configures padding settings.
- LoRA Configuration: It sets up the configuration for LoRA (Long Range Arena) attention, specifying parameters like alpha, dropout, and attention dimension. The `task_type` is set to "CAUSAL_LM."
- Training Parameters: Training parameters are configured, including the output directory, number of training epochs, batch size, optimizer, gradient accumulation steps, learning rate, and more. It also sets up parameters related to mixed-precision training (fp16/bf16) and gradient clipping.
- Supervised Fine-Tuning Trainer: It initializes a trainer (`trainer`) for supervised fine-tuning. This trainer is designed for tasks like causal language modeling and is configured with the model, training dataset, LoRA configuration, tokenization settings, and training arguments.
- Train Model: The code initiates the training process using `trainer.train()`, which fine-tunes the model on the

provided dataset.

- Save Trained Model: After training, it saves the trained model to the specified directory (`new_model`) for later use or evaluation.

This code prepares and fine-tunes a language model for a specific task, making use of custom quantization, BitsAndBytes configuration, and LoRA attention. The trained model can then be used for tasks like generating text or smart job descriptions.

```
# Load tokenizer and model with QLoRA configuration

compute_dtype = getattr(torch, bnb_4bit_compute_dtype)

bnb_config = BitsAndBytesConfig(

load_in_4bit=use_4bit,

bnb_4bit_quant_type=bnb_4bit_quant_type,

bnb_4bit_compute_dtype=compute_dtype,

bnb_4bit_use_double_quant=use_nested_quant,

)

# Check GPU compatibility with bfloat16

if compute_dtype == torch.float16 and use_4bit:

major, _ = torch.cuda.get_device_capability()

if major >= 8:

print("=" * 80)

print("Your GPU supports bfloat16: accelerate training with bf16=True")

print("=" * 80)

# Load base model

model = AutoModelForCausalLM.from_pretrained(

model_name,

quantization_config=bnb_config,

device_map=device_map

)

model.config.use_cache = False

model.config.pretraining_tp = 1
```

```
# Load LLaMA tokenizer

tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)

tokenizer.add_special_tokens({'pad_token': '[PAD]'})

tokenizer.pad_token = tokenizer.eos_token

tokenizer.padding_side = "right"

# Load LoRA configuration

peft_config = LoraConfig(

lora_alpha=lora_alpha,

lora_dropout=lora_dropout,

r=lora_r,

bias="none",

task_type="CAUSAL_LM",

)

# Set training parameters

training_arguments = TrainingArguments(

output_dir=output_dir,

num_train_epochs=num_train_epochs,

per_device_train_batch_size=per_device_train_batch_size,

gradient_accumulation_steps=gradient_accumulation_steps,

optim=optim,

save_steps=save_steps,

logging_steps=logging_steps,

learning_rate=learning_rate,

weight_decay=weight_decay,

fp16=fp16,

bf16=bf16,

max_grad_norm=max_grad_norm,

max_steps=max_steps,
```

```
warmup_ratio=warmup_ratio,
group_by_length=group_by_length,
lr_scheduler_type=lr_scheduler_type,
report_to="tensorboard"
)
# Set supervised fine-tuning parameters
trainer = SFTTrainer(
model=model,
train_dataset=dataset,
peft_config=peft_config,
dataset_text_field="text",
max_seq_length=max_seq_length,
tokenizer=tokenizer,
args=training_arguments,
packing=packing,
)
# Train model
trainer.train()
# Save trained model
trainer.model.save_pretrained(new_model)
```

- This code prepares a structured text block that can be used as a template for a task, with an instruction and input information provided, and a space left for the response to be filled in based on the given struction and input.

```
instruction = "Write a smart jd"
input_q = """Join the Smart Grids and Infrastructure team at Siemens as an experienced
text = ("### Instruction: \n" + instruction +
"\n### Input: \n" + input_q +
"\n### Response :\n" ) # not keeping output column in test dataset
```

- This code sets up a text generation pipeline using a pre-trained language model and generates text in response to a given prompt. The generated text can be used for various natural language processing tasks, such as completing sentences or generating content like smart job descriptions, depending on the specific task and prompt provided.

```
# Ignore warnings

logging.set_verbosity(logging.CRITICAL)

# Run text generation pipeline with our next model

prompt = text

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=80

result = pipe(f"{prompt}\n")

print(result[0]['generated_text'])
```

Code for test the model

```
!pip install -q accelerate==0.21.0 peft==0.4.0 bitsandbytes==0.40.2 transformers==4.31

# ignoring the warnings

import warnings

warnings.filterwarnings("ignore")

import os

import torch

from datasets import load_dataset

from transformers import (

AutoModelForCausalLM,

AutoTokenizer,

BitsAndBytesConfig,

HfArgumentParser,

TrainingArguments,

pipeline,

logging,

)

from peft import LoraConfig, PeftModel
```

```
from trl import SFTTrainer

access_token = "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"

from huggingface_hub import login

login(token = access_token)
```

Run using complete model

- This code prepares a structured text block that can be used as a template for a task. It includes an instruction, input information, and a space left for the response to be filled in based on the given instruction and input. The template appears suitable for tasks where responses or content generation is required, such as writing a smart job description in response to the provided information and instruction.

```
instruction = "Write a smart jd"

input_q = """Join the Smart Grids and Infrastructure team at Siemens as an experienced SDET Lead to drive QA automation strategy and enhance software reliability. Your role involves architecting test automation systems, developing frameworks, and integrating tools within a DevOps environment. With expertise in Java, Selenium, RESTful APIs, and testing methodologies, you'll contribute to bug-free application launches, CI/CD processes, and performance testing. Your leadership skills and problem-solving abilities will be pivotal in shaping the future of software development. This role is based in Chennai and offers the opportunity to work on impactful projects across locations."""

text = ("### Instruction: \n" + instruction +

"\n### Input: \n" + input_q +

"\n### Response :\n" ) # not keeping output column in test dataset
```

- This code sets up the name of the pre-trained language model to be used for fine-tuning, specifies device mapping, and defines the path where the fine-tuned model will be saved after training. It's a crucial step in preparing for the fine-tuning process of a language model for the specific task you have in mind, which appears to involve generating smart job descriptions.

```
# The model that you want to train from the Hugging Face hub

model_name = "meta-llama/Llama-2-7b-chat-hf"

device_map = {"": 0}

# Fine-tuned model name

new_model = "/content/drive/MyDrive/smartjd_python/model_2_1" (#give your model name)
```

- This code reloads the base language model, configures it for low CPU memory usage and precision (FP16), merges it with a fine-tuned model, and reloads the associated tokenizer. This is typically done to prepare a customized model and tokenizer for specific tasks, such as generating smart job descriptions.

```
# Reload model in FP16 and merge it with LoRA weights
```

```
new_model = "/content/drive/MyDrive/smartjd_python/model_2_1"

base_model = AutoModelForCausalLM.from_pretrained(

model_name,

low_cpu_mem_usage=True,

return_dict=True,

torch_dtype=torch.float16,

device_map=device_map,

)

model = PeftModel.from_pretrained(base_model, new_model)

model = model.merge_and_unload()

# Reload tokenizer to save it

tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)

tokenizer.add_special_tokens({'pad_token': '[PAD]'})

tokenizer.pad_token = tokenizer.eos_token

tokenizer.padding_side = "right"
```

- This code prepares a text generation pipeline using a fine-tuned model and tokenizer. It generates text in response to the provided prompt and attempts to print the generated text. Please note that there's a typo in the print statement that should be corrected for the code to work correctly.

```
# Ignore warnings

logging.set_verbosity(logging.CRITICAL)

# Run text generation pipeline with our next model

prompt = text

pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=800)

result = pipe(f"{prompt}\n")

print(result[0]['generated_text'])
```

Saving model to hugging face

- ```
  This code snippet is authenticating a user or application using the provided acces
  write_token = "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"
  ```
  ```

```
from huggingface_hub import login

login(token = write_token)
```

- This code is a combination of a command-line login and Python code to push a fine-tuned model and its associated tokenizer to the Hugging Face model hub, making them available for others to use and download. The `use_temp_dir=False` option indicates that a temporary directory is not used during the push operation.

```
# huggingface-cli login

model.push_to_hub(new_model, use_temp_dir=False)

tokenizer.push_to_hub(new_model, use_temp_dir=False)
```

Testing for our smart JD model

- This code loads a fine-tuned language model and tokenizer, prepares an instruction and input, tokenizes the text, generates new text based on the input, and prints the generated text. The generated text is typically based on the provided instruction and input and can be used for various natural language processing tasks.

```
from transformers import AutoTokenizer, LlamaForCausalLM

import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

print("device : ",device)

from huggingface_hub.hf_api import HfFolder

HfFolder.save_token('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')

model = LlamaForCausalLM.from_pretrained("datasciencevaluematrix/model_2_1",use_auth_t

model.to(device)

tokenizer = AutoTokenizer.from_pretrained("datasciencevaluematrix/model_2_1",use_auth_t

# Move the model to the GPU (if available)

instruction = "Write a smart jd"

input_q = """Join the Smart Grids and Infrastructure team at Siemens as an experienced

text = ("### Instruction: \n" + instruction +

"\n### Input: \n" + input_q +

"\n### Response :\n" ) # not keeping output column in test dataset

inputs = tokenizer(text, return_tensors="pt")
```

```
outputs = model.generate(
input_ids=inputs['input_ids'], # Use 'input_ids' key instead of 'entities'
attention_mask=inputs['attention_mask'], # Use 'attention_mask' key
max_new_tokens=100,
repetition_penalty=1.2,
)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Conclusion

- Data Preparation:
  - Data in the form of job descriptions and other relevant information is loaded from Excel files into Pandas DataFrames (`train_df` and `test_df`).
- Dataset Preparation for Model:
  - A function named `prepare_dataset` is defined to format the data into a structured format suitable for training and testing language models. It includes an instruction, input text, and, in the training dataset, a response or output.
- Hugging Face Dataset Conversion:
  - The Pandas DataFrames are converted into a Hugging Face Dataset object for easier fine-tuning and model training.
- Model Fine-Tuning:
  - A pre-trained language model, "Llama-2-7b-chat-hf," is selected for fine-tuning.
  - Various model configuration parameters, including LoRA (Low-Rank Adaptation) and quantization settings, are specified.
  - The model is fine-tuned using a specific training setup, such as gradient accumulation and optimization settings.
  - The trained model is saved for later use.
- Hugging Face Hub Authentication:
  - An access token is used for authentication with the Hugging Face Hub to enable model and tokenizer uploads and downloads.
- Model and Tokenizer Upload:
  - The fine-tuned model and tokenizer are uploaded to the Hugging Face model hub, making them accessible for others to use.
- Model Usage for Text Generation:
  - The fine-tuned model and tokenizer are loaded for text generation.
  - An instruction and input text are prepared for the text generation task.
  - The model generates text based on the provided input, adhering to certain generation parameters, and the generated text is printed.

The provided code showcases a workflow for fine-tuning and using pre-trained language models for various natural language processing tasks, with a specific focus on generating text based on instructions and input. The Hugging Face Transformers library and Hugging Face Hub are utilized to facilitate model fine-tuning, sharing, and usage in a collaborative and efficient

manner.