Preeti AI · Follow
15 min read · Oct 22

Instruction-tuning Llama-2–7B for News Classification

The purpose of this notebook is to provide a comprehensive, step-by-step tutorial for fine-tuning any LLM (Large Language Model).

This guide will be divided into two parts:

Part 1: Setting up and Preparing for Fine-Tuning
1. Installing and loading the required modules
2. Steps to get approval for Meta's Llama 2 family of models
3. Setting up Hugging Face CLI and user authentication
4. Loading a pre-trained model and its associated tokenizer
5. Loading the training dataset
6. Preprocessing the training dataset for model fine-tuning

Part 2: Fine-Tuning and Open-Sourcing
1. Configuring PEFT (Parameter Efficient Fine-Tuning) method QLoRA for efficient fine-tuning
2. Fine-tuning the pre-trained model
3. Saving the fine-tuned model and its associated tokenizer
4. Pushing the fine-tuned model to the Hugging Face Hub for public usage

Let's get started!

Note that running this on a CPU is practically impossible. If running on Google Colab, go to Runtime > Change runtime type. Change Hardware accelarator to GPU. Change GPU type to T4. Change Runtime shape to High-RAM.

Installing Required Libraries

First, we will install some required libraries.

`transformers`: for loading a large language model and fine-tuning it.

`bitsandbytes`: for loading the model in 4-bit precision.

`accelerate`: for training models and performing inference at scale.

`peft`: for fine-tuning a small number of parameters.

`trl`: for training transformer language models using Reinforcement Learning.
“””

```
!pip install -q accelerate==0.21.0 — progress-bar off
!pip install -q peft==0.4.0 — progress-bar off
!pip install -q bitsandbytes==0.40.2 — progress-bar off
!pip install -q transformers==4.31.0 — progress-bar off
!pip install -q trl==0.4.7 — progress-bar off
```

“””### Loading Required Libraries

Next, we will load the required libraries for fine-tuning a Large Language Model (LLM) like Llama 2. We will look at each imported class in greater detail in subsequent sections.
“””

```
import os
from random import randrange
from functools import partial
import torch
from datasets import load_dataset
from transformers import (AutoModelForCausalLM,
AutoTokenizer,
BitsAndBytesConfig,
HfArgumentParser,
Trainer,
TrainingArguments,
DataCollatorForLanguageModeling,
EarlyStoppingCallback,
```

```
pipeline,
logging,
set_seed)

import bitsandbytes as bnb
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training,
PeftModel, AutoPeftModelForCausalLM
from trl import SFTTrainer
from google.colab import drive
drive.mount('/content/drive')

"""### Hugging Face Hub Login

Meta's family of Llama 2 models is gated. You will require approval to access it using the
Hugging Face Hub.

Below are the steps to request permission for the Llama-2–7B model:
1. Get approval from Hugging Face (https://huggingface.co/meta-llama/Llama-2-7b-hf).
2. Get approval from Meta (https://ai.meta.com/resources/models-and-libraries/llama-
downloads/).
3. Create a WRITE access token on Hugging Face (https://huggingface.co/settings/tokens).
4. Execute `!huggingface-cli login` in Google Colab Notebook, enter the token, and enter
"Y."

Note: Make sure your email address on your Hugging Face account is the same as the one
you enter on Meta's website for approval.

If you don't want to perform the above steps, use a cloned version of Llama-2–7B, such as
https://huggingface.co/daryl149/llama-2-7b-chat-hf. Additionally, you'll have to set
`use_auth_token` to `False` while loading the model and its tokenizer.
"""

!huggingface-cli login

"""### Creating Bitsandbytes Configuration

Before loading the model, we will define a function `create_bnb_config` to define the
`bitsandbytes` configuration. The `bitsandbytes` library allows model quantization.
Quantization is a technique used to compress deep learning models by reducing the
number of bits used to represent their weights and activations. This compression allows for
```

faster inference and reduced memory consumption, making it possible to deploy these models on edge devices with limited resources.

By using 4-bit transformer language models, we can achieve impressive results while significantly reducing memory and computational requirements.

Hugging Face Transformers (`transformers`) is closely integrated with `bitsandbytes`. The `BitsAndBytesConfig` class from the `transformers` library allows configuring the model quantization method.

Parameters:

`load_in_4bit`: Load the model in 4-bit precision, i.e., divide memory usage by 4.

`bnb_4bit_use_double_quant`: Use nested quantization techniques for more memory-efficient inference at no additional cost.

`bnb_4bit_quant_type`: Set quantization data type. The options are either FP4 (4-bit precision), which is the default quantization data type, or NF4 (Normal Float 4), a new 4-bit data type adapted for weights that have been initialized using a normal distribution.

`bnb_4bit_compute_dtype`: Set the computational data type for 4-bit models. Default value: torch.float32
“””

def create_bnb_config(load_in_4bit, bnb_4bit_use_double_quant, bnb_4bit_quant_type, bnb_4bit_compute_dtype):
“””
Configures model quantization method using bitsandbytes to speed up training and inference

:param load_in_4bit: Load model in 4-bit precision mode
:param bnb_4bit_use_double_quant: Nested quantization for 4-bit model
:param bnb_4bit_quant_type: Quantization data type for 4-bit model
:param bnb_4bit_compute_dtype: Computation data type for 4-bit model
“””

bnb_config = BitsAndBytesConfig(
load_in_4bit = load_in_4bit,
bnb_4bit_use_double_quant = bnb_4bit_use_double_quant,

```
bnb_4bit_quant_type = bnb_4bit_quant_type,
bnb_4bit_compute_dtype = bnb_4bit_compute_dtype,
)

return bnb_config

"""### Loading Hugging Face Model and Tokenizer

We will now define a function `load_model` that accepts the model name (`model_name`) from Hugging Face Hub and the `bitsandbytes` configuration for model quantization.

In this function, we will perform the following steps:
1. Get the number of GPUs available.
2. Set the maximum GPU memory.
3. Use the from_pretrained` method from the `AutoModelForCausalLM` class to load a pre-trained Hugging Face model in 4-bit precision using the model name and the quantization configuration.
4. Set which device to send the model to using `device_map`. Passing `device_map = 0` means putting the whole model on GPU 0. Other inputs could be `cpu`, `cuda:1`, etc. Setting `device_map = auto` will let `accelerate` compute the most optimized `device_map` automatically.
5. Set `max_memory`, a dictionary device identifier, to maximum memory, which will default to the maximum memory available for each GPU and the available CPU RAM if unset.
6. Load the model tokenizer from the model name on Hugging Face.
7. Set a padding token to ensure shorter sequences will have the same length as the longest sequence in a batch. In this case, we will set the EOS (End of Sentence) token as the padding token.

**Important Note: A tokenizer for a model will preprocess and tokenize (convert letters/words/sub-words to tokens or numbers) the input in a way that the model expects. Model tokenizers are also responsible for correctly applying special tokens and certain special embeddings or positional encoders specific to a model in the input.**
"""

def load_model(model_name, bnb_config):
"""
Loads model and model tokenizer
```

```
:param model_name: Hugging Face model name
:param bnb_config: Bitsandbytes configuration
"""

# Get number of GPU device and set maximum memory
n_gpus = torch.cuda.device_count()
max_memory = f'{40960}MB'

# Load model
model = AutoModelForCausalLM.from_pretrained(
model_name,
quantization_config = bnb_config,
device_map = "auto", # dispatch the model efficiently on the available resources
max_memory = {i: max_memory for i in range(n_gpus)},
)

# Load model tokenizer with the user authentication token
tokenizer = AutoTokenizer.from_pretrained(model_name, use_auth_token = True)

# Set padding token as EOS token
tokenizer.pad_token = tokenizer.eos_token

return model, tokenizer

"""### Initializing Transformers and Bitsandbytes Parameters

We will now initialize input parameters for the `transformers` and `bitsandbytes`
modules.
"""

################################################################################
# transformers parameters
################################################################################

# The pre-trained model from the Hugging Face Hub to load and fine-tune
model_name = "meta-llama/Llama-2–7b-hf"
```

```
################################################################################
# bitsandbytes parameters
################################################################################

# Activate 4-bit precision base model loading
load_in_4bit = True

# Activate nested quantization for 4-bit base models (double quantization)
bnb_4bit_use_double_quant = True

# Quantization type (fp4 or nf4)
bnb_4bit_quant_type = "nf4"

# Compute data type for 4-bit base models
bnb_4bit_compute_dtype = torch.bfloat16

"""Finally, we will call the above functions to get `model` and `tokenizer` objects."""

# Load model from Hugging Face Hub with model name and bitsandbytes configuration

bnb_config = create_bnb_config(load_in_4bit, bnb_4bit_use_double_quant,
bnb_4bit_quant_type, bnb_4bit_compute_dtype)

model, tokenizer = load_model(model_name, bnb_config)

"""### Loading Dataset

Now that we have loaded the Llama-2–7B model and its tokenizer, we will move on to
loading our news classification instruction dataset from the previous blog as a Hugging
Face `Datasets`.

Firstly, we will initialize the path of the dataset. In this case, we have a CSV file that
contains 99 records, or prompts. This dataset contains an `instruction` column containing
the instruction to categorize a news article into 18 categories, an `input` column
containing the news article, and an `output` column containing the actual news category
for training.
```

```
We will use the `load_dataset` function and pass the file location. We will define a generic dataset builder name `csv` because our dataset is a CSV file. You can similarly load a JSON file by passing `json` and the dataset location to a JSON file. All the records are assigned to the `train` split by default, which we would retrieve using the `split` parameter.
“””

# The instruction dataset to use
dataset_name = “/content/drive/MyDrive/news_classification.csv”

# Load dataset
dataset = load_dataset(“csv”, data_files = dataset_name, split = “train”)

print(f’Number of prompts: {len(dataset)}’)
print(f’Column names are: {dataset.column_names}’)

“””The `load_dataset` function will convert the CSV file into a dictionary of prompts. We can look at a random prompt in the dataset using a random index.”””

dataset[randrange(len(dataset))]

“””### Creating Prompt Template

After loading the instruction dataset, we will define the `create_prompt_formats` function to create a prompt template against each prompt in our dataset and save it in a new dictionary key `text` for further data preprocessing and fine-tuning.
“””

def create_prompt_formats(sample):
“””
Creates a formatted prompt template for a prompt in the instruction dataset

:param sample: Prompt or sample from the instruction dataset
“””

# Initialize static strings for the prompt template
INTRO_BLURB = “Below is an instruction that describes a task. Write a response that appropriately completes the request.”
INSTRUCTION_KEY = “### Instruction:”
INPUT_KEY = “Input:”
```

```
RESPONSE_KEY = "### Response:"
END_KEY = "### End"

# Combine a prompt with the static strings
blurb = f"{INTRO_BLURB}"
instruction = f"{INSTRUCTION_KEY}\n{sample['instruction']}"
input_context = f"{INPUT_KEY}\n{sample['input']}" if sample["input"] else None
response = f"{RESPONSE_KEY}\n{sample['output']}"
end = f"{END_KEY}"

# Create a list of prompt template elements
parts = [part for part in [blurb, instruction, input_context, response, end] if part]

# Join prompt template elements into a single string to create the prompt template
formatted_prompt = "\n\n".join(parts)

# Store the formatted prompt template in a new key "text"
sample["text"] = formatted_prompt

return sample

create_prompt_formats(dataset[randrange(len(dataset))])

"""### Getting Maximum Sequence Length of the Pre-trained Model

In the next cell, we will define the `get_max_length` function to find out the maximum sequence length of the Llama-2–7B model. This function will pull the model configuration and attempt to find the maximum sequence length from one of the several configuration keys that may contain it. If the maximum sequence length is not found, it will default to 1024. We will use the maximum sequence length during dataset preprocessing to remove records that exceed that context length because the pre-trained model won't accept them.
"""

def get_max_length(model):
"""
Extracts maximum token length from the model configuration

:param model: Hugging Face model
"""
```

```
# Pull model configuration
conf = model.config
# Initialize a "max_length" variable to store maximum sequence length as null
max_length = None
# Find maximum sequence length in the model configuration and save it in "max_length" if found
for length_setting in ["n_positions", "max_position_embeddings", "seq_length"]:
max_length = getattr(model.config, length_setting, None)
if max_length:
print(f"Found max lenth: {max_length}")
break
# Set "max_length" to 1024 (default value) if maximum sequence length is not found in the model configuration
if not max_length:
max_length = 1024
print(f"Using default max length: {max_length}")
return max_length

"""### Tokenizing Dataset Batch

The user-defined `preprocess_batch` function will tokenize a batch of the input dataset (`batch`) using the `tokenizer` object. We will set the maximum sequence length using the `max_length` parameter, which will control the maximum length used by the padding or truncation parameter. `truncation = True` will truncate the input to the maximum length provided by the `max_length` parameter. Similarly, `padding = max_length` will pad the input to the maximum length provided. This function will be called in the `preprocess_dataset` function defined next.
"""

def preprocess_batch(batch, tokenizer, max_length):
"""
Tokenizes dataset batch

:param batch: Dataset batch
:param tokenizer: Model tokenizer
:param max_length: Maximum number of tokens to emit from the tokenizer
"""
```

```
return tokenizer(
batch["text"],
max_length = max_length,
truncation = True,
)

"""### Preprocessing Dataset

To preprocess the complete dataset for fine-tuning, we will define the `preprocess_dataset` function, which will perform the following operations:

1. Create the formatted prompts against each prompt in the instruction dataset using the `create_prompt_formats` function.
2. Tokenize the dataset in batches using the `preprocess_batch` function and removing the original dictionary keys (instruction, input, output, and text).
3. Filter out prompts with input token sizes exceeding the maximum length.
4. Shuffle the dataset using a random seed.
"""

def preprocess_dataset(tokenizer: AutoTokenizer, max_length: int, seed, dataset: str):
"""
Tokenizes dataset for fine-tuning

:param tokenizer (AutoTokenizer): Model tokenizer
:param max_length (int): Maximum number of tokens to emit from the tokenizer
:param seed: Random seed for reproducibility
:param dataset (str): Instruction dataset
"""

# Add prompt to each sample
print("Preprocessing dataset...")
dataset = dataset.map(create_prompt_formats)

# Apply preprocessing to each batch of the dataset & and remove "instruction", "input", "output", and "text" fields
_preprocessing_function = partial(preprocess_batch, max_length = max_length, tokenizer = tokenizer)
dataset = dataset.map(
_preprocessing_function,
```

```
batched = True,
remove_columns = ["instruction", "input", "output", "text"],
)

# Filter out samples that have "input_ids" exceeding "max_length"
dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_length)

# Shuffle dataset
dataset = dataset.shuffle(seed = seed)

return dataset

# Random seed
seed = 33

max_length = get_max_length(model)
preprocessed_dataset = preprocess_dataset(tokenizer, max_length, seed, dataset)

"""We can now look at the preprocessed dataset, which contains tokens or IDs."""

print(preprocessed_dataset)

print(preprocessed_dataset[0])

"""With everything set up, we can move forward to fine-tuning or instruction-tuning
Llama-2–7B on our news classification instruction dataset.

### Creating PEFT Configuration

Fine-tuning pretrained LLMs on downstream datasets results in huge performance gains
when compared to using the pretrained LLMs out-of-the-box. However, as models get
larger and larger, full fine-tuning becomes infeasible to train on consumer hardware. In
addition, storing and deploying fine-tuned models independently for each downstream
task becomes very expensive, because fine-tuned models are the same size as the original
pretrained model. Parameter-Efficient Fine-tuning (PEFT) approaches are meant to
address both problems!

PEFT approaches only fine-tune a small number of (extra) model parameters while
freezing most parameters of the pretrained LLMs, thereby greatly decreasing the
computational and storage costs. It also helps in portability, wherein users can tune models
```

using PEFT methods to get tiny checkpoints worth a few MB compared to the large checkpoints of full fine-tuning.

In short, PEFT approaches enable you to get performance comparable to full fine-tuning while only having a small number of trainable parameters.

Hugging Face provides the PEFT library, which provides the latest Parameter-Efficient Fine-tuning techniques seamlessly integrated with Hugging Face Transformers and Hugging Face Accelerate.

There are several PEFT methods. In the next cell, we will use QLoRA, one of the latest methods that reduces the memory usage of LLM finetuning without performance tradeoffs, using the `LoraConfig` class from the `peft` library.

QLoRA uses 4-bit quantization to compress a pretrained language model. The LM parameters are then frozen, and a relatively small number of trainable parameters are added to the model in the form of Low-Rank Adapters. During finetuning, QLoRA backpropagates gradients through the frozen 4-bit quantized pretrained language model into the Low-Rank Adapters. The LoRA layers are the only parameters being updated during training.
“””

def create_peft_config(r, lora_alpha, target_modules, lora_dropout, bias, task_type):
“””
Creates Parameter-Efficient Fine-Tuning configuration for the model

:param r: LoRA attention dimension
:param lora_alpha: Alpha parameter for LoRA scaling
:param modules: Names of the modules to apply LoRA to
:param lora_dropout: Dropout Probability for LoRA layers
:param bias: Specifies if the bias parameters should be trained
“””
config = LoraConfig(
r = r,
lora_alpha = lora_alpha,
target_modules = target_modules,
lora_dropout = lora_dropout,
bias = bias,

```
task_type = task_type,
)

return config

"""### Finding Modules for LoRA Application

In the next cell, we will define the `find_all_linear_names` function to find the module to apply LoRA to. This function will get the module names from `model.named_modules()` and store it in a set to keep distinct module names.
"""

def find_all_linear_names(model):
"""
Find modules to apply LoRA to.

:param model: PEFT model
"""

cls = bnb.nn.Linear4bit
lora_module_names = set()
for name, module in model.named_modules():
if isinstance(module, cls):
names = name.split('.')
lora_module_names.add(names[0] if len(names) == 1 else names[-1])

if 'lm_head' in lora_module_names:
lora_module_names.remove('lm_head')
print(f"LoRA module names: {list(lora_module_names)}")
return list(lora_module_names)

"""### Calculating Trainable Parameters

We can use the `print_trainable_parameters` function to find out the number and percentage of trainable model parameters. This function will calculate the number of total parameters in `model.named_parameters()` and then those that would get updated.
"""

def print_trainable_parameters(model, use_4bit = False):
"""
```

```
Prints the number of trainable parameters in the model.

:param model: PEFT model
"""

trainable_params = 0
all_param = 0

for _, param in model.named_parameters():
num_params = param.numel()
if num_params == 0 and hasattr(param, "ds_numel"):
num_params = param.ds_numel
all_param += num_params
if param.requires_grad:
trainable_params += num_params

if use_4bit:
trainable_params /= 2

print(
f"All Parameters: {all_param:,d} || Trainable Parameters: {trainable_params:,d} ||
Trainable Parameters %: {100 * trainable_params / all_param}"
)

"""### Fine-tuning the Pre-trained Model

We will create `fine_tune`, our final function, to wrap everything we have done so far and
initiate the fine-tuning process. This function will perform the following model
preprocessing operations to prepare it for training:

1. Enable gradient checkpointing to reduce memory usage during fine-tuning.
2. Use the `prepare_model_for_kbit_training` function from PEFT to prepare the model for
fine-tuning.
3. Call find_all_linear_names` to get the module names to apply LoRA to.
4. Create LoRA configuration by calling the `create_peft_config` function.
5. Wrap the base Hugging Face model for fine-tuning to PEFT using the `get_peft_model`
function.
6. Print the trainable parameters.
```

For training, we will instantiate a `Trainer()` object within the `fine_tune` function. This class requires the model, preprocessed dataset, and training arguments, listed below.

`per_device_train_batch_size`: The batch size per GPU/TPU/CPU for training.

`gradient_accumulation_steps`: Number of update steps to accumulate the gradients for, before performing a backward/update pass.

`warmup_steps`: Number of steps used for a linear warmup from 0 to `learning_rate`.

`max_steps`: If set to a positive number, the total number of training steps to perform.

`learning_rate`: The initial learning rate for Adam.

`fp16`: Whether to use 16-bit (mixed) precision training (through NVIDIA apex) instead of 32-bit training.

`logging_steps`: Number of update steps between two logs.

`output_dir`: The output directory where the model predictions and checkpoints will be written.

`optim`: The optimizer to use for training.

Next, we will use the `train` method on the trainer` object to start the training and log and save the model metrics on the training dataset. Finally, we will save the model checkpoint (model weights, configuration file, and tokenizer) in the output directory and delete the model to free up memory. You can load the model for inference later using its saved checkpoint.

```
def fine_tune(model,
tokenizer,
dataset,
lora_r,
lora_alpha,
lora_dropout,
bias,
task_type,
per_device_train_batch_size,
gradient_accumulation_steps,
```

```
warmup_steps,
max_steps,
learning_rate,
fp16,
logging_steps,
output_dir,
optim):
“””
Prepares and fine-tune the pre-trained model.

:param model: Pre-trained Hugging Face model
:param tokenizer: Model tokenizer
:param dataset: Preprocessed training dataset
“””

# Enable gradient checkpointing to reduce memory usage during fine-tuning
model.gradient_checkpointing_enable()

# Prepare the model for training
model = prepare_model_for_kbit_training(model)

# Get LoRA module names
target_modules = find_all_linear_names(model)

# Create PEFT configuration for these modules and wrap the model to PEFT
peft_config = create_peft_config(lora_r, lora_alpha, target_modules, lora_dropout, bias,
task_type)
model = get_peft_model(model, peft_config)

# Print information about the percentage of trainable parameters
print_trainable_parameters(model)

# Training parameters
trainer = Trainer(
model = model,
train_dataset = dataset,
args = TrainingArguments(
per_device_train_batch_size = per_device_train_batch_size,
gradient_accumulation_steps = gradient_accumulation_steps,
```

```
warmup_steps = warmup_steps,
max_steps = max_steps,
learning_rate = learning_rate,
fp16 = fp16,
logging_steps = logging_steps,
output_dir = output_dir,
optim = optim,
),
data_collator = DataCollatorForLanguageModeling(tokenizer, mlm = False)
)

model.config.use_cache = False

do_train = True

# Launch training and log metrics
print("Training...")

if do_train:
train_result = trainer.train()
metrics = train_result.metrics
trainer.log_metrics("train", metrics)
trainer.save_metrics("train", metrics)
trainer.save_state()
print(metrics)

# Save model
print("Saving last checkpoint of the model...")
os.makedirs(output_dir, exist_ok = True)
trainer.model.save_pretrained(output_dir)

# Free memory for merging weights
del model
del trainer
torch.cuda.empty_cache()

"""Initializing QLoRA and TrainingArguments parameters below for training."""
```

```
################################################################################
# QLoRA parameters
################################################################################

# LoRA attention dimension
lora_r = 16

# Alpha parameter for LoRA scaling
lora_alpha = 64

# Dropout probability for LoRA layers
lora_dropout = 0.1

# Bias
bias = "none"

# Task type
task_type = "CAUSAL_LM"

################################################################################
# TrainingArguments parameters
################################################################################

# Output directory where the model predictions and checkpoints will be stored
output_dir = "./results"

# Batch size per GPU for training
per_device_train_batch_size = 1

# Number of update steps to accumulate the gradients for
gradient_accumulation_steps = 4

# Initial learning rate (AdamW optimizer)
learning_rate = 2e-4
```

```
# Optimizer to use
optim = "paged_adamw_32bit"

# Number of training steps (overrides num_train_epochs)
max_steps = 20

# Linear warmup steps from 0 to learning_rate
warmup_steps = 2

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

# Log every X updates steps
logging_steps = 1

"""Calling the `fine_tune` function below to fine-tune or instruction-tune the pre-trained model on our preprocessed news classification instruction dataset."""

fine_tune(model,
tokenizer,
preprocessed_dataset,
lora_r,
lora_alpha,
lora_dropout,
bias,
task_type,
per_device_train_batch_size,
gradient_accumulation_steps,
warmup_steps,
max_steps,
learning_rate,
fp16,
logging_steps,
output_dir,
optim)

"""With these steps, we have fine-tuned a popular open-source pre-trained model, Llama-2–7B, on an instruction dataset that we created for news classification!
```

We can see from the log that there are 3,540,389,888 parameters in the model, out of which 39,976,960 are trainable. That's approximately 1% of the total parameters. The model trained for 20 steps and converged at a loss value of 1.4. It is possible that the converged weights are not the best weights. We can fix this by adding `EarlyStoppingCallback` to the `trainer`, which would regularly evaluate the model on a validation dataset and keep only the best weights.

Merging Weights & Pushing to Hugging Face

After saving the fine-tuned weights, we can create our fine-tuned model by merging the fine-tuned weights and saving it to a new directory with its tokenizer. By performing this step, we can have a memory-efficient, fine-tuned model and tokenizer for inference. We will also push the fine-tuned model and its associated tokenizer to Hugging Face Hub for public usage.
“””

```
# Load fine-tuned weights
model = AutoPeftModelForCausalLM.from_pretrained(output_dir, device_map = “auto”, torch_dtype = torch.bfloat16)
# Merge the LoRA layers with the base model
model = model.merge_and_unload()

# Save fine-tuned model at a new location
output_merged_dir = “results/news_classification_llama2_7b/final_merged_checkpoint”
os.makedirs(output_merged_dir, exist_ok = True)
model.save_pretrained(output_merged_dir, safe_serialization = True)

# Save tokenizer for easy inference
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.save_pretrained(output_merged_dir)

model

tokenizer

# Fine-tuned model name on Hugging Face Hub
new_model = “sahayk/news-classification-18-llama-2–7b”

# Push fine-tuned model and tokenizer to Hugging Face Hub
model.push_to_hub(new_model, use_auth_token = True)
```

```
tokenizer.push_to_hub(new_model, use_auth_token = True)

"""Check out the fine-tuned model on Hugging Face: https://huggingface.co/sahayk/news-classification-18-llama-2-7b

### References

**1. https://huggingface.co/**

**2. https://huggingface.co/blog**

**3. https://www.philschmid.de/**

**4. https://blog.ovhcloud.com/**
"""
```

Llm | Llamas | Finetune Llm | Classification | AI

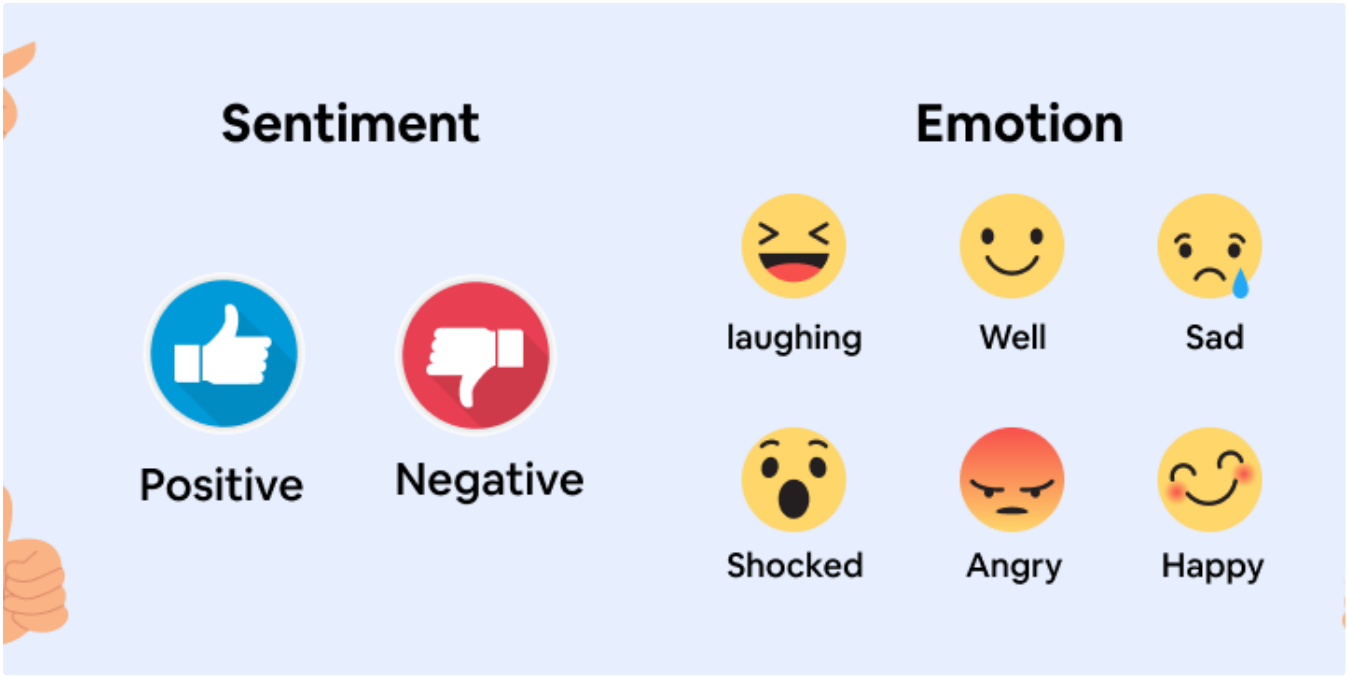
Follow

Written by Preeti AI

14 Followers

Talks about #AI/ML/DL/Computer vision/Data Science/NLP. Connect via:: priyasi.ai7@gmail.com

More from Preeti AI



Preeti AI

Sentiment Analysis-NLP

Data Collection: Gather all the unstructured data files containing text data that you want to analyze for sentiment. This could be from...

2 min read · Jul 29

2

Preeti AI

LLM Project for beginners

This is a basic example on how to use OpenAI's GPT-3 (which is one of their language models) in a Python code snippet. Please note that...

1 min read · Jul 28

Preeti AI

Medical Chatbot using GPT-3

We'll build a simple medical chatbot using the GPT-3 language model from OpenAI. Please note that you'll need access to the OpenAI GPT-3...

2 min read · Jul 28

Preeti AI

Basic NLP Project on Sentiment Analysis

Let's walk through a simple NLP project approach for sentiment analysis using Python and the popular Natural Language Toolkit (NLTK)...

1 min read · Jul 29

2

See all from Preeti AI

Recommended from Medium

Robert John

2 easy ways for fine-tuning LLAMA-2 and other Open source LLMs

Step by step guide on fine-tuning LLAMA-2 and other Open source LLMs

12 min read · Sep 4

68 1

Uday Chandra

Instruction fine-tuning Llama 2 with PEFT's QLoRa method

Llama 2 is a family of open-source large language models released by Meta. They can be used for a variety of tasks, such as writing...

5 min read · Jul 20

107 6

Lists

The New Chatbots: ChatGPT, Bard, and Beyond

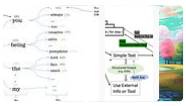
12 stories · 187 saves

Generative AI Recommended Reading

52 stories · 386 saves

What is ChatGPT?

9 stories · 215 saves



Natural Language Processing

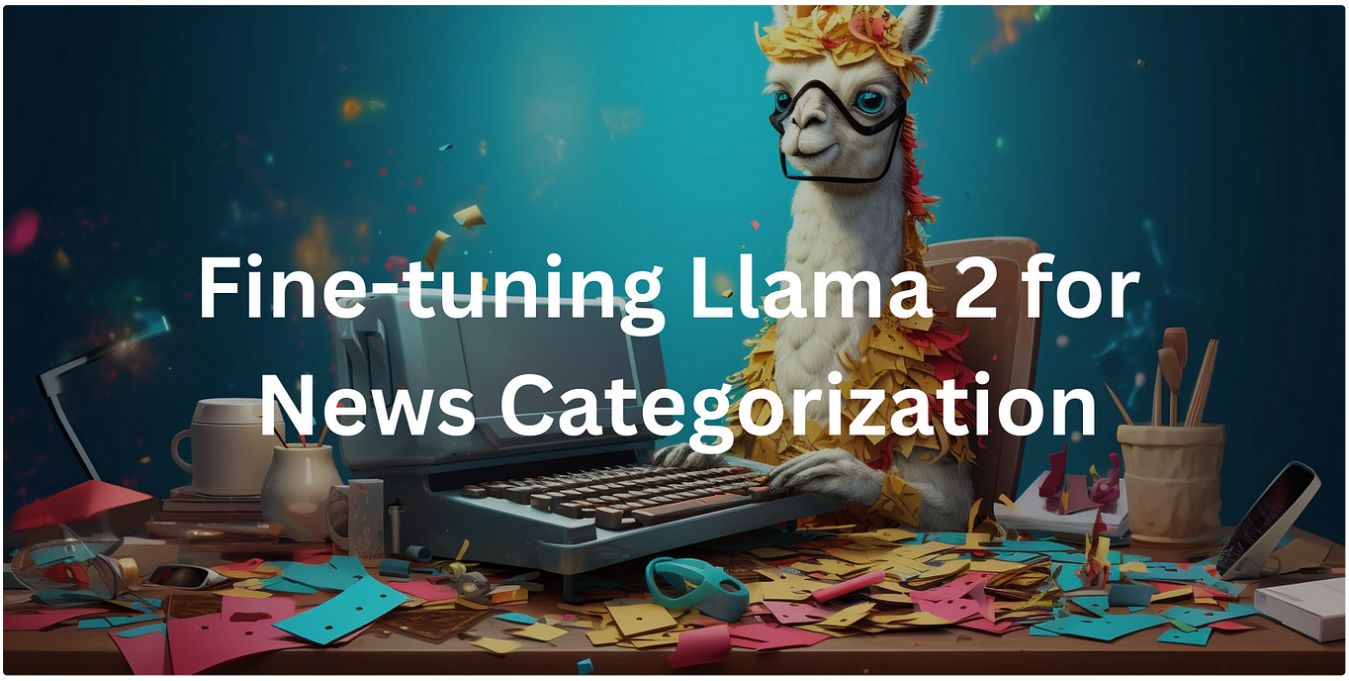
814 stories · 376 saves

datatec.studio in AI MVP

Guide: Fine-Tuning bert-base-uncased Model for Text Comparasion on Colab

· 5 min read · Oct 30

20 1



Kshitiz Sahay

Fine-tuning Llama 2 for news category prediction: A step-by-step comprehensive guide to...

A step-by-step comprehensive guide to fine-tuning any LLM.

14 min read · Aug 7

115 5

Gathnex

Fine-Tuning Llama-2 LLM on Google Colab: A Step-by-Step Guide.

Llama, Llama, Llama: 🦙 A Highly Speakable Model in Recent Times. 🗣️ Llama 2: 🌟 It's like the rockstar of language models, developed by...

12 min read · Sep 18

249 4

Armin Norouzi, Ph.D in Artificial Corner

Mastering Llama 2: A Comprehensive Guide to Fine-Tuning in Google Colab

Dive deep into Llama 2—the cutting-edge NLP model. This guide covers everything from setup and loading to fine-tuning and deployment in...

· 30 min read · Sep 4

345 7