Improving Trained Models with RLHF

Introduction

As previously stated, the RLHF process involves incorporating human feedback into the training process through a reward model that learns the desired patterns to amplify the model's output. For instance, if the goal is to enhance politeness, the reward model will guide the model to generate more polite responses by assigning higher scores to polite outputs. This process can be resource-intensive due to the need to train an additional reward model using a dataset curated by humans, which can be costly. Nevertheless, we will leverage available open-source models and datasets whenever feasible to explore the technique thoroughly while maintaining acceptable costs.

It is recommended to begin the procedure by conducting a supervised fine-tuning phase, which enables the model to adjust to a conversational manner. This procedure can be accomplished by using the `SFTTrainer` class. The next phase involves training a reward model with the desired traits using the `RewardTrainer` class in section 2. Finally, the Reinforcement Learning phase employs the models from the preceding steps to construct the ultimate aligned model, utilizing the `PPOTrainer` class in section 3.

After each subsection, the fine-tuned models, the reports generated from the weights and biases, and the file detailing the requirements for the library can be accessed. Note that different steps might necessitate distinct versions of libraries. We employed the `OPT-1.3B` model as the foundational model and fine-tuned the `DeBERTa` (300M) model as the reward model for our experiments. While these are more compact models and might not incorporate the insights of recent larger models like GPT-4 and LLaMA2, the procedure we are exploring in this tutorial can be readily applied to other existing networks by simply modifying the model key name in the code.

We'll be using a set of 8x100 A100 GPUs in this lesson.

GPU Cloud - Lambda

In this lesson, we'll leverage Lambda, Lambda, the GPU cloud designed by ML engineers for training LLMs & Generative AI, for renting GPUs. We can create an account on it, link a billing account, and rent one instance of the following GPU servers with associated costs. The cost is for the time your instance is up and not only for the time you're training your model, so remember to turn the instance off. For this lesson, we rented an 8x NVIDIA A100 instance comprising 40GB of memory at the price of $8.80/h.

⚠️ Beware of costs when you borrow cloud GPUs. The total cost will depend on the machine type and the up time of the instance. Always remember to monitor your costs in the billing section of Lambda Labs and to spin off your instances when you don't use them.

💡 If you just want to replicate the code in the lesson spending very few money, you can just run the training in your instance and stop it after a few iterations.

Training Monitoring - Weights and Biases

To ensure everything is progressing smoothly, we'll log the training metrics to Weights & Biases, allowing us to see the metrics in real-time in a suitable dashboard.

1. Supervised Fine-Tuning

We thoroughly covered the SFT phase in the previous lessons. If any steps are unclear, refer to the previous lessons.

In this tutorial, the differences are in how we use a distinct dataset called OpenOrca and in applying the QLoRA method, which we will elaborate on in the subsequent sections.

The OpenOrca dataset comprises 1 million interactions with the language model extracted from the original OpenOrca dataset. Each interaction in this collection is comprised of a question paired with a corresponding response. This phase aims to familiarize the model with the conversational structure, thereby teaching it to answer questions rather than relying on its standard auto-completion mechanism.

Begin the process by installing the necessary libraries.

```
pip install -q transformers==4.32.0 bitsandbytes==0.41.1 accelerate==0.22.0 deeplake==3.6.
```

The sample code.

1.1. The Dataset

The initial phase involves streaming the dataset via the Activeloop's performant dataloader to facilitate convenient accessibility. As previously indicated, we employ a subset of the original dataset containing 1 million data points. Nevertheless, the complete dataset (4 million) is accessible at this URL.

```
import deeplake

# Connect to the training and testing datasets
ds = deeplake.load('hub://genai360/OpenOrca-1M-train-set')
ds_valid = deeplake.load('hub://genai360/OpenOrca-1M-valid-set')

print(ds)
```

The sample code.

```
Dataset(path='hub://genai360/OpenOrca-1M-train-set', read_only=True, tensors=['id', 'quest
```

The output.

The dataset has three significant columns. These encompass `question`, also referred to as prompts, which are the queries we have from the LLM: `response`, i.e., the model's output or answers to the questions, and finally, `system_prompt`, i.e., the initial directives guiding the model in establishing its context, such as "you are a helpful assistant."

For simplicity, we exclusively utilize the initial two columns. It could also be beneficial to incorporate the system prompts while formatting the text. This template formats the text in the structure of `Question: xxx\n\nAnswer: yyy` , where the question-and-answer sections are divided by two newline characters. There's room for experimentation with diverse formats, like trying out `System: xxxnnQuestion: yyynnAnswer: zzz` , to effectively integrate the system prompts from the dataset.

```python
def prepare_sample_text(example):
    """Prepare the text from a sample of the dataset."""
    text = f"Question: {example['question'][0]}\n\nAnswer: {example['response'][0]}"
    return text
```

The sample code.

Moving forward, the next step is loading the OPT model tokenizer.

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")
```

The output.

Next, the `ConstantLengthDataset` class will come into play, serving to aggregate samples to maximize utilization within the 2K input size constraint and enhance the efficiency of the training process.

```python
from trl.trainer import ConstantLengthDataset

train_dataset = ConstantLengthDataset(
    tokenizer,
    ds,
    formatting_func=prepare_sample_text,
    infinite=True,
    seq_length=2048
)

eval_dataset = ConstantLengthDataset(
    tokenizer,
    ds_valid,
    formatting_func=prepare_sample_text,
    seq_length=1024
)

iterator = iter(train_dataset)
sample = next(iterator)
print(sample)

train_dataset.start_iteration = 0
```

The sample code.

```
{'input_ids': tensor([ 16, 358, 828,  ..., 137,  79, 362]), 'labels': tensor([ 16, 358, 82
```

The output.

1.2. Initialize the Model and Trainer

Finally, we initialize the model and apply LoRA, which effectively keeps memory requirements low when fine-tuning a large language model.

```
from peft import LoraConfig

lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
```

The sample code.

Now, we instantiate the `TrainingArguments` , which define the hyperparameters governing the training loop.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./OPT-fine_tuned-OpenOrca",
    dataloader_drop_last=True,
    evaluation_strategy="steps",
    save_strategy="steps",
    num_train_epochs=2,
    eval_steps=2000,
    save_steps=2000,
    logging_steps=1,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    learning_rate=1e-4,
    lr_scheduler_type="cosine",
    warmup_steps=100,
    gradient_accumulation_steps=1,
    bf16=True,
    weight_decay=0.05,
    ddp_find_unused_parameters=False,
    run_name="OPT-fine_tuned-OpenOrca",
    report_to="wandb",
)
```

The sample code.

Now, we are using the `BitsAndBytes` library to execute the quantization process and load the model in a 4-bit format. We will employ the NF4 data type designed for weights and implement the nested quantization approach, which effectively reduces memory usage with negligible decline in performance. Finally, we indicate that the training process computations should be carried out using the `bfloat16` format.

The QLoRA method is a recent approach that combines the LoRA technique with quantization to reduce memory usage. When loading the model, it's necessary to provide the `quantization_config` .

```
import torch
from transformers import BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=torch.bfloat16
)
```

The sample code.

The following code segment utilizes the `AutoModelForCasualLM` class to load the pre-trained weights of the OPT model, which holds 1.3 billion parameters. It's important to note that a GPU is required in order to make use of this capability.

```
from transformers import AutoModelForCausalLM
from accelerate import Accelerator

model = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-1.3b",
    quantization_config=quantization_config,
    device_map={"": Accelerator().process_index}
)
```

The sample code.

Prior to initializing the trainer object, we introduce modifications to the model architecture for efficiency. This involves casting specific layers of the model to full precision (32 bits), including LayerNorms and the final language modeling head.

```
from torch import nn

for param in model.parameters():
  param.requires_grad = False
  if param.ndim == 1:
    param.data = param.data.to(torch.float32)
```

```
model.gradient_checkpointing_enable()
model.enable_input_require_grads()

class CastOutputToFloat(nn.Sequential):
  def forward(self, x): return super().forward(x).to(torch.float32)
model.lm_head = CastOutputToFloat(model.lm_head)
```

Lastly, The `SFTTrainer` class will utilize the initialized dataset and model, in combination with the training arguments and LoRA technique, to start the training process.

```
from trl import SFTTrainer

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    peft_config=lora_config,
    packing=True,
)

print("Training...")
trainer.train()
```

The sample code.

The `SFTTrainer` instance will automatically create checkpoints during the training process, as specified by the `save_steps` parameter, and store them in the `./OPT-fine_tuned-OpenOrca` directory.

We're required to merge the LoRA layers with the base model to form a standalone network, a procedure outlined earlier. The subsequent section of code will handle the merging process.

```
from transformers import AutoModelForCausalLM
import torch

model = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-1.3b", return_dict=True, torch_dtype=torch.bfloat16
)

from peft import PeftModel

# Load the Lora model
model = PeftModel.from_pretrained(model, "./OPT-fine_tuned-OpenOrca/<step>")
model.eval()

model = model.merge_and_unload()

model.save_pretrained("./OPT-fine_tuned-OpenOrca/merged")
```

The standalone model will be accessible on the `./OPT-supervised_fine_tuned/merged` directory. This checkpoint will come into play in Section 3.

▸ **Resources**

2. Training a Reward Model

The second step is training a reward model, which learns human preferences from labeled samples and guides the LLM during the final step of the RLHF process. This model will be provided with samples of favored behavior compared to not expected or desired behavior. The reward model will learn to imitate human preferences by assigning higher scores to samples that align with those preferences.

The reward models essentially perform a classification task, where they select the superior choice from a pair of sample interactions using input from human feedback. Various network types can be used and trained to function as reward models. Conversations are focused on the idea that the reward model should match the size of the base model in order to possess sufficient knowledge for practical guidance. Nonetheless, smaller models like DeBERTa or RoBERTa proved to be effective. Indeed, with enough resources available, experimenting with larger models is great. Both these models need to be loaded in the next phase of RLHF, which is Reinforcement Learning.

Begin the process by installing the necessary libraries. (We use a different version of the TRL library just in this subsection.)

```
pip install -q transformers==4.32.0 deeplake==3.6.19 sentencepiece==0.1.99 trl==0.6.0
```

The sample code.

2.1. The Dataset

💡 Please note that the datasets used in this step contain inappropriate language and offensive words. Nevertheless, this approach is the preferred way to align the model's behavior by exposing it to such language and instructing the model not to replicate it.

We will utilize the "helpfulness/harmless" (hh) dataset from Anthropic, specifically curated for the Reinforcement Learning from Human Feedback (RLHF) procedure (you can read more about it here). The Activeloop datasets hub provides access to a dataset that allows us to stream the content effortlessly using a single line of code. The subsequent code snippet will establish the data loader object for both the training and validation sets.

```
import deeplake

ds = deeplake.load('hub://genai360/Anthropic-hh-rlhf-train-set')
ds_valid = deeplake.load('hub://genai360/Anthropic-hh-rlhf-test-set')

print(ds)
```

The sample code.

```
Dataset(path='hub://genai360/Anthropic-hh-rlhf-train-set', read_only=True, tensors=['chose
```

The output.

Moving forward, we need to structure the dataset appropriately for the Trainer class. However, before that, let's load the pretrained tokenizer for the `DeBERTa` model we will use as the reward model. The code should be recognizable; the `AutoTokenizer` class will locate the suitable initializer class and utilize the `.from_pretrained()` method to load the pretrained tokenizer.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-v3-base")
```

The sample code.

As presented in earlier lessons, the `Dataset` class in PyTorch is in charge of formatting the dataset for various downstream tasks. A pair of inputs is necessary to train a reward model. The first item will denote the chosen (favorable) conversation, while the second will represent a conversation rejected by labelers, which we aim to prevent the model from replicating. The concept revolves around the reward model, which assigns a higher score to the chosen sample while assigning lower rankings to the rejected ones. The code snippet below initially tokenizes the samples and then aggregates the pairs into a single Python dictionary.

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, dataset):
        self.dataset = dataset

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):

      chosen = self.dataset.chosen[idx].text()
      rejected = self.dataset.rejected[idx].text()

      tokenized_chosen = tokenizer(chosen, truncation=True, max_length=max_length, padding
      tokenized_rejected = tokenizer(rejected, truncation=True, max_length=max_length, pad

      formatted_input = {
        "input_ids_chosen": tokenized_chosen["input_ids"],
        "attention_mask_chosen": tokenized_chosen["attention_mask"],
        "input_ids_rejected": tokenized_rejected["input_ids"],
        "attention_mask_rejected": tokenized_rejected["attention_mask"],
      }
    return formatted_input
```

The sample code.

The `Trainer` class anticipates receiving a dictionary containing four keys. This includes the tokenized forms for both chosen and rejected conversations (`input_ids_chosen` and `input_ids_rejected`) and their respective attention masks (`attention_mask_chosen` and `attention_mask_rejected`). As we employ a padding token to standardize input sizes (up to the model's maximum input size, 512 in this case), it's important to inform the model that certain tokens at the end don't contain meaningful information and can be disregarded. This is why attention masks are important.

We can create a dataset instance using the previously defined class. Additionally, we can extract a single row from the dataset using the `iter` and `next` methods to verify the output keys and confirm that everything functions as intended.

```
train_dataset = MyDataset(ds)
eval_dataset = MyDataset(ds_valid)

# Print one sample row
iterator = iter(train_dataset)
one_sample = next(iterator)
print(list(one_sample.keys()))
```

The sample code.

```
['input_ids_chosen', 'attention_mask_chosen', 'input_ids_rejected', 'attention_mask_reject
```

The output.

2.2. Initialize the Model and Trainer

The next steps are quite straightforward. We begin by loading the pretrained DeBERTa model using `AutoModelForSequenceClassification` , as our aim is to employ the network for a classification task. Specifying the number of labels (`num_labels`) as 1 is equally important, as we only require a single score to assess a sequence's quality. This score can indicate whether the content is aligned, receiving a high score, or, if it's unsuitable, receiving a low score.

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    "microsoft/deberta-v3-base", num_labels=1
)
```

The sample code.

Then, we can create an instance of `TrainingArguments` , setting the hyperparameters we intend to utilize. There is flexibility to explore various hyperparameters based on the selection of pre-trained models and available resources. For example, if an Out of Memory (OOM) error is encountered, a smaller batch size might be needed.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="DeBERTa-reward-hh_rlhf",
    learning_rate=2e-5,
    per_device_train_batch_size=24,
    per_device_eval_batch_size=24,
    num_train_epochs=20,
    weight_decay=0.001,
    evaluation_strategy="steps",
    eval_steps=500,
    save_strategy="steps",
    save_steps=500,
    gradient_accumulation_steps=1,
    bf16=True,
    logging_strategy="steps",
    logging_steps=1,
    optim="adamw_hf",
    lr_scheduler_type="linear",
    ddp_find_unused_parameters=False,
    run_name="DeBERTa-reward-hh_rlhf",
    report_to="wandb",
)
```

The sample code.

Finally, the `RewardTrainer` class from the TRL library will tie everything together and execute the training loop. It's essential to provide the previously defined variables, such as the model, tokenizer, and dataset.

```
from trl import RewardTrainer

trainer = RewardTrainer(
    model=model,
    tokenizer=tokenizer,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    max_length=max_length
)

trainer.train()
```

The sample code.

The `trainer` will automatically save the checkpoints, which can be utilized in the next and final steps.

▸ **Resources**

3. Reinforcement Learning (RL)

The final step of RLHF! This section will integrate the models we trained earlier. Specifically, we will utilize the previously trained reward model to further align the fine-tuned model with human feedback. In the training loop, a custom prompt will be employed to generate a response from the fine-tuned OPT. The reward model will then assign a score based on how closely the response resembles a hypothetical human-generated output. Within the RL process, mechanisms are also in place to ensure that the model retains its acquired knowledge and doesn't deviate too far from the original model's foundation. We will proceed by introducing the dataset, followed by a more detailed examination of the process in the subsequent subsections.

Begin the process by installing the necessary libraries.

```
pip install -q transformers==4.32.0 accelerate==0.22.0 peft==0.5.0 trl==0.5.0 bitsandbytes
```

The sample code.

3.1. The Dataset

Given that the process falls under the realm of unsupervised learning, we have the flexibility to choose the dataset for this step. Since the reward model assesses the output without relying on a label, the learning process does not require a question-answer pair. In this section, we will employ Alpaca's OpenOrca dataset, a subset of the OpenOrca dataset.

```
import deeplake

# Connect to the training and testing datasets
ds = deeplake.load('hub://genai360/Alpaca-OrcaChat')
print(ds)
```

The sample code.

```
Dataset(path='hub://genai360/Alpaca-OrcaChat', read_only=True, tensors=['input', 'instruct
```

The output.

The dataset comprises three columns: `input`, which denotes the user's prompt to the model; `instruction`, which represents the model's directive; and `output`, which holds the model's response. We only need to use the `input` feature for the RL process. Before defining a dataset class for proper formatting, it's necessary to load the pre-trained tokenizer corresponding to the fine-tuned model in the first section.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b", padding_side='left')
```

The sample code.

In the subsequent subsection, the trainer requires both the query and its tokenized variant. Thus, the `query` will remain in text format, whereas the `input_ids` will represent the token IDs. The dataset class format should be recognizable by now. An important point to highlight is that the `query` variable acts as a template for crafting user prompts, structured as follows: `Question: XXX\n\nAnswer:` in alignment with the format employed during the supervised fine-tuning (SFT) step.

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, ds):
        self.ds = ds

    def __len__(self):
        return len(self.ds)

    def __getitem__(self, idx):

      query = "Question: " + self.ds.input[idx].text() + "\n\nAnswer: "
      tokenized_question = tokenizer(query, truncation=True, max_length=400, padding='max_

      formatted_input = {
        "query": query,
        "input_ids": tokenized_question["input_ids"][0],
      }

      return formatted_input

# Define the dataset object
myTrainingLoader = MyDataset(ds)
```

The sample code.

Additionally, we must establish a collator function responsible for transforming individual samples from the data loader into data batches. This function will later be passed to the Trainer class.

```
def collator(data):
    return dict((key, [d[key] for d in data]) for key in data[0])
```

The sample code.

3.2. Initialize the SFT Models

In this section, we are required to load two models. To begin, let's initiate the process by loading the fine-tuned model, referred to as `OPT-supervised_fine_tuned` in section 1, utilizing the configuration provided by the PPOConfig class. The majority of the parameters have been elaborated on in earlier lessons, except `adapt_kl_ctrl` and `init_kl_coef` . These arguments will be used to control the KL divergence penalty to ensure the model doesn't stray significantly from the pre-trained model. Otherwise, it runs the risk of generating nonsensical sentences.

```
from trl import PPOConfig

config = PPOConfig(
    task_name="OPT-RL-OrcaChat",
    steps=10_000,
    model_name="./OPT-fine_tuned-OpenOrca/merged",
    learning_rate=1.41e-5,
    batch_size=32,
    mini_batch_size=4,
    gradient_accumulation_steps=1,
    optimize_cuda_cache=True,
    early_stopping=False,
    target_kl=0.1,
    ppo_epochs=4,
    seed=0,
    init_kl_coef=0.2,
    adap_kl_ctrl=True,
    tracker_project_name="GenAI360",
    log_with="wandb",
)
```

The sample code.

We also need to use the `set_seed()` function to set the random state for reproducibility, and the `current_device` variable will store your current device id and will be used later on the code.

```
from trl import set_seed
from accelerate import Accelerator

# set seed before initializing value head for deterministic eval
set_seed(config.seed)

# Now let's build the model, the reference model, and the tokenizer.
current_device = Accelerator().local_process_index
```

The sample code.

The next three code blocks are used to load the supervised fine-tuned (SFT) model. It starts by setting the details for the LoRA to accelerate the fine-tuning process.

```
from peft import LoraConfig
```

```
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
```

The sample code.

The LoRA config can then be used alongside the `AutoModelForCausalLMwithValueHead` class to load the pre-trained weights. We utilize the `load_in_8bit` argument to load the model, employing a quantization technique that reduces weight precision. This helps conserve memory during model training. This model is intended for utilization within the RL loop.

```
from trl import AutoModelForCausalLMWithValueHead

model = AutoModelForCausalLMWithValueHead.from_pretrained(
    config.model_name,
    load_in_8bit=True,
    device_map={"": current_device},
    peft_config=lora_config,
)
```

The sample code.

3.3. Initialize the Reward Models

Utilizing the Huggingface pipeline feature makes loading the Reward model straightforward. We need to define the task we are undertaking. In this case, we selected `sentiment-analysis` , as our fundamental objective revolves around binary classification. Furthermore, it is essential to indicate the path to the pre-trained reward model from the previous section using the `model` parameter. Alternatively, it could be possible to utilize a model's name from the HuggingFace Hub if a pre-trained reward model is accessible.

The pipeline will automatically load the appropriate tokenizer, and we can initiate classification by providing any text to the defined object.

```
from transformers import pipeline
import torch

reward_pipeline = pipeline(
    "sentiment-analysis",
    model="./DeBERTa-v3-base-reward-hh_rlhf/checkpoint-1000",
    tokenizer="./DeBERTa-v3-base-reward-hh_rlhf/checkpoint-1000",
    device_map={"": current_device},
    model_kwargs={"load_in_8bit": True},
    return_token_type_ids=False,
)
```

The sample code.

The `reward_pipe` variable containing the reward model will be employed within the reinforcement learning (RL) training loop.

3.4. PPO Training

The final stage involves employing Proximal Policy Optimization (PPO) to enhance the stability of the training loop. It will restrict alterations to the model by preventing excessively large updates. Empirical findings indicate that introducing minor adjustments accelerates the convergence of the training process, it is one of the intuitions behind the PPO.

Before beginning the actual training loop, certain variables need to be defined for integration within the loop. Initially, we establish the `output_length_sampler` object, which draws samples from a specified range spanning from a defined minimum to a maximum number. We want the outputs to be in the range of 32 to 128 tokens.

```
from trl.core import LengthSampler

output_length_sampler = LengthSampler(32, 400) #(OutputMinLength, OutputMaxLength)
```

The sample code.

We need to define two sets of dictionaries that will control the generation process for fine-tuned and reward models. We have the ability to configure arguments that oversee the sampling procedure, truncation, and batch size for each respective network during inference. The code block was concluded by setting the `save_freq` variable, which determines the interval for checkpoint preservation.

```
sft_gen_kwargs = {
    "top_k": 0.0,
    "top_p": 1.0,
    "do_sample": True,
    "pad_token_id": tokenizer.pad_token_id,
    "eos_token_id": 100_000,
}

reward_gen_kwargs = {
    "top_k": None,
    "function_to_apply": "none",
    "batch_size": 16,
    "truncation": True,
    "max_length": 400
}

save_freq = 50
```

The sample code.

The final action before the actual training loop involves the instantiation of the PPO trainer object. The `PPOTrainer` class will take as input an instance of `PPOConfig` , which we defined earlier, the directory of the fine-tuned model from section 1, and the training dataset.

It's worth noting that we have the option to provide a reference model using the `ref_model` parameter, which will serve as the guide for the KL divergence penalty. In cases where this parameter is not specified, the trainer will default to using the original pre-trained model as the reference.

```
from trl import PPOTrainer

ppo_trainer = PPOTrainer(
    config,
    model,
    tokenizer=tokenizer,
    dataset=myTrainingLoader,
    data_collator=collator
)
```

The sample code.

Now, we can proceed to the last component, which is the training loop. The process begins with obtaining a single batch of samples and utilizing the `input_ids` , which are the formatted and tokenized prompts (refer to section 3.1) to generate responses using the fine-tuned model. Subsequently, these responses are decoded and combined with the prompt before being fed to the reward model. This allows the reward model to assess their proximity to a human-generated response by assigning scores.

Finally, the PPO object will adjust the model based on the scores by the reward model.

```
from tqdm import tqdm
tqdm.pandas()

for step, batch in tqdm(enumerate(ppo_trainer.dataloader)):
    if step >= config.total_ppo_epochs:
        break
    question_tensors = batch["input_ids"]

    response_tensors = ppo_trainer.generate(
        question_tensors,
        return_prompt=False,
        length_sampler=output_length_sampler,
        **sft_gen_kwargs,
    )
    batch["response"] = tokenizer.batch_decode(response_tensors, skip_special_tokens=True)

    # Compute reward score
    texts = [q + r for q, r in zip(batch["query"], batch["response"])]
    pipe_outputs = reward_pipeline(texts, **reward_gen_kwargs)
    rewards = [torch.tensor(output[0]["score"]) for output in pipe_outputs]

    # Run PPO step
    stats = ppo_trainer.step(question_tensors, response_tensors, rewards)
    ppo_trainer.log_stats(stats, batch, rewards)

    if save_freq and step and step % save_freq == 0:
        print("Saving checkpoint.")
        ppo_trainer.save_pretrained(f"./OPT-RL-OrcaChat/checkpoint-{step}")
```

The sample code.

Remember to merge the LoRA adaptors with the base model to ensure that the network can be utilized independently as a standalone model in the future. Simply ensure to modify the directory of the saved checkpoint adaptor according to the results.

```
from transformers import AutoModelForCausalLM
import torch

model = AutoModelForCausalLM.from_pretrained(
  "facebook/opt-1.3b", return_dict=True, torch_dtype=torch.bfloat16
)

from peft import PeftModel

# Load the Lora model
model = PeftModel.from_pretrained(model, "./OPT-RL-OrcaChat/checkpoint-400/")
model.eval();

model = model.merge_and_unload()

model.save_pretrained("./OPT-RL-OrcaChat/merged")
```

The sample code.

▸ **Resources**

QLoRA

Earlier, we employed an argument named `load_in_8bit` during the loading of the base model. This quantization technique significantly reduces the memory requirement when loading large models. A 32-bit floating-point format was utilized for model training in the early stages of neural network development. This entailed the representation of each weight using 32 bits, requiring 4 bytes for storage per weight.

Researchers developed diverse methods to mitigate this constraint with the growth of models and the escalating memory requirements. This led to the utilization of lower-precision values for the loading model. Employing an 8-bit representation for numbers reduces the storage requirement to a mere 1 byte.

In more recent times, an additional advancement allows for models to be loaded in a 4-bit format, further reducing memory demands. It is possible to use the BitsAndBytes library while loading a pre-trained model, as the following code presents.

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
import torch

model = AutoModelForCausalLM.from_pretrained(
    model_name_or_path='/name/or/path/to/your/model',
    load_in_4bit=True,
    device_map='auto',
    torch_dtype=torch.bfloat16,
    quantization_config=BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type='nf4'
    ),
    )
```

The sample code.

It's crucial to remember that this approach relates exclusively to storing model weights and does not affect the training process. Additionally, there's a constant balance to strike between employing lower-precision numbers and potentially compromising the language comprehension capabilities of models. While this trade-off is justified in many instances, it's important to acknowledge its presence.

💡 Prior to progressing to the next section to observe the outcomes of the fine-tuned model, it's important to reiterate that the base model employed in this lesson is a relatively small language model with limited capabilities when compared with the state-of-the-art models we are accustomed to by now, such as ChatGPT. Remember that the insights gained from this lesson can be easily applied to train significantly larger variations of the models, leading to notably improved outcomes. (As highlighted in the lesson's introduction, the key used for loading the tokenizer/model can be modified to models with any size like LLaMA2.)

Inference

We can evaluate the fine-tuned model's outputs by employing various prompts. The code below demonstrates how we can utilize Huggingface's `.generate()` method to interact with models effortlessly. The initial stage involves loading the tokenizer and the model, followed by decoding the output generated by the model. We employ the beam search decoding approach with a limitation to generate a maximum of 128 tokens. (Explore these techniques further in the in-depth blog post by Huggingface.)

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-1.3b")

from transformers import AutoModelForCausalLM
from accelerate import Accelerator

model = AutoModelForCausalLM.from_pretrained(
    "./OPT-RL-OrcaChat/merged", device_map={"": Accelerator().process_index}
)
model.eval();

inputs = tokenizer("Question: In one sentence, describe what the following article is abou
generation_output = model.generate(**inputs,
                                   return_dict_in_generate=True,
                                   output_scores=True,
                                   max_new_tokens=128,
                                   num_beams=4,
                                   do_sample=True,
                                   top_k=10,
                                   temperature=0.6)
print( tokenizer.decode(generation_output['sequences'][0]) )
```

The sample code.

The following entries represent the outputs generated by the model using various prompts.

- ▸ In one sentence, describe what the following article is about…
- ▸ Answer the following question given in this paragraph…
- ▸ What the following paragraph is about?…
- ▸ What the following paragraph is about?… (2)

As evidenced by the examples, the model displays the ability to follow instructions and extract information from lengthy content. However, it falls short in terms of answering open-ended questions such as "Explain the raining process?" This is primarily attributed to the model's smaller size, which entails fewer parameters, approximately ranging from 30x to 70x less than state-of-the-art models.

Conclusion

This lesson experimented with the three essential Reinforcement Learning with Human Feedback (RLHF) process stages. It starts by revisiting the Supervised Fine-Tuning (SFT) process, then proceeds with the training of a reward model, and finally concludes with the reinforcement learning phase. We explored and applied methods such as 4-bit quantization and LoRA to enhance the fine-tuning procedure while utilizing fewer resources. In the upcoming chapter, we will introduce the procedure of deploying models and utilizing them in a production environment.