AWS Machine Learning Blog

Instruction fine-tuning for FLAN T5 XL with Amazon SageMaker Jumpstart

by Laurent Callot, Andrey Kan, Jonas Kübler, Baris Kurt, and Ashish Khetan | on 22 MAY 2023 | in Amazon SageMaker JumpStart, Artificial Intelligence, Technical How-to | Permalink | Comments | Share

Generative AI is in the midst of a period of stunning growth. Increasingly capable foundation models are being released continuously, with large language models (LLMs) being one of the most visible model classes. LLMs are models composed of billions of parameters trained on extensive corpora of text, up to hundreds of billions or even a trillion tokens. These models have proven extremely effective for a wide range of text-based tasks, from question answering to sentiment analysis.

The power of LLMs comes from their capacity to learn and generalize from extensive and diverse training data. The initial training of these models is performed with a variety of objectives, supervised, unsupervised, or hybrid. Text completion or imputation is one of the most common unsupervised objectives: given a chunk of text, the model learns to accurately predict what comes next (for example, predict the next sentence). Models can also be trained in a supervised fashion using labeled data to accomplish a set of tasks (for example, is this movie review positive, negative, or neutral). Whether the model is trained for text completion or some other task, it is frequently not the task customers want to use the model for.

To improve the performance of a pre-trained LLM on a specific task, we can tune the model using examples of the target task in a process known as *instruction fine-tuning*. Instruction fine-tuning uses a set of labeled examples in the form of {prompt, response} pairs to further train the pre-trained model in adequately predicting the response given the prompt. This process modifies the weights of the model.

This post describes how to perform instruction fine-tuning of an LLM, namely FLAN T5 XL, using Amazon SageMaker Jumpstart. We demonstrate how to accomplish this using both the Jumpstart UI and a notebook in Amazon SageMaker Studio. You can find the accompanying notebook in the amazon-sagemaker-examples GitHub repository.

Solution overview

The target task in this post is to, given a chunk of text in the prompt, return questions that are related to the text but can't be answered based on the information it contains. This is a useful task to identify missing information in a description or identify whether a query needs more information to be answered.

FLAN T5 models are instruction fine-tuned on a wide range of tasks to increase the zero-shot performance of these models on many common tasks[1]. Additional instruction fine-tuning for a particular customer task can further increase the accuracy of these models, especially if the target task wasn't previously used to train a FLAN T5 model, as is the case for our task.

In our example task, we're interested in generating relevant but unanswered questions. To this end, we use a subset of the version 2 of the Stanford Question Answering Dataset (SQuAD2.0)[2] to fine-tune the model. This dataset contains questions posed by human annotators on a set of Wikipedia articles. In addition to questions with answers, SQuAD2.0 contains about 50,000 unanswerable questions. Such questions are plausible but can't be directly answered from articles' content. We only use the unanswerable questions. Our data is structured as a JSON Lines file, with each line containing a context and a question.

Prerequisites

To get started, all you need is an AWS account in which you can use Studio. You will need to create a user profile for Studio if you don't already have one.

Fine-tune FLAN-T5 with the Jumpstart UI

To fine-tune the model with the Jumpstart UI, complete the following steps:

1. On the SageMaker console, open Studio.
2. Under **SageMaker Jumpstart** in the navigation pane, choose **Models, notebooks, solutions**.

You will see a list of foundation models, including FLAN T5 XL, which is marked as fine-tunable.

3. Choose **View model**.

4. Under **Data source**, you can provide the path to your training data. The source for the data used in this post is provided by default.
5. You can keep the default value for the deployment configuration (including instance type), security, and the hyperparameters, but you should increase the number of epochs to at least three to get good results.
6. Choose **Train** to train the model.

You can track the status of the training job in the UI.

7. When training is complete (after about 53 minutes in our case), choose **Deploy** to deploy the fine-tuned model.

After the endpoint is created (a few minutes), you can open a notebook and start using your fine-tuned model.

Fine-tune FLAN-T5 using a Python notebook

Our example notebook shows how to use Jumpstart and SageMaker to programmatically fine-tune and deploy a FLAN T5 XL model. It can be run in Studio or locally.

In this section, we first walk through some general setup. Then you fine-tune the model using the SQuADv2 datasets. Next, you deploy the pre-trained version of the model behind a SageMaker endpoint, and do the same

with the fine-tuned model. Finally, you can query the endpoints and compare the quality of the output of the pre-trained and fine-tuned model. You will find that the output of the fine-tuned model is of much higher quality.

Set up prerequisites

Begin by installing and upgrading the necessary packages. Restart the kernel after running the following code:

```
!pip install nest-asyncio==1.5.5 --quiet
!pip install ipywidgets==8.0.4 --quiet
!pip install --upgrade sagemaker --quiet
```

Next, obtain the execution role associated with the current notebook instance:

```
import boto3
import sagemaker
# Get current region, role, and default bucket
aws_region = boto3.Session().region_name
aws_role = sagemaker.session.Session().get_caller_identity_arn()
output_bucket = sagemaker.Session().default_bucket()
# This will be useful for printing
newline, bold, unbold = "\n", "\033[1m", "\033[0m"
print(f"{bold}aws_region:{unbold} {aws_region}")
print(f"{bold}aws_role:{unbold} {aws_role}")
print(f"{bold}output_bucket:{unbold} {output_bucket}"
```

You can define a convenient drop-down menu that will list the model sizes available for fine-tuning:

```
import IPython
from ipywidgets import Dropdown
from sagemaker.jumpstart.filters import And
from sagemaker.jumpstart.notebook_utils import list_jumpstart_models
# Default model choice
model_id = "huggingface-text2text-flan-t5-xl"
# Identify FLAN T5 models that support fine-tuning
filter_value = And(
"task == text2text", "framework == huggingface", "training_supported == true"
)
model_list = [m for m in list_jumpstart_models(filter=filter_value) if "flan-t5" in
# Display the model IDs in a dropdown, for user to select
dropdown = Dropdown(
value=model_id,
options=model_list,
description="FLAN T5 models available for fine-tuning:",
style={"description_width": "initial"},
```

```
layout={"width": "max-content"},
)
```

Jumpstart automatically retrieves appropriate training and inference instance types for the model that you chose:

```
from sagemaker.instance_types import retrieve_default
model_id, model_version = dropdown.value, "*"
# Instance types for training and inference
training_instance_type = retrieve_default(
model_id=model_id, model_version=model_version, scope="training"
)
inference_instance_type = retrieve_default(
model_id=model_id, model_version=model_version, scope="inference"
)
print(f"{bold}model_id:{unbold} {model_id}")
print(f"{bold}training_instance_type:{unbold} {training_instance_type}")
print(f"{bold}inference_instance_type:{unbold} {inference_instance_type}")

If you have chosen the FLAN T5 XL, you will see the following output:

model_id: huggingface-text2text-flan-t5-xl

training_instance_type: ml.p3.16xlarge
```

You're now ready to start fine-tuning.

Retrain the model on the fine-tuning dataset

After your setup is complete, complete the following steps:

Use the following code to retrieve the URI for the artifacts needed:

```
from sagemaker import image_uris, model_uris, script_uris
# Training instance will use this image
train_image_uri = image_uris.retrieve(
region=aws_region,
framework=None,  # automatically inferred from model_id
model_id=model_id,
model_version=model_version,
image_scope="training",
instance_type=training_instance_type,
)
# Pre-trained model
train_model_uri = model_uris.retrieve(
model_id=model_id, model_version=model_version, model_scope="training"
)
```

```
# Script to execute on the training instance
train_script_uri = script_uris.retrieve(
model_id=model_id, model_version=model_version, script_scope="training"
)
print(f"{bold}image uri:{unbold} {train_image_uri}")
```

The training data is located in a public Amazon Simple Storage Service (Amazon S3) bucket.

Use the following code to point to the location of the data and set up the output location in a bucket in your account:

```
from sagemaker.s3 import S3Downloader

# We will use the train split of SQuAD2.0
original_data_file = "train-v2.0.json"

# The data was mirrored in the following bucket
original_data_location = f"s3://sagemaker-sample-files/datasets/text/squad2.0/{origi
S3Downloader.download(original_data_location, ".")
```

The original data is not in a format that corresponds to the task for which you are fine-tuning the model, so you can reformat it:

```
import json

local_data_file = "task-data.jsonl"  # any name with .jsonl extension

with open(original_data_file) as f:
data = json.load(f)

with open(local_data_file, "w") as f:
for article in data["data"]:
for paragraph in article["paragraphs"]:
# iterate over questions for a given paragraph
for qas in paragraph["qas"]:
if qas["is_impossible"]:
# the question is relevant, but cannot be answered
example = {"context": paragraph["context"], "question": qas["question"]}
json.dump(example, f)
f.write("\n")

template = {
```

Now you can define some hyperparameters for the training:

```
from sagemaker import hyperparameters

# Retrieve the default hyper-parameters for fine-tuning the model
hyperparameters = hyperparameters.retrieve_default(model_id=model_id, model_version=

# We will override some default hyperparameters with custom values
hyperparameters["epochs"] = "3"
# TODO
# hyperparameters["max_input_length"] = "300"  # data inputs will be truncated at th
# hyperparameters["max_output_length"] = "40"  # data outputs will be truncated at t
# hyperparameters["generation_max_length"] = "40"  # max length of generated output
print(hyperparameters)
```

You are now ready to launch the training job:

```
from sagemaker.estimator import Estimator
from sagemaker.utils import name_from_base

model_name = "-".join(model_id.split("-")[2:])  # get the most informative part of I
training_job_name = name_from_base(f"js-demo-{model_name}-{hyperparameters['epochs']
print(f"{bold}job name:{unbold} {training_job_name}")

training_metric_definitions = [
{"Name": "val_loss", "Regex": "'eval_loss': ([0-9\\.]+)"},
{"Name": "train_loss", "Regex": "'loss': ([0-9\\.]+)"},
{"Name": "epoch", "Regex": "'epoch': ([0-9\\.]+)"},
]

# Create SageMaker Estimator instance
sm_estimator = Estimator(
role=aws_role,
image_uri=train_image_uri,
model_uri=train_model_uri,
source_dir=train_script_uri,
```

Depending on the size of the fine-tuning data and model chosen, the fine-tuning could take up to a couple of hours.

You can monitor performance metrics such as training and validation loss using Amazon CloudWatch during training. Conveniently, you can also fetch the most recent snapshot of metrics by running the following code:

```
from sagemaker import TrainingJobAnalytics

# This can be called while the job is still running
df = TrainingJobAnalytics(training_job_name=training_job_name).dataframe()
```

```
df.head(10)

model uri: s3://sagemaker-us-west-2-802376408542/avkan/training-huggingface-text2tex
job name: jumpstart-demo-xl-3-2023-04-06-08-16-42-738
INFO:sagemaker:Creating training-job with name: jumpstart-demo-xl-3-2023-04-06-08-16
```

When the training is complete, you have a fine-tuned model at `model_uri`. Let's use it!

You can create two inference endpoints: one for the original pre-trained model, and one for the fine-tuned model. This allows you to compare the output of both versions of the model. In the next step, you deploy an inference endpoint for the pre-trained model. Then you deploy an endpoint for your fine-tuned model.

Deploy the pre-trained model

Let's start by deploying the pre-trained model retrieve the inference Docker image URI. This is the base Hugging Face container image. Use the following code:

```
from sagemaker import image_uris

# Retrieve the inference docker image URI. This is the base HuggingFace container im
deploy_image_uri = image_uris.retrieve(
region=None,
framework=None,  # automatically inferred from model_id
model_id=model_id,
model_version=model_version,
image_scope="inference",
instance_type=inference_instance_type,
)
```

You can now create the endpoint and deploy the pre-trained model. Note that you need to pass the Predictor class when deploying model through the Model class to be able to run inference through the SageMaker API. See the following code:

```
from sagemaker import model_uris, script_uris
from sagemaker.model import Model
from sagemaker.predictor import Predictor
from sagemaker.utils import name_from_base

# Retrieve the URI of the pre-trained model
pre_trained_model_uri = model_uris.retrieve(
model_id=model_id, model_version=model_version, model_scope="inference"
)

pre_trained_name = name_from_base(f"jumpstart-demo-pre-trained-{model_id}")

# Create the SageMaker model instance of the pre-trained model
```

```
if ("small" in model_id) or ("base" in model_id):
deploy_source_uri = script_uris.retrieve(
model_id=model_id, model_version=model_version, script_scope="inference"
)
pre_trained_model = Model(
image_uri=deploy_image_uri
```

The endpoint creation and model deployment can take a few minutes, then your endpoint is ready to receive inference calls.

Deploy the fine-tuned model

Let's deploy the fine-tuned model to its own endpoint. The process is almost identical to the one we used earlier for the pre-trained model. The only difference is that we use the fine-tuned model name and URI:

```
from sagemaker.model import Model
from sagemaker.predictor import Predictor
from sagemaker.utils import name_from_base

fine_tuned_name = name_from_base(f"jumpstart-demo-fine-tuned-{model_id}")
fine_tuned_model_uri = f"{output_location}{training_job_name}/output/model.tar.gz"

# Create the SageMaker model instance of the fine-tuned model
fine_tuned_model = Model(
image_uri=deploy_image_uri,
model_data=fine_tuned_model_uri,
role=aws_role,
predictor_cls=Predictor,
name=fine_tuned_name,
)

print(f"{bold}image URI:{unbold}{newline} {deploy_image_uri}")
print(f"{bold}model URI:{unbold}{newline} {fine_tuned_model_uri}")
print("Deploying an endpoint ...")
```

When this process is complete, both pre-trained and fine-tuned models are deployed behind their own endpoints. Let's compare their outputs.

Generate output and compare the results

Define some utility functions to query the endpoint and parse the response:

```
import boto3
import json

# Parameters of (output) text generation. A great introduction to generation
# parameters can be found at https://huggingface.co/blog/how-to-generate
parameters = {
```

```
"max_length": 40,  # restrict the length of the generated text
"num_return_sequences": 5,  # we will inspect several model outputs
"num_beams": 10,  # use beam search
}

# Helper functions for running inference queries
def query_endpoint_with_json_payload(payload, endpoint_name):
encoded_json = json.dumps(payload).encode("utf-8")
client = boto3.client("runtime.sagemaker")
response = client.invoke_endpoint(
EndpointName=endpoint_name, ContentType="application/json", Body=encoded_json
)
return response
```

In the next code snippet, we define the prompt and the test data. The describes our target task, which is to generate questions that are related to the provided text but can't be answered based on it.

The test data consists of three different paragraphs, one on the Australian city of Adelaide from the first two paragraphs of it Wikipedia page, one regarding Amazon Elastic Block Store (Amazon EBS) from the Amazon EBS documentation, and one of Amazon Comprehend from the Amazon Comprehend documentation. We expect the model to identify questions related to these paragraphs but that can't be answered with the information provided therein.

```
prompt = "Ask a question which is related to the following text, but cannot be answe

test_paragraphs = [
"""
Adelaide is the capital city of South Australia, the state's largest city and the fi
"Adelaide" may refer to either Greater Adelaide (including the Adelaide Hills) or th
The demonym Adelaidean is used to denote the city and the residents of Adelaide. The
region are the Kaurna people. The area of the city centre and surrounding parklands

Adelaide is situated on the Adelaide Plains north of the Fleurieu Peninsula, betweer
the Mount Lofty Ranges in the east. Its metropolitan area extends 20 km (12 mi) from
the Mount Lofty Ranges, and stretches 96 km (60 mi) from Gawler in the north to Sell
""",
"""
Amazon Elastic Block Store (Amazon EBS) provides block level storage volumes for use

We recommend Amazon EBS for data that must be quickly accessible and requires long-t
""",
"""
```

You can now test the endpoints using the example articles

```
print(f"{bold}Prompt:{unbold} {repr(prompt)}")
for paragraph in test_paragraphs:
print("-" * 80)
```

```
print(paragraph)
print("-" * 80)
print(f"{bold}pre-trained{unbold}")
generate_questions(pre_trained_name, paragraph)
print(f"{bold}fine-tuned{unbold}")
generate_questions(fine_tuned_name, paragraph)
```

Test data: Adelaide

We use the following context:

```
delaide is the capital city of South Australia, the state's largest city and the fif
"Adelaide" may refer to either Greater Adelaide (including the Adelaide Hills) or th
The demonym Adelaidean is used to denote the city and the residents of Adelaide. The
region are the Kaurna people. The area of the city centre and surrounding parklands

Adelaide is situated on the Adelaide Plains north of the Fleurieu Peninsula, betweer
the Mount Lofty Ranges in the east. Its metropolitan area extends 20 km (12 mi) from
the Mount Lofty Ranges, and stretches 96 km (60 mi) from Gawler in the north to Sell
```

The pre-trained model response is as follows:

```
Response 0: What is the area of the city centre and surrounding parklands called in
Response 1: What is the area of the city centre and surrounding parklands is called
Response 2: What is the area of the city centre and surrounding parklands called in
Response 3: What is the capital city of South Australia?
Response 4: What is the area of the city centre and surrounding parklands known as i
```

The fine-tuned model responses are as follows:

```
Response 0: What is the second most populous city in Australia?
Response 1: What is the fourth most populous city in Australia?
Response 2: What is the population of Gawler?
Response 3: What is the largest city in Australia?
Response 4: What is the fifth most populous city in the world?
```

Test data: Amazon EBS

We use the following context:

```
Amazon Elastic Block Store (Amazon EBS) provides block level storage volumes for use
```

```
We recommend Amazon EBS for data that must be quickly accessible and requires long-t
```

The pre-trained model responses are as follows:

```
esponse 0: What is the difference between Amazon EBS and Amazon Elastic Block Store
Response 1: What is the difference between Amazon EBS and Amazon Elastic Block Store
Response 2: What is the difference between Amazon EBS and Amazon Simple Storage Serv
Response 3: What is Amazon Elastic Block Store (Amazon EBS)?
Response 4: What is the difference between Amazon EBS and a hard drive?
```

The fine-tuned model responses are as follows:

```
Response 0: What type of applications are not well suited to Amazon EBS?
Response 1: What behaves like formatted block devices?
Response 2: What type of applications are not suited to Amazon EBS?
Response 3: What type of applications are not well suited for Amazon EBS?
Response 4: What type of applications are not suited for Amazon EBS?
```

Test data: Amazon Comprehend

We use the following context:

```
Amazon Comprehend uses natural language processing (NLP) to extract insights about t
You can access Amazon Comprehend document analysis capabilities using the Amazon Com
All of the Amazon Comprehend features accept UTF-8 text documents as the input. In a
Amazon Comprehend can examine and analyze documents in a variety of languages, deper
```

The pre-trained model responses are as follows:

```
Response 0: What does Amazon Comprehend use to extract insights about the content of
Response 1: How does Amazon Comprehend extract insights about the content of documer
Response 2: What does Amazon Comprehend use to develop insights about the content of
Response 3: How does Amazon Comprehend develop insights about the content of documer
Response 4: What does Amazon Comprehend use to extract insights about the content of
```

The fine-tuned model responses are as follows:

```
Response 0: What does Amazon Comprehend use to extract insights about the structure
Response 1: How does Amazon Comprehend recognize sentiments in a document?
Response 2: What does Amazon Comprehend use to extract insights about the content of
```

```
Response 3: What does Amazon Comprehend use to extract insights about the content of
Response 4: What type of files does Amazon Comprehend reject as input?
```

The difference in output quality between the pre-trained model and the fine-tuned model is stark. The questions provided by the fine-tuned model touch on a wider range of topics. They are systematically meaningful questions, which isn't always the case for the pre-trained model, as illustrated with the Amazon EBS example.

Although this doesn't constitute a formal and systematic evaluation, it's clear that the fine-tuning process has improved the quality of the model's responses on this task.

Clean up

Lastly, remember to clean up and delete the endpoints:

```
# Delete resources
pre_trained_predictor.delete_model()
pre_trained_predictor.delete_endpoint()
fine_tuned_predictor.delete_model()
fine_tuned_predictor.delete_endpoint()
```

Conclusion

In this post, we showed how to use instruction fine-tuning with FLAN T5 models using the Jumpstart UI or a Jupyter notebook running in Studio. We provided code explaining how to retrain the model using data for the target task and deploy the fine-tuned model behind an endpoint. The target task in this post was to identify questions that relate to a chunk of text provided in the input but can't be answered based on the information provided in that text. We demonstrated that a model fine-tuned for this specific task returns better results than a pre-trained model.

Now that you know how to instruction fine-tune a model with Jumpstart, you can create powerful models customized for your application. Gather some data for your use case, uploaded it to Amazon S3, and use either the Studio UI or the notebook to tune a FLAN T5 model!

References

[1] Chung, Hyung Won, et al. "Scaling instruction-fine tuned language models." arXiv preprint arXiv:2210.11416 (2022).

[2] Rajpurkar, Pranav, Robin Jia, and Percy Liang. "Know What You Don't Know: Unanswerable Questions for SQuAD." *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2018.

About the authors

Laurent Callot is a Principal Applied Scientist and manager at AWS AI Labs who has worked on a variety of machine learning problems, from foundational models and generative AI to forecasting, anomaly detection, causality, and AI Ops.

Andrey Kan is a Senior Applied Scientist at AWS AI Labs within interests and experience in different fields of Machine Learning. These include research on foundation models, as well as ML applications for graphs and time series.

Dr. Ashish Khetan is a Senior Applied Scientist with Amazon SageMaker built-in algorithms and helps develop machine learning algorithms. He got his PhD from University of Illinois Urbana Champaign. He is an active researcher in machine learning and statistical inference and has published many papers in NeurIPS, ICML, ICLR, JMLR, ACL, and EMNLP conferences.

Baris Kurt is an Applied Scientist at AWS AI Labs. His interests are in time series anomaly detection and foundation models. He loves developing user friendly ML systems.

Jonas Kübler is an Applied Scientist at AWS AI Labs. He is working on foundation models with the goal to facilitate use-case specific applications.

Comments