Getting started with NLP using Hugging Face transformers pipelines

by Paul Ogilvie

February 6, 2023 in Engineering Blog

Advances in Natural Language Processing (NLP) have unlocked unprecedented opportunities for businesses to get value out of their text data. Natural Language Processing can be used for a wide range of applications, including text summarization, named-entity recognition (e.g. people and places), sentiment classification, text classification, translation, and question answering. In many cases, you can get high-quality results from machine learning models that have been previously trained on large text datasets. Many of these pre-trained models are available in the open source and are free to use. Hugging Face is one great source of these models, and their Transformers library is an easy-to-use tool for applying the models and also adapting them to your own data. It's also possible to adjust these models using fine-tuning to your own data.

For example, a company with a support team could use pre-trained models to provide human-readable summaries of text to help employees quickly assess key issues in support cases. This company can also easily train world class classification algorithms based on the readily-available foundation models to automatically categorize their support data into their internal taxonomies.

Databricks is a great platform for running Hugging Face Transformers. Previous Databricks articles have discussed the use of transformers for pre-trained model inference and fine-tuning, but this article consolidates those best practices to optimize performance and ease-of-use when working with transformers on the Lakehouse. This document includes in-line code samples alongside explanations of those best practices, and Databricks also provides complete notebook examples for pre-trained model inference and fine-tuning.

Using pre-trained models

For many applications, such as sentiment analysis and text summarization, pre-trained models work well without any additional model training. 🤗 Transformers pipelines wrap the various components required for inference on text into a simple interface. For many NLP tasks, these components consist of a tokenizer and a model. Pipelines encode best practices, making it easy to get started. For example, pipelines make it easy to use GPUs when available and allow batching of items sent to the GPU for better throughput.

```
from transformers import pipeline
import torch
# use the GPU if available
device = 0 if torch.cuda.is_available() else -1
summarizer = pipeline("summarization", device=device)
```

To distribute the inference on Spark, Databricks recommends encapsulating a pipeline in a pandas UDF. Spark uses broadcast to efficiently transmit any objects required by the pandas UDFs to the worker nodes. Spark also automatically reassigns the GPUs to workers, so you can use a multi-GPU multi-machine cluster seamlessly.

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('string')
def summarize_batch_udf(texts: pd.Series) -> pd.Series:
  pipe = summarizer(texts.to_list(), truncation=True, batch_size=1)
  summaries = [summary['summary_text'] for summary in pipe]
  return pd.Series(summaries)

summaries = df.select(df.text, summarize_batch_udf(df.text).alias("summary"))
```

Here's a sample summary for a snapshot of the Wikipedia article on the band Energy Orchard. Note that we did not clean up the Wikipedia markup before passing it to the summarizer.

```
{{Refimprove|date=September 2010}} '''Energy Orchard''' were a [[guitar]]-based [[Rock and roll|rock]] [[musical ensemble|band]] of the late 1980s and early 1990s, from [[Belfast]], [[Northern Ireland]]. Fronted by [[Bap Kennedy]] (brother of [[singer-songwriter]] [[Brian Kennedy (singer)|Brian Kennedy]]), their style drew heavily on the influence of [[Van Morrison]] and other [[rhythm and blues]] acts, but incorporated traditional elements of Irish [[folk music]].{{citation needed|date=January ...
```

Raw Wikipedia text for Energy Orchard

```
''Energy Orchard'' were a Belfast-based band of the late 1980s and early 1990s. Their style drew heavily on the influence of Van Morrison and otherrhythm and blues acts. Despite extensive touring, the breakthrough to mainstream success eluded them. The band having completed their recording contract in 1996 disbanded.
```

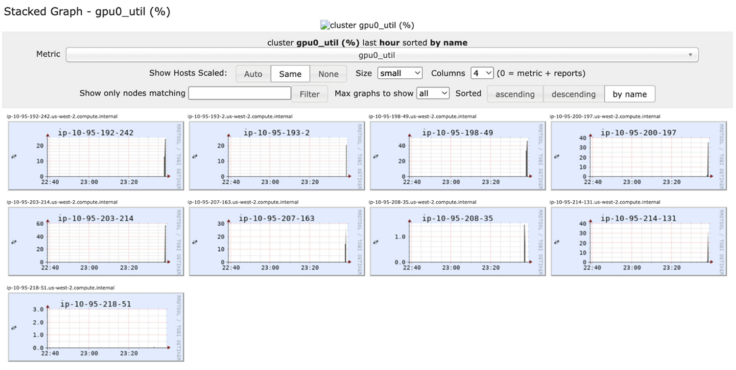
Output summary

This section demonstrated just how easy it is to get started using Hugging Face Transformers to process text at scale on Databricks. The next section describes how you can further tune the performance of these models.

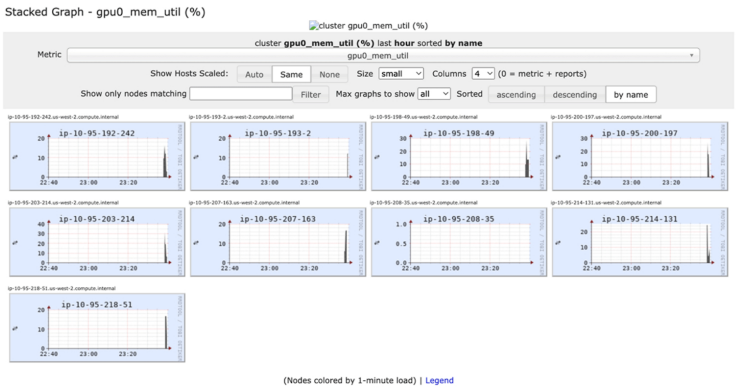
Tuning performance

There are two key aspects to tuning performance of the UDF. The first is that you want to use each GPU effectively, which you can adjust by changing the size of batch sizes for items sent to the GPU by the Transformers pipeline. The second is to make sure your dataframe is well-partitioned to utilize the entire cluster.

Your UDF should work out-of-the box with a `batch_size` of 1. However, this may not be efficiently utilizing the resources available to the workers. To get improved performance, you can tune the batch size to the model and hardware you are using. Databricks recommends trying various batch sizes for the pipeline on your cluster to find the best performance. Read more about pipeline batching and other performance options in Hugging Face documentation. You can monitor GPU performance by viewing the live ganglia metrics for a cluster, and choosing a metric, such as "gpu0-util" for GPU processor utilization or "gpu0_mem_util" for GPU memory utilization.



GPU processor utilization



GPU memory utilization

Your goal with tuning the batch size is to set it large enough so that it drives the full GPU utilization but does not result in "CUDA out of memory" errors.

To make good utilization of the hardware in your cluster, you may need to repartition your Spark DataFrame. Generally some multiple of the number of GPUs on your workers (for GPU clusters) or number of cores across the workers in your cluster (for CPU clusters) works well in practice. Using the UDF is identical to using other UDFs on Spark. For example, you can use it in a select statement to create a column with the results of the model inference.

```
sample = sample.repartition(32)
summaries = sample.select(sample.text, summarize_batch_udf(sample.text).alias("summary"
```

Wrapping pre-built models as MLflow models

Storing a pre-trained model as an MLflow model makes it even easier to deploy a model for batch or real-time inference. This also allows model versioning through the Model Registry, and simplifies model loading code for your inference workloads.

The first step is to create a custom model for your pipeline, which encapsulates loading the model, initializing the GPU usage, and inference function.

```
import mlflow

class SummarizationPipelineModel(mlflow.pyfunc.PythonModel):
  def load_context(self, context):
    device = 0 if torch.cuda.is_available() else -1
    self.pipeline = pipeline("summarization", context.artifacts["pipeline"], device=dev

  def predict(self, context, model_input):
    texts = model_input.iloc[:,0].to_list() # get the first column
    pipe = self.pipeline(texts, truncation=True, batch_size=8)
    summaries = [summary['summary_text'] for summary in pipe]
    return pd.Series(summaries)
```

The code closely parallels the code for creating and using a pandas_udf outlined above. One difference is that the pipeline is loaded from a file made available to the MLflow model's context. This is provided to MLflow when logging a model. Hugging Face transformers pipelines make it easy to save the model to a local file on the driver, which is then passed into the log_model function for the MLflow pyfunc interfaces.

```
summarizer.save_pretrained("./pipeline")
with mlflow.start_run() as run:
  mlflow.pyfunc.log_model(artifacts={'pipeline': "./pipeline"}, artifact_path="summariza
```

Batch inference using MLflow models

MLflow provides an easy interface to load any logged or registered model into a spark UDF. You can look up a model URI from the Model Registry or logged experiment run UI.

```
logged_model_uri = f"runs:/{run.info.run_id}/summarization_model"
loaded_model = mlflow.pyfunc.spark_udf(spark, model_uri=logged_model_uri, result_type='
summaries = df.select(df.title, df.text, loaded_model(df.text).alias("summary"))
```

Fine-tuning models on a single machine using 🤗 Transformers Trainer

Sometimes the pre-trained models do not serve your needs out-of-the-box and you must fine-tune the models on your own data. For example, you may want to create a text classifier based on a foundation model that classifies support tickets into your support teams' ontologies, or you may wish to create a custom spam classifier on your data.

You do not need to leave Databricks in order to fine-tune your models. For moderately sized datasets, you can do this on a single machine with GPU support. The Hugging Face transformers Trainer utility makes it very easy to set up and perform model training. For larger datasets, Databricks also supports distributed multi-machine multi-GPU deep learning.

The process is as follows: create a single machine cluster with GPU support, prepare and download your dataset to the driver, perform model training using Trainer, and log the resulting model to MLflow.

Preparing and downloading data

Start by formatting your training data into a table meeting the expectations of the Trainer. For text classification, this is a table with two columns: a text column and a column of labels. In our example notebook we load some text message spam data:

Table +

	label	text
1	ham	Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
2	ham	Ok lar... Joking wif u oni...
3	spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810075over18's

Hugging Face transformers provides AutoModelForSequenceClassification as a model loader for text classification, which expects integer IDs as the category labels. However, you must also specify mappings from the integer labels to string labels and back. If you have a DataFrame with string labels, you can collect this information as follows:

```
labels = sms.select(sms.label).groupBy(sms.label).count().collect()
id2label = {index: row.label for (index, row) in enumerate(labels)}
label2id = {row.label: index for (index, row) in enumerate(labels)}
```

Then create the integer ids as a label column with a `pandas_udf`:

```
from pyspark.sql.functions import pandas_udf
```

```
import pandas as pd
@pandas_udf('integer')
def replace_labels_with_ids(labels: pd.Series) -> pd.Series:
  return labels.apply(lambda x: label2id[x])

sms_id_labels = sms.select(replace_labels_with_ids(sms.label).alias('label'), sms.text)
```

Split the data into training/testing and make the data available to the filesystem of the driver. One way to do so is to use the DBFS root volume or mount points.

```
(train, test) = sms_id_labels.persist().randomSplit([0.8, 0.2])
# Write the tables to DBFS.
train_dbfs_path = f"{tutorial_path}/sms_train"
test_dbfs_path = f"{tutorial_path}/sms_test"
train_df = train.write.parquet(train_dbfs_path, mode="overwrite")
test_df = test.write.parquet(test_dbfs_path, mode="overwrite")
```

These parquet files are now available to the filesystem on the driver using the mounted /dbfs path. You can specify paths to the parquet files using the 🤗 Datasets utilities to create a training and evaluation dataset.

```
from datasets import load_dataset
train_test = load_dataset("parquet", data_files={"train":f"/dbfs{train_dbfs_path}/*.par
```

The model expects tokenized input, rather than the text in the downloaded data. To ensure compatibility with the base model, use the AutoTokenizer loaded from the base model. HuggingFace datasets allows you to directly apply the tokenizer consistently to both the training and testing data.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(base_model)
def tokenize_function(examples):
    return tokenizer(examples["text"], padding=False, truncation=True)

train_test_tokenized = train_test.map(tokenize_function, batched=True)
```

Constructing a Trainer

The Trainer classes need the user to provide metrics, a base model, and training configuration. By default, the Trainer will compute and use loss as a metric, which isn't very interpretable. Below is an example of creating metrics function that will additionally compute accuracy during model training.

```
import numpy as np
import evaluate
metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

For text classification, use AutoModelForSequenceClassification to load a base model for text classification. Here you provide the number of classes and the label mappings.

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained(base_model, num_labels=2, lab
```

Finally, you must create the training configuration. The `TrainingArguments` class allows specification of the output directory, evaluation strategy, learning rate, and other parameters.

```
from transformers import TrainingArguments, Trainer
training_output_dir = "sms_trainer"
training_args = TrainingArguments(output_dir=training_output_dir, evaluation_strategy="
```

Using a data collator batches input in training and evaluation datasets. Using the `DataCollatorWithPadding` with defaults gives good baseline performance for text classification.

```
from transformers import DataCollatorWithPadding
data_collator = DataCollatorWithPadding(tokenizer)
```

With all of these parameters constructed, you can now create a Trainer.

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_test_dataset["train"],
    eval_dataset=train_test_dataset["test"],
    compute_metrics=compute_metrics,
    data_collator=data_collator,
)
```

Running training and logging the model

Hugging Face interfaces nicely with MLflow, automatically logging metrics during model training using the MLflowCallback. However, you must log the trained model yourself. Similar to the example above, Databricks recommends wrapping the trained model in a transformers pipeline and using MLflow's pyfunc `log_model` capabilities. To do so you will need a custom model class.

```
import mlflow
import torch

pipeline_artifact_name = "pipeline"
class TextClassificationPipelineModel(mlflow.pyfunc.PythonModel):

  def load_context(self, context):
    device = 0 if torch.cuda.is_available() else -1
    self.pipeline = pipeline("text-classification", context.artifacts[pipeline_artifact_

  def predict(self, context, model_input):
    texts = model_input[model_input.columns[0]].to_list()
    pipe = self.pipeline(texts, truncation=True, batch_size=8)
    labels = [prediction['label'] for prediction in pipe]
    return pd.Series(labels)
```

Wrap training in an mlflow run, constructing a transformers pipeline from tokenizer and the trained model, writing it to local disk. Finally, log the model to MLflow.

```
from transformers import pipeline

model_output_dir = "./sms_model"
pipeline_output_dir = "./sms_pipeline"
```

```
model_artifact_path = "sms_spam_model"

with mlflow.start_run() as run:
  trainer.train()
  trainer.save_model(model_output_dir)
  pipe = pipeline("text-classification", model=AutoModelForSequenceClassification.from_p
  pipe.save_pretrained(pipeline_output_dir)
  mlflow.pyfunc.log_model(artifacts={pipeline_artifact_name: pipeline_output_dir}, artif
```

Loading the model for inference is the same as loading the MLflow wrapped pre-trained model.

```
logged_model = "runs:/{run_id}/{model_artifact_path}".format(run_id=run.info.run_id, mod

# Load model as a Spark UDF. Override result_type if the model does not return double va
sms_spam_model_udf = mlflow.pyfunc.spark_udf(spark, model_uri=logged_model, result_type=

test = test.select(test.text, test.label, sms_spam_model_udf(test.text).alias("predictio
display(test)
```

Table +

	text	label	prediction
1	"Happy valentines day" I know its early but i have hundreds of handsomes and beauties to wish. So i thought to finish off aunties and uncles 1st...	0	ham
2	"Response" is one of d powerful weapon 2 occupy a place in others 'HEART'... So, always give response 2 who cares 4 U"... Gud night..swt dreams..take care	0	ham
3	"Wen u miss someone, the person is definitely special for u..... But if the person is so special, why to miss them, just Keep-in-touch" gdeve..	0	ham

This section demonstrated how you can directly use the Hugging Face Transformer Trainer APIs to fine-tune a model to a new text classification problem. You can fine-tune many more NLP models for a wide range of tasks, and the AutoModel classes for Natural Language Processing provide a great foundation.

Conclusion

This blog post demonstrated some best practices and showed how easy it is to get started using transformers for NLP tasks on Databricks.

The key points to recall for inference are:

- 🤗 Transformers pipelines make it easy to use transformers models,
- Repartition your data if needed to utilize the full cluster,
- You can tune your batch size for efficient use of GPUs,
- Spark assigns GPUs automatically on multi-machine GPU clusters,
- Pandas UDFs manage model broadcasting and batching data, and
- pipelines simplify logging transformers models to MLflow.

The key points to recall for single machine model training:

- 🤗 Transformers Trainers provide an accessible way to fine-tune models,
- Prepare your datasets on Spark, mapping any labels to ids if needed for the modeling task, leaving tokenization to 🤗 Transformers,
- Make the datasets available to the driver's filesystem,
- Tokenize the datasets using AutoTokenizer to load the right tokenizer for the model,
- Use Trainer to perform fine-tuning,
- Construct a pipeline from the tokenizer and fine-tuned model, and
- Log the model to MLflow using a custom model that wraps the pipeline.

Databricks continues to invest in simpler ways to scale model training and inference on Databricks. Stay tuned for improvements to data loading, distributed model training, and storing 🤗 Transformers pipelines and models as MLflow models.

To get started with these examples, import these notebooks for using pre-trained models

and fine-tuning.

Try Databricks for free

Get Started

Related posts

Rapid NLP Development With Databricks, Delta, and Transformers

September 9, 2022 by Marshall Carter in Engineering Blog

Free form text data can offer actionable insights unavailable in structured data fields. An insurance company may leverage its claims adjusters' notes to u...

GPU-accelerated Sentiment Analysis Using Pytorch and Huggingface on Databricks

October 28, 2021 by Srijith Rajamohan, Ph.D. in Engineering Blog

Sentiment analysis is commonly used to analyze the sentiment present within a body of text, which could range from a review, an email...

Extracting Oncology Insights From Real-World Clinical Data With NLP

September 22, 2021 by Amir Kermany, Moritz Steller, David Talby and Michael Sanky in Engineering Blog

Preview the solution accelerator notebooks referenced in this blog online or get started right away by downloading and importing the notebooks into your...

See all Engineering Blog posts