Fine-Tuning Transformers with MLflow for Enhanced Model Management

Welcome to our in-depth tutorial on fine-tuning Transformers models with enhanced management using MLflow.

Prerequisites

In order to get started with this lab, we will need to install following packages

To install the dependent packages simply run:

```
pip install evaluate transformers

pip3 install torch torchvision torchaudio

pip install tf-keras

pip install accelerate -U
```

What You Will Learn in This Lab

- Understand the process of fine-tuning a Transformers model.
- Learn to effectively log and manage the training cycle using MLflow.
- Master logging the trained model separately in MLflow.
- Gain insights into using the trained model for practical inference tasks.

Our approach will provide a holistic understanding of model fine-tuning and management, ensuring that you're well-equipped to handle similar tasks in your projects.

Emphasizing Fine-Tuning

Fine-tuning pre-trained models is a common practice in machine learning, especially in the field of NLP. It involves adjusting a pre-trained model to make it more suitable for a specific task. This process is essential as it allows the leveraging of pre-existing knowledge in the model, significantly improving performance on specific datasets or tasks.

Role of MLflow in Model Lifecycle

Integrating MLflow in this process is crucial for:

- **Training Cycle Logging**: Keeping a detailed log of the training cycle, including parameters, metrics, and intermediate results.
- **Model Logging and Management**: Separately logging the trained model, tracking its versions, and managing its lifecycle post-training.
- **Inference and Deployment**: Using the logged model for inference, ensuring easy transition from training to deployment.

```
# Disable tokenizers warnings when constructing pipelines
%env TOKENIZERS_PARALLELISM=false

import warnings

# Disable a few less-than-useful UserWarnings from setuptools and pydantic
warnings.filterwarnings("ignore", category=UserWarning)
```

Preparing the Dataset and Environment for Fine-Tuning

Key Steps in this Section

1. **Loading the Dataset**: Utilizing the `sms_spam` dataset for spam detection.
2. **Splitting the Dataset**: Dividing the dataset into training and test sets with an 80/20 distribution.
3. **Importing Necessary Libraries**: Including libraries like `evaluate`, `mlflow`, `numpy`, and essential components from the `transformers` library.

Before diving into the fine-tuning process, setting up our environment and preparing the dataset is crucial. This step involves loading the dataset, splitting it into training and testing sets, and initializing essential components of the Transformers library. These preparatory steps lay the groundwork for an efficient fine-tuning process.

This setup ensures that we have a solid foundation for fine-tuning our model, with all the necessary data and tools at our disposal. In the following Python code, we'll execute these steps to kickstart our model fine-tuning journey.

```python
import evaluate
import numpy as np
from datasets import load_dataset
from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
    pipeline,
)

import mlflow

# Load the "sms_spam" dataset.
sms_dataset = load_dataset("sms_spam")

# Split train/test by an 8/2 ratio.
sms_train_test = sms_dataset["train"].train_test_split(test_size=0.2)
train_dataset = sms_train_test["train"]
test_dataset = sms_train_test["test"]
```

Tokenization and Dataset Preparation

In the next code block, we tokenize our text data, preparing it for the fine-tuning process of our model.

With our dataset loaded and split, the next step is to prepare our text data for the model. This involves tokenizing the text, a crucial process in NLP where text is converted into a format that's understandable and usable by our model.

Tokenization Process

- **Loading the Tokenizer**: Using the `AutoTokenizer` from the `transformers` library for the `distilbert-base-uncased` model's tokenizer.
- **Defining the Tokenization Function**: Creating a function to tokenize text data, including padding and truncation.
- **Applying Tokenization to the Dataset**: Processing both the training and testing sets for model readiness.

Tokenization is a critical step in preparing text data for NLP tasks. It ensures that the data is in a format that the model can process, and by handling aspects like padding and truncation, it ensures consistency across our dataset, which is vital for training stability and model performance.

```python
# Load the tokenizer for "distilbert-base-uncased" model.
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")


def tokenize_function(examples):
    # Pad/truncate each text to 512 tokens. Enforcing the same shape
    # could make the training faster.
    return tokenizer(
        examples["sms"],
        padding="max_length",
        truncation=True,
        max_length=128,
    )


seed = 22

# Tokenize the train and test datasets
train_tokenized = train_dataset.map(tokenize_function)
train_tokenized = train_tokenized.remove_columns(["sms"]).shuffle(seed=seed)

test_tokenized = test_dataset.map(tokenize_function)
test_tokenized = test_tokenized.remove_columns(["sms"]).shuffle(seed=seed)
```

Model Initialization and Label Mapping

Next, we'll set up label mappings and initialize the model for our text classification task.

Having prepared our data, the next crucial step is to initialize our model and set up label mappings. This involves defining a clear relationship between the labels in our dataset and their corresponding representations in the model.

Setting Up Label Mappings

- **Defining Label Mappings**: Creating bi-directional mappings between integer labels and textual representations ("ham" and "spam").

Initializing the Model

- **Model Selection**: Choosing the `distilbert-base-uncased` model for its balance of performance and efficiency.
- **Model Configuration**: Configuring the model for sequence classification with the defined label mappings.

Proper model initialization and label mapping are key to ensuring that the model accurately understands and processes the task at hand. By explicitly defining these mappings and selecting an appropriate pre-trained model, we lay the groundwork for effective and efficient fine-tuning.

```python
# Set the mapping between int label and its meaning.
id2label = {0: "ham", 1: "spam"}
label2id = {"ham": 0, "spam": 1}

# Acquire the model from the Hugging Face Hub, providing label and id mappings so that
both we and the model can 'speak' the same language.
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
    num_labels=2,
    label2id=label2id,
    id2label=id2label,
)
```

Setting Up Evaluation Metrics

Next, we focus on defining and computing evaluation metrics to measure our model's performance accurately.

After initializing our model, the next critical step is to define how we'll evaluate its performance. Accurate evaluation is key to understanding how well our model is learning and performing on the task.

Choosing and Loading the Metric

- **Metric Selection**: Opting for 'accuracy' as the evaluation metric.
- **Loading the Metric**: Utilizing the `evaluate` library to load the 'accuracy' metric.

Defining the Metric Computation Function

- **Function for Metric Computation**: Creating a function, `compute_metrics`, for calculating accuracy during model evaluation.
- **Processing Predictions**: Handling logits and labels from predictions to compute accuracy.

Properly setting up evaluation metrics allows us to objectively measure the model's performance. By using standardized metrics, we can compare our model's performance against benchmarks or other models, ensuring that our fine-tuning process is effective and moving in the right direction.

```
# Define the target optimization metric
metric = evaluate.load("accuracy")


# Define a function for calculating our defined target optimization metric during training
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

Configuring the Training Environment

In this step, we're going to configure our Trainer, supplying important training configurations via the use of the `TrainingArguments` API.

With our model and metrics ready, the next important step is to configure the training environment. This involves setting up the training arguments and initializing the Trainer, a component that orchestrates the model training process.

Training Arguments Configuration

- **Defining the Output Directory**: We specify the `training_output_dir` where our model checkpoints will be saved during training. This helps in managing and storing model states at different stages of training.
- **Specifying Training Arguments**: We create an instance of `TrainingArguments` to define various parameters for training, such as the output directory, evaluation strategy, batch sizes for training and evaluation, logging frequency, and the number of training epochs. These parameters are critical for controlling how the model is trained and evaluated.

Initializing the Trainer

- **Creating the Trainer Instance**: We use the Trainer class from the Transformers library, providing it with our model, the previously defined training arguments, datasets for training and evaluation, and the function to compute metrics.
- **Role of the Trainer**: The Trainer handles all aspects of training and evaluating the model, including the execution of training loops, handling of data batching, and calling the compute metrics function. It simplifies the training process, making it more streamlined and efficient.

Importance of Proper Training Configuration

Setting up the training environment correctly is essential for effective model training. Proper configuration ensures that the model is trained under optimal conditions, leading to better performance and more reliable results.

In the following code block, we'll configure our training environment and initialize the Trainer, setting the stage for the actual training process.

```
# Checkpoints will be output to this `training_output_dir`.
training_output_dir = "/tmp/sms_trainer"
training_args = TrainingArguments(
    output_dir=training_output_dir,
    evaluation_strategy="epoch",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    logging_steps=8,
    num_train_epochs=3,
)

# Instantiate a `Trainer` instance that will be used to initiate a training run.
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_tokenized,
    eval_dataset=test_tokenized,
    compute_metrics=compute_metrics,
)
```

```
mlflow.set_tracking_uri("http://127.0.0.1:8081")
```

Integrating MLflow for Experiment Tracking

The final preparatory step before beginning the training process is to integrate MLflow for experiment tracking.

MLflow is a critical tool in our workflow, enabling us to log, monitor, and compare different runs of our model training.

Setting up the MLflow Experiment

- **Naming the Experiment**: We use `mlflow.set_experiment` to create a new experiment or assign the current run to an existing experiment. In this case, we name our experiment "Spam Classifier Training". This name should be descriptive and related to the task at hand, aiding in organizing and identifying experiments later.
- **Role of MLflow in Training**: By setting up an MLflow experiment, we can track various aspects of our model training, such as parameters, metrics, and outputs. This tracking is invaluable for comparing different models, tuning hyperparameters, and maintaining a record of our experiments.

Benefits of Experiment Tracking

Utilizing MLflow for experiment tracking offers several advantages:

- **Organization**: Keeps your training runs organized and easily accessible.
- **Comparability**: Allows for easy comparison of different training runs to understand the impact of changes in parameters or data.
- **Reproducibility**: Enhances the reproducibility of experiments by logging all necessary details.

With MLflow set up, we're now ready to begin the training process, keeping track of every important aspect along the way.

In the next code snippet, we'll set up our MLflow experiment for tracking the training of our spam classification model.

```
# Pick a name that you like and reflects the nature of the runs that you will be
recording to the experiment.
mlflow.set_experiment("Spam Classifier Training")
```

Starting the Training Process with MLflow

In this step, we initiate the fine-tuning training run, utilizing the native auto-logging functionality to record the parameters used and loss metrics calculated during the training process.

With our model, training arguments, and MLflow experiment set up, we are now ready to start the actual training process. This step involves initiating an MLflow run, which will encapsulate all the training activities and metrics.

Initiating the MLflow Run

- **Starting an MLflow Run**: We use `mlflow.start_run()` to begin a new MLflow run. This function creates a new run context, under which all the training operations and logging will occur.
- **Training the Model**: Inside the MLflow run context, we call `trainer.train()` to start training our model. This function will run the training loop, processing the data in batches, updating model parameters, and evaluating the model.

Monitoring the Training Progress

During training, the `Trainer` object will output logs that provide valuable insights into the training progress:

- **Loss**: Indicates the model's performance, with lower values signifying better performance.
- **Learning Rate**: Shows the current learning rate used during training.
- **Epoch Progress**: Displays the progress through the current epoch.

These logs are crucial for monitoring the model's learning process and making any necessary adjustments. By tracking these metrics within an MLflow run, we can maintain a comprehensive record of the training process, enhancing reproducibility and analysis.

In the next code block, we will start our MLflow run and begin training our model, closely observing the output to gauge the training progress.

```
with mlflow.start_run() as run:
    trainer.train()
```

Creating a Pipeline with the Fine-Tuned Model

In this section, we're going to create a pipeline that contains our fine-tuned model.

After completing the training process, our next step is to create a pipeline for inference using our fine-tuned model. This pipeline will enable us to easily make predictions with the model.

Setting Up the Inference Pipeline

- **Pipeline Creation**: We use the `pipeline` function from the Transformers library to create an inference pipeline. This pipeline is configured for the task of text classification.
- **Model Integration**: We integrate our fine-tuned model (`trainer.model`) into the pipeline. This ensures that the pipeline uses our newly trained model for inference.
- **Configuring the Pipeline**: We set the batch size and tokenizer in the pipeline configuration. Additionally, we specify the device type, which is crucial for performance considerations.

Device Configuration for Different Platforms

- **Apple Silicon (M1/M2) Devices**: For those using Apple Silicon (e.g., M1 or M2 chips), we set the device type to `"mps"` in the pipeline. This leverages Apple's Metal Performance Shaders for optimized performance on these devices.
- **Other Devices**: If you're using a device other than a MacBook Pro with Apple Silicon, you'll need to adjust the device setting to match your hardware (e.g., `"cuda"` for NVIDIA GPUs or `"cpu"` for CPU-only inference).

Importance of a Customized Pipeline

Creating a customized pipeline with our fine-tuned model allows for easy and efficient inference, tailored to our specific task and hardware. This step is vital in transitioning from model training to practical application.

In the following code block, we'll set up our pipeline with the fine-tuned model and configure it for our device.

```
tuned_pipeline = pipeline(
    task="text-classification",
    model=trainer.model,
    batch_size=8,
    tokenizer=tokenizer,
    device="cpu",
)
```

Validating the Fine-Tuned Model

In this next step, we're going to validate that our fine-tuning training was effective prior to logging the tuned model to our run.

Before finalizing our model by logging it to MLflow, it's crucial to validate its performance. This validation step ensures that the model meets our expectations and is ready for deployment.

Importance of Model Validation

- **Assessing Model Performance**: We need to evaluate the model's performance on realistic scenarios to ensure it behaves as expected. This helps in identifying any issues or shortcomings in the model before it is logged and potentially deployed.
- **Avoiding Costly Redo's**: Given the large size of Transformer models and the computational resources required for training, it's essential to validate the model beforehand. If a model doesn’t perform well, we wouldn't want to log the model, only to have to later delete the run and the logged artifacts.

Evaluating with a Test Query

- **Test Query**: We will pass a realistic test query to our tuned pipeline to see how the model performs. This query should be representative of the kind of input the model is expected to handle in a real-world scenario.
- **Observing the Output**: By analyzing the output of the model for this query, we can gauge its understanding and response to complex situations. This provides a practical insight into the model's capabilities post-fine-tuning.

Validating Before Logging to MLflow

- **Rationale**: The reason for this validation step is to ensure that the model we log to MLflow is of high quality and ready for further steps like deployment or sharing. Logging a poorly performing model would lead to unnecessary complications, especially considering the large size and complexity of these models.

After validating the model and ensuring satisfactory performance, we can confidently proceed to log it in MLflow, knowing it's ready for real-world applications.

In the next code block, we will run a test query through our fine-tuned model to evaluate its performance before proceeding to log it in MLflow.

```
# Perform a validation of our assembled pipeline that contains our fine-tuned model.
quick_check = (
    "I have a question regarding the project development timeline and allocated
resources; "
    "specifically, how certain are you that John and Ringo can work together on
writing this next song? "
    "Do we need to get Paul involved here, or do you truly believe, as you said, 'nah,
they got this'?"
)

tuned_pipeline(quick_check)
```

Model Configuration and Signature Inference

In this next step, we generate a signature for our pipeline in preparation for logging.

After validating our model's performance, the next critical step is to prepare it for logging to MLflow. This involves setting up the model's configuration and inferring its signature, which are essential aspects of the model management process.

Configuring the Model for MLflow

- **Setting Model Configuration**: We define a `model_config` dictionary to specify configuration parameters such as batch size and the device type (e.g., `"mps"` for Apple Silicon). This configuration is vital for ensuring that the model operates correctly in different environments.

Inferring the Model Signature

- **Purpose of Signature Inference**: The model signature defines the input and output schema of the model. Inferring this signature is crucial as it helps MLflow understand the data types and shapes that the model expects and produces.
- **Using mlflow.models.infer_signature**: We use this function to automatically infer the model signature. We provide sample input and output data to the function, which analyzes them to determine the appropriate schema.
- **Including Model Parameters**: Along with the input and output, we also include the `model_config` in the signature. This ensures that all relevant information about how the model should be run is captured.

Importance of Signature Inference

Inferring the signature is a key step in preparing the model for logging and future deployment. It ensures that anyone who uses the model later, either for further development or in production, has clear information about the expected data format, making the model more robust and user-friendly.

With the model configuration set and its signature inferred, we are now ready to log the model into MLflow. This will be our next step, ensuring our model is properly managed and ready for deployment.

```
# Define a set of parameters that we would like to be able to flexibly override at
inference time, along with their default values
model_config = {"batch_size": 8, "device": "mps"}

# Infer the model signature, including a representative input, the expected output,
and the parameters that we would like to be able to override at inference time.
signature = mlflow.models.infer_signature(
    ["This is a test!", "And this is also a test."],
    mlflow.transformers.generate_signature_output(
        tuned_pipeline, ["This is a test response!", "So is this."]
    ),
    params=model_config,
)
```

Logging the Fine-Tuned Model to MLflow

In this next section, we're going to log our validated pipeline to the training run.

With our model configuration and signature ready, the final step in our model training and validation process is to log the model to MLflow. This step is crucial for tracking and managing the model in a systematic way.

Accessing the existing Run used for training

- **Initiating MLflow Run**: We start a new run in MLflow using `mlflow.start_run()` . This new run is specifically for the purpose of logging the model, separate from the training run.

Logging the Model in MLflow

- **Using mlflow.transformers.log_model**: We log our fine-tuned model using this function. It's specially designed for logging models from the Transformers library, making the process streamlined and efficient.

- **Specifying Model Information**: We provide several pieces of information to the logging function:

    - **transformers_model**: The fine-tuned model pipeline.
    - **artifact_path**: The path where the model artifacts will be stored.
    - **signature**: The inferred signature of the model, which includes input and output schemas.
    - **input_example**: Sample inputs to give users an idea of what input the model expects.
    - **model_config**: The configuration parameters of the model.

Importance of Model Logging

Logging the model in MLflow serves multiple purposes:

- **Version Control**: It helps in keeping track of different versions of the model.
- **Model Management**: Facilitates the management of the model lifecycle, from training to deployment.
- **Reproducibility and Sharing**: Enhances reproducibility and makes it easier to share the model with others.

By logging our model in MLflow, we ensure that it is well-documented, versioned, and ready for future use, whether for further development or deployment.

```
# Log the pipeline to the existing training run
with mlflow.start_run(run_id=run.info.run_id):
    model_info = mlflow.transformers.log_model(
        transformers_model=tuned_pipeline,
        artifact_path="fine_tuned",
        signature=signature,
        input_example=["Pass in a string", "And have it mark as spam or not."],
        model_config=model_config,
    )
```

Loading and Testing the Model from MLflow

After logging our fine-tuned model to MLflow, we'll now load and test it.

Loading the Model from MLflow

- **Using mlflow.transformers.load_model**: We use this function to load the model stored in MLflow. This demonstrates how models can be retrieved and utilized post-training, ensuring they are accessible for future use.
- **Retrieving Model URI**: We use the `model_uri` obtained from logging the model to MLflow. This URI is the unique identifier for our logged model, allowing us to retrieve it accurately.

Testing the Model with Validation Text

- **Preparing Validation Text**: We use a creatively crafted text to test the model's performance. This text is designed to mimic a typical spam message, which is relevant to our model's training on spam classification.
- **Evaluating Model Output**: By passing this text through the loaded model, we can observe its performance and effectiveness in a practical scenario. This step is crucial to ensure that the model works as expected in real-world conditions.

Testing the model after loading it from MLflow is essential for several reasons:

- **Validation of Logging Process**: It confirms that the model was logged and loaded correctly.
- **Practical Performance Assessment**: Provides a real-world assessment of the model's performance, which is critical for deployment decisions.
- **Demonstrating End-to-End Workflow**: Showcases a complete workflow from training, logging, loading, to using the model, which is vital for understanding the entire model lifecycle.

In the next code block, we'll load our model from MLflow and test it with a validation text to assess its real-world performance.

```
# Load our saved model in the native transformers format
loaded = mlflow.transformers.load_model(model_uri=model_info.model_uri)

# Define a test example that we expect to be classified as spam
validation_text = (
    "Want to learn how to make MILLIONS with no effort? Click HERE now! See for
yourself! Guaranteed to make you instantly rich! "
    "Don't miss out you could be a winner!"
)

# validate the performance of our fine-tuning
loaded(validation_text)
```

Conclusion: Mastering Fine-Tuning and MLflow Integration

Congratulations on completing this comprehensive tutorial on fine-tuning a Transformers model and integrating it with MLflow! Let’s recap the essential skills and knowledge you've acquired through this journey.

Key Takeaways

1. **Fine-Tuning Transformers Models**: You’ve learned how to fine-tune a foundational model from the Transformers library. This process demonstrates the power of adapting advanced pre-trained models to specific tasks, tailoring their performance to meet unique requirements.

2. **Ease of Fine-Tuning**: We've seen firsthand how straightforward it is to fine-tune these advanced Large Language Models (LLMs). With the right tools and understanding, fine-tuning can significantly enhance a model’s performance on specific tasks.
3. **Specificity in Performance**: The ability to fine-tune LLMs opens up a world of possibilities, allowing us to create models that excel in particular domains or tasks. This specificity in performance is crucial in deploying models in real-world scenarios where specialized understanding is required.

Integrating MLflow with Transformers

1. **Tracking and Managing the Fine-Tuning Process**: A significant part of this tutorial was dedicated to using MLflow for experiment tracking, model logging, and management. You’ve learned how MLflow simplifies these aspects, making the machine learning workflow more manageable and efficient.
2. **Benefits of MLflow in Fine-Tuning**: MLflow plays a crucial role in ensuring reproducibility, managing model versions, and streamlining the deployment process. Its integration with the Transformers fine-tuning process demonstrates the potential for synergy between advanced model training techniques and lifecycle management tools.