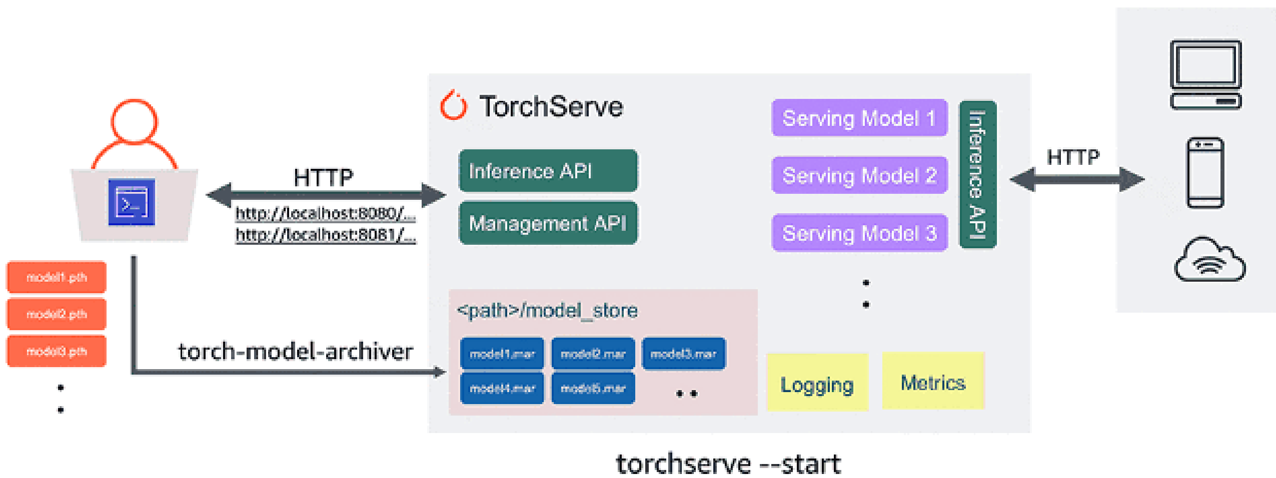
Deploying huggingface‘s BERT to production with pytorch/serve

MFreidank Follow
Apr 26, 2020 · 5 min read



TorchServe architecture. Image first found in an AWS blogpost on TorchServe.

TL;DR: pytorch/serve is a new awesome framework to serve torch models in production. This story teaches you how to use it for *huggingface/transformers* models like BERT.

Traditionally, serving *pytorch* models in production was **challenging**, as no standard framework used to be available for this task. This gap allowed its main competitor *tensorflow* to retain a strong grasp on many production systems, as it provided solid tooling for such deployments in its *tensorflow/serving* framework.

However, nowadays most new models and approaches tend to first be developed and made available in pytorch as researchers enjoy its flexibility for prototyping. This creates a gap between the state-of-the-art developed in research labs and the models typically deployed to production in most companies. In fast-moving fields such as natural language processing (NLP) this gap can be quite pronounced in spite of the efforts of frameworks like *huggingface/transformers* to provide model compatibility for both frameworks. In practice, development and adoption of new approaches tends to happen in *pytorch* first and by the time frameworks and productive systems have caught up and integrated a *tensorflow* version, new and more improved models have already deprecated it.

Most recently, the *pytorch* developers have released their new serving framework *pytorch/serve* to address these issues in a straightforward manner.

Introduction to TorchServe

> *TorchServe is a flexible and easy to use tool for serving PyTorch models.*

TorchServe (repository: pytorch/serve) is a recently (4 days ago at the time of writing) released framework developed by the *pytorch* developers to allow easy and efficient productionalization of trained pytorch models.

I recommend reading this AWS blog post for a thorough overview over *TorchServe*.

Serving Transformer models

huggingface/transformers can be considered a state-of-the-art framework for deep learning on text and has shown itself nimble enough to follow the rapid developments in this fast-moving space.

As this is a very popular framework with many active users (>25k stars on Github) from various different domains, it comes as no surprise that there is already interest (e.g. here, here and here) in serving BERT and other transformer models using TorchServe.

This story will explain how to serve your trained transformer model with TorchServe.

Prerequisites

To avoid unnecessarily bloating this post, I will make an assumption: you already have a trained BERT (or other transformers sentence classifier model) checkpoint.

If you don't, worry not: I will provide references to guides you can follow to get one of your own in no time.

Installing TorchServe

TorchServe provides an easy guide for its installation with pip, conda or docker. Currently, the installation is roughly comprised of two steps:

- Install Java JDK 11
- Install *torchserve* with its python dependencies

Please go through the installation guide linked above to ensure TorchServe is installed on your machine.

Training a huggingface BERT sentence classifier

Many tutorials on this exist and as I seriously doubt my ability to add to the existing corpus of knowledge on this topic, I simply give a few references I recommend:

- https://blog.rosetta.ai/learn-hugging-face-transformers-bert-with-pytorch-in-5-minutes-acee1e3be63d
- https://medium.com/@nikhil.utane/running-pytorch-transformers-on-custom-datasets-717fd9e10fe2

A simple way to get a trained BERT checkpoint is to use the *huggingface* GLUE example for sentence classification:

https://github.com/huggingface/transformers/blob/master/examples/run_glue.py

At the end of training, please ensure that you place trained model checkpoint (*pytorch.bin*), model configuration file (*config.json*) and tokenizer vocabulary file (*vocab.txt*) in the same directory. In what follows below, I will use a trained *"bert-base-uncased"* checkpoint and store it with its tokenizer vocabulary in a folder *"./bert_model"*.

For reference, mine looks like this:

```
serve on master [!?] via v3.7.4 (venv)
↓10% ❯ ls bert_model
bert.mar    config.json    pytorch_model.bin    special_tokens_map.json tokenizer_config.json    vocab.txt
```

Model checkpoint folder, a few files are optional

Defining a TorchServe handler for our BERT model

This is the salt: TorchServe uses the concept of **handlers** to define how requests are processed by a served model. A nice feature is that these handlers can be injected by client code when packaging models, allowing for a great deal of customization and flexibility.

Here is my template for a very basic TorchServe handler for BERT/transformer classifiers:

```
from abc import ABC
import json
import logging
import os

import torch
from transformers import AutoModelForSequenceClassification, AutoTokenizer

from ts.torch_handler.base_handler import BaseHandler

logger = logging.getLogger(__name__)


class TransformersClassifierHandler(BaseHandler, ABC):
    """
    Transformers text classifier handler class. This handler takes a text (string) and
    as input and returns the classification text based on the serialized transformers checkpoin
    """
    def __init__(self):
        super(TransformersClassifierHandler, self).__init__()
        self.initialized = False

    def initialize(self, ctx):
        self.manifest = ctx.manifest

        properties = ctx.system_properties
        model_dir = properties.get("model_dir")
        self.device = torch.device("cuda:" + str(properties.get("gpu_id")) if torch.cuda.is_ava

        # Read model serialize/pt file
        self.model = AutoModelForSequenceClassification.from_pretrained(model_dir)
        self.tokenizer = AutoTokenizer.from_pretrained(model_dir)

        self.model.to(self.device)
        self.model.eval()
```

```python
        self.model.eval()

        logger.debug('Transformer model from path {0} loaded successfully'.format(model_dir))

        # Read the mapping file, index to object name
        mapping_file_path = os.path.join(model_dir, "index_to_name.json")

        if os.path.isfile(mapping_file_path):
            with open(mapping_file_path) as f:
                self.mapping = json.load(f)
        else:
            logger.warning('Missing the index_to_name.json file. Inference output will not incl

        self.initialized = True

    def preprocess(self, data):
        """ Very basic preprocessing code - only tokenizes.
            Extend with your own preprocessing steps as needed.
        """
        text = data[0].get("data")
        if text is None:
            text = data[0].get("body")
        sentences = text.decode('utf-8')
        logger.info("Received text: '%s'", sentences)

        inputs = self.tokenizer.encode_plus(
            sentences,
            add_special_tokens=True,
            return_tensors="pt"
        )
        return inputs

    def inference(self, inputs):
        """
        Predict the class of a text using a trained transformer model.
        """
        # NOTE: This makes the assumption that your model expects text to be tokenized
        # with "input_ids" and "token_type_ids" - which is true for some popular transformer mo
        # If your transformer model expects different tokenization, adapt this code to suit
        # its expected input format.
        prediction = self.model(
            inputs['input_ids'].to(self.device),
            token_type_ids=inputs['token_type_ids'].to(self.device)
        )[0].argmax().item()
        logger.info("Model predicted: '%s'", prediction)
```

```

        if self.mapping:
            prediction = self.mapping[str(prediction)]

        return [prediction]

    def postprocess(self, inference_output):
        # TODO: Add any needed post-processing of the model predictions here
        return inference_output


_service = TransformersClassifierHandler()


def handle(data, context):
    try:
        if not _service.initialized:
            _service.initialize(context)

        if data is None:
            return None

        data = _service.preprocess(data)
        data = _service.inference(data)
        data = _service.postprocess(data)

        return data
    except Exception as e:
        raise e
```

transformers_classifier_torchserve_handler.py hosted with ♡ by GitHub view raw

A few things that my handler does not do, but yours might want to do:

- Custom pre-processing of the text (here we are just tokenizing)
- Any post-processing of the BERT predictions (these can be added in the *postprocess* function).
- Load an ensemble of models. One easy way to achieve this would be to load additional checkpoints in the *initialize* function and provide ensemble prediction logic in the *inference* function.

Converting the trained checkpoint to TorchServe MAR file

TorchServe uses a format called MAR (Model Archive) to package models and version them inside its model store. To make it accessible from TorchServe, we need to convert our trained BERT checkpoint to this format and attach our handler above.

The following command does the trick:

```
torch-model-archiver --model-name "bert" --version 1.0 --serialized-file ./bert_model/pytorch_model.bin --extra-files "./bert_model/config.json,./bert_model/vocab.txt" --handler "./transformers_classifier_torchserve_handler.py"
```

This command attaches the serialized checkpoint of your BERT model (*./bert_model/pytorch_model.bin*) to our new custom handler *transformers_classifier_torchserve_handler.py* described above and adds in extra files for the configuration and tokenizer vocabulary. It produces a file named *bert.mar* that can be understood by TorchServe.

Next, we can start a TorchServe server (by default it uses ports 8080 and 8081) for our BERT model with a model store that contains our freshly created MAR file:

```
mkdir model_store && mv bert.mar model_store && torchserve --start --model-store model_store --models bert=bert.mar
```

That's it! We can now query the model using the inference API:

```
curl -X POST http://127.0.0.1:8080/predictions/bert -T unhappy_sentiment.txt
```

In my case, *unhappy_sentiment.txt* is a file containing example sentences with a negative sentiment. My model correctly predicted a negative sentiment for this text (class 0).

Note that there are many additional interesting facilities available out of the box in the management API. For example we can easily get a list of all registered models, register a new model or new model version and switch served model versions for each model dynamically.

Happy coding and serving!

Transformers Pytorch Machine Learning Deep Learning Deployment