MUNI
FI

Optimizing the Inference of Transformer Based Models

Radoslav Sabol
xsabol@fi.muni.cz

NLP Centre, Faculty of Informatics, Masaryk University

April 25, 2023

Outline

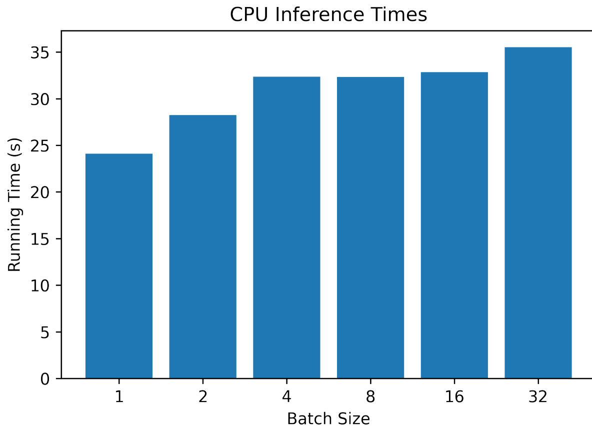
1. How to batch during inference
2. Quantization
3. Handling huge models
4. **Bonus:** PyTorch 2.0

Tweaks - Batching During Inference

- always helpful during training
  - not necessarily true for inference
- can be either 10x speedup or 5x slowdown depending on:
  1. hardware
  2. data
  3. used model

Batching - CPU

- if you are using a CPU, **never** batch

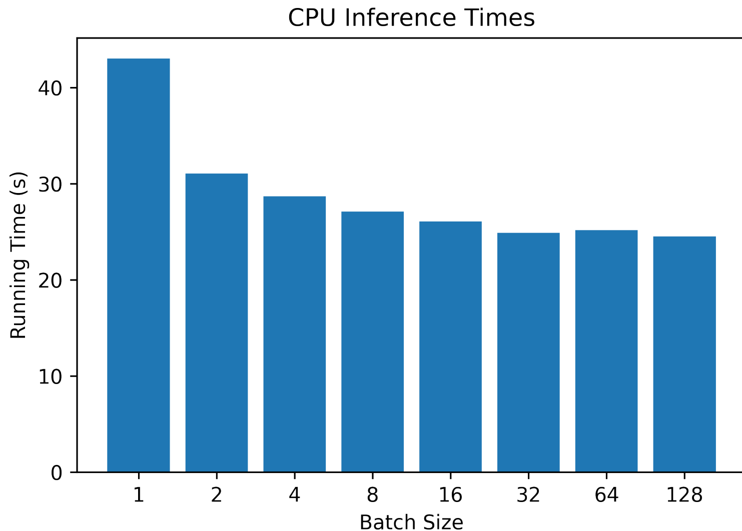


Batching - GPU

- if you are latency constrained (live product doing inference), **don't batch**
- if you have no clue about the size of the sequence length (*natural data*), by default [1] **don't batch**, then:
  - measure, try to tentatively increase batch size until the first OOM
  - **treat OOM** errors nicely as they are inevitable
- Measure performance on your load, with your hardware. **Measure, measure, and keep measuring.** Real numbers are the only way to go.

[1] https://huggingface.co/docs/transformers/main_classes/pipelines#pipeline-batching

GPU Batching - Diminishing Returns



Quantization - Basic Idea

- lowering the inference and memory costs by changing the weight and activation data types [2]
- usually at the cost of reduced accuracy
- typical conversions from `float32`:
  - `float16`, accumulation data type `float16`
  - `int8`, accumulation data type `int32`
- important to keep hardware capabilities in mind

[2] https://huggingface.co/docs/optimum/concept_guides/quantization

Quantization - Calibration

- post-training **static quantization**
  - both weights and activations are quantized in advance
  - needs a *calibration dataset* to adjust the activations
- post-training **dynamic quantization**
  - weights quantized in advance, activations quantized *on the fly*
- **quantization-aware** training
  - performed at training time
  - simulates the error induced by quantization to let the model adapt to it

Quantization Example - ONNX Runtime

```
from optimum.onnxruntime import ORTQuantizer, ORTModelForSequenceClassification
from optimum.onnxruntime.configuration import AutoQuantizationConfig
```

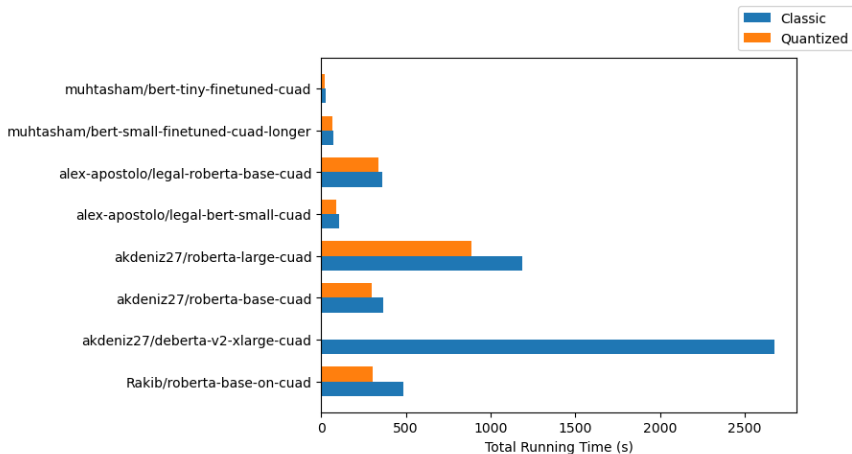
```
# Load PyTorch model and convert to ONNX
onnx_model = ORTModelForSequenceClassification.from_pretrained("UWB-AIR/Czert-A-base-uncased",
                                                                export=True)

# Create quantizer
quantizer = ORTQuantizer.from_pretrained(onnx_model)

# Define the quantization strategy by creating the appropriate configuration
dqconfig = AutoQuantizationConfig.avx512_vnni(is_static=False,
                                              per_channel=False)
```

```
# Quantize the model
model_quantized_path = quantizer.quantize(
    save_dir="path/to/output/model",
    quantization_config=dqconfig,
)
```

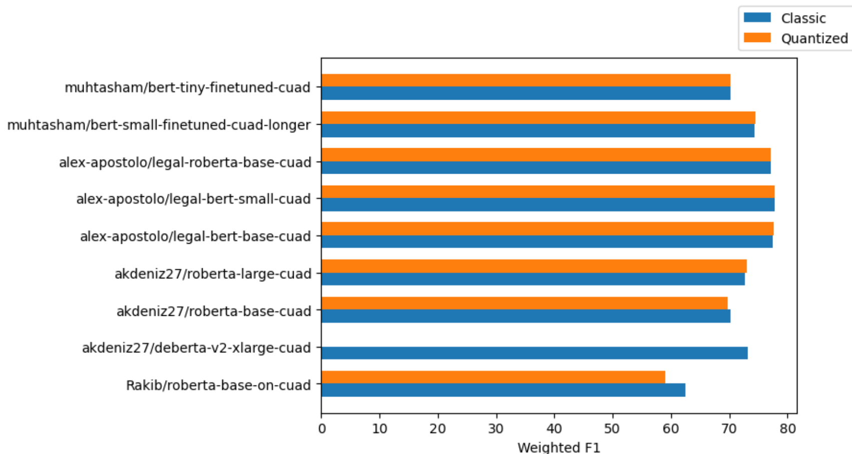
Use Case - Contract Understanding

- extractive task for legal contracts
- **CUAD** - Contract Understanding Atticus Dataset
  - a corpus of 510 commercial legal contracts
  - 41 categories with overall 14,000 annotations
  - *Document Name*, *Parties*, *Expiration Date*, *Solicit of Employees*, ...
- translated as a **question answering task** in the original paper
- **RoBerta** large, extra large, and **DeBerta** extra large report the best experimental results

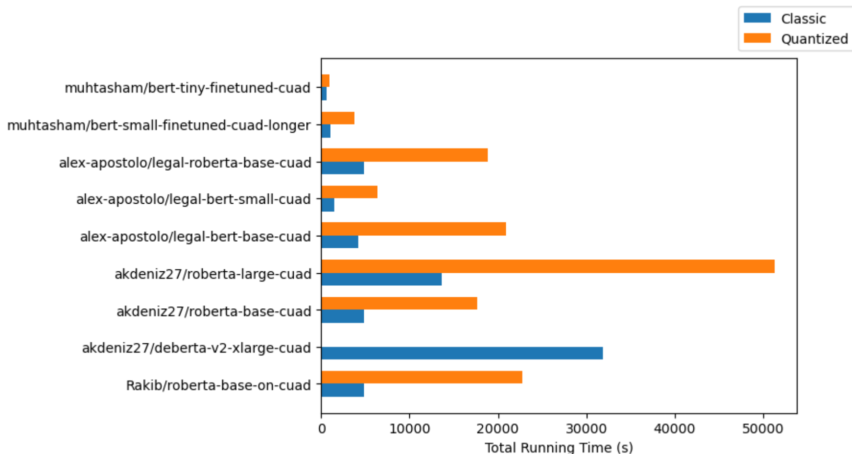
Post-Quantization Results - Inference Time (CPU)



Post-Quantization Results - Weighted F1



Post-Quantization Results - GPU Runtime (Oops)



Handling Huge Models - Intro

```
import torch

my_model = ModelClass(...)
state_dict = torch.load(checkpoint_file)
my_model.load_state_dict(state_dict)
```

1. Instantiate a model with **randomly initialized weights**
2. Load the model weights **into the main memory**
3. **Replace** the randomly initialized weights with the trained ones

Instantiate a Huge Model using `accelerate`

```
from accelerate import init_empty_weights

with init_empty_weights():
    my_model = ModelClass(...)
```

- allows to instantiate a model **without using any RAM**
- but what if the weights **do not fit into the RAM** anyway?

Using Model Shards

- **main idea:** use multiple memory devices (RAM, GPU VRAM, even disk) to load your model
- requires your model to be split into several files (also called **shards**)
- `index.json` contains the required mapping [3]

```
{
  "linear1.weight": "first_state_dict.bin",
  "linear1.bias": "first_state_dict.bin",
  "linear2.weight": "second_state_dict.bin",
  "linear2.bias": "second_state_dict.bin"
}
```

[3] https://huggingface.co/docs/accelerate/usage_guides/big_modeling

Loading Huge Model Weights Using `accelerate`

```
from accelerate import load_checkpoint_and_dispatch

model = load_checkpoint_and_dispatch(
    model, "sharded-gpt-j-6B", device_map="auto", no_split_module_classes=["GPTJBlock"]
)
```

1. uses up **all the available GPU memory**
2. if 1.) is full, uses up **all the CPU RAM**
3. if 2.) is full, the remaining weights are **stored inside of hard drive** as memory-mapped tensors

How Does Inference Work?

1. at each layer, the inputs **are put on the right device**
2. for the weights offloaded on the CPU, **they are put on a GPU just before the forward pass**, and cleaned up just after
3. for the weights offloaded on the hard drive, **they are loaded in RAM then put on a GPU just before the forward pass**, and cleaned up just after

Possible Limitations:

- at least one GPU is required
- GPU offloading is naive and not optimized
- overall, it is an experimental API

BONUS: Pytorch 2.0 - torch.compile()

- **.compile()** method allows to translate the model into TorchScript
- the **performance gain** is claimed to be between **30%-200%** [4] for HuggingFace models
- experimental result with **RoBerta Base** on 100 dummy examples:
  - *Before Compilation: 23.38s*
  - **After Compilation: 16.93s - a 72% increase**

[4] `https://pytorch.org/get-started/pytorch-2.0/#accelerating-hugging-face-and-timm-models-with-pytorch-20`

MUNI
FACULTY
OF INFORMATICS