Today’s tutorial

- How to make inference/training fast?
- How to deal with limited hardware?
- How to get more compute for bigger projects?

How to make inference fast?

- Flash Attention
- Quantization
- BetterTransformer
- Optimum library

Flash Attention

- Time and memory complexity of self-attention is quadratic w.r.t. sequence length
  - The longer the sequence, the more time inference/training takes
- Flash attention introduces additional parallelization over the sequence length to mitigate this
- Smarter partitioning of the workload between GPU threads -> less reads and writes from memory
- Exact algorithm, extension to block-sparse is non-exact
- Only usable for dtypes fp16 and bf16 and on NVIDIA GPUs
- Experimental, only usable without padding tokens

```
pip install flash-attn --no-build-isolation
```

To enable FlashAttention-2, add the `use_flash_attention_2` parameter to [from_pretrained()]():

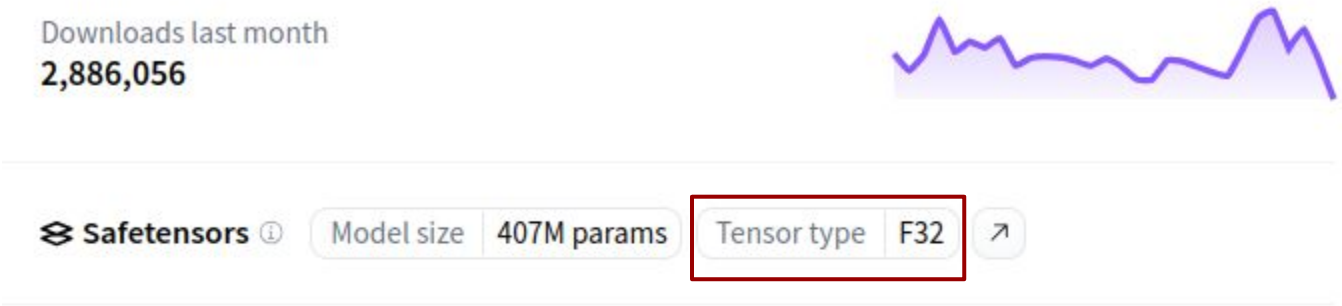
```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, LlamaForCausalLM

model_id = "tiiuae/falcon-7b"
tokenizer = AutoTokenizer.from_pretrained(model_id)

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.bfloat16,
    use_flash_attention_2=True,
)
```

Quantization

- Computers are bad at floating point arithmetic
- We need to choose a certain level of precision whenever we deal with floating point numbers
- Neural networks are just large collections of floating point numbers
- If we reduce the precision of the model, we need much less memory and speed up matrix multiplication considerably
- We may lose some accuracy → applicability dependent on use case

Quantization

-



Quantization: Common Data Types

Data Type	Accumulates to
float16	float16
bfloat16	float32
int16	int32
int8	int32

Accumulation Data Types

- Suppose we want to add two int8 values, A=127, B=127:
- A + B = C
- 127 is the maximum value representable in int8 →accumulate to bigger data type to prevent large precision losses

Quantization with float16

- Same representation as float32 → straightforward quantization
- Questions to answer beforehand:
  - Do operations have a float16 implementation?
  - Does the hardware support operations in float16?
  - Is my operation sensitive to lower precision?
- Some operations in ML are sensitive to changes in very low values (e.g. layer norm)
- With larger scale generative LLMs, quantization to float16 is pretty standard

Quantization to int8

- 256 values representable in int8
- Find best way to project range of float32 value to int8 space
- Consider float x in range [a, b], using affine quantization scheme:

$$X = s*(x_q - Z)$$

- x_q= quantized int8 value associated to x
- S= defined as (float_max-float_min)/(int_max-int_min)
- Z=zeropoint, int8 value corresponding to 0 in float32 realm
    - We do not want to make errors when mapping zero!

Quantization to int8

- Compute x_q:

$$x_q = \mathrm{round}(x/S + Z)$$

- All values outside the [a,b] range are clipped to the closest representable value:

$$x_q = \mathrm{clip}(\mathrm{round}(x/S + Z), \mathrm{round}(a/S + Z), \mathrm{round}(b/S + Z))$$

Quantization in Huggingface Transformers

```
# these versions support 8-bit and 4-bit
pip install bitsandbytes>=0.39.0 accelerate>=0.20.0
```

```python
from transformers import AutoModelForCausalLM

model_name = "bigscience/bloom-2b5"
model_4bit = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto", load_in_4bit=True)
```

```python
from transformers import AutoModelForCausalLM

model_name = "bigscience/bloom-2b5"
model_8bit = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto", load_in_8bit=True)
```

BetterTransformer

- Optimized execution of Huggingface Transformer functions:
  - Fusing multiple sequential operations into single “kernel” to reduce number of computations performed
  - More performant handling of padding tokens

```
python -m pip install optimum
```

```
model = model.to_bettertransformer()
```

```
model = model.reverse_bettertransformer()
model.save_pretrained("saved_model")
```

Optimum

- Similar optimizations to BetterTransformer
- Only for nvidia GPUs

```
from optimum.onnxruntime import ORTModelForSequenceClassification

ort_model = ORTModelForSequenceClassification.from_pretrained(
  "distilbert-base-uncased-finetuned-sst-2-english",
  export=True,
  provider="CUDAExecutionProvider",
)
```

How to make training fast?

- Either speed up computation, optimize memory utilization or both

Method/tool	Improves training speed	Optimizes memory utilization
Batch size choice	Yes	Yes
Gradient accumulation	No	Yes
Gradient checkpointing	No	Yes
Mixed precision training	Yes	(No)
Optimizer choice	Yes	Yes
Data preloading	Yes	No
DeepSpeed Zero	No	Yes
torch.compile	Yes	No

Batch Size Choice

- Always the starting point
- Batch size too large: OOM error
- Batch size too small: inefficient memory usage, longer training time, more cost
- Has to be of size 2^N
- Smaller batch sizes usually converge faster than large batch sizes

Gradient Accumulation

- Computes gradient at smaller steps than for the entire batch at once
- Iteratively compute gradients by performing forward/backward passes through the model
- Accumulate gradients during process
- Once enough gradients have been accumulated, perform model's optimization
- Enables larger batch size than would usually be possible
- But: training process becomes slower → balance gradient accumulation steps and batch size

Gradient Accumulation and Batch Size in Transformers

```
training_args = TrainingArguments(per_device_train_batch_size=1, gradient_accumulation_steps=4,
```

Gradient Checkpointing

- Saving all activations from forward pass result in large memory overhead
- Disregarding and recomputing activations during the backward pass results in large computational overhead
- Gradient Checkpointing strategically select the activations to keep, resulting in only a fraction of the activations to be recomputed

```
training_args = TrainingArguments(
    per_device_train_batch_size=1, gradient_accumulation_steps=4, gradient_checkpointing=True,
)
```

Mixed Precision Training

- Reduces precision of certain variables to achieve computational speed up
- Decreases training time, but can also increase GPU memory utilization, because two representations of the model have to be saved
- Data type to be used dependent on hardware support

```
training_args = TrainingArguments(per_device_train_batch_size=4, fp16=True,
```

```
training_args = TrainingArguments(bf16=True, **default_args)
```

Optimizer Choice

- Optimizes a function (our neural network) w.r.t. an objective function (our loss function)
- To do this, the optimizer has to make changes to all of the weights and biases
- Large impact on training performance/convergence

Optimizer Choice

- Default is Adam(W) which stores rolling averages of the previous gradients
- This results in significant increase of memory usage
- Out of the box alternatives for huggingface transformers:

Optimizer	GPU Memory for 3B Model
AdamW	24GB
Adafactor	12GB
8bit BNB quantized	6GB

Adafactor

- Aggregates the rolling averages Adam uses by summing rolling average row and column wise
- Possibly slower convergence
- However, significant reductions in memory throughput can be achieved

```
training_args = TrainingArguments(per_device_train_batch_size=4, optim="adafactor",
```

8-bit Adam

- Quantizes optimizer states instead of aggregating them

```
training_args = TrainingArguments(per_device_train_batch_size=4, optim="adamw_bnb_8bit",
```

Sharding

- Usable for inference and training
- Divide layers of neural network into single files
- Load files one by one on the GPU during training or inference
- Also used for large distributed LLM applications
- Multiple ways of doing this with huggingface

Two Cases

- I have one GPU and its too small
- I have multiple GPUs and want to distribute inference/training
    - https://huggingface.co/docs/accelerate/concept_guides/big_model_inference

Other techniques

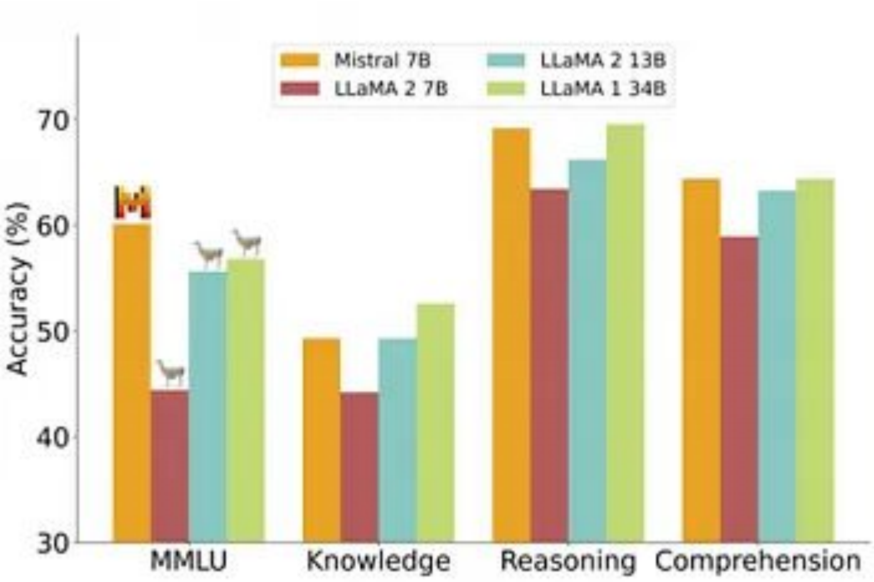
- Data preloading: makes sure the GPU loads as much data as it can possibly handle
  - Set dataloader_num_workers in TrainingArguments to higher value if GPU utilization is far from 100%
- Torch.compile: uses some low-level computational graph magic to optimize training

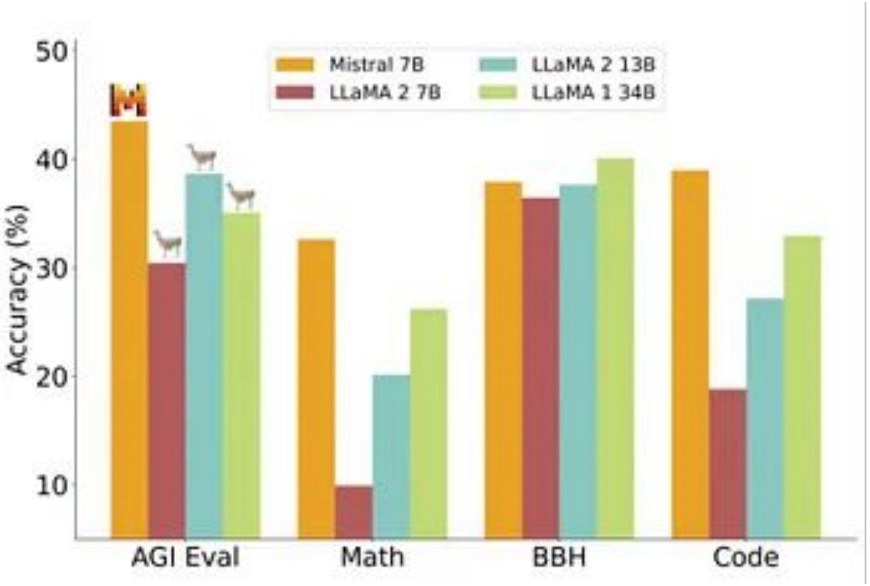
```
training_args = TrainingArguments(torch_compile=True, **default_args)
```

General Advice

- Working with LLMs, especially training, is a applying lot of trial and error while trying to keep your nerves
- Despite all the fancy optimization techniques, more compute is almost always better
- Fine-tuned models can match much larger models on their respective task in a lot of cases
- Guardrails can make models significantly worse

Censored models getting outperformed?





General Advice

- Cloud Providers may have some starting credits to get you hooked ($300 on GCP → great for bigger projects, thesis)
- Being able to access servers remotely and letting your stuff run without having to keep a local machine on is a major advantage

Compute for Assignment 2

- We will rent a 32GB RAM Machine for you to use
- You will have your own user accounts and SSH access
- You will be able to run python scripts/install python packages
- This is (kind of) an experiment, please do not crash the server, distribute your SSH keys anywhere, etc.
- Windows users may install WSL2 to log onto the machine