New issue

Jump to bottom

Possible Bug with KV Caching in Llama (original) model #25420

Closed 2 of 4 tasks **maximkha** opened this issue on Aug 10, 2023 · 23 comments

maximkha commented on Aug 10, 2023 • edited

System Info

transformers==4.31.0

- huggingface_hub version: 0.15.1
- Platform: Linux-5.15.0-78-generic-x86_64-with-glibc2.35
- Python version: 3.10.12
- Running in iPython ?: No
- Running in notebook ?: No
- Running in Google Colab ?: No
- Token path ?: /u/k/h/khanov/.cache/huggingface/token
- Has saved token ?: False
- Configured git credential helpers:
- FastAI: N/A
- Tensorflow: N/A
- Torch: 2.0.0
- Jinja2: 3.0.3
- Graphviz: N/A
- Pydot: N/A
- Pillow: 9.0.1
- hf_transfer: N/A
- gradio: N/A
- numpy: 1.24.2
- ENDPOINT: https://huggingface.co
- HUGGINGFACE_HUB_CACHE: /u/k/h/khanov/.cache/huggingface/hub
- HUGGINGFACE_ASSETS_CACHE: /u/k/h/khanov/.cache/huggingface/assets
- HF_TOKEN_PATH: /u/k/h/khanov/.cache/huggingface/token
- HF_HUB_OFFLINE: False
- HF_HUB_DISABLE_TELEMETRY: False

- HF_HUB_DISABLE_PROGRESS_BARS: None
- HF_HUB_DISABLE_SYMLINKS_WARNING: False
- HF_HUB_DISABLE_EXPERIMENTAL_WARNING: False
- HF_HUB_DISABLE_IMPLICIT_TOKEN: False
- HF_HUB_ENABLE_HF_TRANSFER: False

Who can help?

@ArthurZucker, **@younesbelkada**

Information

- [ ] The official example scripts
- [x] My own modified scripts

Tasks

- [ ] An officially supported task in the `examples` folder (such as GLUE/SQuAD, ...)
- [x] My own task or dataset (give details below)

Reproduction

I was working on a custom decoding method, however, I found a deviation from greedy search when using KV caching.

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

MODEL_PATH = "/nobackup-fast/khanov/llama-7b" # "huggyllama/llama-7b"
GEN_DEV = "cuda:0"

tokenizer = AutoTokenizer.from_pretrained(MODEL_PATH)
model = AutoModelForCausalLM.from_pretrained(MODEL_PATH, torch_dtype=torch.bfloat16).to(GE

def get_input_ids(prompt: str) -> torch.Tensor:
    global model, tokenizer
    tokens = tokenizer(prompt, return_tensors="pt").input_ids.to(GEN_DEV)
    return tokens
def tokens_to_text(tokens: torch.Tensor):
    return tokenizer.batch_decode(tokens, skip_special_tokens=True)


PROMPT = "This is a " # this is just a test prompt

# greedy decoding without caching
tokens = get_input_ids(PROMPT)
for _ in tqdm(range(40)):
    with torch.no_grad():
        mout = model(tokens)
    tokens = torch.hstack((tokens, torch.argmax(mout.logits[0, -1]).unsqueeze(0).unsqueeze
```

```
without_cache = tokens_to_text(tokens)[0]
print(f"{without_cache=}")

# greedy decoding WITH caching
tokens = get_input_ids(PROMPT)
cached = None
for _ in tqdm(range(40)):
    with torch.no_grad():
        if cached is None:
            mout = model(tokens, output_hidden_states=True, use_cache=True)
            cached = mout.past_key_values
        else:
            mout = model(tokens, past_key_values=cached, use_cache=True, output_hidden_st
            cached = mout.past_key_values
    tokens = torch.hstack((tokens, torch.argmax(mout.logits[0, -1]).unsqueeze(0).unsqueez

with_cache = tokens_to_text(tokens)[0]
print(f"{with_cache=}")

# normal greedy search with HF Generate implementation
tokens = get_input_ids(PROMPT)
tokens = model.generate(tokens, num_return_sequences=1, max_new_tokens=40)
generate_output = tokens_to_text(tokens)[0]
print(f"{generate_output=}")

# this matches exactly
assert without_cache == generate_output

# this does not!
assert without_cache == with_cache
```

Expected behavior

I was expecting the results to not change when using the past_key_values kwarg, however, when passing past_key_values, the model assigned different logits to the tokens. This deviates from the model.generate behavior too. This is possibly related to #18809, and #21080.

1

sgugger commented on Aug 10, 2023 Collaborator

cc **@ArthurZucker** and **@gante**

ArthurZucker commented on Aug 10, 2023 Collaborator

Hey! It seems like the problème is from your custom code rather than the `Llama` past key values mechanism as `generate()` uses past key values by default, unless your generation config has `generation_config.use_cache = False` .

I don't know exactly what is wrong with your custom greedy decoding, but would probably say that you are not feeding the positional ID information that is automatically create in `prepare_inputs_for_generation` used in the generation.

gante commented on Aug 10, 2023 Member

Hi **@maximkha** 👋

Thank you for raising this issue! Sadly, our bandwidth is limited, so our capacity to dive into custom code for which a solution already exists is limited :)

As **@ArthurZucker** wrote, you are missing the position IDs, which may have a significant impact on the output. The same is true for the attention mask. Our modeling code makes its best effort to infer these two inputs when they are missing, but it fails in some cases.

My suggestion would be to introduce a `breakpoint()` in `generate`, before the model forward pass, and compare the inputs that go into the model :)

❤️ 1

maximkha commented on Aug 10, 2023 Author

Thanks so so much! Turns out the `prepare_inputs_for_generation` function prepared the positional ID information as you said and after adding that in, the results exactly match! I'll go ahead and close this!

❤️ 4

maximkha closed this as completed on Aug 10, 2023

maximkha commented on Aug 11, 2023 Author

Actually, I'm currently experiencing another issue when using this for Llama for sequential classification. It seems that even when I use prepare_inputs_for_generation, I'm getting values that disagree. I'm not exactly sure what the culprit is, but I have been using the appropriate _reorder_cache function.

maximkha reopened this on Aug 11, 2023

ArthurZucker commented on Aug 11, 2023 Collaborator

maximkha commented on Aug 17, 2023 • edited Author

Hey **@gante**, this isn't an issue with generate specifically, it seems to be that when using the key_value_caching and bfloat16, the logits are significantly different from the non-cached version (some precision loss I'm assuming). There is no generation involved, just using key_values with bfloat16 skews the logits.

I'm not sure if this level of precision loss is to be expected or not.

TL;DR this is a problem with precision + caching, not generate.

Also, sorry for all the messages, but this level of precision loss is impacting my results.

csarron mentioned this issue on Sep 4, 2023

Batch Decoding of LMs will cause different outputs with different batch size #25921

Open

4 tasks

huggingface deleted a comment from **github-actions** bot on Sep 13, 2023

ArthurZucker mentioned this issue on Sep 22, 2023

usage of past_key_values produces different output than the whole sequence at once #26344

Closed

4 tasks

huggingface deleted a comment from **github-actions** bot on Oct 13, 2023

gante commented on Oct 23, 2023 • edited Member

Hey folks 👋 I've done a deep dive on this issue, and I will link related issues to this comment that attempts to summarize findings.

cc:

- **@maximkha**, who has been rightly pursuing us to figure out this mismatch;
- **@ArthurZucker**, who has been seeing other issues like this

TL;DR

Using KV caches, assisted generation, left-padding, and batching will change the `logits`. This happens in most, if not all models at all precisions, but it is almost imperceptible in FP32. With 16 bits, the difference becomes non-negligible. The model was not trained with KV caches or left-padding, so this is introducing a distribution shift -- it's part of the cost of using a lower precision and other related optimizations. The effect is more visible when `do_sample=True`, as greedy decoding (`do_sample=False`) often selects the same token despite the differences.

Why does this happen?

A key operation in neural networks is matrix multiplication, where values are multiplied and accumulated. Unless you have infinite precision, different implementations or different shapes (e.g. crop a few rows of the first matrix) may produce different outputs, as the intermediary calculations must remain in the specified precision and are subject to rounding. For instance, our models with TF and JAX implementations never have the exact output as the PyTorch implementation, they tend to differ by a maximum `1e-5` at FP32 for the same exact input, due to minor differences in the frameworks' inner implementations.

When using KV caches (and, in some models, left-padding), we are changing the input shape to some matrix multiplication operations. For instance, in Llama, when you apply the linear projection to obtain the QKV for the attention layer, the input shape will be different depending on whether you're using left-padding and/or KV caches. Therefore, the output of these operations may be different, and these tiny differences build up across layers and across generated tokens, especially at lower resolutions.

If you place a breakpoint inside the model, and see what happens with and without KV caches, you'll see:

1. During prefill (parsing the input prompt), the KV caches and the hidden states are exactly the same, as the inputs contain the same values and shapes.
2. When generating one token at a time, you will see a divergence happening in the hidden states and the QKV after operations like linear layers.

How big is this difference?

Let's do a simple experiment: for the same set of inputs, let's measure the hidden states' and the logits' maximum difference for the first generated token, with and without KV caching. I created the following test script from an example given in a related issue (#26344). TL;DR it averages the maximum value for the variables described above over 1000 runs:

▼ Test script

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch
from datasets import load_dataset
from tqdm import tqdm


TOTAL_NUM_SAMPLES = 1000
INPUT_LEN = 64

model_name = "codellama/CodeLlama-7b-hf"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name, torch_dtype=torch.float16, low_cpu_mem_usage=True, device_map="auto"
)
```

```
# model = AutoModelForCausalLM.from_pretrained(model_name)

ds = load_dataset("bigcode/the-stack", data_dir="data/python", split="train", streaming=T
ds_iterator = iter(ds.take(TOTAL_NUM_SAMPLES))
max_diffs = {}
for _ in tqdm(range(TOTAL_NUM_SAMPLES)):
    next_data = next(ds_iterator)["content"]
    all_input_ids = tokenizer(
        [next_data], return_tensors="pt", max_length=INPUT_LEN, truncation=True
    ).input_ids.to(model.device)

    # process the whole sequence
    all_outputs = model(all_input_ids, output_hidden_states=True, return_dict=True)
    # get logits for the last token
    last_token_logits = all_outputs.logits[0][-1:]

    # process the sequence except the last token
    kv = model(all_input_ids[:, :-1]).past_key_values
    # input only the last token with previous kv_cache
    new_output = model(all_input_ids[:, -1:], past_key_values=kv, output_hidden_states=Tr
    # extract the last token logits
    new_last_token_logits = new_output.logits[0][-1:]

    for layer_idx in range(len(all_outputs.hidden_states)):
        max_diff = torch.abs(
            all_outputs.hidden_states[layer_idx][:, -1, :] - new_output.hidden_states[lay
        ).max()
        max_diffs.setdefault(f"layer {layer_idx}", []).append(max_diff.cpu().item())

    # theese two distributions should be equal, but they are not.
    max_diffs.setdefault("logits", []).append(torch.abs(last_token_logits - new_last_toke

for key, value in max_diffs.items():
    print(f"{key}: {sum(value) / len(value)}")
```

Here are the results I got for `CodeLlama` (which uses the same code as Llama and Llama2), with `GPT2` in FP16 for comparison:

▼ Llama, FP32

```
layer 0: 0.0
layer 1: 4.981691017746925e-07
layer 2: 2.5094859302043914e-06
layer 3: 2.6547210291028024e-06
layer 4: 2.8776237741112707e-06
layer 5: 3.2249726355075836e-06
layer 6: 3.5362401977181435e-06
layer 7: 3.871295601129532e-06
layer 8: 4.376612603664398e-06
layer 9: 4.956845194101334e-06
layer 10: 5.649109371006489e-06
layer 11: 6.595022976398468e-06
layer 12: 6.92228227853775e-06
layer 13: 7.3333755135536194e-06
layer 14: 7.672600448131561e-06
layer 15: 8.006669580936431e-06
layer 16: 8.94695520401001e-06
```

```
layer 17: 9.912904351949691e-06
layer 18: 1.0702745988965035e-05
layer 19: 1.2084681540727615e-05
layer 20: 1.3510849326848984e-05
layer 21: 1.4993250370025634e-05
layer 22: 1.5627190470695495e-05
layer 23: 1.9214315339922905e-05
layer 24: 1.9937701523303985e-05
layer 25: 2.1439727395772934e-05
layer 26: 2.1951720118522644e-05
layer 27: 2.3870080709457398e-05
layer 28: 2.5171246379613875e-05
layer 29: 2.6149518787860 87e-05
layer 30: 2.8442054986953734e-05
layer 31: 3.540612757205963e-05
layer 32: 1.0248859878629445e-05
logits: 1.5035882592201234e-05
```

▼ Llama, FP16 (the expected 16-bit format to use)

```
layer 0: 0.0
layer 1: 0.000550079345703125
layer 2: 0.00298907470703125
layer 3: 0.0033966217041015625
layer 4: 0.0039486083984375
layer 5: 0.00466839599609375
layer 6: 0.00533612060546875
layer 7: 0.00594580078125
layer 8: 0.006715240478515625
layer 9: 0.00763134765625
layer 10: 0.008845230102539063
layer 11: 0.01030645751953125
layer 12: 0.011149169921875
layer 13: 0.011803375244140626
layer 14: 0.01296966552734375
layer 15: 0.013913818359375
layer 16: 0.015769287109375
layer 17: 0.01764404296875
layer 18: 0.01888623046875
layer 19: 0.02110791015625
layer 20: 0.023257568359375
layer 21: 0.025254150390625
layer 22: 0.02687548828125
layer 23: 0.03120947265625
layer 24: 0.032493896484375
layer 25: 0.03505859375
layer 26: 0.037328369140625
layer 27: 0.0409736328125
layer 28: 0.0434375
layer 29: 0.0456640625
layer 30: 0.04978125
layer 31: 0.060069580078125
layer 32: 0.015433685302734375
logits: 0.016572296142578127
```

▼ Llama, BF16 (the wrong 16-bit format to use with Llama)

```
layer 0: 0.0
layer 1: 0.00433740234375
layer 2: 0.03967041015625
layer 3: 0.0434326171875
layer 4: 0.047635498046875
layer 5: 0.0537783203125
layer 6: 0.058983642578125
layer 7: 0.0638212890625
layer 8: 0.0715574951171875
layer 9: 0.0787001953125
layer 10: 0.0854931640625
layer 11: 0.09280859375
layer 12: 0.09901171875
layer 13: 0.107640625
layer 14: 0.11785498046875
layer 15: 0.1256083984375
layer 16: 0.1408369140625
layer 17: 0.156142578125
layer 18: 0.17044140625
layer 19: 0.191591796875
layer 20: 0.20652734375
layer 21: 0.2248125
layer 22: 0.239251953125
layer 23: 0.272525390625
layer 24: 0.2862265625
layer 25: 0.30887890625
layer 26: 0.329537109375
layer 27: 0.359927734375
layer 28: 0.3814072265625
layer 29: 0.400908203125
layer 30: 0.44475390625
layer 31: 0.5362109375
layer 32: 0.13218017578125
logits: 0.1447247314453125
```

▼ GPT2, FP16

```
layer 0: 0.0
layer 1: 0.010214111328125
layer 2: 0.011416259765625
layer 3: 0.0163514404296875
layer 4: 0.0228807373046875
layer 5: 0.0232802734375
layer 6: 0.0260006103515625
layer 7: 0.02941253662109375
layer 8: 0.03486376953125
layer 9: 0.04135888671875
layer 10: 0.0513974609375
layer 11: 0.0786591796875
layer 12: 0.190262451171875
logits: 0.1796796875
```

As we can see:

1. The error propagates (and increases) across layers
2. Lower precisions greatly increase the mismatch between using KV cache or not
3. BF16 is more sensible to this difference than FP16 -- this is expected, BF16 dedicates more bits to the exponent, so rounding errors are larger
4. This phenomenon also happens in battle-tested models like `GPT2`

What can we do about it?

First of all: the benefits of using variables with lower precision and KV caching is obvious. Are the downsides worth it? My advice is to measure the model on metrics relevant to your task (e.g. perplexity), and compare the cost-benefits on your use case. I suspect using KV caching will remain cost-effective :)

Secondly: there may be ways to reduce this mismatch, but so far I haven't found any. A common trick is to upcast some sensible operations to FP32 (like the on the attention layers' softmax). For completeness, on Llama, I tried:

1. Upcasting the `Linear` layers in the attention layer
2. Running the whole attention layer in FP32
3. Running `apply_rotary_pos_emb` in FP32 (while keeping `sin` and `cos` in FP32 as well)
4. In the decoder layer, upcasting `self.input_layernorm(hidden_states)`
5. In the decoder layer, upcasting `self.post_attention_layernorm(hidden_states)`

Most had no impact, some reduced the mismatch at a high throughput cost.

Finally, regarding left-padding: We might be able to mitigate problems here when we migrate batched generation to nested tensors, which don't need padding.

I hope this comprehensive analysis helps you understand what's going on 🤗 And, who knows, be the spark that ignites a solution to this issue 🪄

😀 👍 3 🎉 8 ❤️ 26 🚀 25

gante mentioned this issue on Oct 23, 2023

bfloat16/float16 llama model logits are significantly different when the same input is in a batch #26869

Closed

4 tasks