Handling big models for inference

When loading a pretrained model in PyTorch, the usual workflow looks like this:

```
import torch

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

In plain English, those steps are:

1. Create the model with randomly initialized weights
2. Load the model weights (in a dictionary usually called a state dict) from the disk
3. Load those weights inside the model

While this works very well for regularly sized models, this workflow has some clear limitations when we deal with a huge model: in step 1, we load a full version of the model in RAM, and spend some time randomly initializing the weights (which will be discarded in step 3). In step 2, we load another full version of the model in RAM, with the pretrained weights. If you're loading a model with 6 billions parameters, this means you will need 24GB of RAM for each copy of the model, so 48GB in total (half of it to load the model in FP16).

This API is quite new and still in its experimental stage. While we strive to provide a stable API, it's possible some small parts of the public API will change in the future.

How the Process Works: A Quick Overview

How the Process Works: Working with Code

Instantiating an empty model

The first tool 🤗 Accelerate introduces to help with big models is a context manager init_empty_weights() that helps you initialize a model without using any RAM, so that step 1 can be done on models of any size. Here is how it works:

```
from accelerate import init_empty_weights

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

For instance:

```
with init_empty_weights():
    model = nn.Sequential(*[nn.Linear(10000, 10000) for _ in range(1000)])
```

initializes an empty model with a bit more than 100B parameters. Behind the scenes, this relies on the meta device introduced in PyTorch 1.9. During the initialization under the context manager, each time a parameter is created, it is instantly moved on that device.

You can’t move a model initialized like this on CPU or another device directly, since it doesn’t have any data. It’s also very likely that a forward pass with that empty model will fail, as not all operations are supported on the meta device.

Sharded checkpoints

It’s possible your model is so big that even a single copy won’t fit in RAM. That doesn’t mean it can’t be loaded: if you have one or several GPUs, this is more memory available to store your model. In this case, it’s better if your checkpoint is split in several smaller files that we call checkpoint shards.

🤗 Accelerate will handle sharded checkpoints as long as you follow the following format: your checkpoint should be in a folder, with several files containing the partial state dicts, and there should be an index in the JSON format that contains a dictionary mapping parameter names to the file containing their weights. For instance we could have a folder containing:

```
first_state_dict.bin
index.json
second_state_dict.bin
```

with index.json being the following file:

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

and `first_state_dict.bin` containing the weights for `"linear1.weight"` and `"linear1.bias"`, `second_state_dict.bin` the ones for `"linear2.weight"` and `"linear2.bias"`

Loading weights

The second tool 🤗 Accelerate introduces is a function load_checkpoint_and_dispatch(), that will allow you to load a checkpoint inside your empty model. This supports full checkpoints (a single file containing the whole state dict) as well as sharded checkpoints. It will also automatically dispatch those weights across the devices you have available (GPUs, CPU RAM), so if you are loading a sharded checkpoint, the maximum RAM usage will be the size of the biggest shard.

Here is how we can use this to load the GPT-J-6B model. You clone the sharded version of this model with:

```
git clone https://huggingface.co/sgugger/sharded-gpt-j-6B
cd sharded-gpt-j-6B
git-lfs install
git lfs pull
```

then we can initialize the model with

```
from accelerate import init_empty_weights
from transformers import AutoConfig, AutoModelForCausalLM

checkpoint = "EleutherAI/gpt-j-6B"
config = AutoConfig.from_pretrained(checkpoint)

with init_empty_weights():
    model = AutoModelForCausalLM.from_config(config)
```

Note that loading the model with `from_config` in Transformers does not tie the weights, which may cause issue when loading a checkpoint that does not contain duplicate keys for the tied weights. So you should tie the weights before loading the checkpoint.

```
model.tie_weights()
```

Then load the checkpoint we just downloaded with:

```
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"
)
```

By passing `device_map="auto"`, we tell 🤗 Accelerate to determine automatically where to put each layer of the model depending on the available resources:

- first we use the maximum space available on the GPU(s)
- if we still need space, we store the remaining weights on the CPU
- if there is not enough RAM, we store the remaining weights on the hard drive as memory-mapped tensors

`no_split_module_classes=["GPTJBlock"]` indicates that the modules that are `GPTJBlock` should not be split on different devices. You should set here all blocks that include a residual connection of some kind.

You can see the `device_map` that 🤗 Accelerate picked by accessing the `hf_device_map` attribute of your model:

```
model.hf_device_map
```

```
{'transformer.wte': 0,
 'transformer.drop': 0,
 'transformer.h.0': 0,
 'transformer.h.1': 0,
 'transformer.h.2': 0,
 'transformer.h.3': 0,
 'transformer.h.4': 0,
 'transformer.h.5': 0,
 'transformer.h.6': 0,
 'transformer.h.7': 0,
 'transformer.h.8': 0,
 'transformer.h.9': 0,
 'transformer.h.10': 0,
 'transformer.h.11': 0,
 'transformer.h.12': 0,
 'transformer.h.13': 0,
 'transformer.h.14': 0,
 'transformer.h.15': 0,
 'transformer.h.16': 0,
 'transformer.h.17': 0,
 'transformer.h.18': 0,
 'transformer.h.19': 0,
 'transformer.h.20': 0,
 'transformer.h.21': 0,
 'transformer.h.22': 0,
 'transformer.h.23': 0,
 'transformer.h.24': 1,
 'transformer.h.25': 1,
 'transformer.h.26': 1,
 'transformer.h.27': 1,
 'transformer.ln_f': 1,
 'lm_head': 1}
```

You can also design your `device_map` yourself, if you prefer to explicitly decide where each layer should be. In this case, the command above becomes:

```
model = load_checkpoint_and_dispatch(model, "sharded-gpt-j-6B", device_map=my_device_m
```

Run the model

Now that we have done this, our model lies across several devices, and maybe the hard drive. But it can still be used as a regular PyTorch model:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
inputs = tokenizer("Hello, my name is", return_tensors="pt")
inputs = inputs.to(0)
output = model.generate(inputs["input_ids"])
tokenizer.decode(output[0].tolist())
```

Behind the scenes, 🤗 Accelerate added hooks to the model, so that:

- at each layer, the inputs are put on the right device (so even if your model is spread across several GPUs, it works)
- for the weights offloaded on the CPU, they are put on a GPU just before the forward pass, and cleaned up just after
- 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

This way, you model can run for inference even if it doesn't fit on one of the GPUs or the CPU RAM!

> This only supports inference of your model, not training. Most of the computation happens behind `torch.no_grad()` context managers to avoid spending some GPU memory with intermediate activations.

Designing a device map

You can let 🤗 Accelerate handle the device map computation by setting `device_map` to one of the supported options (`"auto"`, `"balanced"`, `"balanced_low_0"`, `"sequential"`) or create one yourself, if you want more control over where each layer should go.

> You can derive all sizes of the model (and thus compute a `device_map`) on a model that is on the meta device.

All the options will produce the same result when you don't have enough GPU memory to accommodate the whole model (which is to fit everything that can on the GPU, then offload weights on the CPU or even on the disk if there is not enough RAM).

When you have more GPU memory available than the model size, here the difference between each option:

- `"auto"` and `"balanced"` evenly split the model on all available GPUs, making it possible for you to use a batch size greater than 1.
- `"balanced_low_0"` evenly splits the model on all GPUs except the first one, and only puts on GPU 0 what does not fit on the others. This option is great when you need to use GPU 0 for some processing of the outputs, like when using the `generate` function for Transformers models
- `"sequential"` will fit what it can on GPU 0, then move on GPU 1 and so forth (so won't use the last GPUs if it doesn't need to).

> The options `"auto"` and `"balanced"` produce the same results for now, but the behavior of `"auto"` might change in the future if we find a strategy that makes more sense, while `"balanced"` will stay stable.

First note that you can limit the memory used on each GPU by using the `max_memory` argument (available in `infer_auto_device_map()` and in all functions using it). When setting `max_memory`, you should pass along a dictionary containing the GPU identifiers (for instance `0`, `1` etc.) and the `"cpu"` key for the maximum RAM you want used for CPU offload. The values can either be an integer (in bytes) or a string representing a number with its unit, such as `"10GiB"` or `"10GB"`.

Here is an example where we don't want to use more than 10GiB on each of two GPUs and no more than 30GiB of CPU RAM for the model weights:

```
from accelerate import infer_auto_device_map

device_map = infer_auto_device_map(my_model, max_memory={0: "10GiB", 1: "10GiB", "cpu"
```

> When a first allocation happens in PyTorch, it loads CUDA kernels which take about 1-2GB of memory depending on the GPU. Therefore you always have less usable memory than the actual size of the GPU. To see how much memory is actually used do `torch.ones(1).cuda()` and look at the memory usage.

Therefore when you create memory maps with `max_memory` make sure to adjust the avaialble memory accordingly to avoid out-of-memory errors.

Additionally, if you do some additional operations with your outputs without placing them back on the CPU (for instance inside the `generate` method of Transformers) and if you placed your inputs on a GPU, that GPU will consume more memory than the others (Accelerate always place the output back to the device of the input). Therefore if you would like to optimize the maximum batch size and you have many GPUs, give the first GPU less memory. For example, with BLOOM-176B on 8x80 A100 setup the close to ideal map is:

```
max_memory = {0: "30GIB", 1: "46GIB", 2: "46GIB", 3: "46GIB", 4: "46GIB", 5: "46GIB",
```

as you can see we gave the remaining 7 GPUs ~50% more memory than GPU 0.

If you opt to fully design the `device_map` yourself, it should be a dictionary with keys being module names of your model and values being a valid device identifier (for instance an integer for the GPUs) or `"cpu"` for CPU offload, `"disk"` for disk offload. The keys need to cover the whole model, you can then define your device map as you wish: for instance if your model has two blocks (let's say `block1` and `block2`) which each contain three linear layers (let's say `linear1`, `linear2` and `linear3`), a valid device map can be:

```
device_map = {"block1": 0, "block2": 1}
```

another one that is valid could be:

```
device_map = {"block1": 0, "block2.linear1": 0, "block2.linear2": 1, "block2.linear3":
```

On the other hand, this one is not valid as it does not cover every parameter of the model:

```
device_map = {"block1": 0, "block2.linear1": 1, "block2.linear2": 1}
```

> To be the most efficient, make sure your device map puts the parameters on the GPUs in a sequential manner (e.g. don't put one of the first weights on GPU 0, then weights on GPU 1 and the last weight back to GPU 0) to avoid making many transfers of data between the GPUs.

Limits and further development

We are aware of the current limitations in the API:

- While this could theoretically work on just one CPU with potential disk offload, you need at least one GPU to run this API. This will be fixed in further development.
- `infer_auto_device_map()` (or `device_map="auto"` in load_checkpoint_and_dispatch()) tries to maximize GPU and CPU RAM it sees available when you execute it. While PyTorch is very good at managing GPU RAM efficiently (and giving it back when not needed), it's not entirely true with Python and CPU RAM. Therefore, an automatically computed device map might be too intense on the CPU. Move a few modules to the disk device if you get crashes due to lack of RAM.
- `infer_auto_device_map()` (or `device_map="auto"` in load_checkpoint_and_dispatch()) attributes devices sequentially (to avoid moving things back and forth) so if your first layer is bigger than the size of the GPU you have, it will end up with everything on the CPU/Disk.
- load_checkpoint_and_dispatch() and `load_checkpoint_in_model()` do not perform any check on the correctness of your state dict compared to your model at the moment (this will be fixed in a future version), so you may get some weird errors if trying to load a checkpoint with mismatched or missing keys.
- The model parallelism used when your model is split on several GPUs is naive and not optimized, meaning that only one GPU works at a given time and the other sits idle.
- When weights are offloaded on the CPU/hard drive, there is no pre-fetching (yet, we will work on this for future versions) which means the weights are put on the GPU when they are needed and not before.
- Hard-drive offloading might be very slow if the hardware you run on does not have fast

communication between disk and CPU (like NVMes).

← Example Zoo How to perform distributed inference with normal resources →