title	thumbnail	authors
How 🤗 Accelerate runs very large models thanks to PyTorch	/blog/assets/104_accelerate-large-models/thumbnail.png	user: sgugger

How 🤗 Accelerate runs very large models thanks to PyTorch

Load and run large models

Meta AI and BigScience recently open-sourced very large language models which won't fit into memory (RAM or GPU) of most consumer hardware. At Hugging Face, part of our mission is to make even those large models accessible, so we developed tools to allow you to run those models even if you don't own a supercomputer. All the examples picked in this blog post run on a free Colab instance (with limited RAM and disk space) if you have access to more disk space, don't hesitate to pick larger checkpoints.

Here is how we can run OPT-6.7B:

```
import torch
from transformers import pipeline

# This works on a base Colab instance.
# Pick a larger checkpoint if you have time to wait and enough disk s
checkpoint = "facebook/opt-6.7b"
generator = pipeline("text-generation", model=checkpoint, device_map=

# Perform inference
generator("More and more large language models are opensourced so Hug
```

We'll explain what each of those arguments do in a moment, but first just consider the traditional model loading pipeline in PyTorch: it usually consists of:

1. Create the model
2. Load in memory its weights (in an object usually called `state_dict`)
3. Load those weights in the created model
4. Move the model on the device for inference

While that has worked pretty well in the past years, very large models make this approach challenging. Here the model picked has 6.7 *billion* parameters. In the default precision, it means that just step 1 (creating the model) will take roughly **26.8GB** in RAM (1 parameter in float32 takes 4 bytes in memory). This can't even fit in the RAM you get on Colab.

Then step 2 will load in memory a second copy of the model (so another 26.8GB in RAM in default precision). If you were trying to load the largest models, for example BLOOM or OPT-176B (which both have 176 billion parameters), like this, you would need 1.4 **terabytes** of CPU RAM. That is a bit excessive! And all of this to just move the model on one (or several) GPU(s) at step 4.

Clearly we need something smarter. In this blog post, we'll explain how Accelerate leverages PyTorch features to load and run inference with very large models, even if they don't fit in RAM or one GPU. In a nutshell, it changes the process above like this:

1. Create an empty (e.g. without weights) model
2. Decide where each layer is going to go (when multiple devices are available)
3. Load in memory parts of its weights
4. Load those weights in the empty model
5. Move the weights on the device for inference

6. Repeat from step 3 for the next weights until all the weights are loaded

Creating an empty model

PyTorch 1.9 introduced a new kind of device called the *meta* device. This allows us to create tensor without any data attached to them: a tensor on the meta device only needs a shape. As long as you are on the meta device, you can thus create arbitrarily large tensors without having to worry about CPU (or GPU) RAM.

For instance, the following code will crash on Colab:

```
import torch

large_tensor = torch.randn(100000, 100000)
```

as this large tensor requires `4 * 10**10` bytes (the default precision is FP32, so each element of the tensor takes 4 bytes) thus 40GB of RAM. The same on the meta device works just fine however:

```
import torch

large_tensor = torch.randn(100000, 100000, device="meta")
```

If you try to display this tensor, here is what PyTorch will print:

```
tensor(..., device='meta', size=(100000, 100000))
```

As we said before, there is no data associated with this tensor, just a shape.

You can instantiate a model directly on the meta device:

```
large_model = torch.nn.Linear(100000, 100000, device="meta")
```

But for an existing model, this syntax would require you to rewrite all your modeling code so that each submodule accepts and passes along a `device` keyword argument. Since this was impractical for the 150 models of the Transformers library, we developed a context manager that will instantiate an empty model for you.

Here is how you can instantiate an empty version of BLOOM:

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

config = AutoConfig.from_pretrained("bigscience/bloom")
with init_empty_weights():
    model = AutoModelForCausalLM.from_config(config)
```

This works on any model, but you get back a shell you can't use directly: some operations are implemented for the meta device, but not all yet. Here for instance, you can use the `large_model` defined above with an input, but not the BLOOM model. Even when using it, the output will be a tensor of the meta device, so you will get the shape of the result, but nothing more.

As further work on this, the PyTorch team is working on a new class `FakeTensor`, which is a bit like tensors on the meta device, but with the device information (on top of shape and dtype)

Since we know the shape of each weight, we can however know how much memory they will all consume once we load the pretrained tensors fully. Therefore, we can make a decision on how to split our model across CPUs and GPUs.

Computing a device map

Before we start loading the pretrained weights, we will need to know where we want to put them. This way we can free the CPU RAM each time we have put a weight in its right place. This can be done with the empty model on the meta device, since we only need to know the shape of each tensor and its dtype to compute how much space it will take in memory.

Accelerate provides a function to automatically determine a *device map* from an empty model. It will try to maximize the use of all available GPUs, then CPU RAM, and finally flag the weights that don't fit for disk offload. Let's have a look using OPT-13b.

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

config = AutoConfig.from_pretrained("facebook/opt-13b")
with init_empty_weights():
    model = AutoModelForCausalLM.from_config(config)

device_map = infer_auto_device_map(model)
```

This will return a dictionary mapping modules or weights to a device. On a machine with one Titan RTX for instance, we get the following:

```
{'model.decoder.embed_tokens': 0,
 'model.decoder.embed_positions': 0,
 'model.decoder.final_layer_norm': 0,
 'model.decoder.layers.0': 0,
 'model.decoder.layers.1': 0,
 ...
 'model.decoder.layers.9': 0,
 'model.decoder.layers.10.self_attn': 0,
 'model.decoder.layers.10.activation_fn': 0,
 'model.decoder.layers.10.self_attn_layer_norm': 0,
 'model.decoder.layers.10.fc1': 'cpu',
 'model.decoder.layers.10.fc2': 'cpu',
 'model.decoder.layers.10.final_layer_norm': 'cpu',
 'model.decoder.layers.11': 'cpu',
 ...
 'model.decoder.layers.17': 'cpu',
 'model.decoder.layers.18.self_attn': 'cpu',
 'model.decoder.layers.18.activation_fn': 'cpu',
 'model.decoder.layers.18.self_attn_layer_norm': 'cpu',
 'model.decoder.layers.18.fc1': 'disk',
 'model.decoder.layers.18.fc2': 'disk',
 'model.decoder.layers.18.final_layer_norm': 'disk',
 'model.decoder.layers.19': 'disk',
 ...
 'model.decoder.layers.39': 'disk',
 'lm_head': 'disk'}
```

Accelerate evaluated that the embeddings and the decoder up until the 9th block could all fit on the GPU (device 0), then part of the 10th block needs to be on the CPU, as well as the following weights until the 17th layer. Then the 18th layer is split between the CPU and the disk and the following layers must all be offloaded to disk

Actually using this device map later on won't work, because the layers composing this model have residual connections (where the input of the block is added to the output of the block) so all of a given layer should be on the same device. We can indicate this to Accelerate by passing a list of module names that shouldn't be split with the `no_split_module_classes` keyword argument:

```
device_map = infer_auto_device_map(model, no_split_module_classes=["O
```

This will then return

```
'model.decoder.embed_tokens': 0,
 'model.decoder.embed_positions': 0,
 'model.decoder.final_layer_norm': 0,
 'model.decoder.layers.0': 0,
 'model.decoder.layers.1': 0,
 ...
 'model.decoder.layers.9': 0,
 'model.decoder.layers.10': 'cpu',
 'model.decoder.layers.11': 'cpu',
 ...
 'model.decoder.layers.17': 'cpu',
 'model.decoder.layers.18': 'disk',
 ...
 'model.decoder.layers.39': 'disk',
 'lm_head': 'disk'}
```

Now, each layer is always on the same device.

In Transformers, when using `device_map` in the `from_pretrained()` method or in a `pipeline`, those classes of blocks to leave on the same device are automatically provided, so you don't need to worry about them. Note that you have the following options for `device_map` (only relevant when you have more than one GPU):

- `"auto"` or `"balanced"`: Accelerate will split the weights so that each GPU is used equally;
- `"balanced_low_0"`: Accelerate will split the weights so that each GPU is used equally except the first one, where it will try to have as little weights as possible (useful when you want to work with the outputs of the model on one GPU, for instance when using the `generate` function);
- `"sequential"`: Accelerate will fill the GPUs in order (so the last ones might not

be used at all).

You can also pass your own `device_map` as long as it follows the format we saw before (dictionary layer/module names to device).

Finally, note that the results of the `device_map` you receive depend on the selected dtype (as different types of floats take a different amount of space). Providing `dtype="float16"` will give us different results:

```
device_map = infer_auto_device_map(model, no_split_module_classes=["O
```

In this precision, we can fit the model up to layer 21 on the GPU:

```
{'model.decoder.embed_tokens': 0,
 'model.decoder.embed_positions': 0,
 'model.decoder.final_layer_norm': 0,
 'model.decoder.layers.0': 0,
 'model.decoder.layers.1': 0,
 ...
 'model.decoder.layers.21': 0,
 'model.decoder.layers.22': 'cpu',
 ...
 'model.decoder.layers.37': 'cpu',
 'model.decoder.layers.38': 'disk',
 'model.decoder.layers.39': 'disk',
 'lm_head': 'disk'}
```

Now that we know where each weight is supposed to go, we can progressively load the pretrained weights inside the model.

Sharding state dicts

Traditionally, PyTorch models are saved in a whole file containing a map from parameter name to weight. This map is often called a `state_dict`. Here is an excerpt from the PyTorch documentation on saving on loading:

```
# Save the model weights
torch.save(my_model.state_dict(), 'model_weights.pth')

# Reload them
new_model = ModelClass()
new_model.load_state_dict(torch.load('model_weights.pth'))
```

This works pretty well for models with less than 1 billion parameters, but for larger models, this is very taxing in RAM. The BLOOM model has 176 billions parameters; even with the weights saved in bfloat16 to save space, it still represents 352GB as a whole. While the super computer that trained this model might have this amount of memory available, requiring this for inference is unrealistic.

This is why large models on the Hugging Face Hub are not saved and shared with one big file containing all the weights, but **several** of them. If you go to the BLOOM model page for instance, you will see there is 72 files named `pytorch_model_xxxxx-of-00072.bin`, which each contain part of the model weights. Using this format, we can load one part of the state dict in memory, put the weights inside the model, move them on the right device, then discard this state dict part before going to the next. Instead of requiring to have enough RAM to accommodate the whole model, we only need enough RAM to get the biggest checkpoint part, which we call a **shard**, so 7.19GB in the case of BLOOM.

We call the checkpoints saved in several files like BLOOM *sharded checkpoints*, and we have standardized their format as such:

- One file (called `pytorch_model.bin.index.json`) contains some metadata and a map parameter name to file name, indicating where to find each weight
- All the other files are standard PyTorch state dicts, they just contain a part of the model instead of the whole one. You can have a look at the content of the index file here.

To load such a sharded checkpoint into a model, we just need to loop over the various shards. Accelerate provides a function called `load_checkpoint_in_model` that will do this for you if you have cloned one of the repos of the Hub, or you can directly use the `from_pretrained` method of Transformers, which will handle the downloading and caching for you:

```
import torch
from transformers import AutoModelForCausalLM

# Will error
checkpoint = "facebook/opt-13b"
model = AutoModelForCausalLM.from_pretrained(checkpoint, device_map="
```

If the device map computed automatically requires some weights to be offloaded on disk because you don't have enough GPU and CPU RAM, you will get an error indicating you need to pass an folder where the weights that should be stored on disk will be offloaded:

```
ValueError: The current `device_map` had weights offloaded to the dis
`offload_folder` for them.
```

Adding this argument should resolve the error:

```
import torch
from transformers import AutoModelForCausalLM

# Will go out of RAM on Colab
checkpoint = "facebook/opt-13b"
model = AutoModelForCausalLM.from_pretrained(
    checkpoint, device_map="auto", offload_folder="offload", torch_dt
)
```

Note that if you are trying to load a very large model that require some disk offload on top of CPU offload, you might run out of RAM when the last shards of the checkpoint are loaded, since there is the part of the model staying on CPU taking space. If that is the case, use the option `offload_state_dict=True` to temporarily offload the part of the model staying on CPU while the weights are all loaded, and reload it in RAM once all the weights have been processed

```
import torch
from transformers import AutoModelForCausalLM

checkpoint = "facebook/opt-13b"
model = AutoModelForCausalLM.from_pretrained(
    checkpoint, device_map="auto", offload_folder="offload", offload_
)
```

This will fit in Colab, but will be so close to using all the RAM available that it will go out of RAM when you try to generate a prediction. To get a model we can use, we need to offload one more layer on the disk. We can do so by taking the `device_map` computed in the previous section, adapting it a bit, then passing it to the `from_pretrained` call:

```
import torch
from transformers import AutoModelForCausalLM

checkpoint = "facebook/opt-13b"
device_map["model.decoder.layers.37"] = "disk"
model = AutoModelForCausalLM.from_pretrained(
    checkpoint, device_map=device_map, offload_folder="offload", offl
)
```

Running a model split on several devices

One last part we haven't touched is how Accelerate enables your model to run with its weight spread across several GPUs, CPU RAM, and the disk folder. This is done very simply using hooks.

> hooks are a PyTorch API that adds functions executed just before each forward called

We couldn't use this directly since they only support models with regular arguments and no keyword arguments in their forward pass, but we took the same idea. Once the model is loaded, the `dispatch_model` function will add hooks to every module and submodule that are executed before and after each forward pass. They will:

- make sure all the inputs of the module are on the same device as the weights;
- if the weights have been offloaded to the CPU, move them to GPU 0 before the forward pass and back to the CPU just after;
- if the weights have been offloaded to disk, load them in RAM then on the GPU 0 before the forward pass and free this memory just after.

The whole process is summarized in the following video:

<iframe width="560" height="315" src="https://www.youtube.com/embed/MWCSGj9jEAo" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope;

picture-in-picture" allowfullscreen></iframe>
This way, your model can be loaded and run even if you don't have enough GPU RAM and CPU RAM. The only thing you need is disk space (and lots of patience!) While this solution is pretty naive if you have multiple GPUs (there is no clever pipeline parallelism involved, just using the GPUs sequentially) it still yields pretty decent results for BLOOM. And it allows you to run the model on smaller setups (albeit more slowly).

To learn more about Accelerate big model inference, see the documentation.