Getting Started with Google's Gemma LLM using HuggingFace Libraries

Coldstart Coder · Follow

10 min read · Mar 6, 2024

 35    

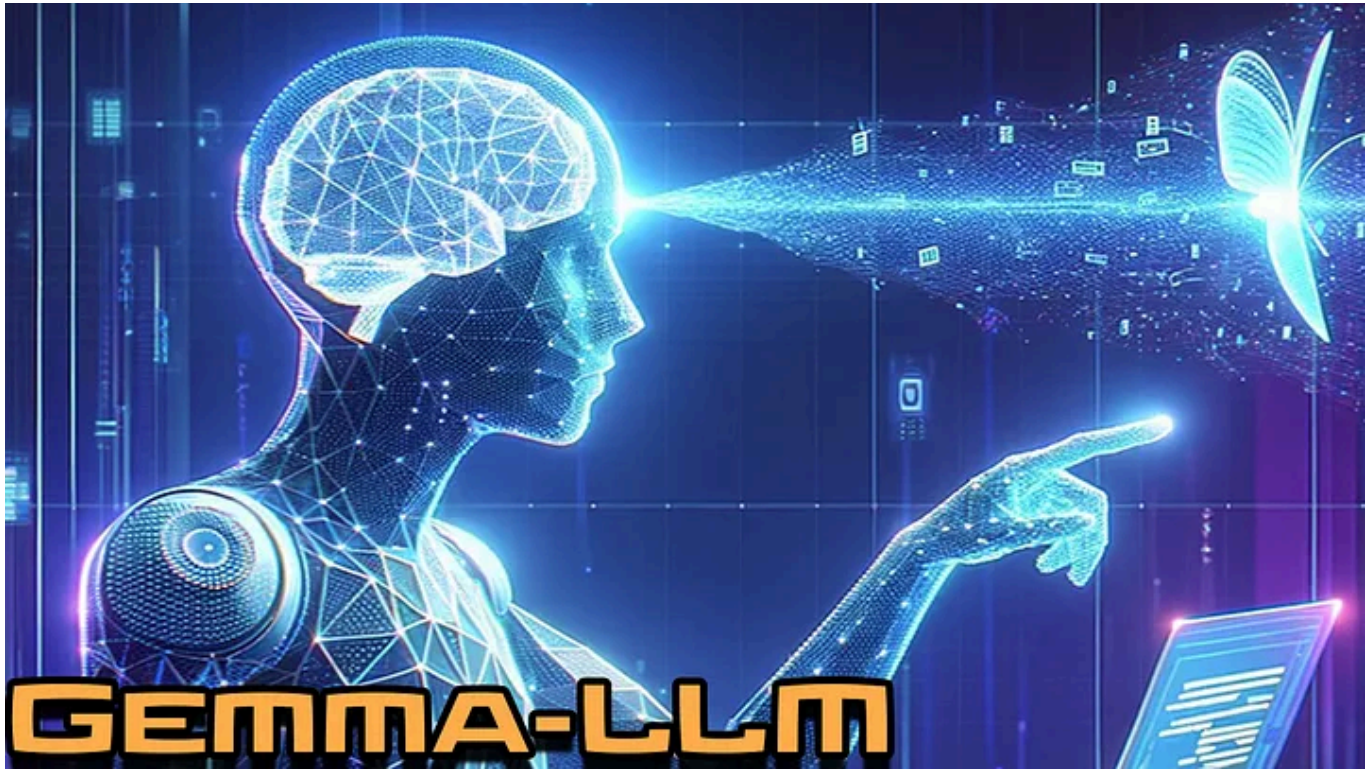

Image Generated through Chat-GPT and further edited by author

(This article is also available as a notebook on Google Collab you can find the notebook Here)

Google recently released a series of open source LLMs based on their Gemini flagship model. These smaller models are built using the same research and training methodologies that Google used to create Gemini and come with very big promises in how they will reshape the open source LLM space. The models come in 2 sizes, 2B and 7B making them small enough that they can

sit on consumer level hardware, and even Google's own Collab service if users don't want or don't have access to their own personal GPUs. The Gemma models are exciting entries into the LLM race and I'm excited to explore them. In this notebook I'll go over how to access these models and run them in your own environment using Huggingface's libraries and tools.

Accessing the models

The Gemma family of models have been uploaded to HuggingFace, so anyone can request access and download the model. All you need is a huggingface account and then request access on the model cards on their website. This initial requirement is mostly to show that if you use the models you accept their terms and conditions to not misuse them. The terms of service are pretty open on what you can do, you can even use these models for commercial purposes. However google still wants users to agree to not use these models for evil, so you have to agree to their terms before you can download them.

There are 4 models available from Google(and plenty of fine tuned ones from other sources): gemma-2b, gemma-7b, gemma-2b-it and gemma-7b-it. The -it variants mean those are instruction tuned, meaning that Google has

put in some extra training to have those models ready to perform tasks for users rather than just try and complete/continue the text. This additional tuning makes them able to act as chatbots. For most initial experiments I would recommend starting with gemma-7b-it. You can find that model Here, that page also links to the other models in the family if you choose to download a different variant.

After your request for access is approved (for me it was instantaneous) you will then be able to pull the model from huggingface. For that you will need the transformers and accelerate packages provided by huggingface that makes it easy to work with models in their repository. So install those packages and make sure they are up to date:

> *!pip install — upgrade transformers accelerate*

Logging into HuggingFace from your Environment

Since these are gated models hugging face is going to require some authorization. Just to prove you have accepted the terms for using the model. If you are in a notebook you can use the following code that will embed a login widget that will let you put\ in your access token from huggingface (you can find it in your profile settings on the hugging face website).

```
from huggingface_hub import notebook_login
notebook_login()
```

Loading the Model

Now to download the model. We'll import some packages, and include torch as well. The gemma models will be loaded as a pytorch model, so we can use torch to specify datatypes and other aspects of our model when we load it in. By default they will be loaded as torch.float32, meaning each weight will be made up of 32 bits of information, for our 7b model this means it will take about 32gb of memory needed to load this model. For a gpu like an a100 this isn't an issue since you have 40gb of space, but if you are on a smaller gpu or plan to use gpu memory for other functions (such as fine tuning) then you may want to opt instead to load the model as a bfloat16 datatype which will cut the memory requirements in half. This is smaller so it will take up less space, and although this means we'll lose a bit of model precision in practice I've found it doesn't change it so drastically to impact usability. Another option is to look at quantization methods to reduce the size as well, but we won't be using any advanced quantization methods in this notebook (maybe in a future notebook).

For this notebook we'll assume you have enough gpu space to load in the full float32 weights, but I've included the code to load it in with bfloat16 instead.

So now select our model checkpoint, load in the tokenizer and the model!

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, set_seed
set_seed(1234)

# identify which checkpoint we want, this is the repository on huggingface
# that we'll pull the model from,
model_checkpoint = "google/gemma-7b-it"

# load in the tokenizer,
# just in case you aren't familiar with llms the tokenizer takes the raw text
# and converts it into a vector that can be processed by our model,
# so basically it's a converter to convert english into numbers for our model
# to compute on
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)


# load the model using torch.bfloat16 to take up less space on the gpu
#model = AutoModelForCausalLM.from_pretrained(model_checkpoint, torch_dtype=torc


# load the model in it's default torch.float32 datatype, be sure to identify
# the device map as cuda to tell huggingface to put the model on the gpu
# you could still run it on the cpu, but it will be SLOW,
model = AutoModelForCausalLM.from_pretrained(model_checkpoint, device_map="cuda"
```

Next for fun let's take a look at the model summary, we can do this by just calling print on our model. We'll also do a sanity check to make sure our model was loaded with the proper datatype.

```
# print out the model layers
print(model)
print(model.dtype)
```

Here's the output:

```
GemmaForCausalLM(
  (model): GemmaModel(
    (embed_tokens): Embedding(256000, 3072, padding_idx=0)
    (layers): ModuleList(
      (0-27): 28 x GemmaDecoderLayer(
        (self_attn): GemmaAttention(
          (q_proj): Linear(in_features=3072, out_features=4096, bias=False)
          (k_proj): Linear(in_features=3072, out_features=4096, bias=False)
          (v_proj): Linear(in_features=3072, out_features=4096, bias=False)
          (o_proj): Linear(in_features=4096, out_features=3072, bias=False)
          (rotary_emb): GemmaRotaryEmbedding()
        )
        (mlp): GemmaMLP(
          (gate_proj): Linear(in_features=3072, out_features=24576, bias=False)
          (up_proj): Linear(in_features=3072, out_features=24576, bias=False)
          (down_proj): Linear(in_features=24576, out_features=3072, bias=False)
```

```
            (act_fn): GELUActivation()
          )
          (input_layernorm): GemmaRMSNorm()
          (post_attention_layernorm): GemmaRMSNorm()
        )
      )
      (norm): GemmaRMSNorm()
    )
    (lm_head): Linear(in_features=3072, out_features=256000, bias=False)
)
torch.float32
```

Looks like everything is in order.

Next up, running inference on our input!

Generating Text

Gemma 7b has a context window of 8192 tokens, meaning we can feed it a lot of data to help it generate it's text. An important thing to remember is that words and tokens are not 1 to 1, a single word may translate to a single token or several. A good rule of thumb I found online is about 1.5 tokens per word, but your mileage may vary.

So let's start running some inference on some input. We'll start with a completion prompt, so we'll provide some text that the model will

complete/continue. This is how causal language models are trained, and it's later fine tuned for other tasks, so the model should perform pretty well in this task. First we define our prompt and convert that into tokens.

```
# create a completion prompt that leaves plenty of room for the model to generat
# new text
prompt = "Dr.Pepper is the best soda because"
# convert our prompt to tokens and send it to the gpu.
token_inputs = tokenizer(prompt, return_tensors="pt").to('cuda')
# just to see what the tokens look like for our prompt, we'll print them
token_inputs
```

As we can see the tokenizer returns a dict that has a single entry "input_ids" which holds the converted text. Technically these are the ids for what tokens the text are converted into, and not the tokens themselves, but that might just be splitting hairs. For other models or inputs it can return additional arguments to pass to the model but for now this is all we get, just the text converted into token ids.

Now we run those input tokens through our model. We set the max_new_tokens both to prevent the model from getting into an endless

loop and for stability reasons. I've found that the model sometimes encounters runtime errors if this value isn't set.

these models will generate tokens one at a time, predicting what token should come next in the sequence. So the model output layer is a giant matrix that contains the score for each token in it's corpus and how likely that token is to be next.

By setting do_sample to true the model will use a probability distribution to select the next token. Without this the model will always select the next most likely token, the one with the highest score, which for some tasks isn't desirable, we want a bit of variation in our output. temperature helps control that randomness, the higher the temperature the more likely that lower ranked tokens will be selected as the next output. Whether this is desirable for your use case is something you need to decide for yourself, but generally a bit of randomness is preferred. Not so much that the model generates gibberish, but enough that it varies a bit from call to call.

```
token_outputs = model.generate(input_ids=token_inputs['input_ids'], max_new_toke
# since this is exploratory we'll decode special tokens as well. for this prompt
# it will be the beginning <bos> token and the ending <eos> tokens.
```

```
decoded_output = tokenizer.decode(token_outputs[0], skip_special_tokens=False)
decoded_output
```

Here's the output for the above:

> *<bos>Dr.Pepper is the best soda because it has the most flavor.*
>
> *This statement is biased because it assumes that Dr.Pepper has the most flavor. There are many other sodas that have more flavor than Dr.Pepper.<eos>*

And there we go! That's the basics for how to call the model!

Chatting with the Model

The next step is to get a bit more structure to the prompts we send to our model. We don't just want the model to complete sentences, we want the model to be able to act as a chatbot, complete simple tasks and generally be more interactive.

To do this Google has trained the -it variant models to identify conversation prompts, when it sees a particular structure in the text we feed in it will respond as a chatbot.

For our bot(and most llm chatbots out there) we consider a conversation to be a series of turns. First the user provides input, then the bot, then the user ect. There is a specific format that the gemma models can identify and recognize that it needs to respond as a chatbot and be helpful, not just try and continue the text.

The structure of the prompt and how to use it is as follows:

> ```
>
> *<start_of_turn>user*
>
> *User text goes here<end_of_turn>*
>
> *<start_of_turn>model*
>
> ```

we introduce special tokens for starting and ending a turn that the model can recognize. After a start turn token we idenfity who's turn it is, either the user or the model. For the models turn we don't include the end of turn, it

will generate that, we then hand this prompt over and the bot will generate it's response to the conversation so far.

An important note is there is no system or instruction role, it only identifies user and model for this prompt. If you want to try and load the model with instructions you'll have to input them as an initial user turn.

It's pretty straightforward. So here's a small example of us constructing this prompt by hand:

```
# conversation prompt
prompt = \
'''<bos><start_of_turn>user
I'm hosting a backyard pool party this weekend, what sorts of foods should I ser
<start_of_turn>model
'''
token_inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
token_outputs = model.generate(input_ids=token_inputs['input_ids'], do_sample=Tr
# in this case we only care about what the model says next, so we won't
# print out special tokens this time,
decoded_output = tokenizer.decode(token_outputs[0], skip_special_tokens=True)
print(decoded_output)
```

You'll notice that the decoded_output has both our original prompt and the new content generated by the model. If we wanted to just decode and use the new text we can do that using some indexing with the following snippet of code:

```
# what this index is doing is grabbing everything that comes after
# our original prompt,
new_tokens = token_outputs[0][token_inputs['input_ids'].shape[-1]:]
decoded_output = tokenizer.decode(new_tokens, skip_special_tokens=True)
print(decoded_output)
```

So now our agent is able to carry on a conversation with us. This opens us up to very interesting possibilities, we can ask the model to perform certain tasks as the user (such as summarization, question/answering, ect.) and we now have a level of interactivity with it to provide feedback it can use to improve it's answer.

Chat Prompt Creation with Tokenizer

One thing we can do is instead of manually crafting this prompt we can actually use the tokenizer to take in a structured list of inputs and construct this prompt for us. If you've ever used the openai chat api this format will

seem very similar. We will create a list of dict objects, those dict objects will have a role and content key which will tell us who's going during that turn and what their input is. In code we can then just pass in this list into the tokenizer and it will automatically create the conversation prompt from before.

```
# create a list of turns for our conversation, each conversation has to start
# with a user turn and must alternate between user/model
chat = [
    {"role": "user", "content": "Write a hello world program in Python please."}
]
# next we'll use the apply_chat_template from our tokenizer, this will convert
# our list into the conversation prompt and at the same time
# convert the prompt to tokens all in one call.
token_inputs = tokenizer.apply_chat_template(chat, tokenize=True, return_tensors

# an important thing to note about the apply_chat_template, it returns the tenso
# unlike how we called the tokenizer before and got back a dict with the input_i
# it's not a huge change, just need to keep in mind this returns a slightly diff

# call inference on the model
token_outputs = model.generate(input_ids=token_inputs, do_sample=True, max_new_t
new_tokens = token_outputs[0][token_inputs.shape[-1]:]
decoded_output = tokenizer.decode(new_tokens, skip_special_tokens=True)
print(decoded_output)
```

Awesome, that makes managing the conversation a lot easier! Let's extend the example to have a running conversation:

```
chat = [

]
print("model: Hello! How can I help you?")
while user_input := input("User:"):
    # add in a quick check to break this loop cleanly.
    if user_input == "exit":
      break
    user_turn = {"role": "user", "content": user_input}
    chat.append(user_turn)
    token_inputs = tokenizer.apply_chat_template(chat, tokenize=True, return_ten
    token_outputs = model.generate(input_ids=token_inputs, do_sample=True, max_n
    new_tokens = token_outputs[0][token_inputs.shape[-1]:]
    decoded_output = tokenizer.decode(new_tokens, skip_special_tokens=True)
    model_turn = {"role": "model", "content": decoded_output}
    chat.append(model_turn)
    print("model: ", decoded_output)
```

Using the above you can have an interactive chat with the model!

Closing Thoughts

And that's the basics for how to use the gemma models using huggingface libraries. From here there's a lot we can explore, such as quantization to

make the model smaller and faster, or we can attempt to fine tune the model for our particular use case or to make it have a particular tone, or we can just explore prompt engineering and support systems to build applications such as question answering over documents or interactive interfaces.

There's a lot we can explore, and I'm hoping to write more articles exploring them soon. But this is enough to get started!

Hope you found this interesting and learned something! Happy Coding!

Hugging Face | Python | Llm | Gemma