l6

March 20, 2025

1 Lesson 6: Model Example

In this lesson, you will reinforce your understanding of the transformer architecture by exploring the decoder-only model `microsoft/Phi-3-mini-4k-instruct`.

1.1 Setup

We start with setting up the lab by installing the required libraries (`transformers` and `accelerate`) and ignoring the warnings. The `accelerate` library is required by the `Phi-3` model. But you don't need to worry about installing these libraries, the requirements for this lab are already installed.

Access requirements.txt file: If you'd like to access the requirements file: 1) click on the "File" option on the top menu of the notebook and then 2) click on "Open". For more help, please see the "Appendix – Tips, Help, and Download" Lesson.

[ ]:
```
# !pip install transformers>=4.41.2 accelerate>=0.31.0
```

[23]:
```
# Warning control
import warnings
warnings.filterwarnings('ignore')
```

1.2 Loading the LLM

Let's first load the model and its tokenizer. For that you will first import the classes: `AutoModelForCausalLM` and `AutoTokenizer`. When you want to process a sentence, you can apply the tokenizer first and then the model in two separate steps. Or you can create a pipeline object that wraps the two steps and then apply the pipeline to the sentence. You'll explore both approaches in this notebook. This is why you'll also import the `pipeline` class.

FYI: The transformers library has two types of model classes: AutoModelForCausalLM and AutoModelForMaskedLM. Causal language models represent the decoder-only models that are used for text generation. They are described as causal, because to predict the next token, the model can only attend to the preceding left tokens. Masked language models represent the encoder-only models that are used for rich text representation. They are described as masked, because they are trained to predict a masked or hidden token in a sequence.

[24]:
```
# import the required classes
# from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
```

[25]:
```
# Load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("../models/microsoft/
 ↪Phi-3-mini-4k-instruct")

model = AutoModelForCausalLM.from_pretrained(
    "../models/microsoft/Phi-3-mini-4k-instruct",
    device_map="cpu",
    torch_dtype="auto",
    trust_remote_code=True,
)
```

```
Special tokens have been added in the vocabulary, make sure the associated word
embeddings are fine-tuned or trained.
`flash-attention` package not found, consider installing for better performance:
No module named 'flash_attn'.
Current `flash-attention` does not support `window_size`. Either upgrade or use
`attn_implementation='eager'`.

Loading checkpoint shards:   0%|          | 0/2 [00:00<?, ?it/s]
```

Note: You'll receive a warning that the flash-attention package is not found. That's because flash attention requires certain types of GPU hardware to run. Since the model of this lab is not using any GPU, you can ignore this warning.

Now you can wrap the model and the tokenizer in a pipeline object that has "text-generation" as task.

[26]:
```
# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False, # False means to not include the prompt text in the␣
 ↪returned text
    max_new_tokens=50,
    do_sample=False, # no randomness in the generated text
)
```

1.3 Generating a Text Response to a Prompt

You'll now use the pipeline object (labeled as generator) to generate a response consisting of 50 tokens to the given prompt.

Note: The model might take around 2 minutes to generate the output.

[27]:
```
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap.␣
 ↪Explain how it happened. "

output = generator(prompt)
```

```
print(output[0]['generated_text'])
```

```
Email to Sarah:

Subject: Sincere Apologies for the Gardening Mishap


Dear Sarah,


I hope this message finds you well. I am writing to express my deepest ap
```

1.4 Exploring the Model's Architecture

You can print the model to take a look at its architecture.

[28]:
```
model
```

[28]:
```
Phi3ForCausalLM(
  (model): Phi3Model(
    (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
    (embed_dropout): Dropout(p=0.0, inplace=False)
    (layers): ModuleList(
      (0-31): 32 x Phi3DecoderLayer(
        (self_attn): Phi3Attention(
          (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
          (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
          (rotary_emb): Phi3RotaryEmbedding()
        )
        (mlp): Phi3MLP(
          (gate_up_proj): Linear(in_features=3072, out_features=16384,
bias=False)
          (down_proj): Linear(in_features=8192, out_features=3072, bias=False)
          (activation_fn): SiLU()
        )
        (input_layernorm): Phi3RMSNorm()
        (resid_attn_dropout): Dropout(p=0.0, inplace=False)
        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
        (post_attention_layernorm): Phi3RMSNorm()
      )
    )
    (norm): Phi3RMSNorm()
  )
  (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
```

```
)
```

The vocabulary size is 32064 tokens, and the size of the vector embedding for each token is 3072.

[29]:
```
model.model.embed_tokens
```

[29]:
```
Embedding(32064, 3072, padding_idx=32000)
```

You can just focus on printing the stack of transformer blocks without the LM head component.

[30]:
```
model.model
```

[30]:
```
Phi3Model(
  (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
  (embed_dropout): Dropout(p=0.0, inplace=False)
  (layers): ModuleList(
    (0-31): 32 x Phi3DecoderLayer(
      (self_attn): Phi3Attention(
        (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
        (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
        (rotary_emb): Phi3RotaryEmbedding()
      )
      (mlp): Phi3MLP(
        (gate_up_proj): Linear(in_features=3072, out_features=16384, bias=False)
        (down_proj): Linear(in_features=8192, out_features=3072, bias=False)
        (activation_fn): SiLU()
      )
      (input_layernorm): Phi3RMSNorm()
      (resid_attn_dropout): Dropout(p=0.0, inplace=False)
      (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
      (post_attention_layernorm): Phi3RMSNorm()
    )
  )
  (norm): Phi3RMSNorm()
)
```

There are 32 transformer blocks or layers. You can access any particular block.

[31]:
```
model.model.layers[0]
```

[31]:
```
Phi3DecoderLayer(
  (self_attn): Phi3Attention(
    (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
    (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
    (rotary_emb): Phi3RotaryEmbedding()
  )
  (mlp): Phi3MLP(
    (gate_up_proj): Linear(in_features=3072, out_features=16384, bias=False)
```

```
    (down_proj): Linear(in_features=8192, out_features=3072, bias=False)
    (activation_fn): SiLU()
  )
  (input_layernorm): Phi3RMSNorm()
  (resid_attn_dropout): Dropout(p=0.0, inplace=False)
  (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
  (post_attention_layernorm): Phi3RMSNorm()
)
```

1.5 Generating a Single Token to a Prompt

You earlier used the Pipeline object to generate a text response to a prompt. The pipeline provides an abstraction to the underlying process of text generation. Each token in the text is actually generated one by one.

Let's now give the model a prompt and check the first token it will generate.

[32]:
```
prompt = "The capital of France is"
```

You'll need first to tokenize the prompt and get the ids of the tokens.

[33]:
```
# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
input_ids
```

[33]:
```
tensor([[ 450, 7483,  310, 3444,  338]])
```

Let's now pass the token ids to the transformer block (before the LM head).

[34]:
```
# Get the output of the model before the lm_head
model_output = model.model(input_ids)
```

The transformer block outputs for each token a vector of size 3072 (embedding size). Let's check the shape of this output.

[35]:
```
# Get the shape the output the model before the lm_head
model_output[0].shape
```

[35]:
```
torch.Size([1, 5, 3072])
```

[36]:
```
# ([1, 5, 3072])
# 1 is batch size
# 5 is number of tokens
# 3072 is number of output tokens
```

The first number represents the batch size, which is 1 in this case since we have one prompt. The second number 5 represents the number of tokens. And finally 3072 represents the embedding size (the size of the vector that corresponds to each token).

Let's now get the output of the LM head.

[37]:
```
# Get the output of the lm_head
lm_head_output = model.lm_head(model_output[0])
```

[38]:
```
lm_head_output.shape
```

[38]:
```
torch.Size([1, 5, 32064])
```

The LM head outputs for each token in the input prompt, a vector of size 32064 (vocabulary size). So there are 5 vectors, each of size 32064. Each vector can be mapped to a probability distribution, that shows the probability for each token in the vocabulary to come after the given token in the input prompt.

Since we're interested in generating the output token that comes after the last token in the input prompt ("is"), we'll focus on the last vector. So in the next cell, `lm_head_output[0,-1]` is a vector of size 32064 from which you can generate the token that comes after ("is"). You can do that by finding the id of the token that corresponds to the highest value in the vector `lm_head_output[0,-1]` (using `argmax(-1)`, -1 means across the last axis here).

[39]:
```
token_id = lm_head_output[0,-1].argmax(-1)
token_id
```

[39]:
```
tensor(3681)
```

Finally, let's decode the returned token id.

[40]:
```
tokenizer.decode(token_id)
```

[40]:
```
'Paris'
```

Download Notebooks: If you'd like to donwload the notebook: 1) click on the "File" option on the top menu of the notebook and then 2) click on "Download as" and select "Notebook (.ipynb)". For more help, please see the "Appendix – Tips, Help, and Download" Lesson.