Exercise: Generating one token at a time

In this exercise, we will get to understand how an LLM generates text--one token at a time, using the previous tokens to predict the following ones.

Step 1. Load a tokenizer and a model

First we load a tokenizer and a model from HuggingFace's transformers library. A tokenizer is a function that splits a string into a list of numbers that the model can understand.

In this exercise, all the code will be written for you. All you need to do is follow along!

In [1]:
```
from transformers import AutoModelForCausalLM, AutoTokenizer

# To load a pretrained model and a tokenizer using HuggingFace, we c
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

# We create a partial sentence and tokenize it.
text = "Udacity is the best place to learn about generative"
inputs = tokenizer(text, return_tensors="pt")

# Show the tokens as numbers, i.e. "input_ids"
inputs["input_ids"]
```

```
tokenizer_config.json: 100%  26.0/26.0 [00:00<00:00, 3.01kB/s]

config.json: 100%  665/665 [00:00<00:00, 101kB/s]

vocab.json: 100%  1.04M/1.04M [00:00<00:00, 11.1MB/s]

merges.txt: 100%  456k/456k [00:00<00:00, 8.76MB/s]

tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 4.75MB/s]

model.safetensors: 100%  548M/548M [00:02<00:00, 210MB/s]

generation_config.json: 100%  124/124 [00:00<00:00, 21.7kB/s]
```

Out[1]:
```
tensor([[  52,   67, 4355,  318,  262, 1266, 1295,  284, 2193,  54
6, 1152,  876]])
```

Step 2. Examine the tokenization

Let's explore what these tokens mean!

In [2]:
```
# Show how the sentence is tokenized
import pandas as pd


def show_tokenization(inputs):
    return pd.DataFrame(
        [(id, tokenizer.decode(id)) for id in inputs["input_ids"][0]],
        columns=["id", "token"],
    )


show_tokenization(inputs)
```

Out[2]:

	id	token
0	tensor(52)	U
1	tensor(67)	d
2	tensor(4355)	acity
3	tensor(318)	is
4	tensor(262)	the
5	tensor(1266)	best
6	tensor(1295)	place
7	tensor(284)	to
8	tensor(2193)	learn
9	tensor(546)	about
10	tensor(1152)	gener
11	tensor(876)	ative

Subword tokenization

The interesting thing is that tokens in this case are neither just letters nor just words. Sometimes shorter words are represented by a single token, but other times a single token represents a part of a word, or even a single letter. This is called subword tokenization.

Step 2. Calculate the probability of the next token

Now let's use PyTorch to calculate the probability of the next token given the previous ones.

In [3]:
```
# Calculate the probabilities for the next token for all possible ch
# top 5 choices and the corresponding words or subwords for these to

import torch

with torch.no_grad():
    logits = model(**inputs).logits[:, -1, :]
    probabilities = torch.nn.functional.softmax(logits[0], dim=-1)


def show_next_token_choices(probabilities, top_n=5):
    return pd.DataFrame(
        [
            (id, tokenizer.decode(id), p.item())
            for id, p in enumerate(probabilities)
            if p.item()
        ],
        columns=["id", "token", "p"],
    ).sort_values("p", ascending=False)[:top_n]


show_next_token_choices(probabilities)
```

Out[3]:

	id	token	p
8300	8300	programming	0.157593
4673	4673	learning	0.148413
4981	4981	models	0.048504
17219	17219	biology	0.046483
16113	16113	algorithms	0.027796

Interesting! The model thinks that the most likely next word is "programming", followed up closely by "learning".

In [4]:
```
# Obtain the token id for the most probable next token
next_token_id = torch.argmax(probabilities).item()

print(f"Next token id: {next_token_id}")
print(f"Next token: {tokenizer.decode(next_token_id)}")
```

```
Next token id: 8300
Next token:  programming
```

In [5]:
```
# We append the most likely token to the text.
text = text + tokenizer.decode(8300)
text
```

Out[5]:
```
'Udacity is the best place to learn about generative programming'
```

Step 3. Generate some more tokens

The following cell will take `text`, show the most probable tokens to follow, and append the most likely token to text. Run the cell over and over to see it in action!

In [6]:
```
# Press ctrl + enter to run this cell again and again to see how the

from IPython.display import Markdown, display

# Show the text
print(text)

# Convert to tokens
inputs = tokenizer(text, return_tensors="pt")

# Calculate the probabilities for the next token and show the top 5
with torch.no_grad():
    logits = model(**inputs).logits[:, -1, :]
    probabilities = torch.nn.functional.softmax(logits[0], dim=-1)

display(Markdown("**Next token probabilities:**"))
display(show_next_token_choices(probabilities))

# Choose the most likely token id and add it to the text
next_token_id = torch.argmax(probabilities).item()
text = text + tokenizer.decode(next_token_id)
```

```
Udacity is the best place to learn about generative programming
```

Next token probabilities:

	id	token	p
13	13	.	0.352222
11	11	,	0.135989
290	290	and	0.109372
287	287	in	0.069530
8950	8950	languages	0.058291

Step 4. Use the `generate` method

In [*]:
```
from IPython.display import Markdown, display

# Start with some text and tokenize it
text = "Once upon a time, generative models"
inputs = tokenizer(text, return_tensors="pt")

# Use the `generate` method to generate lots of text
output = model.generate(**inputs, max_length=100, pad_token_id=toker

# Show the generated text
display(Markdown(tokenizer.decode(output[0])))
```

That's interesting...

You'll notice that GPT-2 is not nearly as sophisticated as later models like GPT-4, which you may have experience using. It often repeats itself and doesn't always make much sense. But it's still pretty impressive that it can generate text that looks like English.

Congrats for completing the exercise! 🎉

Give yourself a hand. And please take a break if you need to. We'll be here when you're refreshed and ready to learn more!