Sheet 3.3: Prompting & Decoding

Author: Polina Tsvilodub & Michael Franke

This sheet provides more details on concepts that have been mentioned in passing in the previous sheets, and provides some practical examples and exercises for prompting techniques that have been covered in lecture four. Therefore, the learning goals for this sheet are:

- take a closer look and understand various decoding schemes,
- understand the temperature parameter,
- see a few practical examples of prompting techniques from the lecture.

Decoding schemes

This part of this sheet is a close replication of this sheet.

This topic addresses the following question: Given a language model that outputs a next-word probability, how do we use this to actually generate naturally sounding text? For that, we need to choose a single next token from the distribution, which we will then feed back to the model, together with the preceding tokens, so that it can generate the next one. This inference procedure is repeated, until the EOS token is chosen, or a maximal sequence length is achieved. The procedure of how exactly to get that single token from the distribution is call *decoding scheme*. Note that "decoding schemes" and "decoding strategies" refer to the same concept and are used interchangeably.

We have already discussed decoding schemes in lecture 02 (slide 25). The following introduces these schemes in more detail again and provides example code for configuring some of them.

> Exercise 3.3.1: Decoding schemes
>
> Please read through the following introduction and look at the provided code.

1. With the help of the example and the documentation, please complete the code (where it says "### YOUR CODE HERE ####") for all the decoding schemes.

Common decoding strategies are:

- **pure sampling**: In a pure sampling approach, we just sample each next word with exactly the probability assigned to it by the LM. Notice that this process, therefore, is non-determinisitic. We can force replicable results, though, by setting a *seed*.
- **Softmax sampling**: In soft-max sampling, the probablity of sampling word w_i is $P_{LM}(w_i \mid w_{1:i-1}) \propto \exp\left(\frac{1}{\tau} P_{LM}(w_i \mid w_{1:i-1})\right)$, where τ is a *temperature parameter*.
  - The *temperature parameter* is also often available for closed-source models like the GPT family. It is often said to change the "creativity" of the output.
- **greedy sampling**: In greedy sampling, we don't actually sample but just take the most likely next-word at every step. Greedy sampling is equivalent to setting $\tau = 0$ for soft-max sampling. It is also sometimes referred to as *argmax* decoding.

- **beam search**: In simplified terms, beam search is a parallel search procedure that keeps a number k of path probabilities open at each choice point, dropping the least likely as we go along. (There is actually no unanimity in what exactly beam search means for NLG.)
- **top-k sampling**: his sampling scheme looks at the k most likely next-words and samples from so that:

$$P_{\text{sample}}(w_i \mid w_{1:i-1}) \propto \begin{cases} P_M(w_i \mid w_{1:i-1}) & \text{if } w_i \text{ in top-}k \\ 0 & \text{otherwise} \end{cases}$$

- **top-p sampling**: Top-p sampling is similar to top-k sampling, but restricts sampling not to the top-k most likely words (so always the same number of words), but the set of most likely words the summed probability of which does not exceed threshold p.

The within the `transformers` package, for all causal LMs, the `.generate()` function is available which allows to sample text from the model (remember the brief introduction in sheet 2.5). Configuring this function via different values and combinations of various parameters allows to sample text with the different decoding schemes described above. The respective documentation can be found here. The same configurations can be passed to the `pipeline` endpoint which we have seen in the same sheet.

Check out this blog post for very noce visualizations and more detials on the *temperature* parameter.

Please complete the code below. GPT-2 is used as an example model, but this works exactly the same with any other causal LM from HF.

```
# import relevant packages
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# load the tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# add the EOS token as PAD token to avoid warnings
model = GPT2LMHeadModel.from_pretrained("gpt2",
pad_token_id=tokenizer.eos_token_id)

# convenience function for nicer output
def pretty_print(s):
    print("Output:\n" + 100 * '-')
    print(tokenizer.decode(s, skip_special_tokens=True))

# encode context the generation is conditioned on
input_ids = tokenizer.encode('I enjoy walking with my cute dog',
return_tensors='pt')

# set a seed for reproducibility (if you want)
torch.manual_seed(199)

# below, greedy decoding is implemented
```

```
# NOTE: while it is the default for .generate(), it is NOT for
pipeline()

greedy_output = model.generate(input_ids, max_new_tokens=10)
print(pretty_print(greedy_output[0]))

# here, beam search is shown
# option `early_stopping` implies stopping when all beams reach the
end-of-sentence token
beam_output = model.generate(
    input_ids,
    max_new_tokens=10,
    num_beams=3,
    early_stopping=True
)

pretty_print(beam_output[0])


#  pure sampling
sample_output = model.generate(
    input_ids,        # context to continue
    #### YOUR CODE HERE ####
    max_new_tokens=10, # return maximally 10 new tokens (following the
input)
)

pretty_print(sample_output[0])

# same as pure sampling before but with `temperature`` parameter
SM_sample_output = model.generate(
    input_ids,        # context to continue
    #### YOUR CODE HERE ####
    max_new_tokens=10,
)

pretty_print(SM_sample_output[0])

# top-k sampling
top_k_output = model.generate(
    input_ids,
    ### YOUR CODE HERE ####
    max_new_tokens=10,
)

pretty_print(top_k_output[0])

# top-p sampling
top_p_output = model.generate(
    input_ids,
```

```
    ### YOUR CODE HERE ####
    max_length=50,
)

pretty_print(top_p_output[0])
```

Exercise 3.3.2: Understanding decoding schemes

Think about the following questions about the different decoding schemes.

1. Why is the temperature parameter in softmax sampling sometimes referred to as a creativity parameter? Hint: Think about the shape distribution and from which the next word is sampled, and how it compares to the "pure" distribution when the temperature parameter is varied.
2. Just for yourself, draw a diagram of how beam decoding that starts with the BOS token and results in the sentence "BOS Attention is all you need" might work, assuming k=3 and random other tokens of your choice.
3. Which decoding scheme seems to work best for GPT-2?
4. Which of the decoding schemes included in this work sheet is a special case of which other decoding scheme(s)? E.g., X is a special case of Y if the behavior of Y is obtained when we set certain paramters of X to specific values.
5. Can you see pros and cons to using some of these schemes over others?

Outlook

There are also other more recent schemes, e.g., locally typical sampling introduced by Meister et al. (2022).

Prompting strategies

The lecture introduced different prompting techniques. (Note: "prompting technique" and "prompting strategy" refer to the same concept and are used interchangeably) Prompting techniques refer to the way (one could almost say -- the art) of constructing the inputs to the LM, so as to get optimal outputs for your task at hand. Note that prompting is complementary to choosing the right decoding scheme -- one still has to choose the decoding scheme for predicting the completion, given the prompt constructed via a particulat prompting strategy.

Below, a practical example of a simple prompting strategy, namely *few-shot prompting* (which is said to elicit *in-context learning*), and a more advanced example, namely *generated knowledge prompting* are provided. These should serve as inspiration for your own implementations and explorations of other prompting schemes out there. Also, feel free to play around with the examples below to build your intuitions! Of course, you can also try different models, sentences, ...

Note

You might have already experienced rate limits of accessing the GPU on Colab. To try to avoid difficulties with completing the tasks on GPU, if you want to use Colab, here are a few potential aspects (approximated by experience, definitely non-exhaustive and inofficial) that might lead to rate limits: requesting GPU runtimes and then not utilizing the GPU, requesting a lot of GPU

runtimes (e.g., multiple per day), running very long jobs (multiple hours). To try to work around this, one possibility is to debug and test code that doesn't require GPUs in non-GPU runtimes, and only request those when actually needed.

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch
import pandas as pd
import numpy as np

# define computational device
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Device: {device}")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
    print(f"Device: {device}")
else:
    device = torch.device("cpu")
    print(f"Device: {device}")


tokenizer = AutoTokenizer.from_pretrained("EleutherAI/Pythia-1.4b")
model = AutoModelForCausalLM.from_pretrained(
    "EleutherAI/Pythia-1.4b",
    # trust_remote_code=True,
    torch_dtype=torch.float16,
).to(device)

# few shot prompting

few_shot_prompt = """
Input: This class is awesome. Sentiment: positive
Input: This class is terrible. Sentiment: neutral
Input: The class is informative. Sentiment: neutral
"""
input_text = "The class is my favourite!"

full_prompt = few_shot_prompt + "Input: " + input_text + " Sentiment:
"

input_ids = tokenizer(full_prompt,
return_tensors="pt").input_ids.to(device)
few_shot_prediction = model.generate(
    input_ids,
    max_new_tokens=10,
    do_sample=True,
    temperature=0.4,
)

print(tokenizer.decode(few_shot_prediction[0],
skip_special_tokens=False))
```

Example of generated knowledge prompting (somewhat approximated, based on code from this class), as introduced by Liu et al. (2022). This prompting technique is used to answer this multiple-choice question from the CommonsenseQA benchmark: "Where would you expect to find a pizzeria while shopping?". The answer options are: A = ["chicago", "street", "little italy", "food court", "capital cities"]

As a reminder, the overall idea of generated knowledge prompting is the following:

- knowledge generation: given question Q and a few-shot example, generate a set K_Q of k knowledge statements
  - we will load the few-shot examples from a csv file here.
- knowledge integration: given Q and K_Q, retrieve the log probabilities of each answer option $a_i \in A$ and select the option with the highest probability.
  - in the paper, this is done separately for each knowledge statement in K_Q. As a simplification, we will concatenate all K_Q and compare the answer options given this combined prompt.

```
# 1. construct few-shot example

question = "Where would you expect to find a pizzeria while shopping?"
answers = ["chicago", "street", "little italy", "food court", "capital
cities"]

examples_df = pd.read_csv("files/knowledge_examples.csv", sep = "|")

few_shot_template = """{q} We know that {k}"""

few_shot_prompt = "\n".join([
    few_shot_template.format(
        q=examples_df.loc[i, "input"],
        k=examples_df.loc[i, "knowledge"].lower()
    )
    for i in range(len(examples_df))
])
print("Constructed few shot prompt\n", few_shot_prompt)

# 2. generate knowledge statements
# tokenize few shot prompt together with our actual question
prompt_input_ids = tokenizer(
    few_shot_prompt + "\n" + question + " We know that ",
    return_tensors="pt"
).input_ids.to(device)

knowledge_statements = model.generate(
    prompt_input_ids,
    max_new_tokens=15,
    do_sample=True,
    temperature=0.5
)
# access the knowledge statements (i.e., only text that comes after
```

```
prompt)
knowledge = tokenizer.decode(
    knowledge_statements[0, prompt_input_ids.shape[-1]:],
    skip_special_tokens=True
)
print(tokenizer.decode(knowledge_statements[0]))
print("Generated knowledge ", knowledge)

# 3. Score each answer to the question based on the knowledge
statements
# as the score, we take the average log probability of the tokens in
the answer

answer_log_probs = []
# iterate over the answer options
# NOTE: This can take a moment
for a in answers:
    # construct the full prompt
    prompt = f"{knowledge} {question} {a}"
    # construct the prompt without the answer to create a mask which
will
    # allow to retrieve the token probabilities for tokens in the
answer only
    context_prompt = f"{knowledge} {question}"
    # tokenize the prompt
    input_ids = tokenizer(prompt,
                          return_tensors="pt").input_ids.to(device)
    # tokenize the context prompt
    context_input_ids = tokenizer(context_prompt,
                                  return_tensors="pt").input_ids
    # create a mask with -100 for all tokens in the context prompt
    # the -100 indicates that the token should be ignored in the loss
computation
    masked_labels = torch.ones_like(input_ids) * -100
    masked_labels[:, context_input_ids.shape[-1]:] = input_ids[:,
context_input_ids.shape[-1]:]
    print("Mask ", masked_labels)
    # generate the answer
    preds = model(
        input_ids,
        labels=masked_labels
    )
    # retrieve the average log probability of the tokens in the answer
    log_p = preds.loss.item()
    answer_log_probs.append(-log_p)
    print("Answer ", a, "Average log P ", log_p)

# 4. retrieve the answer option with the highest score
# find max probability
print("All answers ", answers)
```

```
print("Answer probabilities ", answer_log_probs)
max_prob_idx = np.argmax(answer_log_probs)
print("Selected answer ", answers[max_prob_idx], "with log P ",
answer_log_probs[max_prob_idx])
```

Exercise 3.3.3: Prompting techniques

For the following exercises, use the same model as used above.

1. Using the code for the generated knowledge approach, score the different answers to the question *without* any additional knowledge. Compare your results to the result of generated knowledge prompting. Did it improve the performance of the model?

2. Implement an example of a few-shot chain-of-thought prompt.
3. Try to vary the few-shot and the chain-of-thought prompt by introducing mistakes and inconsistencies. Do these mistakes affect the result of your prediction? Feel free to use any example queries of your choice or reuse the examples above.

Outlook

As always, here are a few optional resources on this topic to llok at (although there is definitely much more online):

- a prompting webbook providing an overview of various approaches
- a framework / package, LangChain, which provides very useful utilities for more complex schemes like tree of thought prompting (spoiler: we will look closer at this package in future sessions, but you can already take a look if you are curious!)