Advanced Methods in Text Analytics
Exercise 8: Large Language Models - Part 2
Solutions

Daniel Ruffinelli

FSS 2025

1 Making Predictions with LLMs

(a)
- GPT3-Neo 1.3B: vocab. size 50257, 24 layers with hidden size 2048, 24 layers, MLP up projection to 8192 and GELU activation.
- Llama3.2-1B: vocab size 128256, 16 layers with hidden size 2048, MLP up projection to 8192 and Swish activation.

(b) The output we get from the tokenizer (`__call__`) is a `BatchEncoding` object (read details here). This is a dictionary with keys `input_ids` and `attention_mask`. The former contains the list of token ids used to encode the prompt, given as a tensor of size batch size $\times$ sequence length, where batch size is typically because we feed models multiple input sequences at once during training (i.e. in this case batch size = 1). So, computing the length of the last dimension gives us the number of tokens used to encode the prompt.

The `attention_mask` component is a tensor of the same shape, where each element is 1 if the corresponding token should attended to by the model, and 0 otherwise. This is commonly used for things like padding tokens, which we don't want the model to attend to, e.g. when computing training loss.

(c) The output we get from the model is a `CausalLMOutputWithPast` object (read details here). This is a dictionary with keys `logits` and `past_key_values`. The former contains the logits of the model in a tensor of size batch size $\times$ input sequence length $\times$ vocab size. So, as expected from an autoregressive language model, we have a distribution over the vocabulary for each token in the input sequence, and we need the distribution corresponding to the last token to make our next-token prediction.

The `past_key_values` components contains pre-computed keys and values in self-attention, useful to more efficiently decode tokens at each step (decoding step $n + 1$ uses many of the same keys and values used by the model in decoding step n). The object contains this information for each of the layers in the model.

(d) See code for implementation. We used sorting to keep the solution more general, but if the top k elements is all you need, then using torch.topk is a more efficient solution. The top k tokens make sense given the prompt. E.g. we can see that the top token is a new line token, possibly indicating the model wants to start a new sentence in a new line. Given the prompt *Hello?*, we also see words like *Hello*, *you*, *are*, *my*, etc. The same holds for both models. This function is useful for sampling, e.g. greedy, top k.

2 Prompting

(a) All prompts we test in the code here are natural language questions, e.g. *What is the capital of France?*. If we look at the top k tokens, the answer is never the top token, nor is it in the top k tokens (exception: GPT3 does have *Paris* when asked the question above). As before, the top token in GPT3 is always a new line, and for both models, the top tokens seems similar to one another, e.g. words like *what* or *how*, but not necessarily similar to the expected answer. But even if the answer were in the top k tokens, how could we use it to consistently make a good prediction? In factual questions, we want a deterministic answer. This is where prompting comes in.

Autoregressive models are designed to generate sequences, so unless specifically asked for, it's not fair to ask models to give a single expected answer in the first token they produce. How do we specifically ask for that? One simple way that works since the days of GPT3 is an ICL-like prompt. Then the top token is the correct answer. Note that this too is brittle because of tokenization issues, e.g. Rome is often tokenized with a space, so we need to add the space to the prompt to see the correct answer, and this is not consistent across models or examples. Try to out!

(b) ICL prompts typically have three components: instructions of the task, set of demonstrations of the task, and the question we ask the model. The code produces prompts that clearly show these components. Note that in the code we have two different *templates* to construct ICL prompts, and that using instructions is optional, as is typically the case in ICL settings. Finally, we are randomizing the set and order of demonstrations, so that running the code multiple times produces different prompts.

(c) Most tasks are not clear unless we provide demonstrations. This is true even if we asked humans to solve the tasks. When adding demonstrations, we see the models being able to predict the expected answer most of the time, but this is still (as always) brittle to the chosen prompt. In particular, we can see here that the choice of template makes a difference in the top token predicted, since one of the templates requires a leading whitespace, which can result in models producing a single token with the correct answer (recall that many words end up tokenized with a leadins whitespace). Additionally, the translating task is interesting because while Llama3.2 does support French officially, GPT-3 does not.

(d) Using natural language instructions does help in a zero-shot setting. For example, adding the question *What is the capital of the following country?* results in both GPT3 and Llama returning the correct answer in the top token even without using demonstrations. However, the choice of prompt is important. For example, asking Llama to simply *Translate to French* does not work, while asking it to *Translate the following word to French* does.

(e) Given that we were already able to construct prompts that result in models understanding the task, either via demonstrations or natural language instructions, the main difference we see with an instruction tuned model is that it can better handle natural language questions. E.g. now when asked *What is the capital of France?*, the top token is indeed the correct answer, which we did not get with previous models unless we used clever ICL prompts. Do you find any other differences? Try all of the ICL tasks provided in the code. More generally, we are still looking at top k tokens and trying to use greedy decoding, but autoregressive mdoels are designed to generate sequences longer than a single token, and instruction tuned models (often referred to as *chat* models) are designed to generate longer sequences in a more conversational way. So, we explore decoding longer sequences in the next task.

3 Generating Longer Responses

(a) See code.

(b) If we go back to our natural language questions, e.g. *What is the capital of France?* and decode a sequence of several tokens, we now see the models were indeed sometimes trying to generate coherent sentences that we could not see by simply inspecting the top tokens produced in the first decoding step. However, we also get to see interesting things about these models in this case. For example, we see that if we keep decoding tokens, the models just "keep talking", often revealing the type of data used to train them (they may reveal a multiple choice type of setting). Also, Llama3.2-1B is able to answer the question about the capital of France, but it may reproduce the question first, which is an unintuitive things to do (though this does not really happen with the instruction-tuned model). GPT-3 is very sensitive to whether we include a whitespace at the end of the prompt or not. If not, GPT-3 will often produce an endless sequence of new lines, but if we include the whitespace, it sometimes produces more coherent sequences. Try that with the simple prompts *"Hello?"* and *"Hello? "*.

(c) Indeed different sampling methods make a difference, as expected. Try it for both prompts that expect a factual answer as well as prompts that have more open-ended tasks. In particular, it is interesting to note that it's difficult for the instruct variant of Llama3.2 to produce an incoherent sentence, but when increasing the temperature in random sampling, we do get the model to rant about relevant things instead of answerinng the factual question we ask, e.g. *What is the capital of France?*.

(d) Here we can get an idea of how to use pre-trained LLMs to construct chatbots. We set up a system prompt, i.e. instructions given to the model that the user does not see, and we keep feeding the model a new prompt that contains the system prompt, the chat history and the new dialogue entry from the user. Recall that these models do not possess long-term memory, which is why there has been significant effort in recent years to increase the size of the context window in LLMs.

Try different system prompts. Do all models follow the instructions? How about differences with instruction-tuned models? For example, instruction-tuned models seem to be capable of knowing when to stop generating the sequence (perhaps by producing the end-of-sequence EOS token), but if asked the non instructon-tuned Llama to generate many tokens in this setting, it will just complete a dialogue between the user and the assistant.