Chapter 8

Generation

The traditional generation task involved mapping some kind of semantic representation to text. As we've seen, defining a semantic representation is difficult, prompting the question, "Generation from what?!" (Wilks). In this chapter, we're going to sidestep this question by focusing exclusively on generation from vector representations.

8.1 What to generate?

First I'd like to take a step back to think about what generation is supposed to do. Suppose we have some model $\hat{P}(y \mid x)$ that is an estimate of some true distribution $P(y \mid x)$. For now, you can assume that x is some user prompt (like a question) and y is the machine's response.

What is this true distribution supposed to be? All human beings? A particular (real or imaginary) human being? An omniscient person?

Supposing that we can estimate $\hat{P}(y \mid x)$ perfectly, how do we generate responses from it, given x? Do we sample randomly from $\hat{P}(y \mid x)$, or do we choose $\arg\max_y \hat{P}(y \mid x)$, or something else?

To see why these are hard questions, consider how we would want a computer to respond to:

- What is your name?
- Please tell me a joke.
- What is the 123,456,789th digit of π?
- Please write me a sonnet on the subject of the Forth Bridge.
- Will a baseball dropped on the moon fall down?[1]

I don't have a good answer to the question of the true distribution; here, we just assume that we are given data drawn from somewhere. As for the question

[1]In a survey of 305 respondents, mostly studying to become elementary school teachers, only 32.8% correctly answered "yes" (Stein et al., "A study of common beliefs and misconceptions in physical science", *Journal of Elementary Science Education* 20:2, 2008, pages 1–11).

of what y we should choose, it seems that there are some tasks (machine translation, factual question answering) where we want the argmax, and some (creative writing) where we want to sample, and some tasks where we might want something in between. In the next section, we'll see some additional technical reasons that contribute to this question.

8.2 Generation from language models

We've already seen an example of a model for generation, as part of machine translation. We used an encoder–decoder for machine translation, but here we're going to consider language models based on decoder-only transformers.

8.2.1 Model

The input to the model is a prefix $\texttt{BOS} \cdot w_1 \cdots w_{t-1}$, and the output is a distribution over the next word, $P(w_t)$.

$$\begin{aligned}
\mathbf{V}_0, \ldots, \mathbf{V}_{t-1} &= \text{Embedding}^{\boxed{1}}(\texttt{BOS} \cdot w_1 \cdots w_{t-1}) \\
\mathbf{H}_0, \ldots, \mathbf{H}_{t-1} &= \text{Transformer}^{\boxed{2}}(\mathbf{V}_0, \ldots, \mathbf{V}_{t-1}) \\
\mathbf{y}^{(t)} &= \log(\text{softmax}(\text{LinearLayer}^{\boxed{3}}(\mathbf{H}_{t-1}))).
\end{aligned}$$

The transformer uses masked self-attention. Then $\mathbf{y}^{(t)}$ is the vector of log-probabilities $\log P(w_t)$.

Now suppose you want to ask the model a question $x = x_1 \cdots x_m$. The model doesn't immediately give a distribution over answers y; it only gives distributions over next words. The mathematically most correct way to get a distribution over answers would be

$$\begin{aligned}
P(y \mid x) = {} & P(y_1 \mid \texttt{BOS} \cdot x_1 \cdots x_m) \\
& \cdot P(y_2 \mid \texttt{BOS} \cdot x_1 \cdots x_m \cdot y_1) \\
& \vdots \\
& \cdot P(y_n \mid \texttt{BOS} \cdot x_1 \cdots x_m \cdot y_1 \cdots y_{n-1}) \\
& \cdot P(\texttt{EOS} \mid \texttt{BOS} \cdot x_1 \cdots x_m \cdot y_1 \cdots y_n)
\end{aligned}$$

where $n = |y|$. (Perhaps there could be some separator symbol between x_m and y_1.)

8.2.2 Ancestral sampling

Then there are two natural choices for an algorithm to choose an answer. First, we could randomly choose one. This algorithm is called *ancestral sampling*:

- $w \leftarrow \texttt{BOS} \cdot x$
- while w does not end in $\texttt{EOS}$

  - sample a from $P(a \mid w)$
  - $w \leftarrow w \cdot a$
- output w

But, in addition to the problems alluded to above (when there is a correct answer, the user usually wants the correct answer and not a random answer), the randomly-generated answer will be fairly incoherent. The following example is from Holtzman et al. (2020) using GPT-2:

- Prompt: `In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.`
- Response: `They were cattle called Bolivian Cavalleros; they live in a remote desert uninterrupted by town, and they speak huge, beautiful, paradisiacal Bolivian linguistic thing. They say, 'Lunch, marge.' They don't tell what the lunch is," director Professor Chuperas Omwell told Sky News. "They've only been talking to scientists, like we're being interviewed by TV reporters. We don't even stick around to be interviewed by TV reporters. Maybe that's how they figured out that they're cosplaying as the Bolivian Cavalleros."`

Recall that to avoid overfitting, and in particular to avoid assigning a zero probability to any word, we applied *smoothing* to n-gram language models, adding some probability to words that have never been seen in a given context. Neural language models don't require smoothing, but there are various factors that achieve a similar effect:

- The next-word distribution is a softmax, which cannot output zero
- The model works with vectors whose size is smaller than the vocabulary size
- The training of the model is stopped early, when perplexity on a validation set is minimized

The result is that the model assigns a too-high probability to weird choices. That makes ancestral sampling too willing to actually make a weird choice (e.g., *cattle*), and the more weird words get added to the context, the weirder future choices will become.

8.2.3 Exact search and beam search

How about the alternative, choosing the most probable y? First of all, this is in general an NP-hard problem (Higuera and Oncina, 2013). It's possible to use heuristics to solve it slowly (Stahlberg and Byrne, 2019), but in practice we always use an approximate search, *beam search*.

The idea of beam search is to generate a string from left-to-right, and at each time step, we keep only the best k strings so far:

```
• $b^{(i)} \leftarrow \texttt{BOS} \cdot x$ for $i = 1, \ldots, k$
• while any $b^{(i)}$ does not end in EOS
    – $B \leftarrow \emptyset$
    – for $i = 1, \ldots, k$
        * if $b_i$ ends in EOS, $B \leftarrow B \cup \{b^{(i)}\}$
        * else, $B \leftarrow B \cup \{b^{(i)} \cdot a \mid a \in \Sigma\}$
    – $b^{(1)}, \ldots, b^{(k)} \leftarrow$ the best $k$ members of $B$
• Output the best of $\{b^{(i)} \mid i = 1, \ldots, k\}$
```

This is what's typically used in machine translation (with typical k being quite small, like $k = 4$). But this suffers from problems of its own. The best output may be empty, or highly repetitive, or a copy of x.

Why empty? Since the probability of a string is the product of the probability of the words, and multiplying probabilities always makes them smaller, it's easy for shorter strings to have higher probability than longer strings. Even if the model knows that $P(\texttt{EOS} \mid \texttt{BOS})$ is low, it needs to be lower than the probability of the whole correct string, and that's not easy to do (Murray and Chiang, 2018).

Why a copy? Even if the data has just a few examples of copying in it, the fact that there's only one way to copy a sentence but many ways to actually respond to it, the probability of a copy may be higher than the probability of any legitimate response (Ott et al., 2018).

As an example of repetition, beam search for the above unicorn prompt gives (Holtzman et al., 2020):

- Response: `"The study, published in the Proceedings of the National Academy of Sciences of the United States of America (PNAS), was conducted by researchers from the Universidad Nacional Autónoma de México (UNAM) and the Universidad Nacional Autónoma de México (UNAM/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de ..."`

8.2.4 Compromises

In machine translation, the usual fix is to do something simple like divide the log-probability of a translation by its length (Koehn and Knowles, 2017). Additionally, the fact that beam search is approximate turns out to be helpful, so keeping k low is good. The special case $k = 1$ is called *greedy search*, which you will be asked to implement:

```
• $w \leftarrow \texttt{BOS} \cdot x$
• while $w$ does not end in EOS
    – $a \leftarrow \arg\max_a P(a \mid w)$
    – $w \leftarrow w \cdot a$
```

- output w

But in other settings, the common practice is to use some kind of compromise between ancestral sampling and greedy search. All of the following affect only the line that chooses a:

- Sample a from $P(a \mid w)$, but modify the model *during generation only* so that
$$\mathbf{o}^{(t)} = \log(\text{softmax}(\text{LinearLayer}^{\boxed{3}}(\mathbf{H}_{t-1})/T))$$
where T is called the *temperature* (and in fact corresponds to the temperature in a Boltzmann distribution). At $T = 1$, this is ancestral sampling, and as T approaches 0, this becomes greedy search. (Warning: sometimes people incorrectly call $1/T$ the temperature.)

- Top-k sampling: Assume a fixed $k > 0$. Let $a_1, \ldots, a_k$ be the symbols that have the top k values of $P(a \mid w)$, then sample a from the distribution
$$\text{Top}(a) = \frac{P(a \mid w)}{\sum_{i=1}^{k} P(a_i \mid w)} \qquad (a \in \{a_1, \ldots, a_k\}).$$

- Nucleus or top-p sampling (Holtzman et al., 2020): Assume a fixed $p > 0$. Let $a_1, \ldots, a_k$ be the smallest subset of Σ such that $\sum_{i=1}^{k} P(a_i \mid w) \geq p$. (That is, sort the alphabet in decreasing order according to $P(a \mid w)$, and go down the list until the total probability is p or more.) Then sample a from the distribution $\text{Top}(a)$ as above.

All of these methods work reasonably well in practice, but a truly principled account of how to sample from language models is still a topic of research.

8.3 Reinforcement learning with human feedback

A language model trained on a mountain of text can be used for completing texts, but isn't necessarily great at particular applications, like dialogue. Questions or requests aren't all that frequent in most texts, and their completions are often not direct responses to those questions or requests. ChatGPT is further trained using *reinforcement learning with human feedback* or RLHF (Ouyang et al., 2022).[2]

The short but inaccurate story is: Define $r_{\text{human}}(x, y)$ to be a numeric score that a human would assign to a response y for a prompt x. Let ϕ be the parameters of the language model. We train the language model using maximum-likelihood to get parameter values ϕ_0, and then we want to further fine-tune the language model by maximizing the following objective function:

$$L = \sum_{w} \gamma \log P_\phi(w) + \sum_{x} \sum_{y} P_\phi(y \mid x)\, r_{\text{human}}(x, y). \tag{8.1}$$

The first term is the usual log-likelihood; the summation over w is over the original training data. The second term is the expected score: the summation over x is

[2]This blog post is also helpful: `https://huyenchip.com/2023/05/02/rlhf.html`

over some collection of prompts, and the summation over y is theoretically over all possible responses.

This creates a chain of problems. First, the summation over y is infinite. Instead, we approximate this sum by randomly generating some y's from the language model, $P_\phi(y \mid x)$. This approximation is typical in reinforcement learning. For example, to train a computer to play a video game, you can define a probability distribution over sequences of moves, but you can't maximize the expected final score. Instead, you have the computer play the game, reward it according to the final score, and repeat, and hopefully the computer will play better and better each time.

But this still isn't practical, because in RLHF the rewards are assigned by humans. Even industrial research groups apparently have no more than 1M judgements, which is not enough. So we create another model, called the *reward model*, $r_\theta(x, y)$, to mimic $r_{\text{human}}(x, y)$. The subscript θ stands for the parameters of the reward model. The idea is to train this model on the human-generated examples, then plug this model into (8.1):

$$L = \sum_w \gamma \log P_\phi(w) + \sum_x \sum_y P_\phi(y \mid x)\, r_\theta(x, y). \tag{8.2}$$

Note that θ is kept fixed while maximizing L.

The next problem is that humans actually aren't that good at assigning numeric scores. It's much more reliable to present a human with two responses y and y' and choose which one is better (a *pairwise judgement*). So wrap the reward model inside another model that chooses between two responses:

$$P(y \text{ better than } y') = \text{sigmoid}(r_\theta(x, y) - r_\theta(x, y')).$$

Then given a collection of human pairwise judgements, we can learn a θ that maximizes their log-likelihood. This is the same method used to compute Elo ratings of chess players. Every response y is a "player," every pairwise judgement is a "match" between y and y', and the response that is judged to be better is the "winner." In a step of gradient ascent, if y is the "winner," its "rating" $r_\theta(x, y)$ goes up, and if y' is the "loser," its "rating" $r_\theta(x, y')$ goes down.

The final problem is that the reward model isn't perfect. You may be familiar with examples of images of (say) dogs that have been manipulated to fool a neural network into classifying it as (say) a cat. Similarly, there's a danger that the above training will just train the language model (P_ϕ) to generate bad outputs that trick the reward model (r_θ) into saying they are good. So a third term is added to the objective function (8.2) to keep the model close to the original (pre-RLHF) model:

$$L = \sum_w \gamma \log P_\phi(w) + \sum_x \sum_y P_\phi(y \mid x)\, r_\theta(x, y) - \beta\, \text{KL}[P_\phi \| P_{\phi_0}]$$

where KL is the *Kullback-Leibler divergence*, a measure of how different two probability distributions are. That, finally, is the objective function used to fine-tune InstructGPT and (as far as I know) ChatGPT.

8.4 Limitations of language models

Current research on language models for generation focuses on a number of limitations:

- *Hallucination*, where generated text is fluent but factually incorrect.
- *Sycophany*, where the model is too willing to agree with the user.
- Limited ability to plan text (e.g., generating a sentence that ends with a specified word)

Here, I would like to focus on limitations of transformers at solving reasoning tasks.

Recall that a transformer decoder takes as input words $w_1 \cdots w_{t-1}$ and outputs a probability distribution over the next word, $P(w_t \mid w_1 \cdots w_{t-1})$. The computation of this probability distribution has a fixed *depth*. If the computation is thought of as a graph where each node has a value that is a function of the values of its in-neighbor nodes, then the length of the longest path from an input node to an output node is called the depth of the computation graph, and in a transformer this depth is independent of t. (By contrast, in an RNN, there is a path that goes from w_1 through $\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(t)}$ to the output, so the depth is $O(t)$.

This implies that if you ask the language model a question and expect it to answer immediately, there is a limit on how computationally difficult the question can be. For example, if x is an addition problem like `123+123=`, you know from doing grade-school addition that generating each digit of the answer requires a fixed amount of work (add two digits and possibly a carried 1). So you might expect transformers to be able to learn to do addition, and they can. But if x is a multiplication problem like `123*123=`, you know that grade-school multiplication requires more work:

```
  123
 ×123
-----
  369
 246
123
-----
15129
```

In total there are $O(|x|^2)$ steps, so generating each digit of y takes an average of $O(|x|)$ steps. So it should come as no surprise that transformers seem to be unable to learn to multiply numbers beyond a few digits.

Other examples (Wei et al., 2022) include:

- The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
- Is the following sentence plausible? "Joao Moutinho caught the screen pass in the NFC championship."
- A coin is heads up. Maybelle flips the coin. Shalonda does not flip the coin. Is the coin still heads up?

The currently popular solution is called *chain-of-thought* or a *scratchpad*: if you don't require the answer to appear immediately, but allow intermediate symbols to be generated first, then these symbols effectively increase the depth of the computation, increasing the model's reasoning ability. Continuing with the multiplication example, perhaps it helps to write (as in grade-school multiplication) the numbers `369+246+123=` before generating the final answer `15129`. Formally, allowing intermediate symbols makes transformers as powerful as Turing machines (Pérez, Barceló, and Marinkovic, 2021). Practically, this has been shown to actually improve performance on reasoning tasks. In principle we could fine-tune the model to generate intermediate symbols, but the more common approach is to design prompts with instructions like "Let's think step by step" and/or demonstrations of thinking step-by-step. For example (Wei et al., 2022):

> Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
>
> A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
>
> Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

How to make transformers learn to take intermediate steps *silently* is therefore an important goal for current research.

References

Higuera, Colin de la and Jose Oncina (2013). "Computing the Most Probable String with a Probabilistic Finite State Machine". In: *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing (FSMNLP)*, pp. 1–8. URL: https://aclanthology.org/W13-1801.

Holtzman, Ari et al. (2020). "The Curious Case of Neural Text Degeneration". In: *Proc. ICLR*.

Koehn, Philipp and Rebecca Knowles (2017). "Six Challenges for Neural Machine Translation". In: *Proceedings of the First Workshop on Neural Machine Translation (WNMT)*, pp. 28–39. DOI: 10.18653/v1/W17-3204. URL: https://aclanthology.org/W17-3204.

Murray, Kenton and David Chiang (2018). "Correcting Length Bias in Neural Machine Translation". In: *Proceedings of the Third Conference on Machine Translation (WMT)*, pp. 212–223. DOI: 10.18653/v1/W18-6322. URL: https://aclanthology.org/W18-6322.

Ott, Myle et al. (2018). "Analyzing Uncertainty in Neural Machine Translation". In: *Proc. ICML*, pp. 3956–3965. URL: https://proceedings.mlr.press/v80/ott18a.html.

Ouyang, Long et al. (2022). "Training language models to follow instructions with human feedback". In: *Advances in Neural Information Processing Systems*. Vol. 35, pp. 27730–27744. URL: https://proceedings.neurips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

Pérez, Jorge, Pablo Barceló, and Javier Marinkovic (2021). “Attention is Turing-Complete”. In: *J. Mach. Learn. Res.* 22, 75:1–75:35. URL: `http://jmlr.org/papers/v22/20-302.html`.

Stahlberg, Felix and Bill Byrne (2019). “On NMT Search Errors and Model Errors: Cat Got Your Tongue?” In: *Proc. EMNLP-IJCNLP*. Ed. by Kentaro Inui et al., pp. 3356–3362. DOI: `10.18653/v1/D19-1331`. URL: `https://aclanthology.org/D19-1331`.

Wei, Jason et al. (2022). “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Proc. NeurIPS*. URL: `https://arxiv.org/pdf/2201.11903.pdf`.