We've spent a lot of time figuring out how better to estimate probability distributions p_θ over sequences of text. In this note, we'll go over strategies for taking this distribution and generating text. These strategies will depend on the kind of text we're generating, and what errors we think exist in the distribution our model has learned.

Sampling and finding the argmax

We've trained our model autoregressively, conditioning on tokens used as input (and those we've generated so far) and predicting the next token. We'll sample from it similarly. Let $x_1, \ldots, x_k$ be an input token sequence, and $y_1, \ldots, y_{t-1}$ be a (possibly empty) string of output tokens generated so far. For brevity, we will denote these sequences as x and $y_{<t}$ respectively. We sample a next token as:

$$y_t \sim p_\theta(y_t \mid x, y_{<t}) \qquad \text{(sampling, or ancestral sampling)} \tag{1}$$

The sampling process is a loop of sampling subsequent y until a stopping condition is met — either a special `end of text` token is generated, or a maximum sequence length is met.

Overall, as models become increasingly strong, and as we clean the data they're trained on, this simple sampling method becomes better and better. Intuitively, we work really hard to estimate the distribution p_θ, and the better it is, the less we need to do afterward to "fix" it. But consider that some text is noisy, our model isn't perfect, and the model likely spreads some probability mass over output sequences that aren't what we want. Assuming our model is good, we might instead want find the *most likely* output under the model:

$$y_{1:T} = \arg\max_{y \in \mathcal{V}*} p_\theta(y \mid x) \qquad \text{(argmax)} \tag{2}$$

This is much easier to write out than it is to compute. Whereas during sampling, we run the model's forward pass once per token we end up generating, finding the argmax of our model is a search through an exponentially large space (base is the vocab size, exponent is the maximum sequence length) $(|\mathcal{V}|^T)$.[1] In practice, people don't really do this, but it's a good thought experiment to consider whether you think the maximum-probability output of your model will be meaningfully better than a sample.

Indeed, approximations to finding the argmax are often used in practice. One is **greedy decoding**, so named as it's the greedy algorithm for approximating the argmax search. We set:

$$y_t = \arg\max_{w \in \mathcal{V}} p_\theta(w \mid x, y_{<t}) \qquad \text{(greedy decoding)} \tag{3}$$

This picks the most likely output word at each step. This is a common generation strategy because, like sampling, it costs one forward pass per token generated, but like the argmax, it avoids low-probability sequences. It's a nice default to try if you're playing with a model.

Truncation sampling and temperature scaling

One issue with mode-seeking generation strategies (the mode is the highest-probability element of a distribution, so mode-seeking just means it's attempting to find the mode,

[1] However, you can still try this if you're interested — you can run Dijkstra's algorithm or similar — to find the shortest path (highest likelihood generation) from the starting sequence to the generation of the special end of sequence token. (In this graph analogy, nodes are prefixes of text, edges (labeled with words) are weighted by negative log probabilities, so finding the shortest path finds the maximum-probability terminating sequence.)

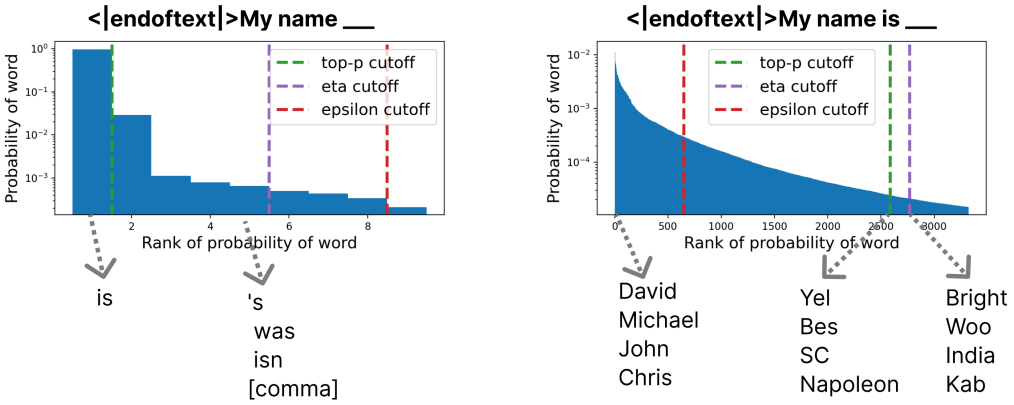

Figure 1: Two examples of truncation of a language model's distribution, from [Hewitt et al., 2022]. On the left, the distribution over the next word after the start of the document and *My name*. Note that most probability is on the word

like, e.g., greedy decoding) is that you always get the same response if you generate multiple times.[2] Sampling from the trained distribution p_θ certainly gives you many possible generations (if there's entropy in the trained distribution) but may generate low-probability, and potentially low-quality, outputs.

The most successful strategies for hitting a nice tradeoff between how much of the distribution we keep (coverage/diversity) and how high-quality our outputs are on average, are in a family of algorithms called (by some, including me) *truncation sampling*. These algorithms have the following form

$$\mathcal{A} = \text{select}(\mathcal{V}, p_\theta, x, y_{<t}) \quad \text{(choose accepted set } \mathcal{A} \subseteq \mathcal{V}) \tag{4}$$

$$p_{\mathcal{A},\theta}(w \mid x, y_{<t}) = \begin{cases} p_\theta(w \mid x, y_{<t})/Z & w \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

$$y_t \sim p_{\mathcal{A},\theta}(w \mid x, y_{<t}) \quad \text{(sample from accepted set)} \tag{6}$$

where $Z = \sum_{w \in \mathcal{A}} p_\theta(w \mid x, y_{<t})$ is the normalization constant — the sum of probabilities of words that we're keeping in the accepted set. The function select is where the fun happens; we get to use a range of heuristics to decide which words stay in the accepted set, and which ones have probabilities set to zero. Usually, these heuristics relate to the probability of the word in that context — high-probability words are probably good continuations and are kept; low-probability words are more likely to be bad continuations and may be cut. We can re-frame greedy decoding under this family by setting the select function to only have the most likely next word in $\mathcal{A}$ at each step. Note that the accept set $\mathcal{A}$ is computed for each prefix (we're omitting a subscript for brevity.)

There are many ways to implement the "high probability is probably a good continu-

[2] In fact, at the large, distributed scale at which modern LMs are often built and served, this is…not true. Little variations in the forward passes of modern models can lead to differences in probabilities and then different generations. At small scale, it's still true. Here's one example of what can happen. Recall that we run our models using floating point numbers represented with very few bits. Small deviations in behavior might lead to relatively large changes in the floating point value (because there aren't many values for a number to "be", unlike in the real numbers. When sampling, differences in, e.g., the CUDA kernels used can lead to different floating point approximations, which can then compound as the network's forward pass progresses. But, the spirit of this statement about not having variation in outputs still holds; we're not sampling from the distribution we estimated; indeed it would be hard to characterize the kind of variation we *accidentally* get from odd distributed systems things.

ation; low-probability not so much" intuition. One very simple method is to just set some minimum threshold of probabilities for the accept set:

$$\mathcal{A} = \{w \in \mathcal{V} \mid p_\theta(w \mid x, y_{<t}) > \epsilon\} \tag{7}$$

Intuitively, low probabilities are not just indicative of unlikely continuations, it's also just hard to estimate the probability of a thing whose true probability is very very low. This algorithm **epsilon sampling** [Hewitt et al., 2022] implements this intuition. There are some details here that one has to keep in mind however; for example, it is possible that no words are above the ϵ threshold, in which case often implementations will just generate the argmax word at that timestep.

By far the most popular truncation method is **top-p sampling**, also called nucleus sampling [Holtzman et al., 2020]. The intuition of top-p sampling is that the most likely p percent of the model's distribution at any timestep is "good" (should be kept) and the remaining $1 - p$ lowest-probability percent is bad.

$$w^{(1)}, \ldots, w^{(|\mathcal{V}|)} = \arg\operatorname{sort}_{w \in \mathcal{V}} \; p_\theta(w \mid x, y_{<t}) \tag{8}$$

$$k = \min\{i \in \mathcal{N} \mid \sum_{j=1}^{i} p_\theta(w^{(j)} \mid x, y_{<t}) \geq p\} \tag{9}$$

$$\mathcal{A} = \{w^{(1)}, \ldots, w^{(k)}\} \tag{10}$$

To interpret this, think of (1) sorting the vocabulary in decreasing order of probability, and then taking the k most probable words such that k is the minimal set whose sum of probabilities it at least p.

Each of these algorithms makes different assumptions about what is "wrong'' about the distribution we've learned. Each also includes a parameter (p or ϵ) which controls the tradeoff of how much you cut of the distribution (which you'd like to avoid) vs how much you avoid potentially generating low-quality outputs (by cutting off the low-likelihood words.)

A final method we must discuss is **temperature sampling**, which is not a truncation sampling algorithm. It does not explicitly set any probabilities to zero; instead intuitively it interpolates in log space between the uniform distribution (arbitrarily high temperature) and the most likely token (zero temperature). The distribution is as follows:

$$p_{\tau,\theta}(w \mid x, y_{<t}) = \frac{p_\theta(w \mid x, y_{<t})^{1/\tau}}{\sum_{w \in \mathcal{V}} p_\theta(w \mid x, y_{<t})^{1/\tau}} \tag{11}$$

Note the exponent — when τ approaches zero, logits are exponentiated, making the largest *even* larger relative to the others. When τ approaches infinity, logits are raised to the power of roughly 0, so each gets probability roughly $\frac{1}{\mathcal{V}}$.

References

[Hewitt et al., 2022] Hewitt, J., Manning, C. D., and Liang, P. (2022). Truncation sampling as language model desmoothing. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3414–3427.

[Holtzman et al., 2020] Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. In *International Conference on Learning Representations*.