Generation Algorithms

CS 6120 Natural Language Processing
Northeastern University

Si Wu

Materials based on various chapters from Jurafsky & Martin

Logistics

- All feedbacks on project initial pitch will be out by tonight
  - Reminder: if you didn’t get 100%, make sure you reply to my comments, and if necessary, resubmit in order to get 100%

- Today:
  - A more advanced topic: generation algorithms
  - We are still by default talking about under transformer architecture

Generation used to be a really hard task!

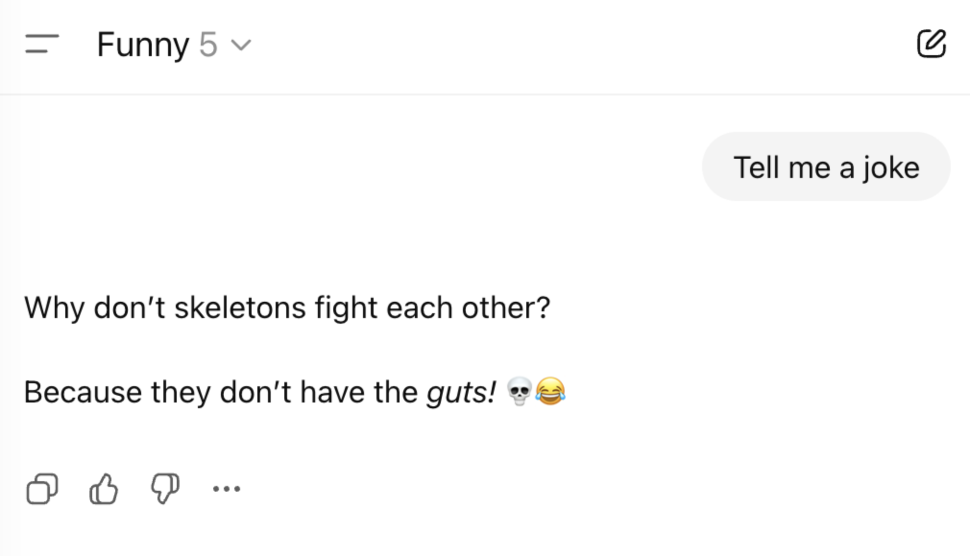
- Before LLMs, we struggle with natural language generation
  - Repetition used to be a big problem
    - Looping: I love dogs. I love dogs. I love dogs. I love dogs.
    - Generic: “This is a great movie. It is very interesting. It is very good.”
  - Forgot information from much earlier text
  - Decoding strategy can help a little, but
    - limited context windows: n-gram, RNN/LSTM
    - Difficulty modeling long-term dependencies: vanishing gradient in RNN (transformer solved this with self-attention!)
    - Etc.

- Now, we have different concerns with text that are so “realistic”, we worry its factuality and try to detect hallucination.

Text generation

Use cases

- Any text generation task really:
  - Machine translation
  - Summarization
  - Chatbot
  - Generate narratives, poetry, etc.
  - Non natural human language: code generation



What affects text generation

- Model
  - Transformer
  - RNN
  - Encoder-decoder
- Data
- Context window size
- Decoding strategy: once we have the text probability, how to choose?
- (And other hyperparameters related to training a model)

Decoding strategy

- Greedy decoding
- Beam search

- Top-k sampling
- Top-p (nucleus) sampling

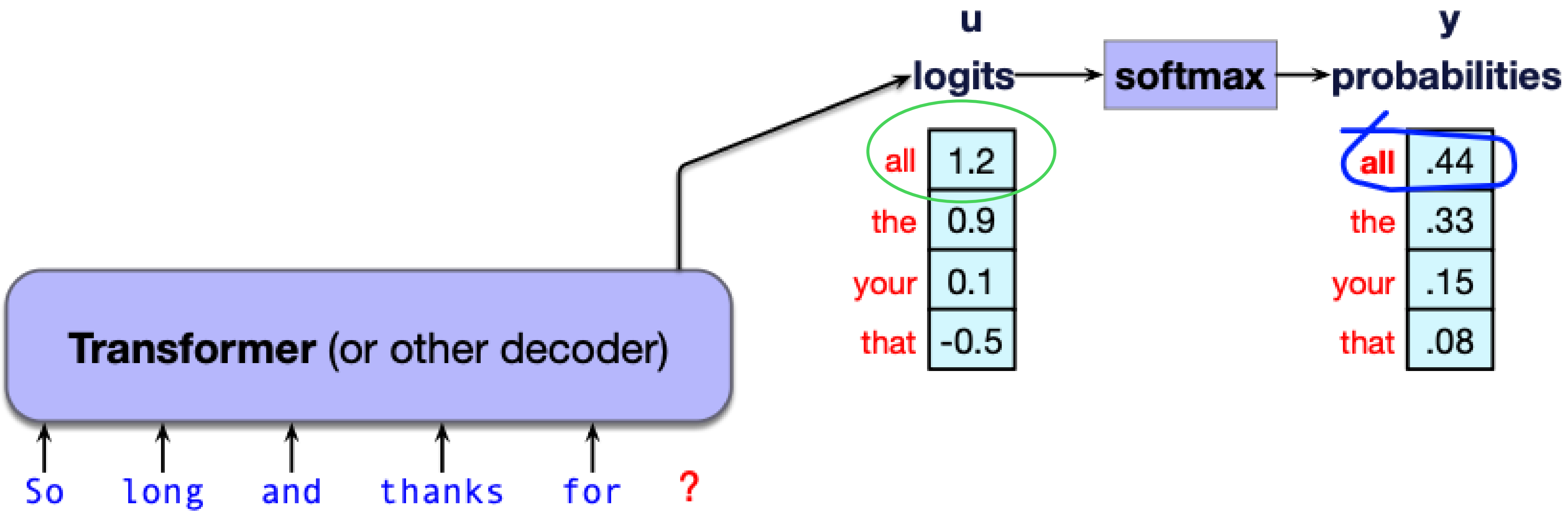
- Temperature scaling
- Contrastive search

- And some other more advanced ones

The simplest: Greedy decoding

- Just like **greedy algorithm** in your first algorithm class
  - "Choose the most optimal at each step"
- At each time step in generation, we choose the **token with the highest probability** in the vocabulary
  - It's that simple!
- In practice, we don't use greedy decoding with LLMs.
  - The token it chooses are extremely predictable, so the resulting text is generic and repetitive.
  - At time step t+1, the word t might turn out to be the wrong choice…

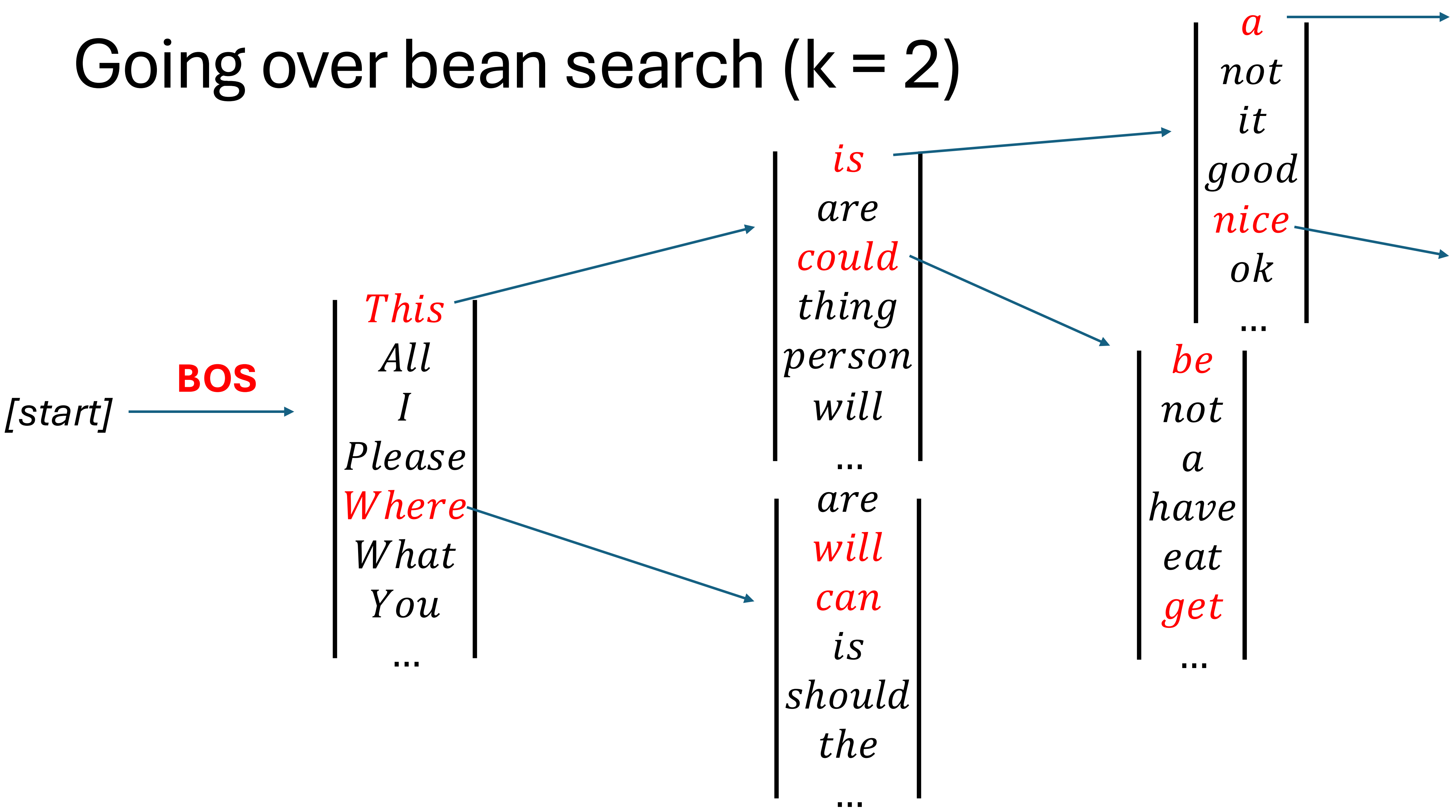
Greedy decoding



Beam search

- It’s **best-first** search and **breath-first** search
- The problem with greedy: at t+1, the choice at t could have been wrong → beam search maintains all choices and decide later
- High-level:
  - Traversing through a tree of possible sequences, where each node represents a state (a sequence built so far),
  - Each branch corresponds to a possible next word, weighted by its probability.
  - But an exhaustive search is too expensive, so beam search narrows the search by keeping only k (beam width) most promising options at each step

Going over bean search (k = 2)

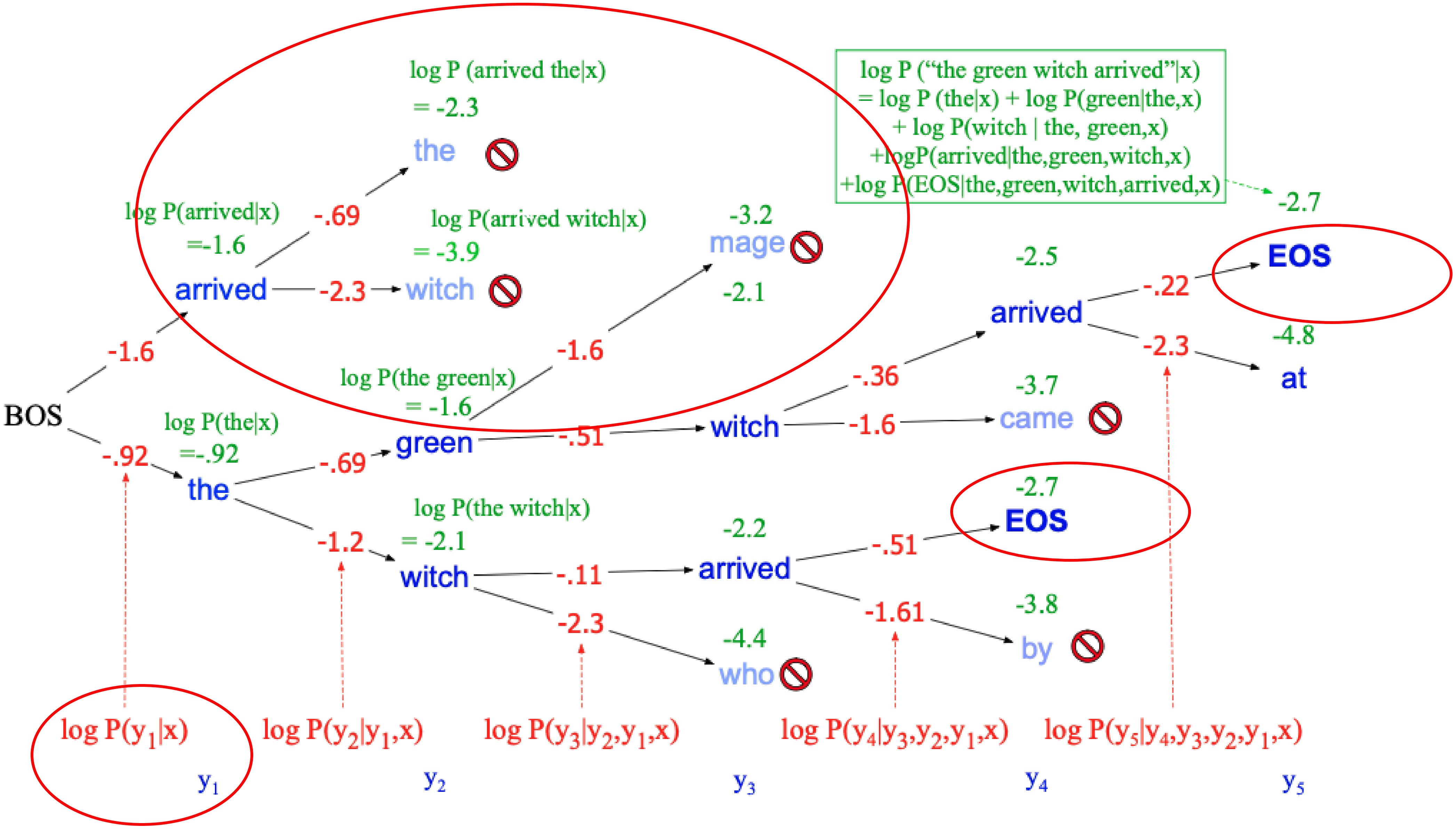


Beam search

- Not only do we keep k **best tokens** at each step

$$\left|\begin{matrix} \color{red}{This} \\ All \\ I \\ Please \\ \color{red}{Where} \\ What \\ You \end{matrix}\right|$$
$$\dots$$

- We are also only keeping k **best paths** at each step
- This continues until EOS (end-of-sentence) token is generated. Then we remove the completed sequence, and reduce k by 1.
- Continues, until k = 0. By then, we will have k complete sequences.



log P (arrived the|x)
= -2.3
the
log P(arrived|x)
=-1.6
-.69
log P(arrived witch|x)
= -3.9
arrived
-2.3
witch
-3.2
mage
-2.1
-1.6
-1.6
log P(the green|x)
= -1.6
BOS
log P(the|x)
=-.92
-.92
the
-.69
green
-.51
witch
-.36
-1.6
came
-3.7
arrived
-2.5
-.22
EOS
-2.3
at
-4.8
log P ("the green witch arrived"|x)
= log P (the|x) + log P(green|the,x)
+ log P(witch | the, green,x)
+logP(arrived|the,green,witch,x)
+log P(EOS|the,green,witch,arrived,x)
-2.7
log P(the witch|x)
= -2.1
-1.2
witch
-.11
arrived
-2.2
-.51
EOS
-2.7
-1.61
by
-3.8
-2.3
who
-4.4
log P(y1|x)
log P(y2|y1,x)
log P(y3|y2,y1,x)
log P(y4|y3,y2,y1,x)
log P(y5|y4,y3,y2,y1,x)
y1
y2
y3
y4
y5

Beam search

- We use chain rule at each node.
- Each node is the probability of each word given its prior context, which we can turn into a sum of logs

$$\begin{aligned} score(y) &= \log P(y|x) \\ &= \log\left(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)\ldots P(y_t|y_1,\ldots,y_{t-1},x)\right) \\ &= \sum_{i=1}^{t} \log P(y_i|y_1,\ldots,y_{i-1},x) \end{aligned}$$

Beam search

- For machine translation, we generally use beam widths k between 5 and 10, giving 5-10 hypotheses at the end.
- Then we can pass down all k and their scores, or just keep the best one.

Sampling

Sampling

- Both are about generating text from a LM's probability distribution over words, but they differ in how they pick the next token
- Sampling → randomly drawing the next token from the probability distribution, not just picking the best one
- For sampling, we care about two important factors in generation: **quality** and **diversity**
- Most probable words are accurate, coherent, factual, but also very likely to be boring and repetitive.
- Methods that give more weights to the middle-probability words tend to be creative and diverse, but less factual and more likely to be incoherent or low-quality.

Top-k sampling

- Instead of choosing the single most probable words, we first truncate the distribution to the top k most likely words, renormalize to produce a legitimate probability distribution
- Then randomly sample from within these k words according to their renormalized probabilities
  - Both beam search and top-k sampling have the parameter k, but they are different!
    - the renormalization and the random sampling makes top-k sampling very different from beam search, even though it’s also kind of keeping k at a time.
    - Also one is a sampling method and the other is decoding

Top k sampling (high-level)

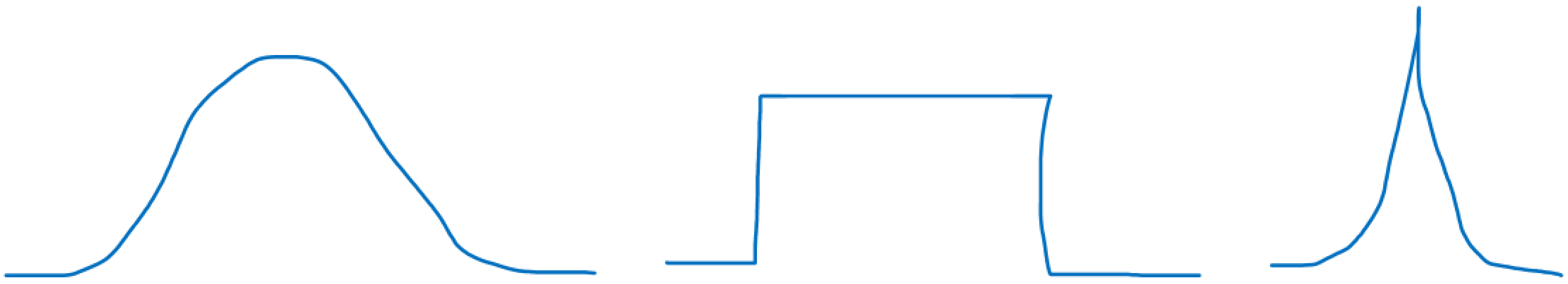
1. Choose choose k words in advance
2. For each word in the vocabulary V, use the language model to compute the likelihood of this word given the context $p(w_t \,|\boldsymbol{w}_{<t})$
3. Sort the words by their likelihood, and only keeping the top k most probable words
4. Renormalize the scores of the k words to be a legitimate probability distribution. → because the vocab size has changed, total probability doesn't sum up to 1
5. Randomly sample a word from the k remaining words

Top k sampling

- When k = 1, top-k sampling is identical to greedy decoding
- Why?
  - Random sampling from a set of size 1 is not random at all
  - Nothing to renormalize

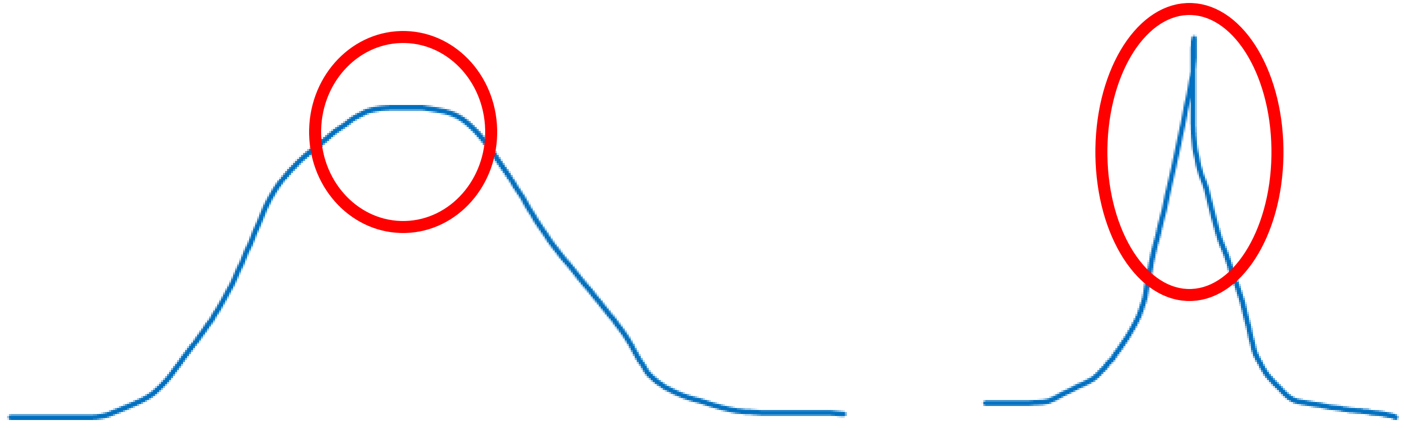
Top-p (nucleus) sampling

- In top-k sampling, k is fixed, but the shape of the probability distribution over words differs in different context.
- For example, sometimes top 10 words make up of 90% of the probability, other times top 10 words only make up of 10% if the distribution is more uniform.



Top-p (nucleus) sampling

- Top-p sampling keep not the top k words, but **the top p percent of the probability mass**
  - The goal is similar: truncate the distribution to remove unlikely words and improve efficiency
- Measuring probability instead of the number of words, the hope is that the measure will be more robust in different context
  - Dynamically increasing and decreasing the pool of word candidates



Top-p (nucleus) sampling

- Given a distribution, we sort the distribution from most probable
- Then using the top p percentage of the vocabulary
- Similar to top-k, we need to renormalize after truncation
- After renormalization, we then random sampling from the truncated vocabulary

More on sampling

- As you can see, sampling really is just a statistic problem, so there are many other sampling methods that can be applied here
  - Typical Sampling, Meister et al., 2022
    - Filter surprising and unsurprising tokens
  - Epsilon Sampling, Hewitt et al., 2022
    - Only sample from probability above a certain threshold epsilon
  - Etc.

Temperature sampling

- Temperature sampling = **temperature scaling** + [whatever sampling you want to use after]

- Temperature scaling: **reshape the probability distribution** to increase the probability of the high probability tokens, and decrease the probability of the low probability tokens
  - The rich gets richer, the poor gets poorer
  - *Temperature is a hyperparameter* that you tune

Intuition: Temperature sampling

- Imagine if you have a distribution that’s almost like a uniform distribution, but 3 tokens have slightly higher values
  - temperature scaling will help them standout before whatever sampling method you use next
- Another example:
  - Before temperature scaling, if you only use top p sampling, 10 tokens make up 90%
  - After temperature scaling, if you use top p sampling, 5 tokens make up 90%, or 20 tokens make up 90%, depending on the temperature you are using.

Temperature scaling

Math trick!

$$P_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

T = temperature

z_i is the logit to token i

T < 1, makes the distribution sharper → spiky !!!

T > 1 , makes the distribution flatter → uniform ...

T = 1, no change

Temperature scaling

$$P_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$$

- Scale the logits by 1/T
- Apply softmax
  - Remember logits are the raw values before softmax!
- Then sample from the new probabilities

Temperature sampling = **temperature scaling** + [whatever sampling you want to use after]

Temperature scaling

T < 1, makes the distribution sharper → spiky !!!

less diverse, probability distribution is reshaped to concentrate on top tokens

T > 1 , makes the distribution flatter → uniform ...

more diverse, more like uniform distribution

More advanced algorithms

Minimum bayes risk decoding

- Works better than beam search and temperature scaling
- Often used on machine translation (Kumar and Byrne, 2004), speech recognition
- High-level idea: instead of choosing the most probable, choose the one likely to have least error (low risk).

- **Risk**, here, means, rather than picking the most probable sequence, we instead do some *risk assessment*:
  - According some metrics (BLEU, chrF, BERTScore),
  - Comparing to some known good translation (minimizing expected loss/risk)

Minimum bayes risk decoding

- In practice, we don't know the perfect set of translation for a given sentence, we instead, we choose the candidate translation which is most similar with some set of candidate translation
  - First beam search or sampling
  - Then pairwise similarity
- Essentially, we are approximating the enormous space of all possible translations U with a smaller set of possible candidate translations Y.
- Given this set of possible candidate translation Y, and some similarity function $util$, we choose the best translation which is the most similar to the other candidate translations.

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{c \in \mathcal{Y}} \operatorname{util}(y, c)$$