Assignment 2 – TUT2

Tian Yu

Timeline

Tutorial 1 & A2 Release	Feb 6, 2026
Tutorial 2	Feb 13, 2026
Office Hour	Feb 27, 2026
A2 Deadline	Mar 5, 2026

TUT2 Outline

- Language Model Decoding
- BLEU
- Model Training
- Analysis

Typo in assignment

Typo in type annotation for a2_transformer_model.py

```
def finalize_beams_for_beam_search(self, top_beams: Tensor,
device: torch.device) -> Tensor:
```

update to

```
def finalize_beams_for_beam_search(self, top_beams: List[Tensor],
device: torch.device) -> Tensor:
```

Language Model Decoding

Neural Machine Translation - Goal

- Assume we have the oracle probability P*, the goal of a **translation problem** is to find

$$Y^* = Argmax_Y \; P^*(Y|X)$$

- In NMT, we train the network to approximate

$$P^*(Y|X) \approx P_{data}(Y|X) \approx P_\theta(Y|X)$$

Assuming $Y = (y_0, y_1, \dots, y_k)$ Then

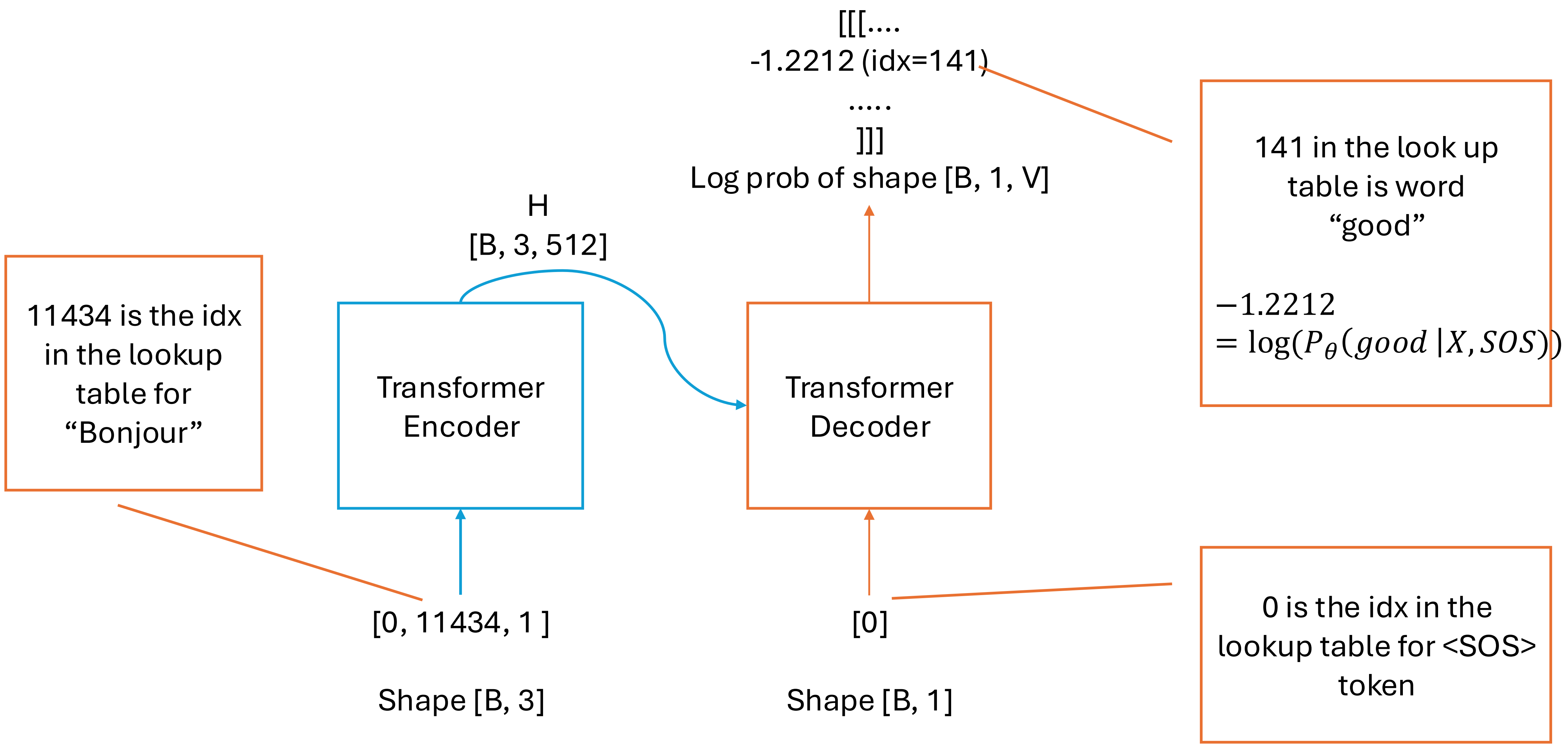
$$P_\theta(Y|X) = P_\theta(y_0|X) * P_\theta(y_1|X, y_0) * \dots * P_\theta(y_k|X, y_{<k})$$

$$P_\theta(Y|X) = \prod_{t=0}^{k} P_\theta(y_t|X, y_{<t}) \text{ OR } \boxed{\log\big(P_\theta(Y|X)\big) = \sum_{t=0}^{k} \log P_\theta(y_t|X, y_{<t})}$$

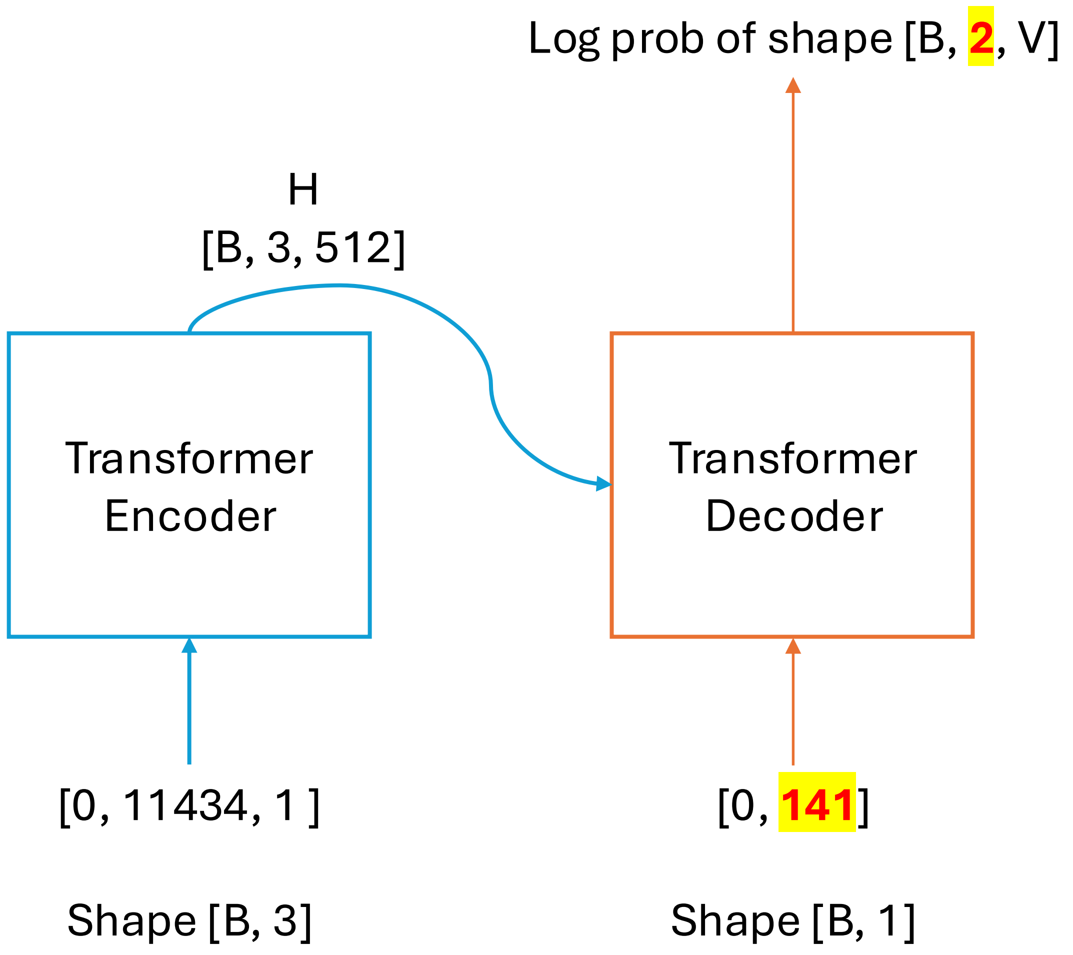
Some Details

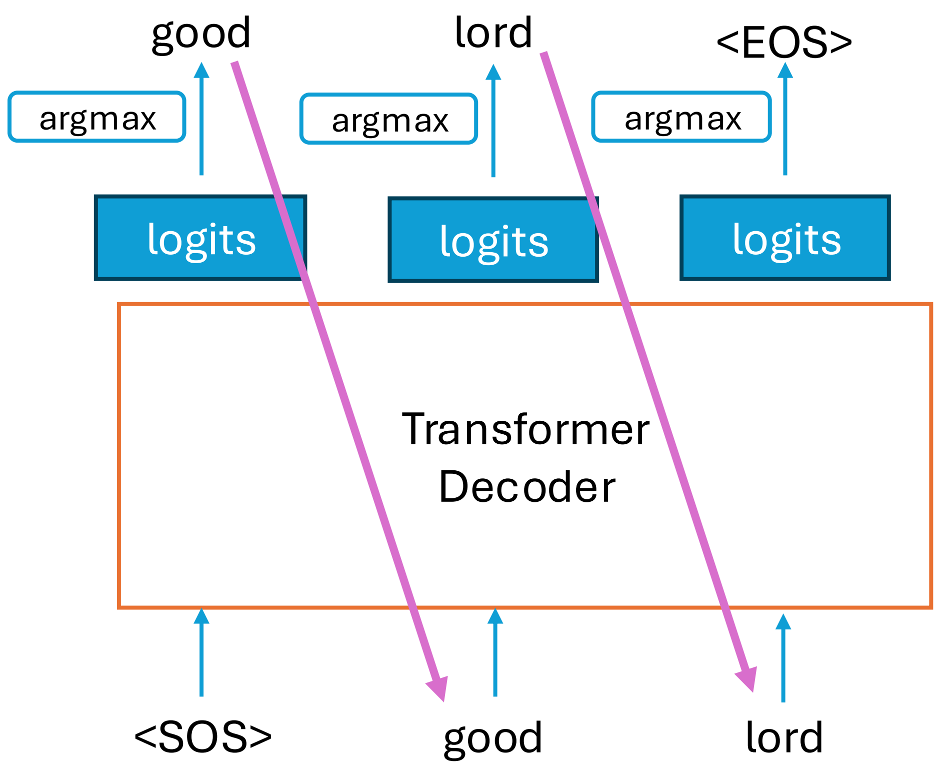
- If $Y = (y_0, y_1, \dots, y_k)$, what is y_0 and y_k ?
    - <SOS> & <EOS>
- What is $P_\theta(y_0|X)$?
    - 1

An Example to translate “Bonjour”


[[[....
-1.2212 (idx=141)
.....
]]]
Log prob of shape [B, 1, V]
141 in the look up table is word “good”
$-1.2212 = \log(P_\theta(good \mid X, SOS))$
H
[B, 3, 512]
11434 is the idx in the lookup table for “Bonjour”
Transformer Encoder
Transformer Decoder
[0, 11434, 1]
Shape [B, 3]
[0]
Shape [B, 1]
0 is the idx in the lookup table for <SOS> token

An Example to translate “Bonjour” - Greedy





What is greedy / beam decoding?

- Remember what we want?

$$Y^* = Argmax_Y \, P_\theta(Y|X)$$

- But it’s infeasible to search the entire search space
- Greedy / Beam search uses the model generated next token probability as a local score to provide an efficient **APPROXIMATION** to the Maximum A Posteriori
- Is best score guaranteed?
- NO!!!

Beam Search - Motivation

- Greedy decoding has no way to undo decisions!
  - Input: *il a m'entarté* ----- *(he hit me with a pie)*
  - → *he* ___
  - → *he hit* _
  - → *he hit a* _____
- How to fix this?
- At each step, we can have multiple candidates instead of one

Beam search decoding

- Core idea: On each step of the decoder, we expand m possible candidates, keep track of the *k* most probable partial translations (which we call *hypotheses*)
  - m is the branching factor, *k* is the beam size (in practice around 5 to 10, in NMT)

- A hypothesis $\log\left(P_\theta(Y|X)\right) = \sum_{t=0}^{k} \log P_\theta(y_t|X, y_{<t})$ has a score which is its log probability:
  - $\text{Score(Y)} = \log\left(P_\theta(Y|X)\right) = \sum_{t=0}^{k} \log P_\theta(y_t|X, y_{<t})$
  - Scores are all **negative**, and **higher score is better**
  - We search for high-scoring hypotheses, tracking top *k* on each step

- Beam search is not guaranteed to find optimal solution
- But much more efficient than exhaustive search!

Beam search decoding: example

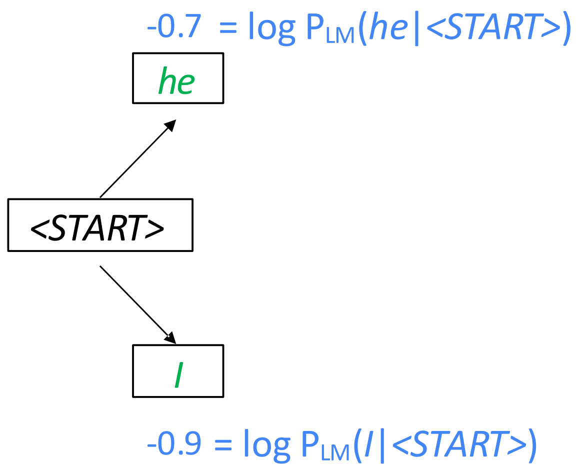
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$

<START>

Calculate prob dist of next word

Beam search decoding: example

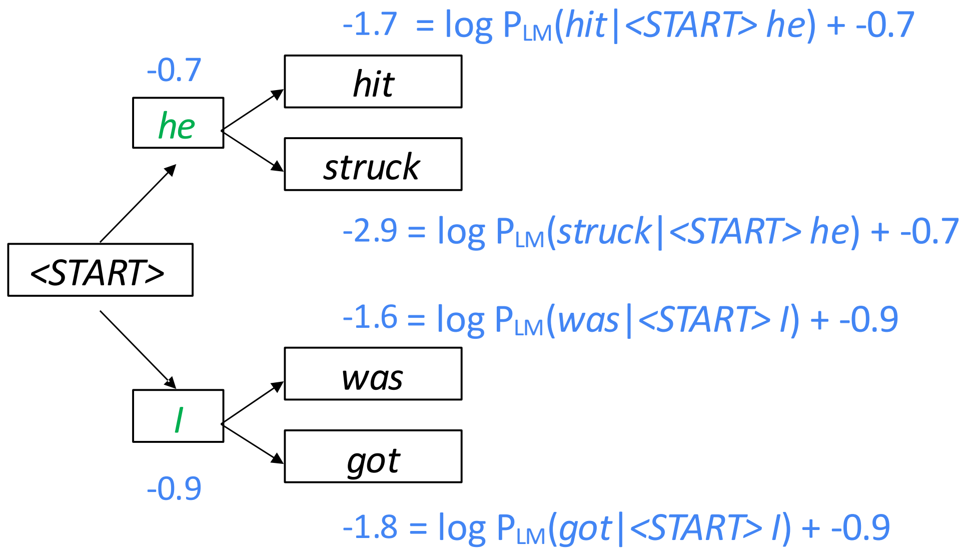
Branching factor=m=2   Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$


-0.7 = log P_LM(*he*|*<START>*)
he
<START>
I
-0.9 = log P_LM(*I*|*<START>*)

Take top *k* words and compute scores

Beam search decoding: example

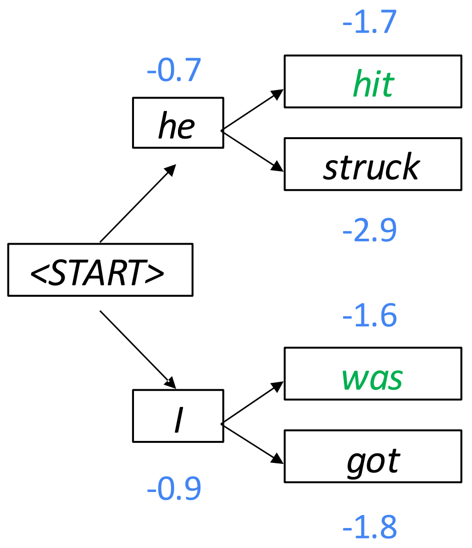
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$


-1.7 = log P_LM(hit|<START> he) + -0.7
-0.7
he
hit
struck
-2.9 = log P_LM(struck|<START> he) + -0.7
<START>
-1.6 = log P_LM(was|<START> I) + -0.9
was
I
got
-0.9
-1.8 = log P_LM(got|<START> I) + -0.9

For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

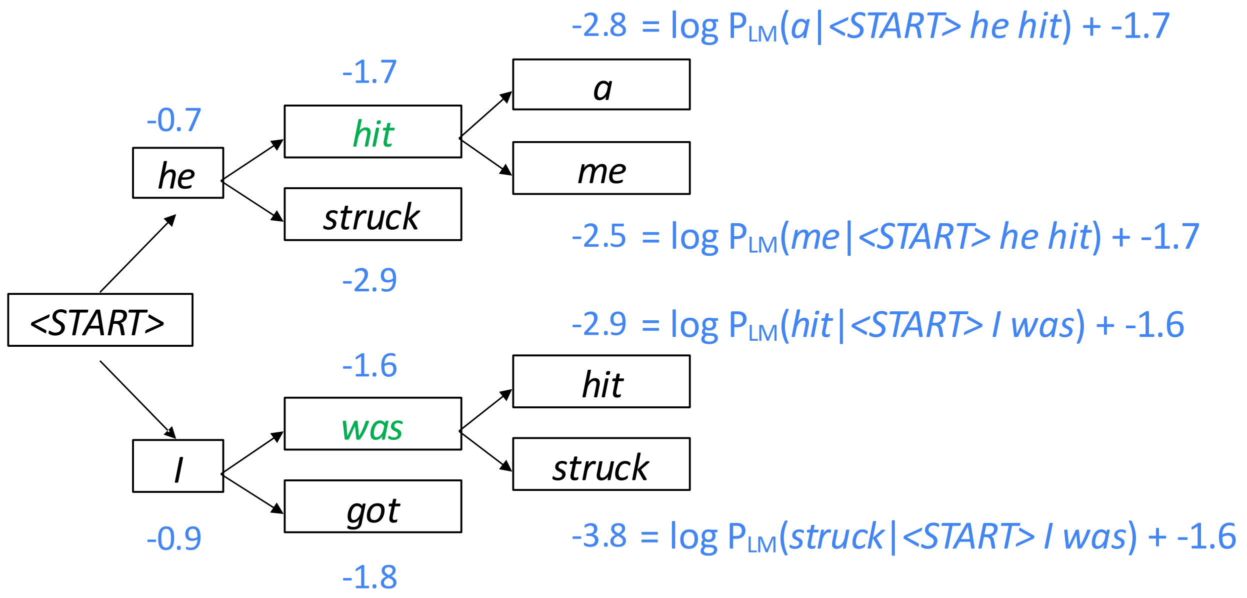
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1,\ldots,y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1,\ldots,y_{i-1},x)$



Of these k^2 hypotheses,
just keep *k* with highest scores

Beam search decoding: example

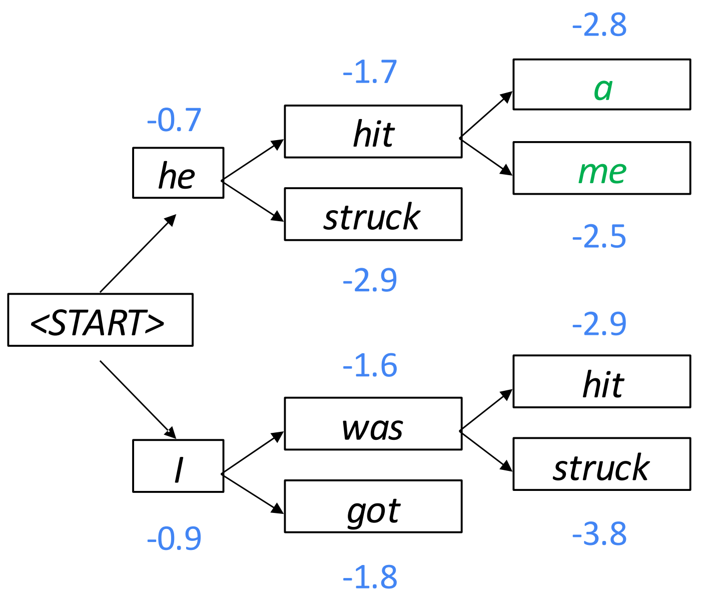
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$


-2.8 = log P_LM(a|<START> he hit) + -1.7
-1.7
a
-0.7
hit
me
he
struck
-2.5 = log P_LM(me|<START> he hit) + -1.7
-2.9
<START>
-2.9 = log P_LM(hit|<START> I was) + -1.6
-1.6
hit
was
struck
I
got
-0.9
-3.8 = log P_LM(struck|<START> I was) + -1.6
-1.8

For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

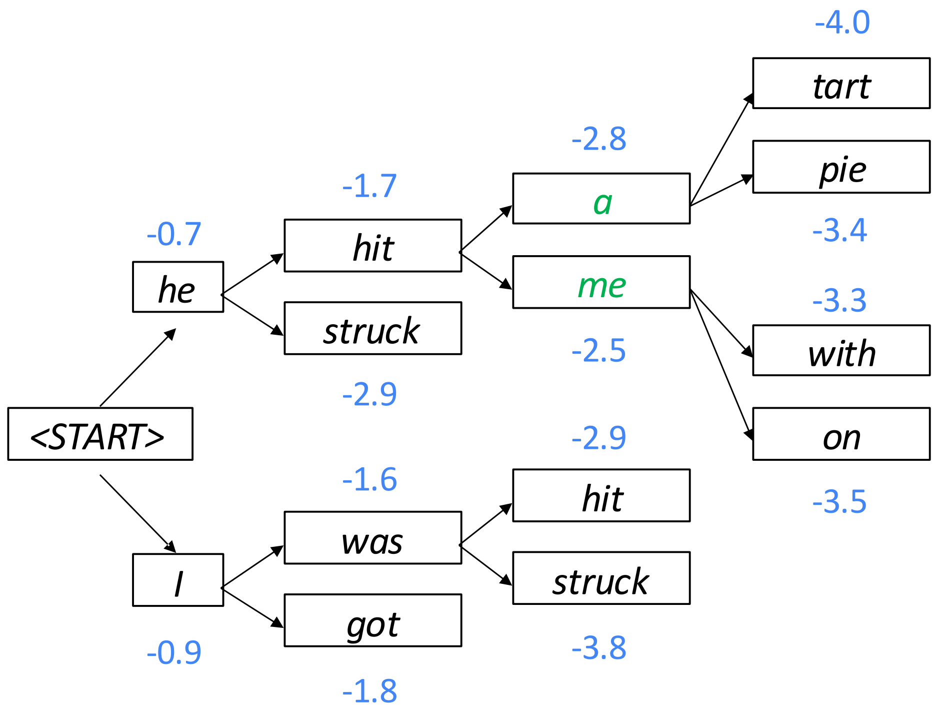
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

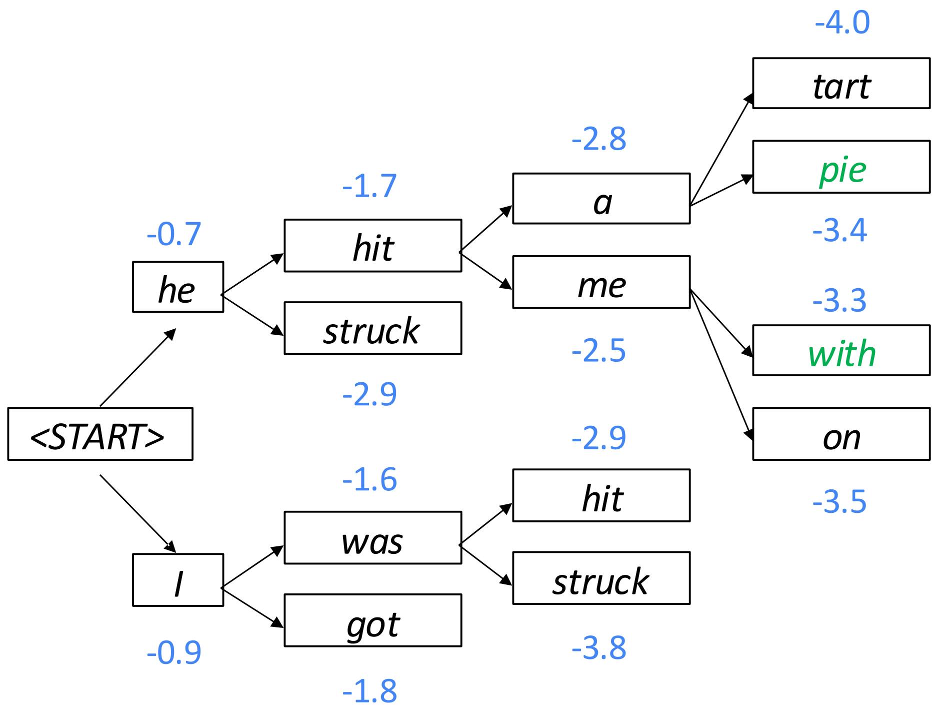
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

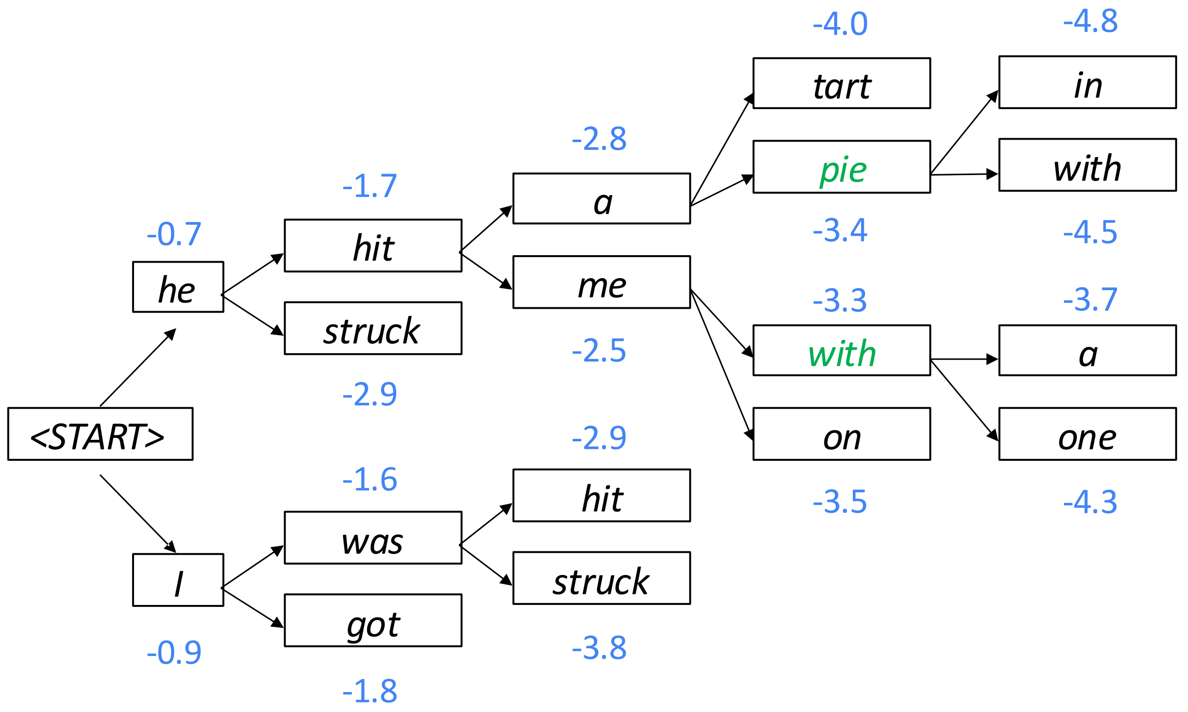
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1,\ldots,y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i|y_1,\ldots,y_{i-1},x)$



Of these k^2 hypotheses,
just keep *k* with highest scores

Beam search decoding: example

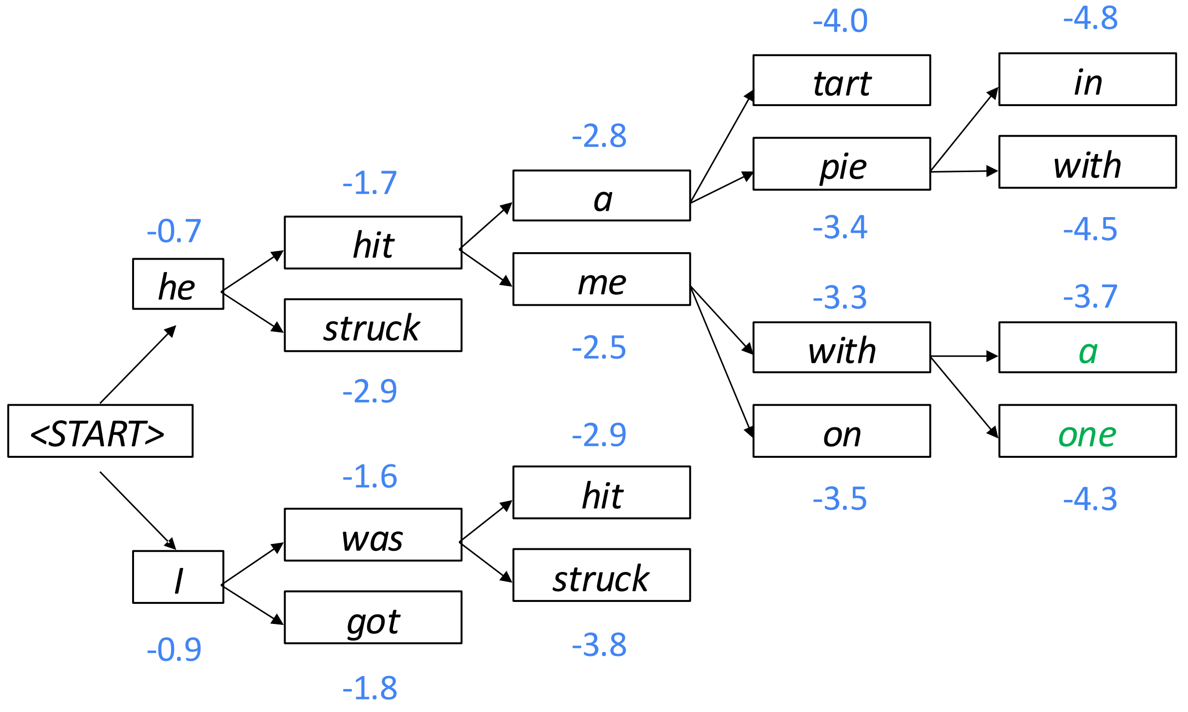
Branching factor=m=2   Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

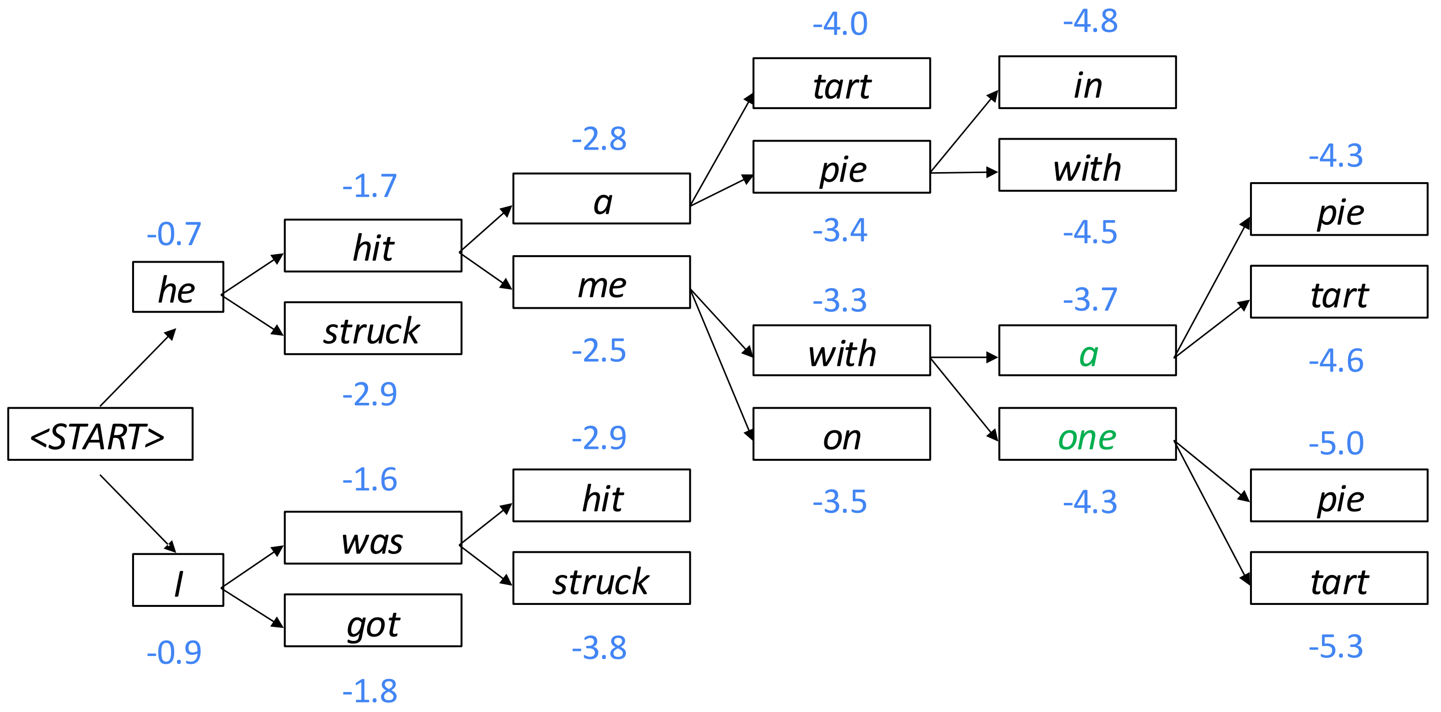
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1,\ldots,y_t)=\sum_{i=1}^{t}\log P_{\text{LM}}(y_i|y_1,\ldots,y_{i-1},x)$



Of these k^2 hypotheses, just keep *k* with highest scores

Beam search decoding: example

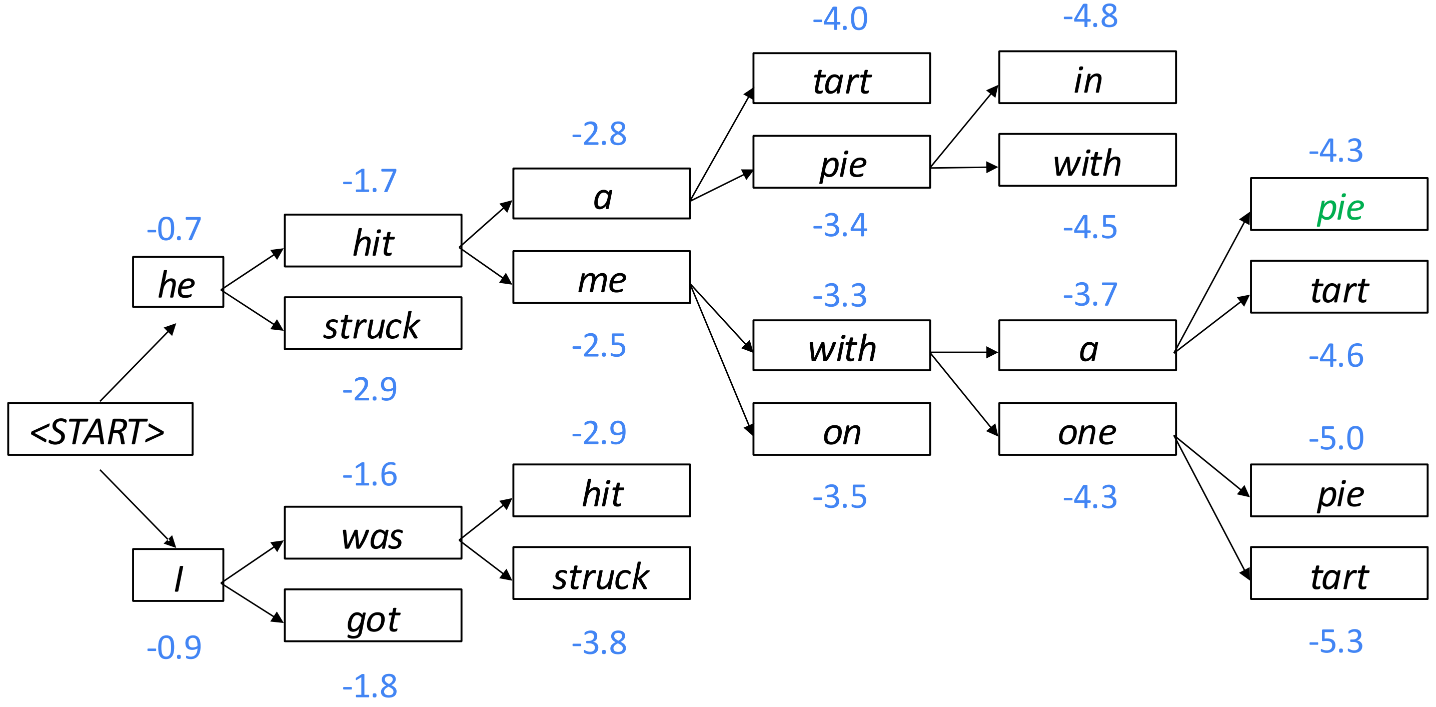
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



For each of the *k* hypotheses, find top *k* next words and calculate scores

Beam search decoding: example

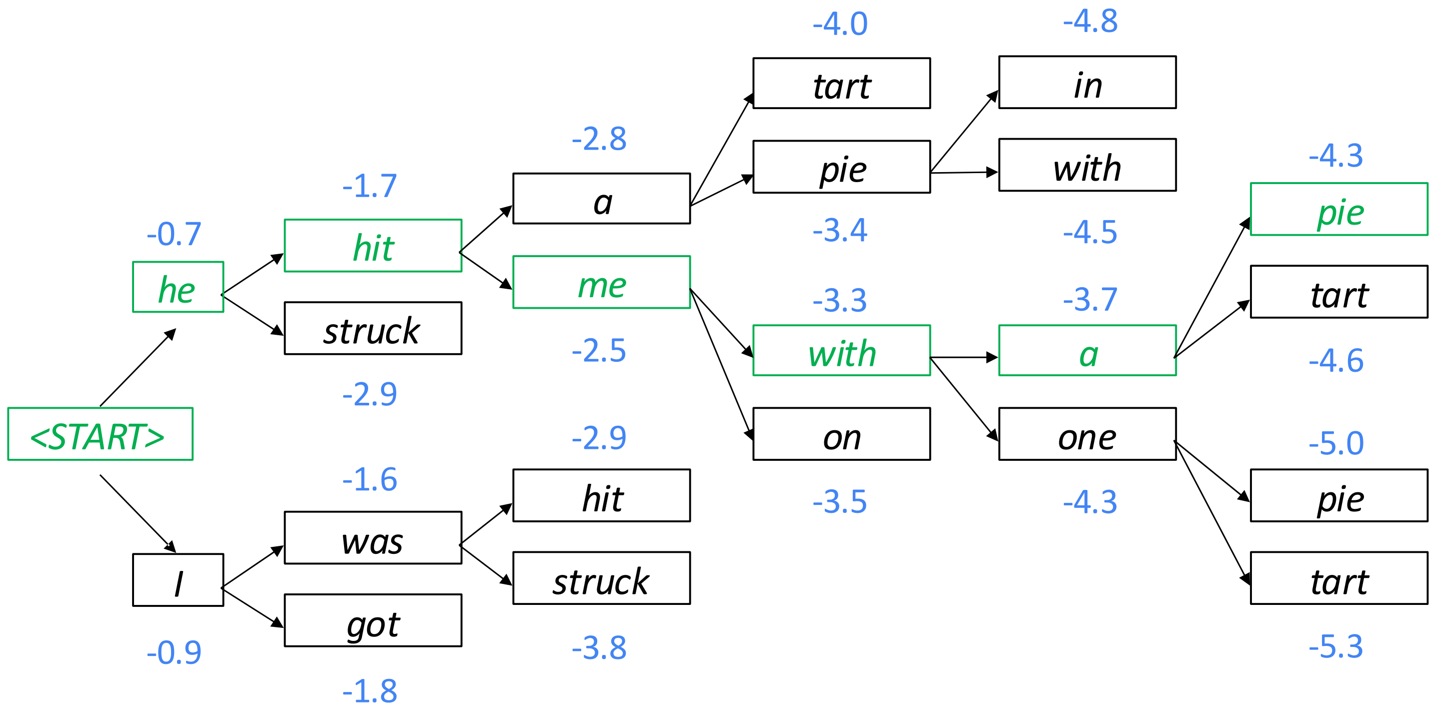
Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



This is the top-scoring hypothesis!

Beam search decoding: example

Branching factor=m=2 Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Backtrack to obtain the full hypothesis

Beam search decoding: stopping criterion

- In greedy decoding, usually we decode until the model produces an <END> token
  - **For example:** *<START> he hit me with a pie <END>*

- In beam search decoding, different hypotheses may produce <END> tokens on
  different timesteps
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via beam search.

- Usually we continue beam search until:
  - We reach timestep *T* (where *T* is some pre-defined cutoff), or
  - We have at least *n* completed hypotheses (where *n* is pre-defined cutoff)

Beam Search Methods

initialize_beams_for_beam_search()

- Takes first decoder step and uses the top-k outputs to initialize beams
- There are several steps listed in the docstring -- follow them carefully
- Tip: you need to call the encoder first (look at how this is done in decode_greedy())

expand_encoder_for_beam_search()

- This is a helper method called at the end of the previous method.
- Goal: Expands source embeddings and mask to have shape [batch_size * k, ...] instead of [batch_size, ...]
- This gives the src embeddings (encoder output) a similar shape to the decoder beams, letting us process things in parallel
- Relevant pytorch method: expand()

Beam Search Methods

- ***repeat_and_reshape_for_beam_search()***
  - We expand [batch_size * k, cur_len] -> [batch_size * k, expan, cur_len] so we can get n=expan completions for each of the current k translations per beam.
  - We reshape [batch_size * k, expan, cur_len] -> [batch_size, k * expan, cur_len], so that (later) we can select the best k per sentence in the batch.
  - Relevant pytorch method: expand()
- ***score_sequence_for_beam_search()***
  - You only need to do the second step (scoring) the sentences by summing log probabilities.

Beam Search Methods

- ***finalize_beams_for_beam_search()***
  - This pads the generated sequences so they are all the same length.
  - We need to do this because beam search removes finished beams at each step (so the generated sequences can have different lengths)

BLEU

BLEU evaluation

- **BLEU (BiLingual Evaluation Understudy)** is an automatic and popular method for evaluating MT.
  - It uses **multiple** human **reference** translations, and looks for local matches, allowing for phrase movement.

  - **Candidate:** *n.* a translation produced by a machine.

- There are a few parts to a **BLEU score**…

[1]Papineni, Kishore, et al. "Bleu: a method for automatic evaluation of machine translation." *Proceedings of the 40th ACL*. 2002. [link]

Example of BLEU evaluation

- **Reference 1**: *It is a guide to action that ensures that the military will forever heed Party commands*
- **Reference 2**: *It is the guiding principle which guarantees the military forces always being under command of the Party*
- **Reference 3**: *It is the practical guide for the army always to heed the directions of the party*

- **Candidate 1**: *It is a guide to action which ensures that the military always obeys the commands of the party*
- **Candidate 2**: *It is to insure the troops forever hearing the activity guidebook that party direct*

BLEU: Unigram precision

- The **unigram precision** of a candidate is

$$\frac{C}{N}$$

where N is the number of words in the **candidate**
and C is the number of words in the **candidate** which are in **at least one reference**.

- e.g., **Candidate 1**: *It is a guide to action which ensures that the military always obeys the commands of the party*
  - **unigram precision** = 17 / 18

(*obeys* appears in none of the three references).

BLEU: Modified unigram precision

- **Reference 1**: *The lunatic is on the grass*
- **Reference 2**: *There is a lunatic upon the grass*
- **Candidate**: *The the the the the the the*
  - Unigram precision = $\frac{7}{7} = 1$
- **Capped unigram precision:**

  A candidate word type w can only be correct a **maximum** of $cap(\mathrm{w})$ times.
  - e.g., with $cap\ (the) = 2$, the above gives $p_1 = \frac{2}{7}$

BLEU: Generalizing to *N*-grams

- Generalizes to higher-order *N*-grams.
  - **Reference 1**: ***It is*** *a guide to action that ensures that the military will forever heed Party commands*
  - **Reference 2**: ***It is*** *the guiding principle which*
  - *guarantees the military forces always being under command of the Party*
  - **Reference 3**: ***It is*** *the practical guide for the army*
  - *always to heed the directions of the party*

Bigram precision, p_2

- **Candidate 1**: ***It is*** *a guide to action which ensures that the military always obeys the commands of the party*

$p_2 = 10/17$

- **Candidate 2**: ***It is*** *to insure the troops forever hearing the activity guidebook that party direct*

$p_2 = 1/13$

BLEU: Precision is not enough

- **Reference 1**: It is a guide to action that ensures that the *military will forever heed Party commands*
- **Reference 2**: It is the guiding principle which guarantees the *military forces always being under command* **of the** Party
- **Reference 3**: It is the practical guide for the army always to *heed the directions* **of the** party

- **Candidate 1**: **of the**

Unigram precision = $p_1 = \frac{2}{2} = 1$ Bigram precision= $p_2 = \frac{1}{1} = 1$

BLEU: Brevity

- Solution: Penalize brevity.
- **Step 1**: for each candidate,
  find the reference **most similar in length**.
- **Step 2**: c_i is the length of the i^{th} candidate, and
  r_i is the nearest length among the references,

$$brevity_i = \frac{r_i}{c_i}$$

Bigger = too brief

- **Step 3**: multiply precision by the **brevity penalty**:

$$BP_i = \begin{cases} 1 & \text{if } brevity_i < 1 \\ e^{1-brevity_i} & \text{if } brevity_i \geq 1 \end{cases}$$

$(r_i < c_i)$

$(r_i \geq c_i)$

BLEU: Final score

- On slide 96, $r_1 = 16, r_2 = 17, r_3 = 16$, and $c_1 = 18$ and $c_2 = 14$,

$$brevity_1 = \frac{17}{18} \qquad BP_1 = 1$$

$$brevity_2 = \frac{16}{14} \qquad BP_2 = e^{1-\left(\frac{8}{7}\right)} = 0.8669$$

- **Final score** of candidate C:

$$\boxed{BLEU_C = BP_C \times (p_1 p_2 \ldots p_n)^{1/n}}$$

where p_n is the n-gram precision. (You can set n empirically)

BLEU: summary

- BLEU is a **geometric mean** over n-gram precisions.
  - These precisions are **capped** to avoid strange cases.
    - E.g., the translation "*the the the the*" is not favoured.

  - This geometric mean is **weighted** (*brevity penalty*) so as not to favour unrealistically short translations, e.g., "*the*"

- Initially, evaluations showed that BLEU predicted human judgements very well, but:
  - People started **optimizing** MT systems to **maximize** BLEU. Correlations between BLEU and humans **decreased**.

BLEU Score

`grouper()`

- Extract all n-grams from a sequence
- Use a sliding window approach to generate n-grams

`n_gram_precision()`

- Calculates the precision for a given order of n-gram
- First generate n-grams for both reference and candidate sequences
- Then count how many candidate n-grams in the reference n-grams and divide by the total

`brevity_penality()`

- Calculates the brevity penalty between a reference and candidate

`BLEU_score()`

- Compute the n-gram precisions for all orders from 1 to n
- Apply the formula

Model Training

Training Loop

train_for_epoch()

- Follow the instructions in the docstring
- Don’t forget to scale the loss (due to gradient accumulation for large batch size)

train_input_target_split()

- Split target tokens into input and target for maximum likelihood training (teacher forcing)
- model inputs exclude the last token in each sequence, and outputs exclude the first token in each sentence

Training Loop

train_step_optimizer_and_scheduler()

- Step the optimizer, zero out the gradient, and step scheduler

compute_batch_total_bleu()

- Computes bleu score for a batch of sentences
- tip: don’t pass sos, eos, and pad tokens to bleu_score_func

teach.cs with GPU: srun

- First make sure your code works in cpu mode! Debugging in CUDA mode is much more difficult

- Basic usage:
  - `srun -p csc401 --gres gpu your_regular_command`
    - `srun -p csc2511 --gres gpu` if you enrolled in CSC 2511
- Check current queue: `squeue -p csc401`
- Keep training after disconnecting: Use `screen`

Analysis

Let’s translate some sentences!

- Here, you translate 8 sentences from French to English, using the following
- models:
  - The model you trained
  - A fine-tuned pre-trained transformer model (T5 MT model or Bart MT model)
  - A large, established model (Google Translate or ChatGPT)
- Then, you answer four questions by comparing them.

Q&A