HW3a (Transformers) Recitation

November 4, 2022

10-417/617 Intermediate Deep Learning

Agenda

1. HW3a Overview
2. Written portion
   - We won't really go through this in detail, but we will provide a few tips, reminders, things to watch out for
3. Transformer architecture
4. Decoding (beam search)
5. Evaluation (BLEU score)
6. Programming portion

HW3a Overview

- Released: Wednesday (November 2, 2022)
- Due: Monday (November 14, 2022)
- 80 points for 417; 100 points for 617
- Written (30+10 points):
    - 2 questions for all (15 points each)
    - 1 questions for 617 only (10 points)
- Programming (50+10 points):
    - Auto-grader (24 + 6 points)
    - Experiments / analysis (26 + 4 points)
- **Start early!**

Written portion

Q1: Vanishing/Exploding Gradients in RNNs

- Main ideas/tools: derivatives, chain rule

Q2: Deriving PyTorch’s GELU

- Part 1 → change of variables (integration)
- Part 4 → make sure you remove all terms with higher order than 3 at the very first step

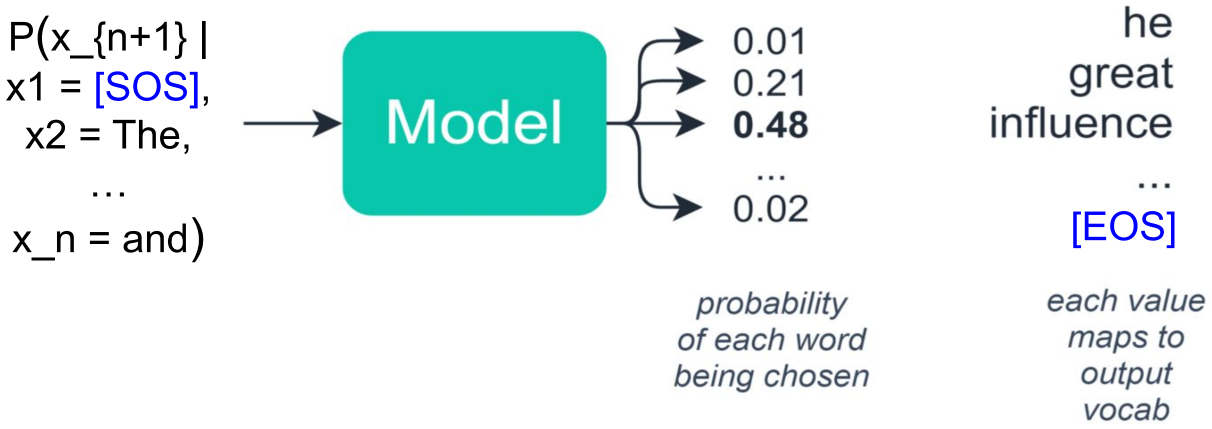
Written portion

Q3: Gradients in RBMs

- Part 2: beta(v) and gamma(v, h) are functions for you to define. Make sure you define these correctly
- Part 3: Theta is just a general variable. Apply chain rule
- Part 4: Use result from part 3 (and fill in theta accordingly)

Language Modeling

A language model is a **probability distribution over a set words,** usually represented by a neural network



Note the special [SOS] and [EOS] tokens here. These are used to represent the starts and ends of sentences. In the assignment, they are index 0 and 1 in our vocabulary.

Applications of Language Modeling

Since we can predict the probability of the next work, we can use language models in open-ended generation tasks and seq2seq tasks:

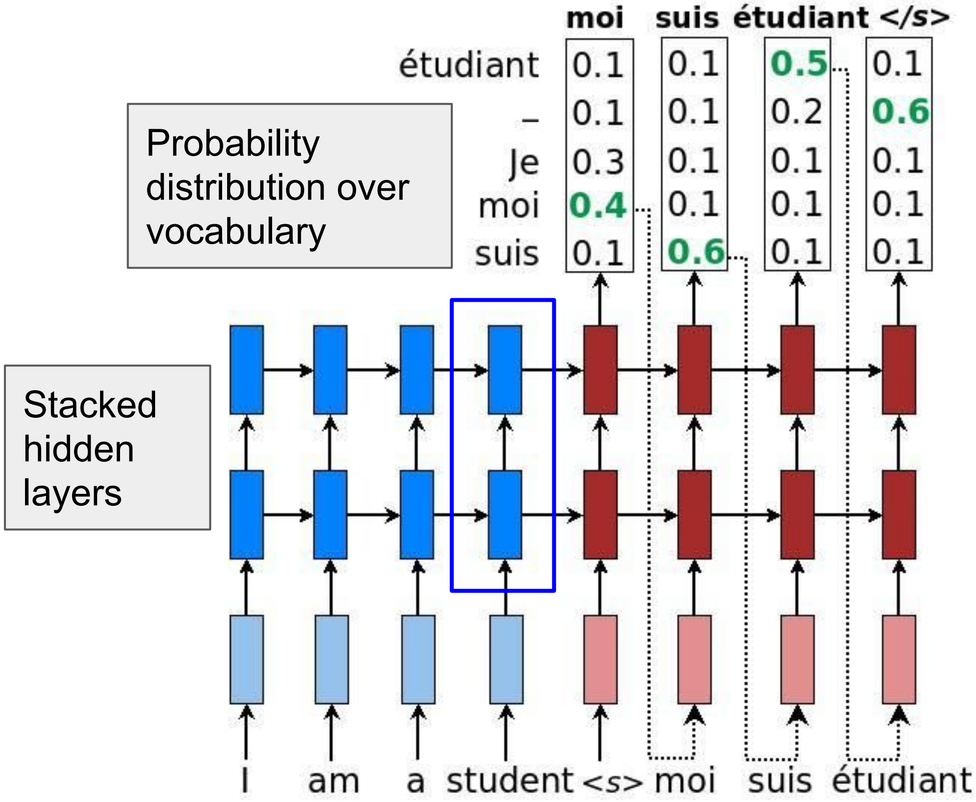
- Text generation (e.g. essay generation, story generation, etc.)
- Summarization
- **Machine translation**

In this HW, we will be working on the task of French-to-English translation

Language Modelling with RNNs

- Seq2seq translation example
- Model generates words sequentially

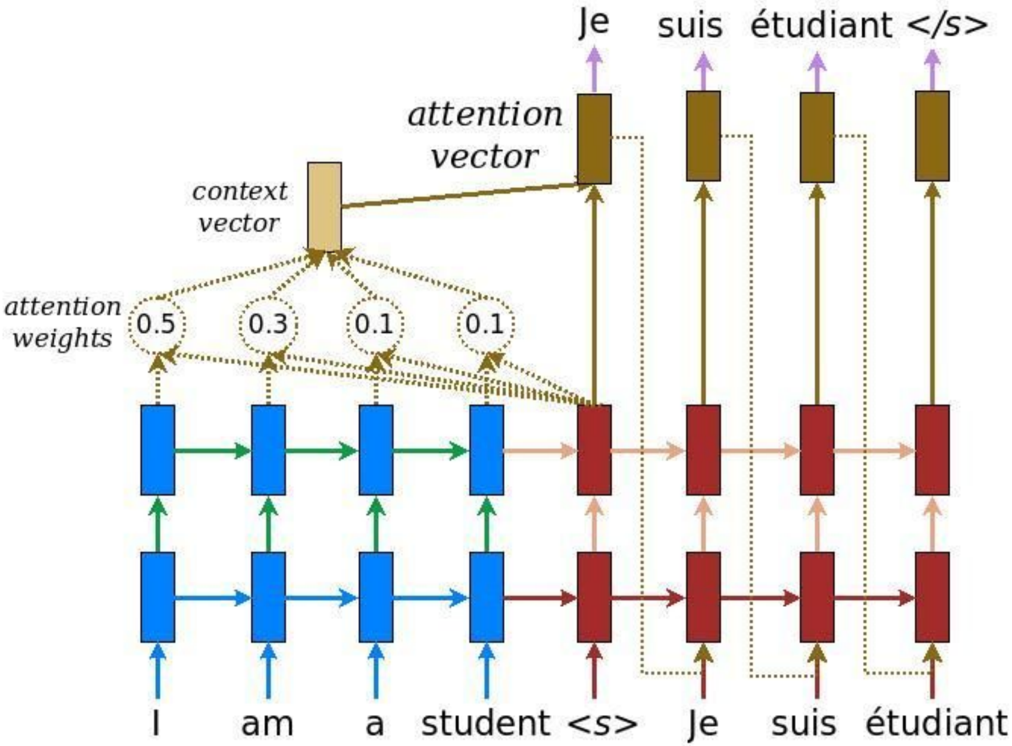
- One main downside is that the effect of the earlier words gets diminished over time (all the input information is encoded into just one single vector)



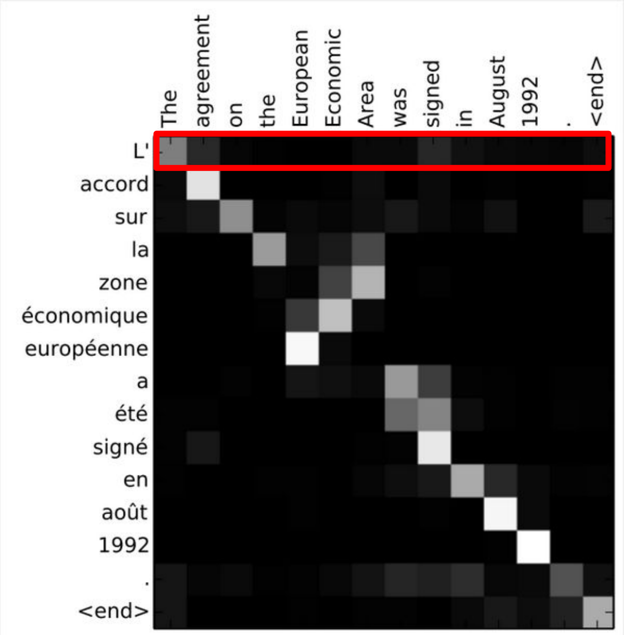
Attention in RNNs

- **Attention** is a key way to address this issue
    1. Take the pairwise **dot products** between the current vector and each input vector to get attention weights
    2. Multiple the attention weights by each input vector and add to get the final context vector

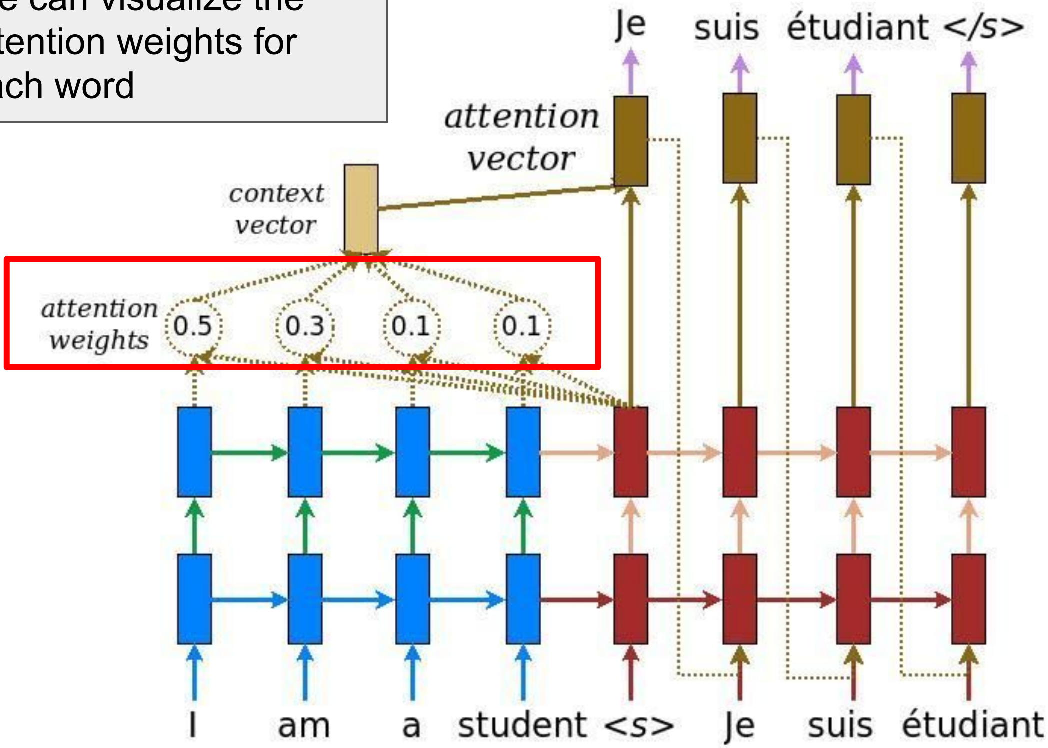
- Using attention, each output is able to take into consideration each input word



Attention in RNNs



We can visualize the attention weights for each word

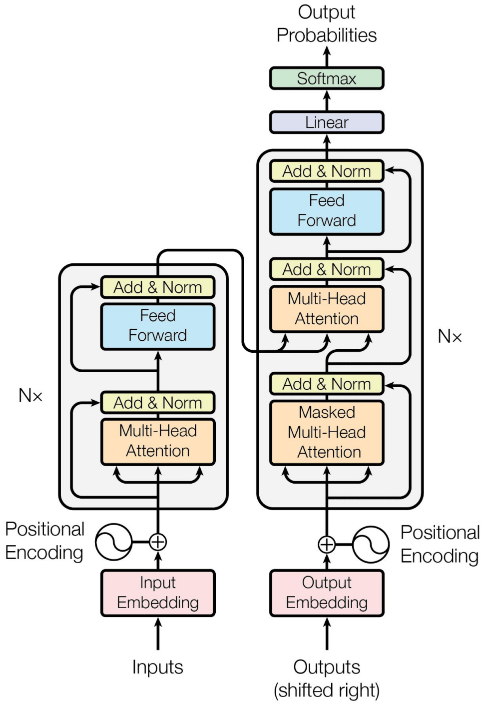


A good resource/tutorial for attention can be found here:
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Transformers

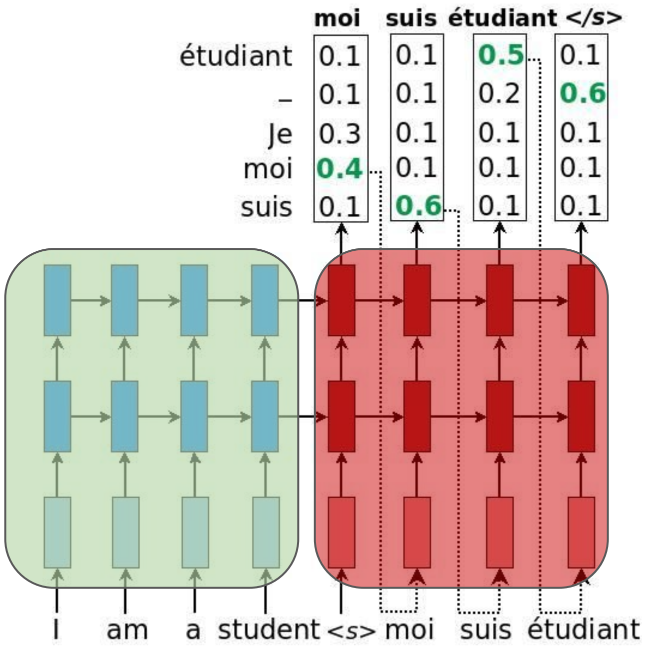
- Transformers are also language models, but they don't use any recurrent structure unlike RNNs or LSTMs

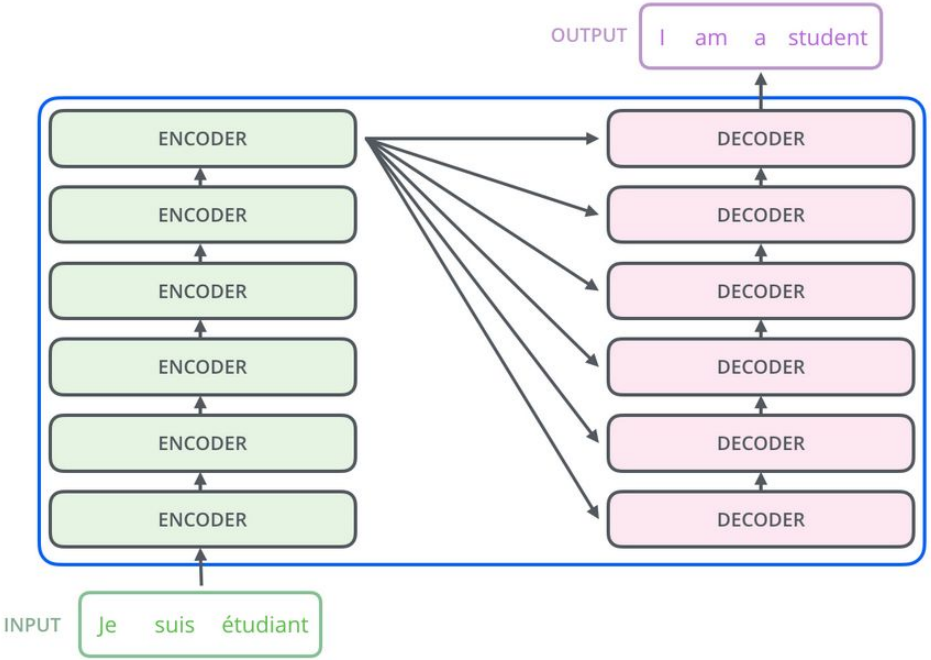
- Rather, they introduce the idea of **self-attention**
- "Attention is All You Need" [1]



[1] Vaswani et al (2017) https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
Some really good resources/tutorials for transformers: The Illustrated Transformer and The Annotated Transformer

Encoder + Decoder Architecture

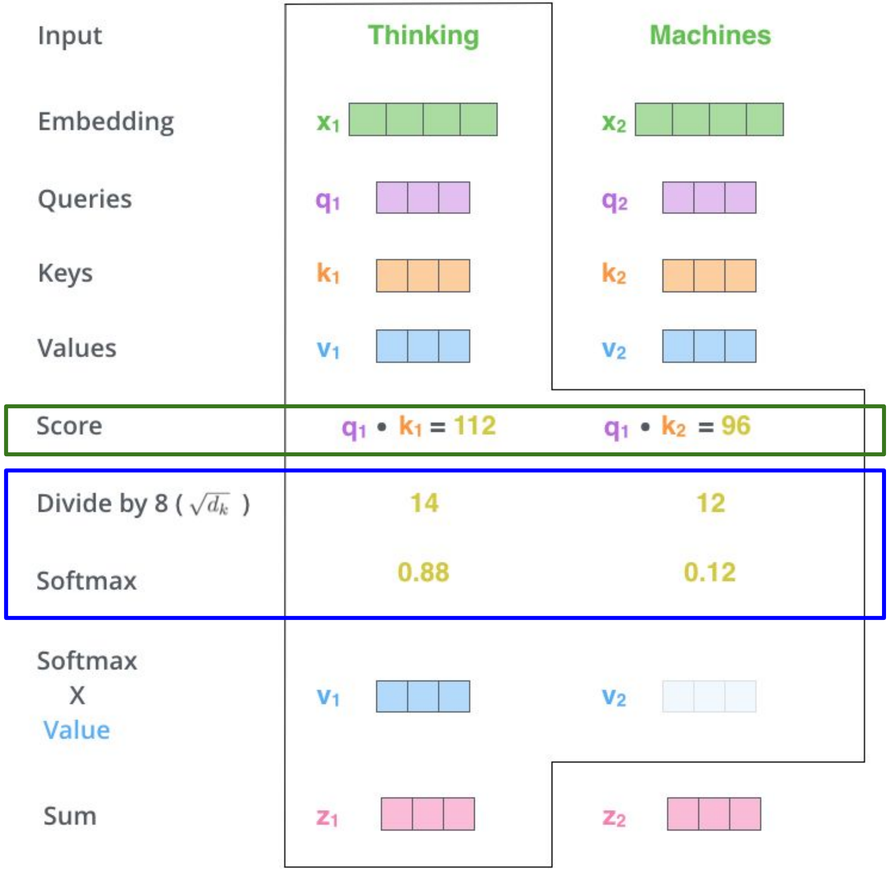




Similar to RNNs, transformers follow an encoder-decoder structure with stacked layers/blocks.

Self-Attention

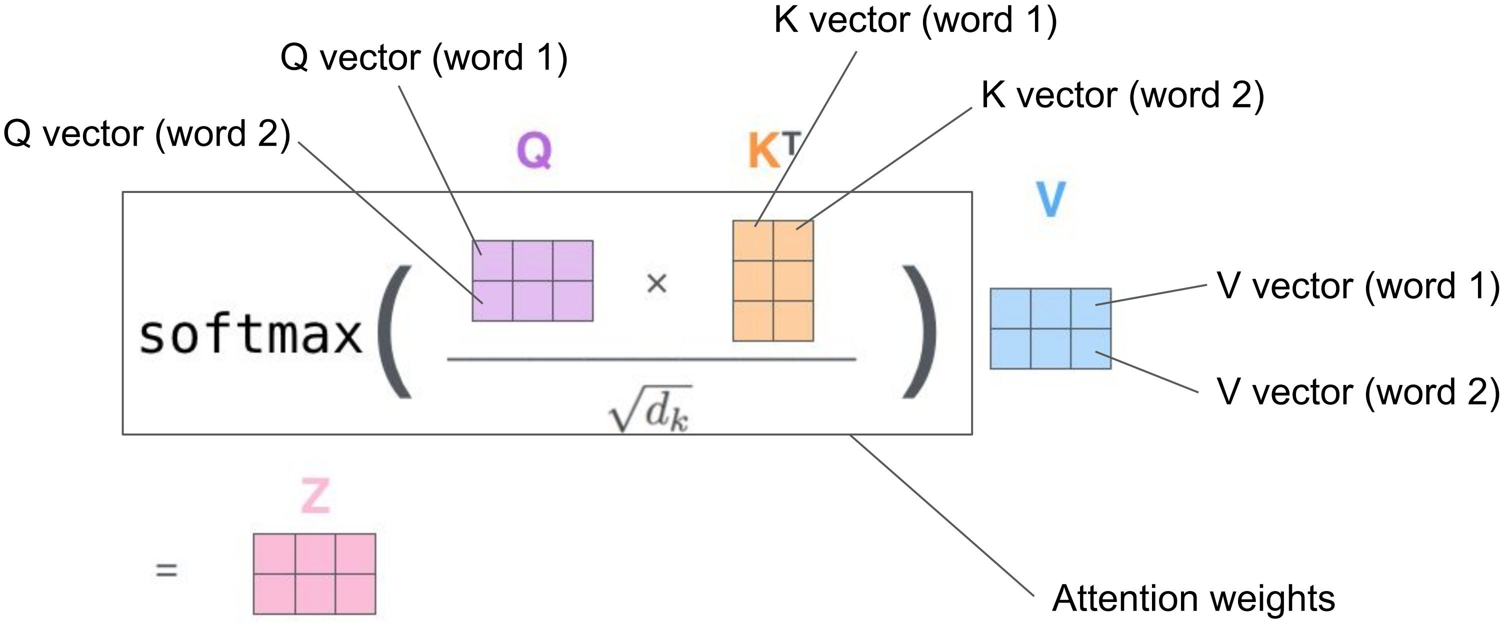
- Each input pays attention to each other input (hence "self"-attention)
- In addition to the input vector (embedding), each input also has a corresponding query, key, and value vector
- How to get the attention weights:
  - Similar to RNN – take a dot product
  - Dot product of query vector of current word with key vector of each of the other input words
  - Scale and softmax to get final weights
- How to get final output:
  - Weighted sum of value vectors (using attention weights)



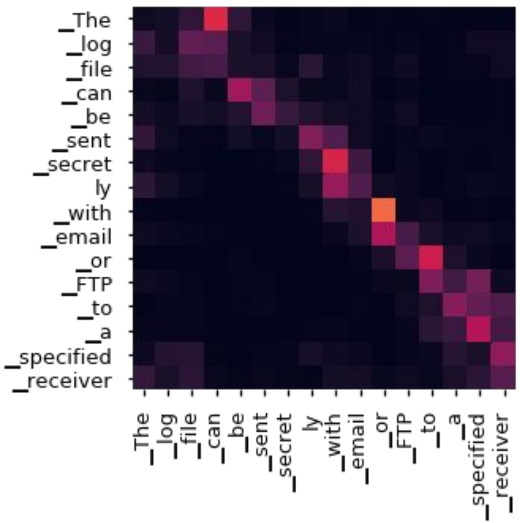
Self-Attention

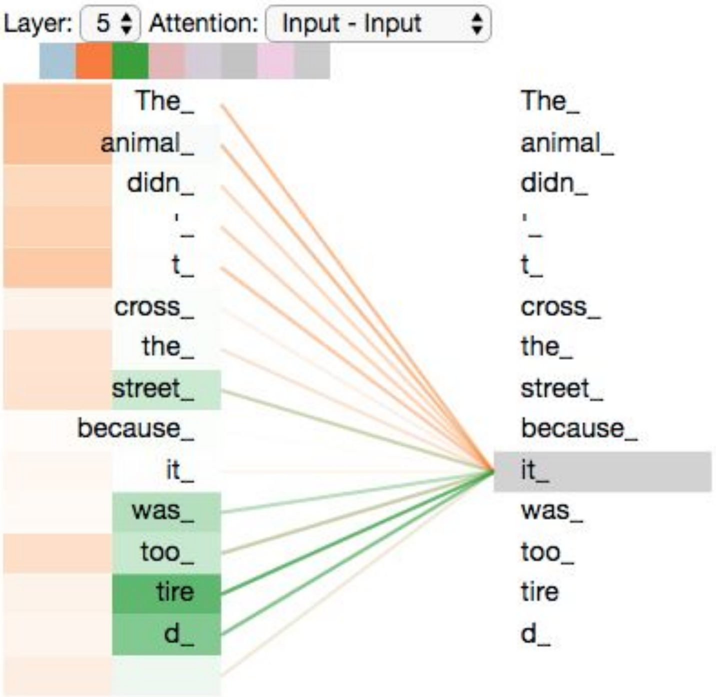
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

In matrix form:



Self-Attention





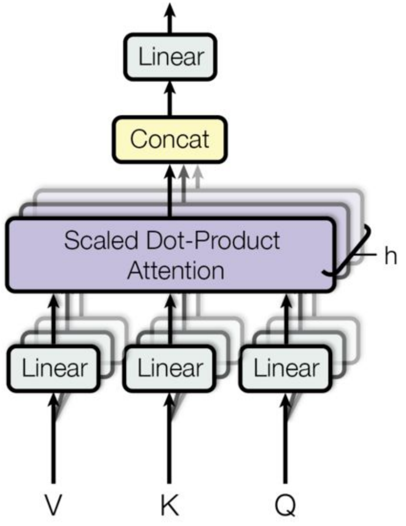
We can plot the (self-)attention weights similar to how we previously plotted attention weights for RNNs

Self-Attention

- In practice, rather than directly calling $\text{Attention}(Q, K, V)$

  We instead call $\text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ where the Ws are parameter (weight) matrices (basically a linear layer)

- This allows us to have multiple "heads" for attention

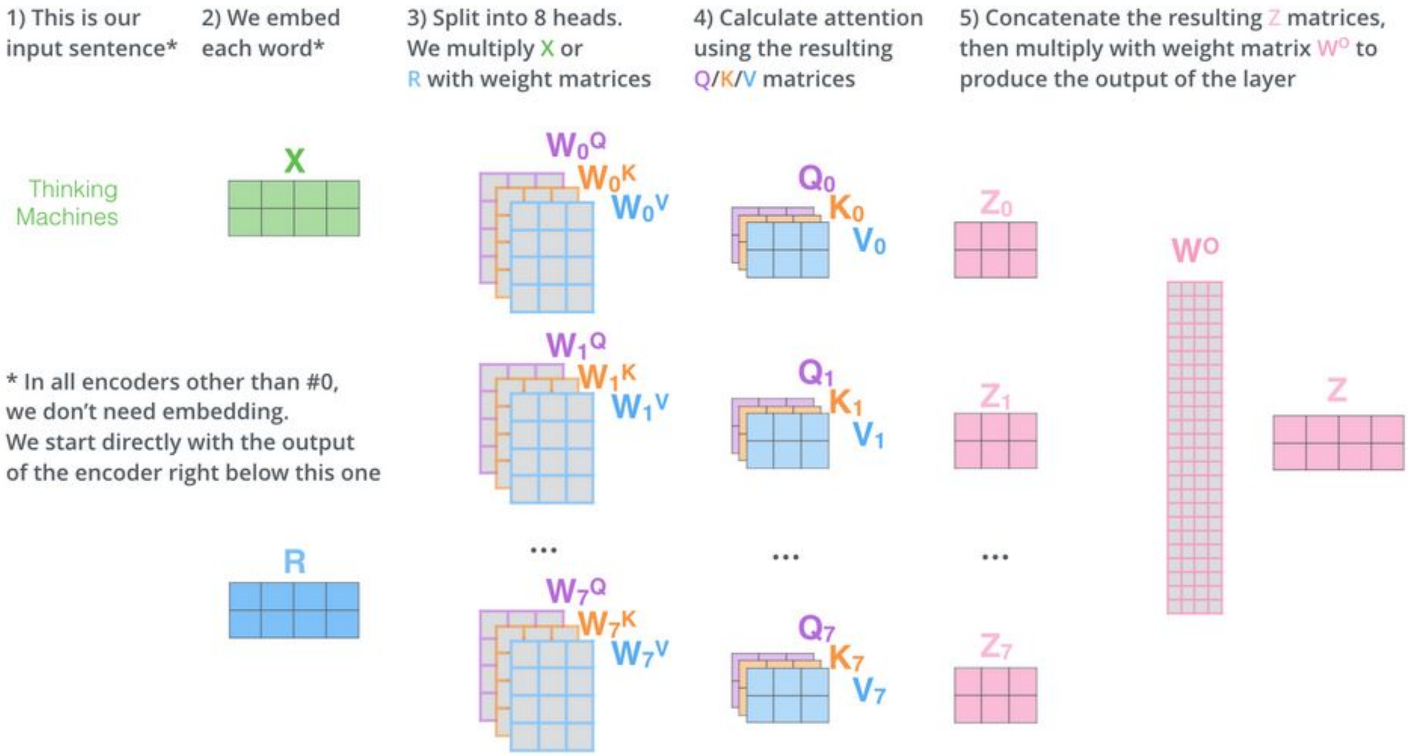


Multi-Headed Attention

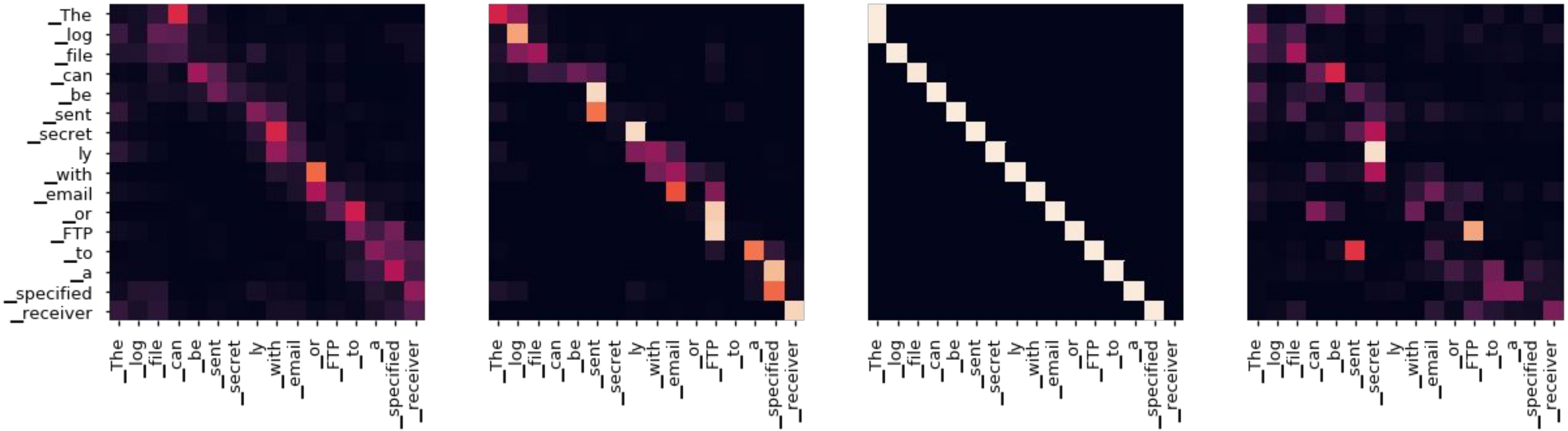
Z_i is the result of a single attention head i

$$\text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Then we concatenate Z1, Z2, … Zk and multiply by a final weight vector to get the final Z



Multi-Headed Attention

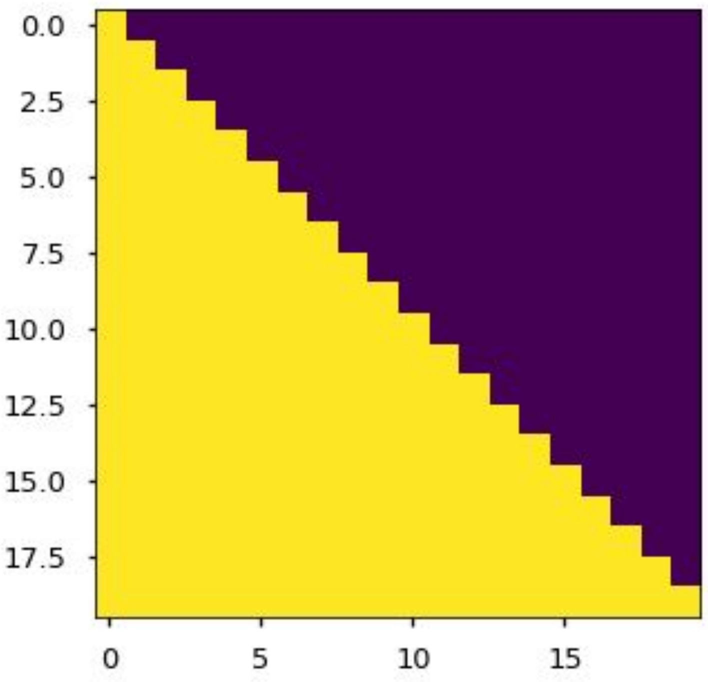


Different attention heads can pay attention to different parts of the input

Decoding Masking

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions

In practice, a “mask” is a binary matrix (0s and 1s), where the 0s represent the positions to “mask out”

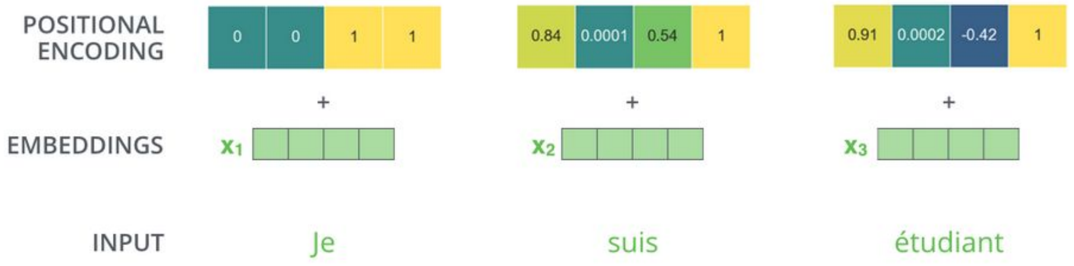


Positional Encoding

- As the name suggests, we need a way to encode the position of the tokens
  - Introduce a "position matrix" to add to the input matrix
  - This "position matrix" is determined as follows:

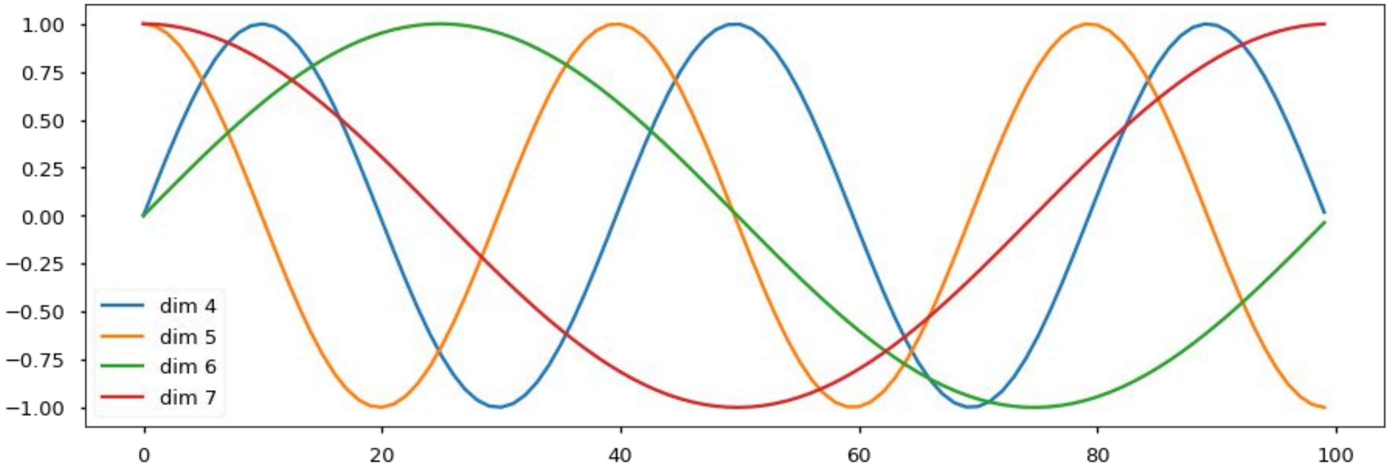
$$\mathbf{P}_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right) \qquad \mathbf{P}_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

- The final input is the sum of the position matrix and the input matrix



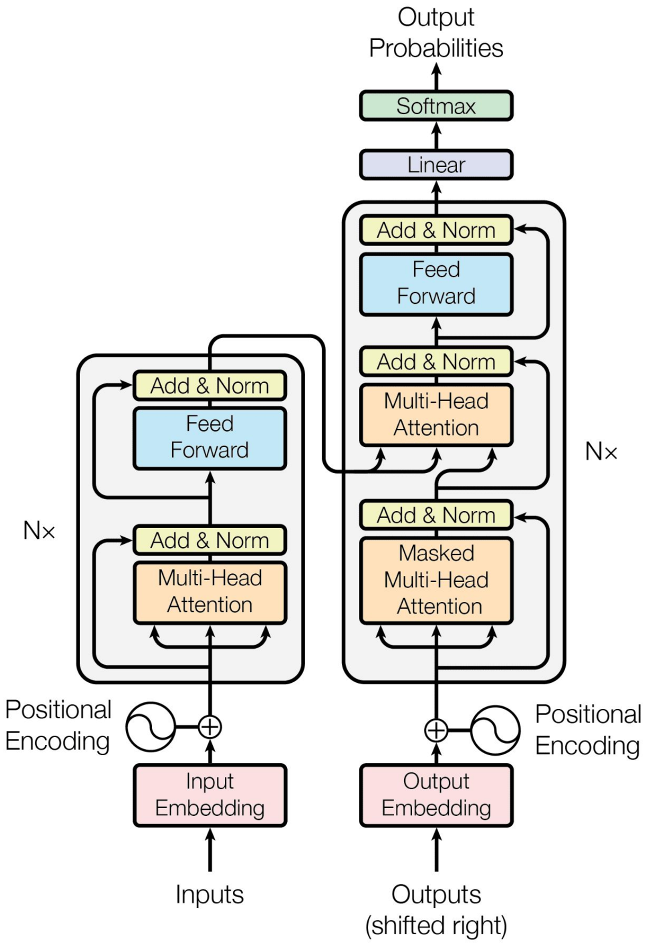
Positional Encoding

$$\mathbf{P}_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right) \quad \mathbf{P}_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right)$$



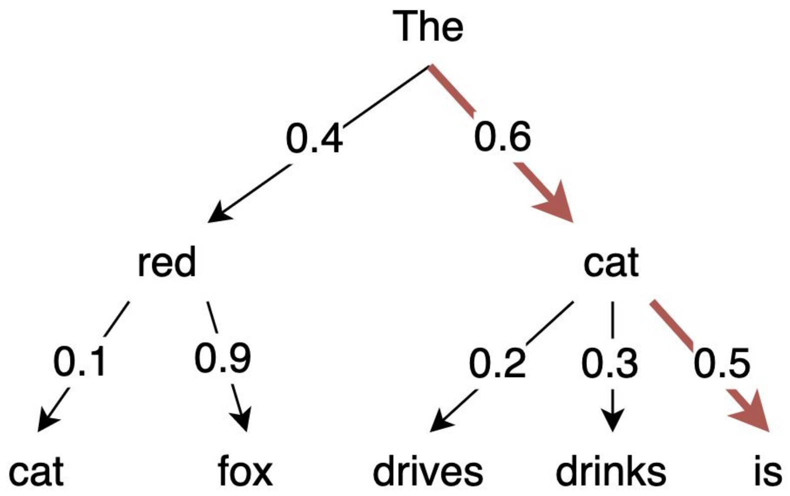
Here, *pos* is the index of the word in the sentence, *d* is the number of dimensions (i.e. embedding size), and *2i (or 2i+1)* is the index along the *d* axis.

Putting it all together



Decoding: Greedy

- Once the model is trained, how do we generate results?
  - Recall that a language model is a **probability distribution over words**.
  - We can simply take argmax at each step!

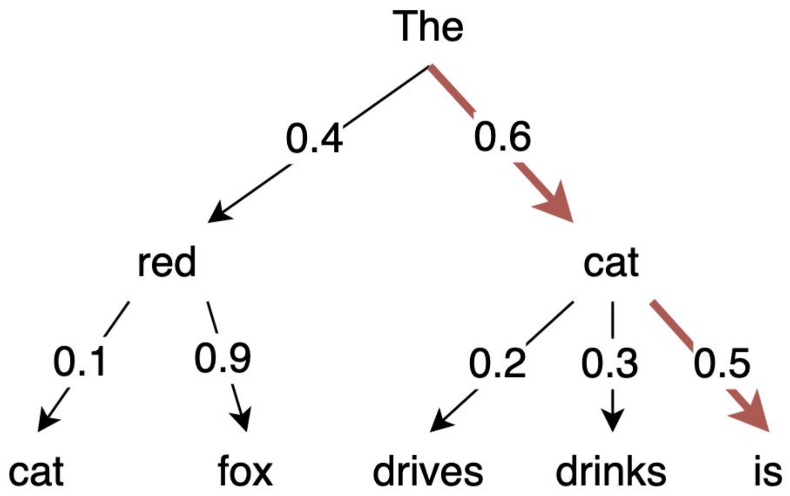


Downside of greedy decoding: This may not necessarily lead to the most probable sequence. In this example, “The red fox” is actually more probably (0.36) than “The cat is” (0.30).

Decoding: Greedy

How do we capture the sentence with the highest probability?
We will need to search the entire tree. This can get computationally expensive (intractable) since each node will introduce V new children (where V=vocabulary size).

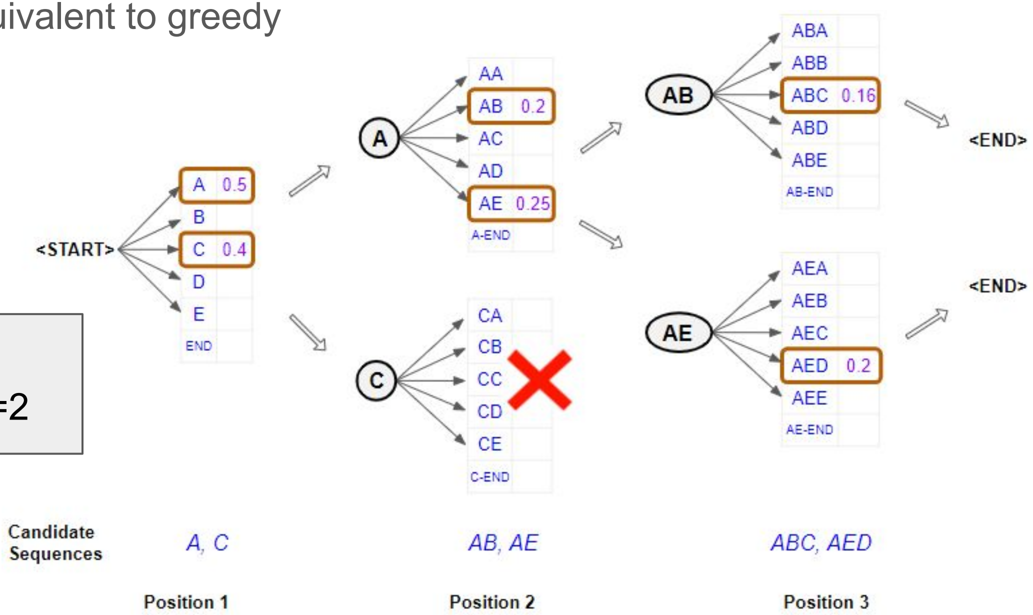
Can we introduce some kind of compromise?



Downside of greedy decoding:
This may not necessarily lead to the most probable sequence. In this example, “The red fox” is actually more probably (0.36) than “The cat is” (0.30).

Decoding: Beam Search

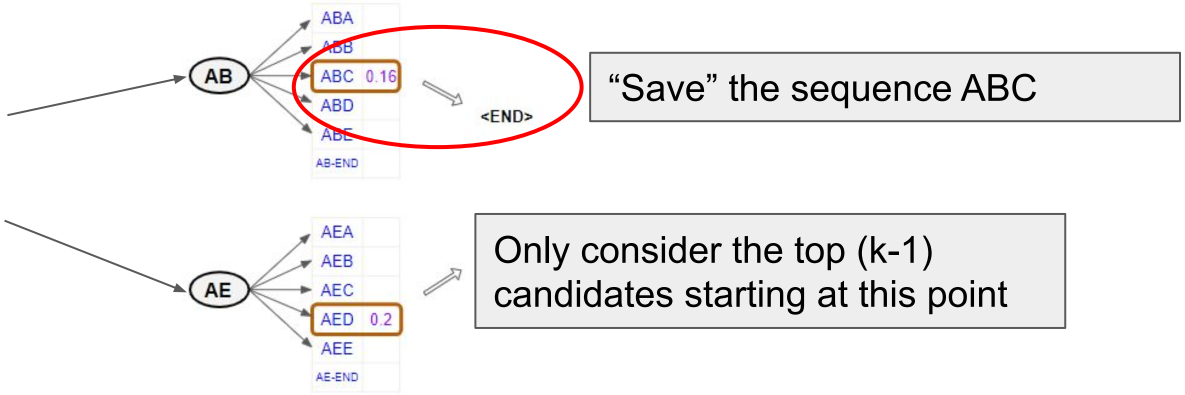
- Instead of keeping only the top candidate at each step, we keep the top k candidates at each step
  - k is called the “beam width” or “beam size”
  - k=1 is equivalent to greedy



Beam search example with k=2

Decoding: Beam Search

- Instead of keeping only the top candidate at each step, we keep the top k candidates at each step
  - k is called the “beam width” or “beam size”
  - k=1 is equivalent to greedy
- What happens when we reach the end-of-sentence [EOS] token?
  - Save that sequence as a possible “final candidate”.
  - Reduce your beam size by one – only consider the top (k-1) candidates starting from now



Decoding: Beam Search

- Instead of keeping only the top candidate at each step, we keep the top k candidates at each step
  - k is called the “beam width” or “beam size”
  - k=1 is equivalent to greedy
- What happens when we reach the end-of-sentence [EOS] token?
  - Save that sequence as a possible “final candidate”.
  - Reduce your beam size by one – only consider the top (k-1) candidates starting from now
- How do we select the final answer?
  - Consider all beams from the final timestep AND all “final candidates” from the [EOS] step
  - Select the sentence with the highest probability
    - In practice, we add log-likelihood probabilities instead of multiplying probabilities (i.e. compute log(p1)+log(p2)+log(p3) instead of p1p2p3.)

Evaluation (BLEU Score)

- This is used to evaluate the quality of our translations
- Score between 0 and 1
- Based on **n-gram matching**
- An n-gram is a contiguous sequence of words of size n
- e.g. “The dog is a happy dog”
    - 1-grams: { [The], [dog], [is], [a], [happy], [dog] }
    - 2-grams: { [The, dog], [dog, is], [is, a], [a, happy], [happy, dog] }
    - 3-grams: { [The, dog, is], [dog, is, a], [is, a, happy], [a, happy, dog] }

BLEU Score Calculation

- We will build up towards this formula:

$$P_n(y, \hat{y}) \triangleq \frac{\sum_{x \in \{\text{unique n-grams in } \hat{y}\}} \min(C(\hat{y}, x), C(y, x))}{|\hat{y}| - n + 1}$$

BLEU Score Calculation

- Let's consider 1-grams for now.
- Target sentence: y = { [The], [dog], [is], [a], [happy], [dog] }
- Predicted sentence: y_hat = { [The], [cat], [is], [a], [very], [happy], [cat] }
- **Initial attempt (wrong)**: Calculate number of n-gram overlaps between prediction and target
  - "The" = 1
  - "cat" = 0
  - "is" = 1
  - "a" = 1
  - "very" = 0
  - "happy" = 1
  - "cat" = 0
- Total = sum(counts) / total_ngrams_in_y_hat = 4/7

Why is this not a good idea?

BLEU Score Calculation

- Let's consider 1-grams for now.
- Target sentence: y = { [The], [dog], [is], [a], [happy], [dog] }
- Predicted sentence: y_hat = { [happy], [happy], [happy], [happy], [happy] }
- **Initial attempt (wrong)**: Calculate number of n-gram overlaps between prediction and target
  - "happy" = 1
  - "happy" = 1
  - "happy" = 1
  - "happy" = 1
  - "happy" = 1
- Total = sum(counts) / total_ngrams_in_y_hat = **5/5**

We need to find a way to deal with repeated words.

BLEU Score: Clipped Precision

- Previously:

  For each n-gram in the predicted sentence (y_hat), count the number of times it appears in the target sentence (y). Take the sum of these counts and divide by total_ngrams_in_y_hat

- With clipped precision:

  For each **unique** n-gram in the predicted sentence (y_hat), count the **min(number of times it appears in y, number of times it appears in y_hat)**. Take the sum of these counts and divide by total_ngrams_in_y_hat

BLEU Score: Clipped Precision

$$P_n(y, \hat{y}) \triangleq \frac{\boxed{\sum_{x \in \{\text{unique n-grams in } \hat{y}\}}} \boxed{\min(C(\hat{y}, x), C(y, x))}}{\boxed{|\hat{y}| - n + 1}}$$

- <u>With clipped precision:</u>

  For each **unique** n-gram in the predicted sentence (y_hat), count the **min(number of times it appears in y, number of times it appears in y_hat)**. Take the sum of these counts and divide by total_ngrams_in_y_hat

BLEU Score Calculation

- Let's consider 1-grams for now.
- Target sentence: y = { [The], [dog], [is], [a], [happy], [dog] }
- Predicted sentence: y_hat = { [happy], [happy], [happy], [happy], [happy] }
- **With clipped precision**:
  - "happy" = min(1, 5) = 1
- Total = sum(counts) / total_ngrams_in_y_hat = 1/5

Are we done? Is this good enough?

BLEU Score Calculation

- Let's consider 1-grams for now.
- Target sentence: y = { [The], [dog], [is], [a], [happy], [dog] }
- Predicted sentence: y_hat = { [happy], [happy], [happy], [happy], [happy] }
- **With clipped precision**:
    - "happy" = min(1, 5) = 1
- Total = sum(counts) / total_ngrams_in_y_hat = 1/5

- Predicted sentence: y_hat = { [happy] }
- **With clipped precision:**
    - "happy" = min(1, 1) = 1
- Total = sum(counts) / total_ngrams_in_y_hat = **1/1**

How do we deal with short predictions?

BLEU Score: Brevity Penalty

- We introduce a **brevity penalty**
- If prediction is shorter than the target:

  penalty = exp(1 - (len_target / len_prediction))

- Otherwise, no penalty:

  penalty = 1

- We multiply this penalty with our BLEU score

BLEU Score Calculation

- Let's consider 1-grams for now.
- Target sentence: y = { [The], [dog], [is], [a], [happy], [dog] }
- Predicted sentence: y_hat = { [happy] }
- **With clipped precision and brevity penalty**:
    - "happy" = min(1, 1) = 1
- Total = [sum(counts) / total_ngrams_in_y_hat] * brevity_penalty

  = (1/1) * exp(1 - 6/1) = 1 * exp(-5) = **0.0067**

BLEU Score: Final Calculation

- Previously, we were only considering 1-grams.
- Repeat this same process for 2-grams, 3-grams, … until k-grams.
- This gives scores P1, P2, … Pk
- The final BLEU score is the **geometric mean** of (P1, P2, … Pk)

BLEU Score: Worked Example

- Target sentence: {"the", "fat", "cat", "ate", "the", "fat", "rat"}
- Predicted sentence: {"the", "fat", "cat", "ate" "the", "cat"}

- Suppose k=3. We use "c()" here to denote the clipped precision
- P1: c(["the"])=min(2,2), c(["fat"])=min(2,1), c(["cat"])=min(1,2), c(["ate"])=min(1,1)
- P1 = (2+1+1+1) / total_num_1_grams_in_pred = **5/6**
- P2: c(["the fat"])=1, c(["fat cat"])=1, c(["cat ate"])=1, c(["ate the"])=1, c(["the cat"])=0
- P2 = (1+1+1+1+0) / total_num_2_grams_in_pred = **4/5**
- P3: c(["the fat cat"])=1, c(["fat cat ate"])=1, c(["cat ate the"])=1, c(["ate the cat"])=0
- P3 = (1+1+1+0) / total_num_3_grams_in_pred = **3/4**
- Final score = geometric_mean(5/6, 4/5, 3/4) * brevity_penalty

  = (3/6)^(1/3) * exp(1 - 7/6) = **0.672**

Programming Portion: Auto-grader

1. Positional Encoding
2. Self-attention Layer
3. Lookahead Mask
4. Beam Search Prediction
5. BLEU Score (617 only)

Programming Portion: Experiments

1. (autograder)
2. Plotting train and test loss
   - This should be relatively straightforward (similar to previous assignments)
   - Provided function: train()
   - Make sure to save your models, as you will be loading/using them in the next questions
3. Decoding (beam search)
   - Task: Generate translations for a few sentences and report the generations
   - Provided function: decode_sentence()
4. Visualizing Attention
   - Provided function: visualize_attention()
   - How to generate attention matrix: output of Transformer.forward

Programming Portion: Experiments

5. Beam Search

- Task: Investigate the effect of different beam sizes

6. BLEU Score (617 only)

- Generate output first
- Then take BLEU_Score(output_prediction, ground_truth_target)