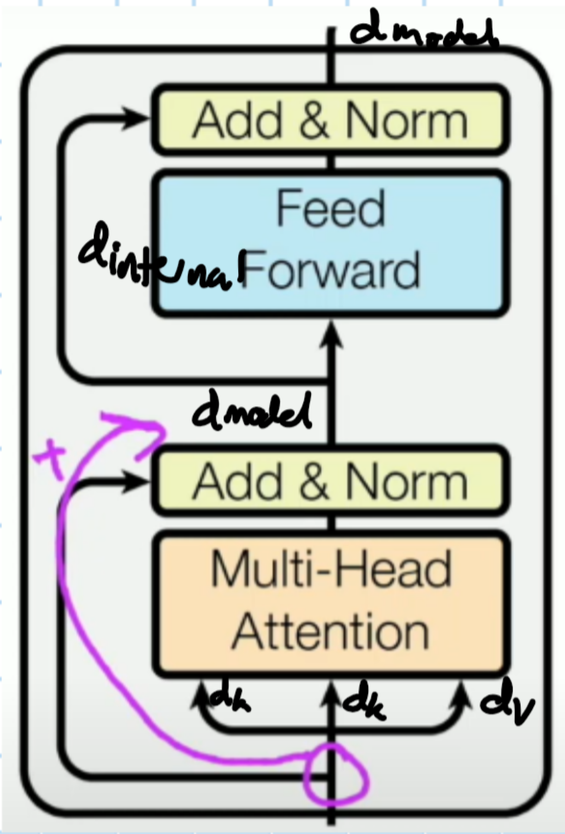
Transformer Architecture

Transformers are not only self attention, you also have FFN layers

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

usually simple, one hidden layer but where most params are

input is added to output (represented by the arrows into "Add & Norm"



vectors: d_{model}
Queries/keys: d_k, always smaller than d_{model} (saves computation)
Values: seperate dim d_v, output is multiplied by W^o which is $d_v \times d_{model}$ to get back to d_{model}

FFN can explode the dimension with W_1 and collapse it back with W_2

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

layers

	N	d_{model}	d_{ff}	h	d_k	d_v
base	6	512	2048	8	64	64

gets expensive quickly

- From Vaswani et al.

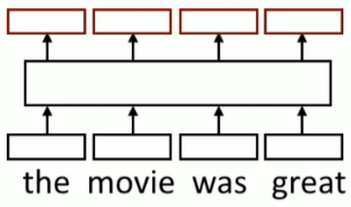
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}
GPT-3 Small	125M	12	768	12	64
GPT-3 Medium	350M	24	1024	16	64
GPT-3 Large	760M	24	1536	16	96
GPT-3 XL	1.3B	24	2048	24	128
GPT-3 2.7B	2.7B	32	2560	32	80
GPT-3 6.7B	6.7B	32	4096	32	128
GPT-3 13B	13.0B	40	5140	40	128
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128

- From GPT-3; d_{head} is our d_k

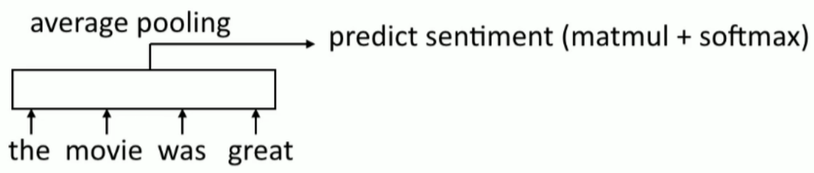
80% of the computation is in the FFN layers
other is attention
most of the information in models like GPT is from the FFN params

Using Transformers

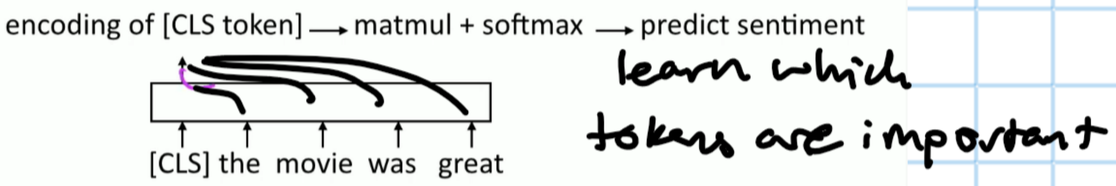
What do Transformers produce?



- Encoding of each word — can pass this to another layer to make a prediction (like predicting the next word for language modeling)
- Like RNNs, Transformers can be viewed as a transformation of a sequence of vectors into a sequence of context-dependent vectors



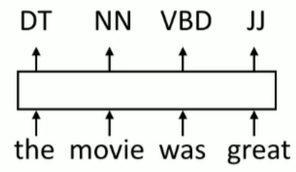
- Alternative: use a placeholder [CLS] token at the start of the sequence.



- Because [CLS] attends to everything with self-attention, it can do the pooling for you!

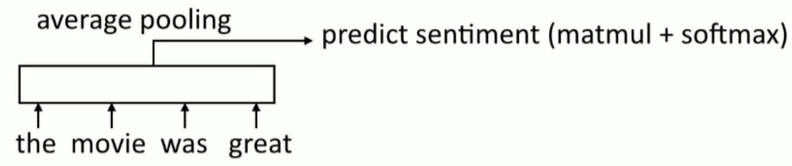
Transformer Uses

- Transducer: make some prediction for each element in a sequence



output **y** = score for each tag, then softmax

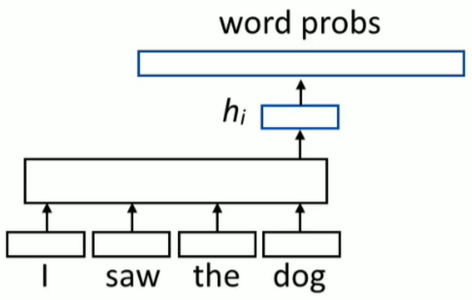
- Classifier: encode a sequence into a fixed-sized vector and classify that



Transformer Language Modeling

Goal: predict next word

Transformer Language Modeling



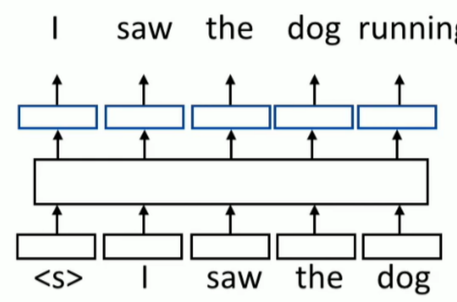
$$P(w|\text{context}) = \frac{\exp(\mathbf{w} \cdot \mathbf{h_i})}{\sum_{w'} \exp(\mathbf{w}' \cdot \mathbf{h_i})}$$

equivalent to

$$P(w|\text{context}) = \text{softmax}(W\mathbf{h}_i)$$

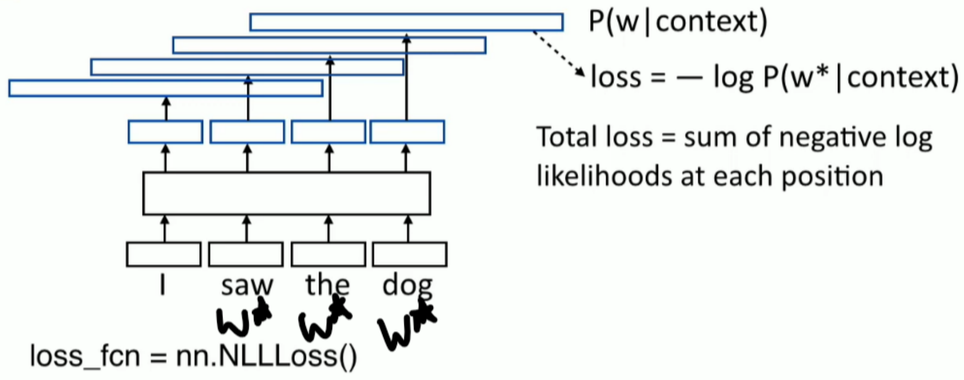
- h_i is the embedding of *dog* produced by the Transformer
- W is a (vocab size) x (hidden size) matrix; linear layer in PyTorch (rows are word embeddings)

Training Transformer LMs



- Input is a sequence of words, output is those words shifted by one
- Allows us to train on predictions across several timesteps simultaneously (similar to batching but this is NOT what we refer to as batching)

Training Transformer LMs



Total loss = sum of negative log likelihoods at each position

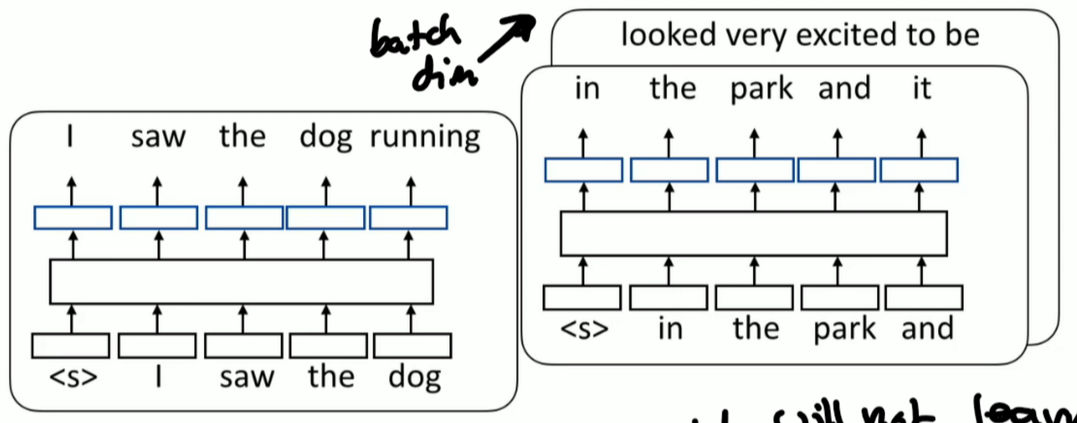
```
loss_fcn = nn.NLLLoss()
loss += loss_fcn(log_probs, ex.output_tensor)
                 [seq len, num output classes]  [seq len]
```

- Batching is a little tricky with NLLLoss: need to collase [batch, seq len, num classes] to [batch * seq len, num classes]. You do not need to batch

Batched LM Training

I saw the dog running | in the park and it | looked very excited to be | there

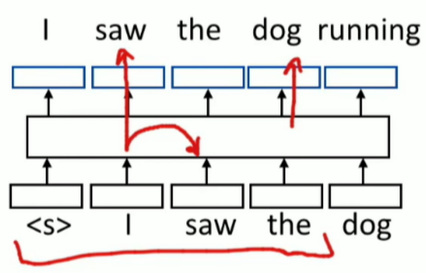
batch dim



if you have pos encod., the model will not learn to attend to stuff far away.

A Small Problem with Transformer LMs

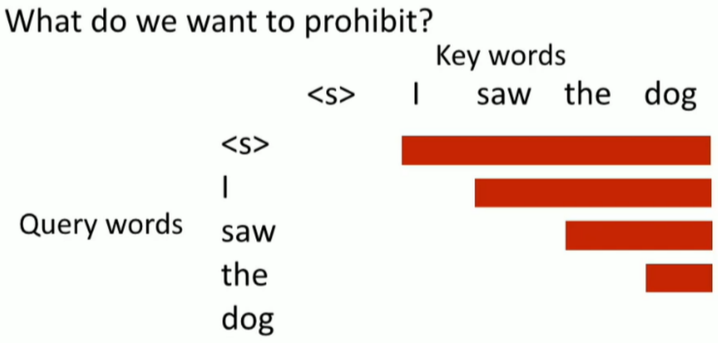
- This Transformer LM as we've described it will *easily* achieve perfect accuracy. Why?



- With standard self-attention: "I" attends to "saw" and the model is "cheating". How do we ensure that this doesn't happen?

Attention Masking

- What do we want to prohibit?



- This is called a **causal** mask (also, causal self-attention / causal Transformers). Only things in the "past" can influence the "present"

Implementation in PyTorch

- nn.TransformerEncoder can be built out of nn.TransformerEncoderLayers, can accept an input and a mask for language modeling:

```
# Inside the module; need to fill in size parameters
layers = nn.TransformerEncoderLayer([...])
transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers=[...])
[. . .]
# Inside forward(): puts negative infinities in the red part
mask = torch.triu(torch.ones(len, len) * float('-inf'), diagonal=1)
output = transformer_encoder(input, mask=mask)
```

Transformers Extension

Scaling Laws

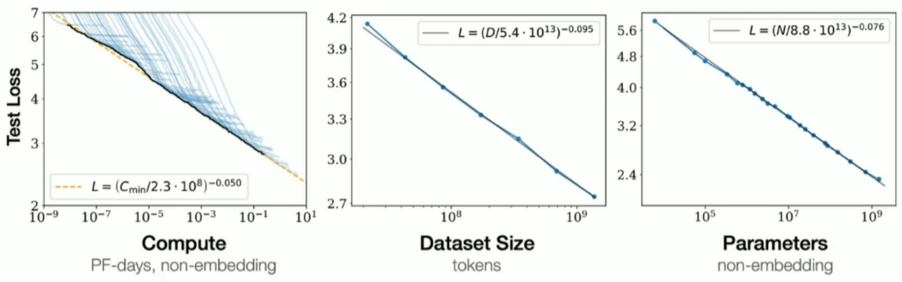


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

- Transformers scale really well!

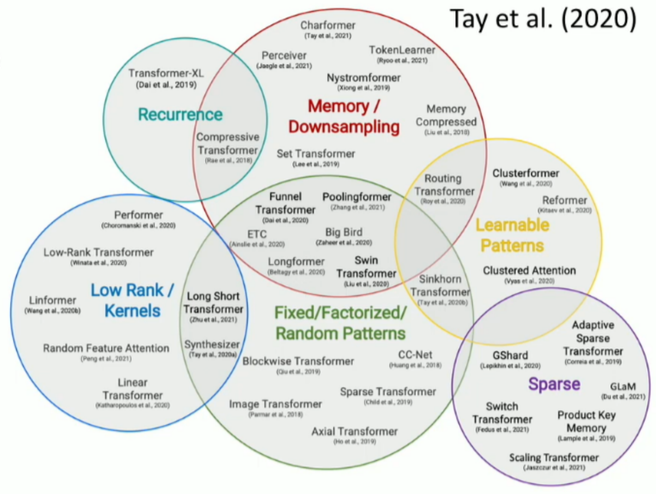
Kaplan et al. (2020)

just increasing training probably will decrease loss, which is rare for such models

Transformer Runtime

- Even though most parameters and FLOPs are in feedforward layers, Transformers are still limited by quadratic complexity of self-attention
- Many ways proposed to handle this

Tay et al. (2020)



Performers

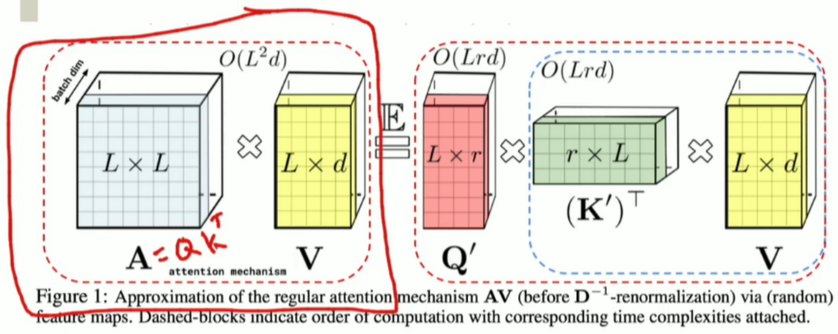
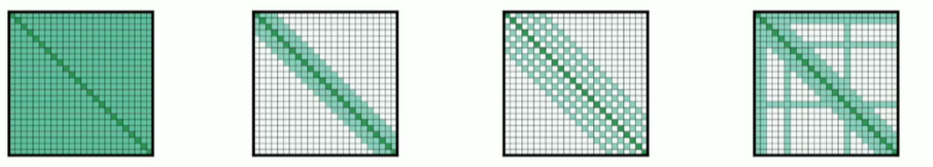


Figure 1: Approximation of the regular attention mechanism $\mathbf{AV}$ (before $\mathbf{D}^{-1}$-renormalization) via (random) feature maps. Dashed-blocks indicate order of computation with corresponding time complexities attached.

- No more len[2] term, but we are fundamentally approximating the self-attention mechanism (cannot form **A** and take the softmax)

Choromanski et al. (2020)

Longformer



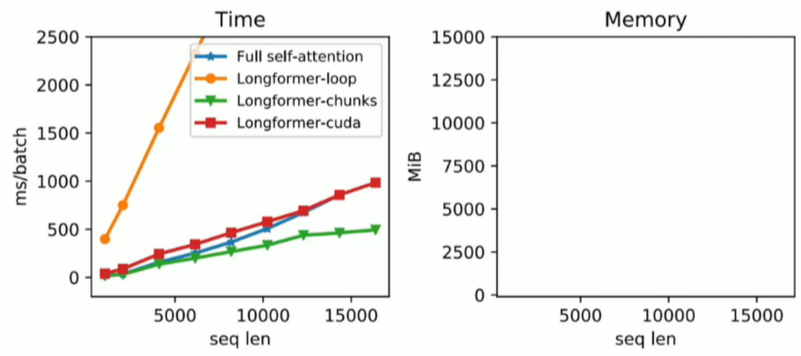
(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

Figure 2: Comparing the full self-attention pattern and the configuration of attention patterns in our Longformer.

- Use several pre-specified self-attention patterns that limit the number of operations while still allowing for attention over a reasonable set of things
- Scales to 4096-length sequences

Beltagy et al. (2021)

Attention Maps



- Loop = non-vectorized version
- What will the memory profile look like?

Beltagy et al. (2021)

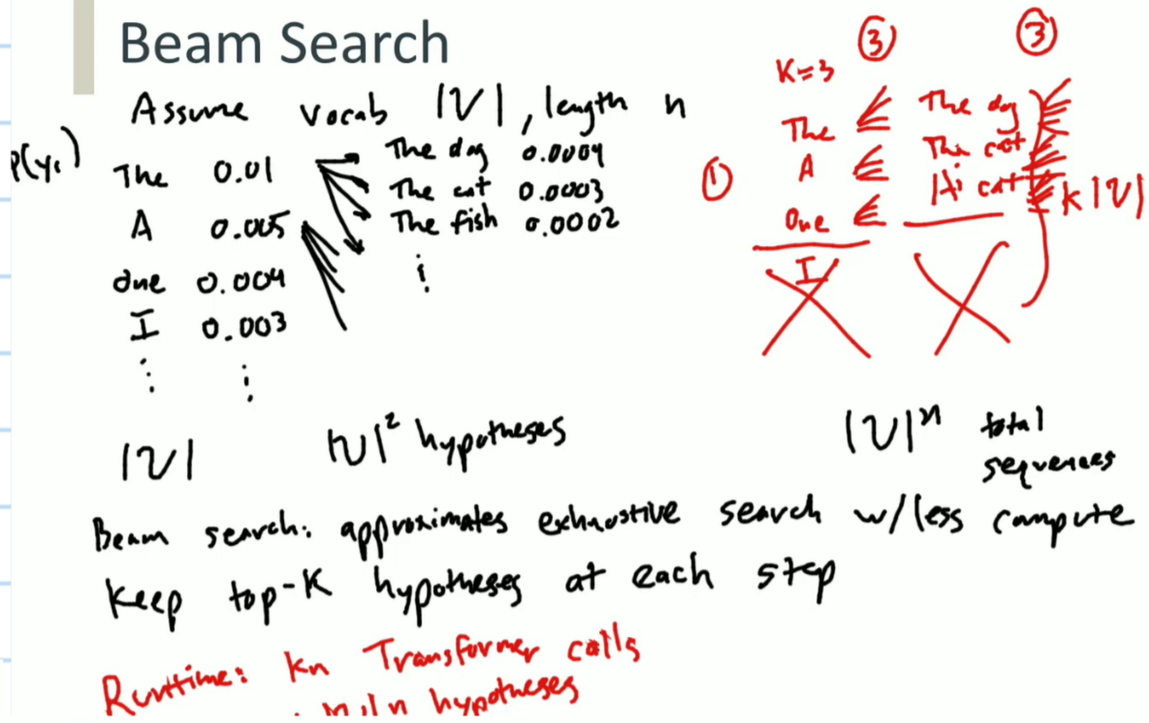
Beam Search (and other decoding strats)

cut-off non-top k

Decoding Strategies

- LMs place a distribution $P(y_i | y_1, ..., y_{i-1})$
- seq2seq models place a distribution $P(y_i | \mathbf{x}, y_1, ..., y_{i-1})$
- Generation from both models looks similar; how do we do it?
  - Option 1: max y_i $P(y_i | y_1, ..., y_{i-1})$ — take greedily best option
  - Option 2: use beam search to find the sequence with the highest prob.
  - Option 3: sample from the model; draw y_i from that distribution
- This segment: beam search

Beam Search



Beam Search: Applications

- Beam search is used frequently in seq2seq conditional generation settings like machine translation where finding the highest-probability hypothesis is really important
- Better models make beam search less crucial. The models are better at "planning ahead" internally so it's less likely that they assign a sequence $y_1 \ldots y_{n-1}$ high probability if there's no good y_n for it

Story Generation (GPT-2)

Context: 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.

Beam Search, *b*=32:
"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 ..."

Pure Sampling:
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."

- Beam search degenerates and starts repeating. If you see a fragment repeated 2-3x, it has very high probability to keep repeating
- Sampling is too noisy, introduces many grammatical errors

Nucleus Sampling

Decoding Strategies

- LMs place a distribution $P(y_i | y_1, ..., y_{i-1})$
- seq2seq models place a distribution $P(y_i | \mathbf{x}, y_1, ..., y_{i-1})$
- Generation from both models looks similar; how do we do it?
  - Option 1: max y_i $P(y_i | y_1, ..., y_{i-1})$: greedily take best option
  - Option 2: use beam search to find the sequence with the highest prob.
  - Option 3: sample from the model; draw y_i from that distribution
- Beam search is great for applications like machine translation or question answering where the answers are somewhat constrained. But LLMs are increasingly being used for **open-ended** generation tasks where there is not one right answer. How do these compare here?

more creativity

Drawbacks of Beam Search: Degeneration

- Beam search fails because the model is *locally normalized*
- Let's look at all the individual decisions that get made here

Beam Search, *b*=32:
"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 ..."

P(*Nacional | ... Universidad*) is high
P(*Autónoma | ... Universidad Nacional*) is high
P(*de | ... Universidad Nacional Autónoma*) is high
P(*México | Universidad Nacional Autónoma de*) is high
P(*/ | ... México*) and P(*Universidad | ... México /*) — these probabilities may be low. But those are just 2/6 words of the repeating fragment

- Each word is likely given the previous words, but the sequence is bad. (Globally-normalized models can judge this, but are computationally hard to use)

Holtzman et al. (2019)

Drawbacks of Sampling: Long Tail

- Sampling is "too random"

Pure Sampling:
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

P(*y* | ... they live in a remote desert uninterrupted by)

0.01 roads
0.01 towns
0.01 people
0.005 civilization

Good options, maybe accounting for 90% of the total probability mass. So a 90% chance of getting something good

...
0.0005 town

Long tail with 10% of the mass

- On average, every 10 words we will get something from the 10% tail of the distribution. 100 words -> 1%. Such words can really derail us!

Holtzman et al. (2019)

Nucleus Sampling

P(*y* | ... they live in a remote desert uninterrupted by)

0.01 roads
0.01 towns
0.01 people
0.005 civilization

→ renormalize and sample

cut off after *p*% of mass

- Define a threshold *p*. Keep the most probable options account for *p*% of the probability mass (the *nucleus*), then sample among these.
- To implement: sort options by probability, truncate the list once the total exceeds *p*, then renormalize and sample from it

Decoding Strategies

- LMs place a distribution $P(y_i | y_1, ..., y_{i-1})$
- seq2seq models place a distribution $P(y_i | \mathbf{x}, y_1, ..., y_{i-1})$
- Generation from both models looks similar; how do we do it?
  - Option 1: max y_i $P(y_i | y_1, ..., y_{i-1})$ — take greedily best option
  - Option 2: use beam search to find the sequence with the highest prob.
  - ~~Option 3: sample from the model; draw y_i from that distribution~~
  - Option 4: nucleus sampling

ChatGPT probably uses something like Nucleus Sampling