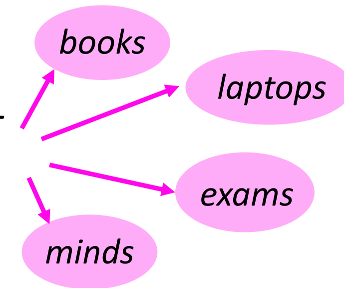
Lecture 22: Large Language Models and RLHF

What is a language model?

- **Language Modeling** is the task of predicting what word comes next

the students opened their ______



- More formally: given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$, compute the probability distribution of the next word $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $\boldsymbol{x}^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

- A system that does this is called a **Language Model**

Probabilistic language models

- You can also think of a Language Model as a system that assigns a probability to a piece of text

- For example, if we have some text $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

$$\begin{aligned} P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) &= P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} |\, \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} |\, \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)}) \\ &= \prod_{t=1}^{T} \underbrace{P(\boldsymbol{x}^{(t)} |\, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what our LM provides}} \end{aligned}$$

N-gram models

the students opened their ______

- **Question**: How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n*-gram Language Model!

- **Definition:** An *n*-gram is a chunk of *n* consecutive words.
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - four-grams: “the students opened their”

- **Idea:** Collect statistics about how frequent different n-grams are and use these to predict next word.

N-gram models and Markov assumption

- First we make a Markov assumption: $x^{(t+1)}$ depends only on the preceding n-1 words

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$ (assumption)

prob of a n-gram

prob of a (n-1)-gram

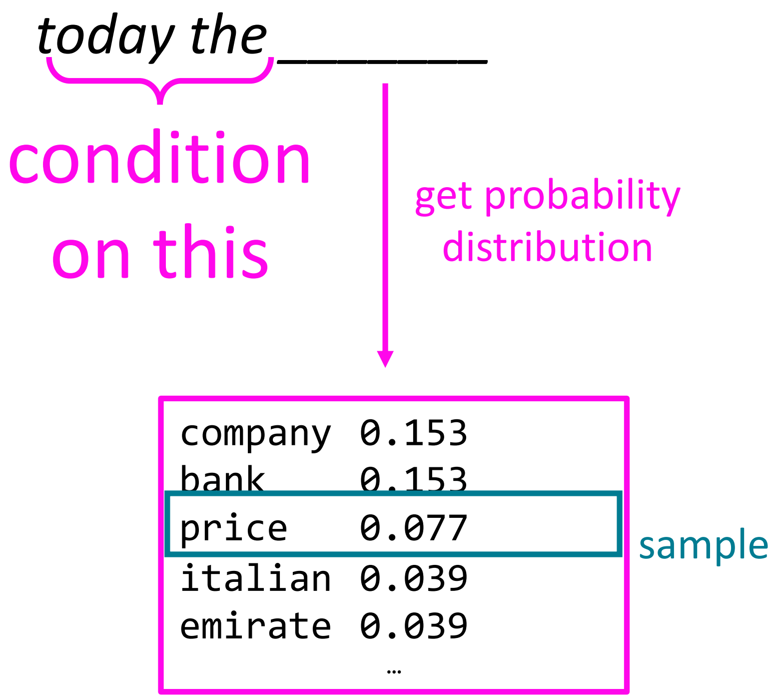
$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}$$ (definition of conditional prob)

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\mathrm{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\mathrm{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}$$ (statistical approximation)

Generating text

You can also use a Language Model to generate text



Generating text

You can also use a Language Model to generate text

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

Surprisingly grammatical!

…but **incoherent.** We need to consider more than three words at a time if we want to model language well.

But increasing *n* worsens sparsity problem, and increases model size…

How good are n-gram models?

You can also use a Language Model to generate text

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

Surprisingly grammatical!

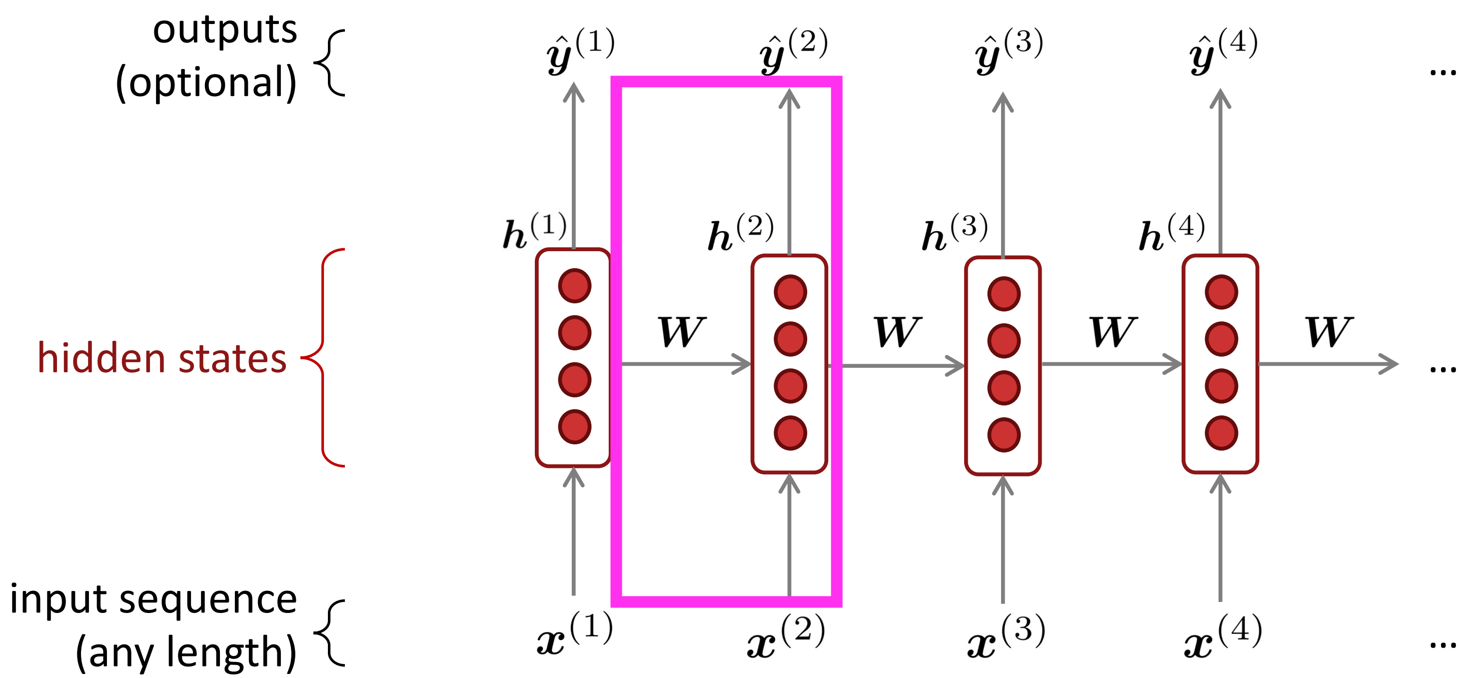
…but **incoherent.** We need to consider more than three words at a time if we want to model language well.

But increasing *n* worsens sparsity problem, and increases model size…

Problems with n-gram models

- Small n means model is not good enough
- Large n means that many combinations do not occur in the data - *sparsity*
- Generally speaking, fixed n is very rigid

Recurrent nets (RNNs)



A Simple RNN Language Model

$\hat{\boldsymbol{y}}^{(4)} = P(\boldsymbol{x}^{(5)}|\text{the students opened their})$

output distribution

$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states

$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$

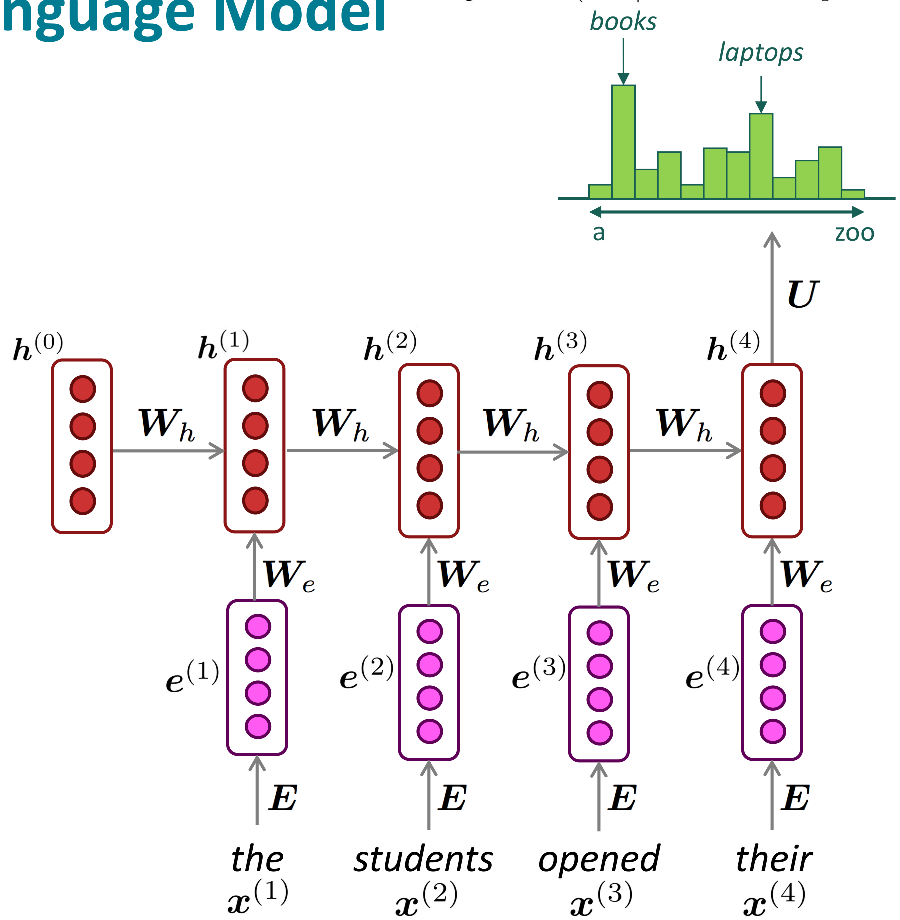
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

words / one-hot vectors

$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



Note: *this input sequence could be much longer now!*

RNN training

- Get a big corpus of text which is a sequence of words $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{\boldsymbol{y}}^{(t)}$ for *every step t*.
  - i.e., predict probability dist of *every word*, given words so far

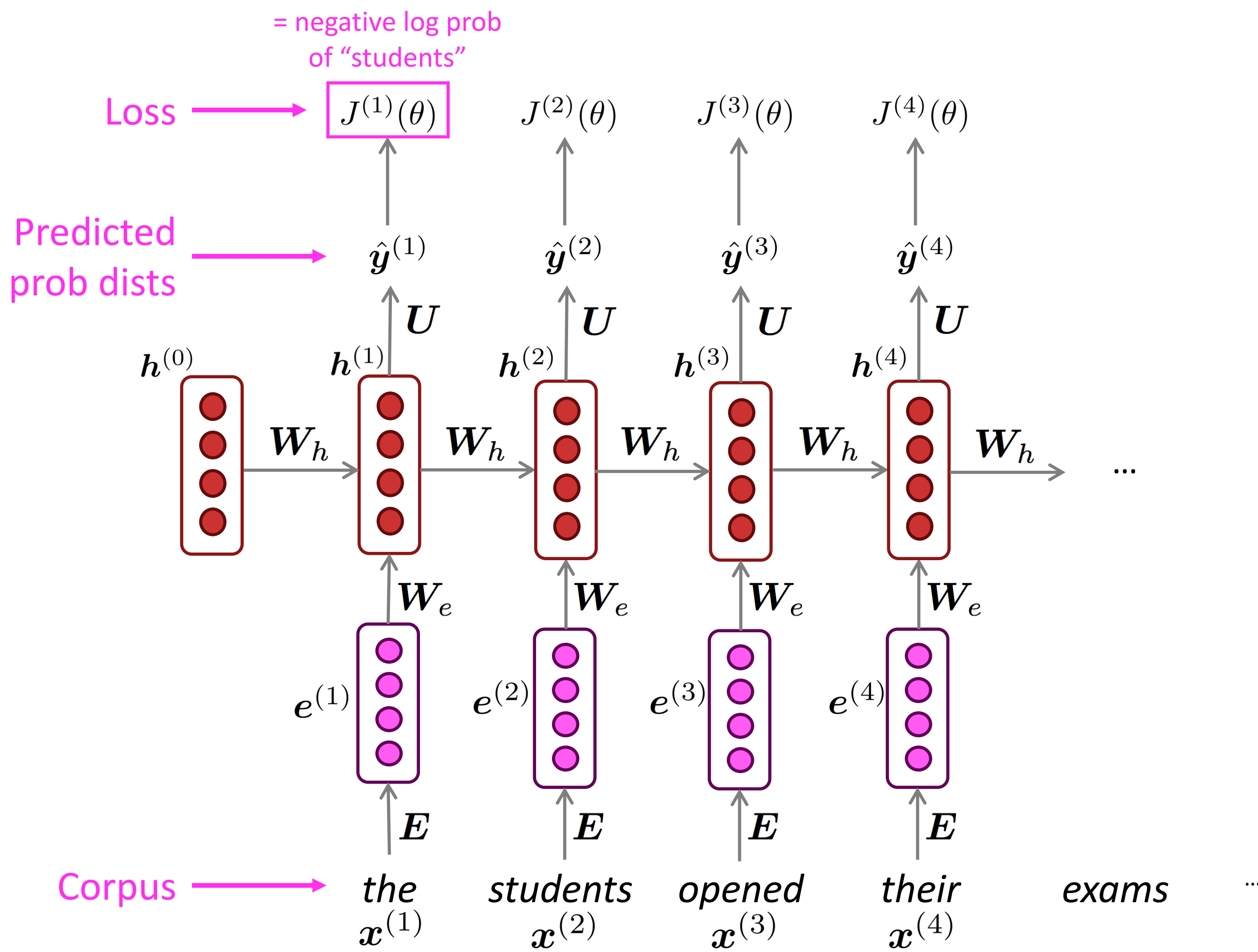
- Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $\boldsymbol{x}^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Average this to get overall loss for entire training set:

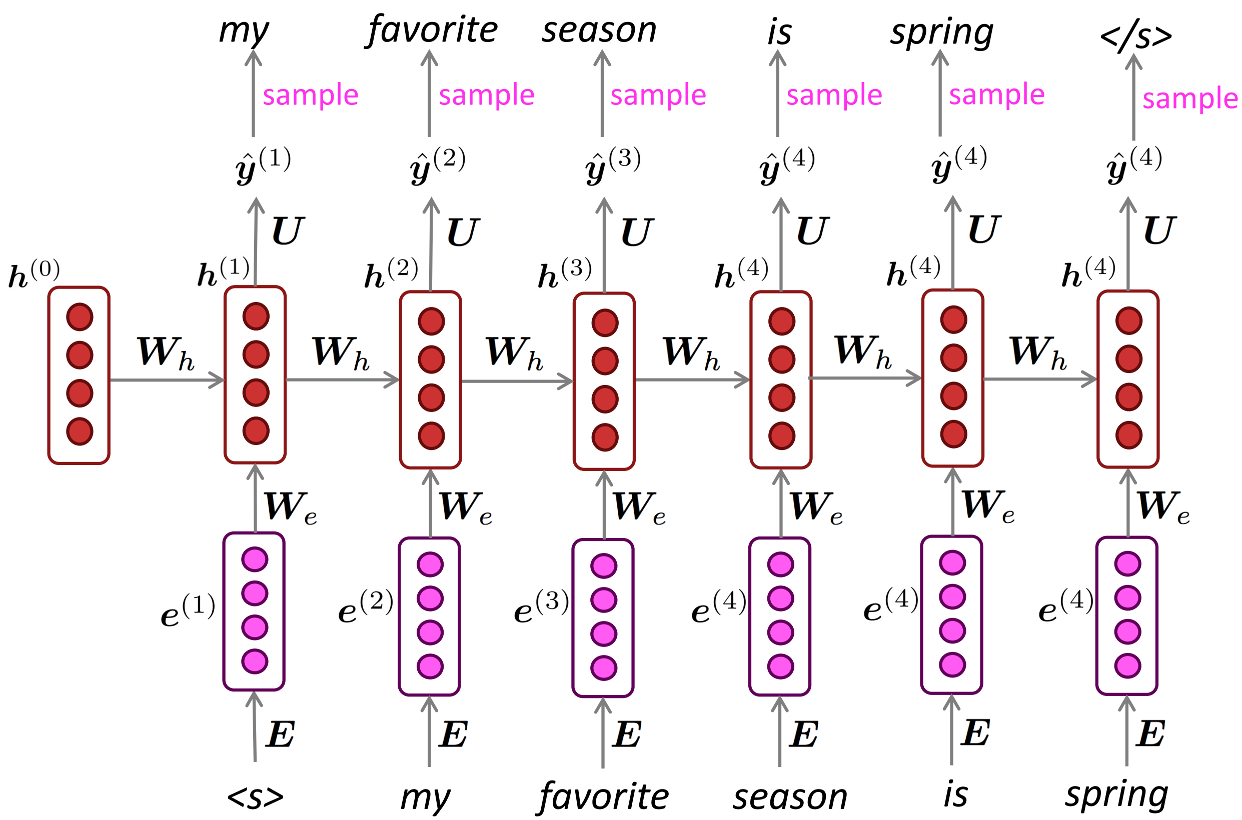
$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

RNN training



Generating text with RNNs

Just like an n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output becomes next step's input.



RNN example

Let's have some fun!

- You can train an RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on *Harry Potter*:

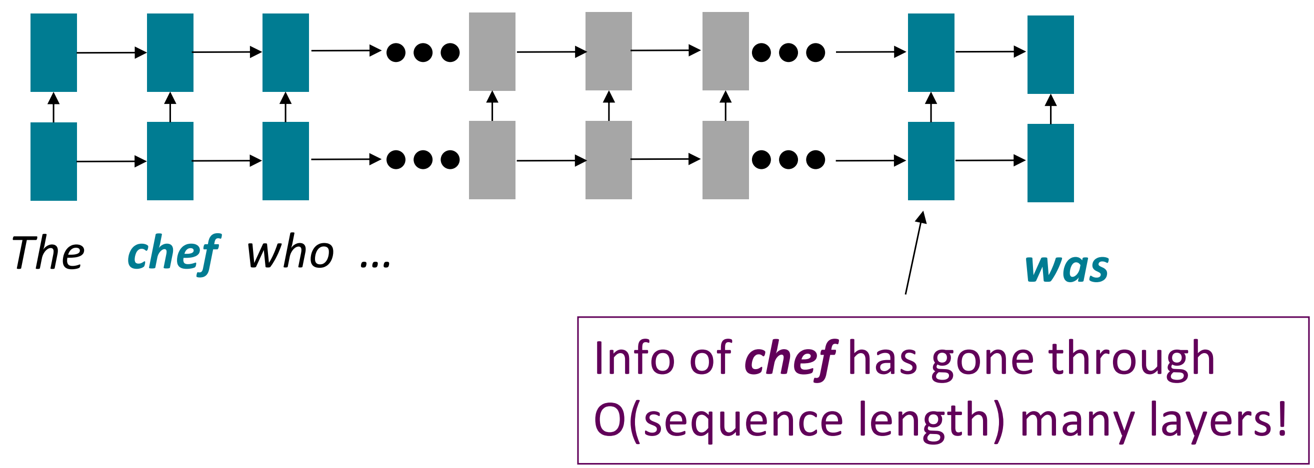
"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

Source: https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6

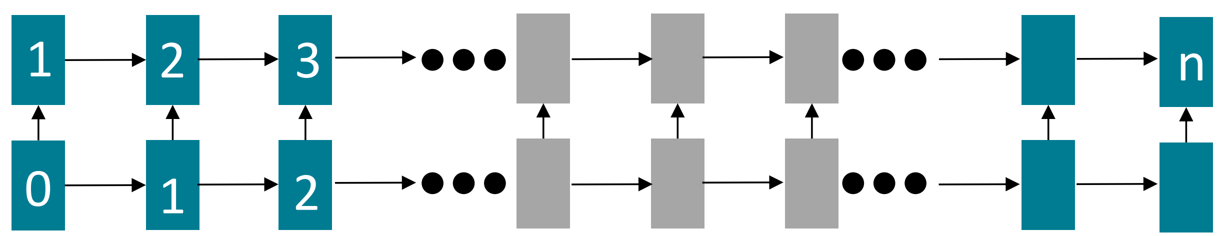
Issues with RNNs: Linear interaction distance

- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…

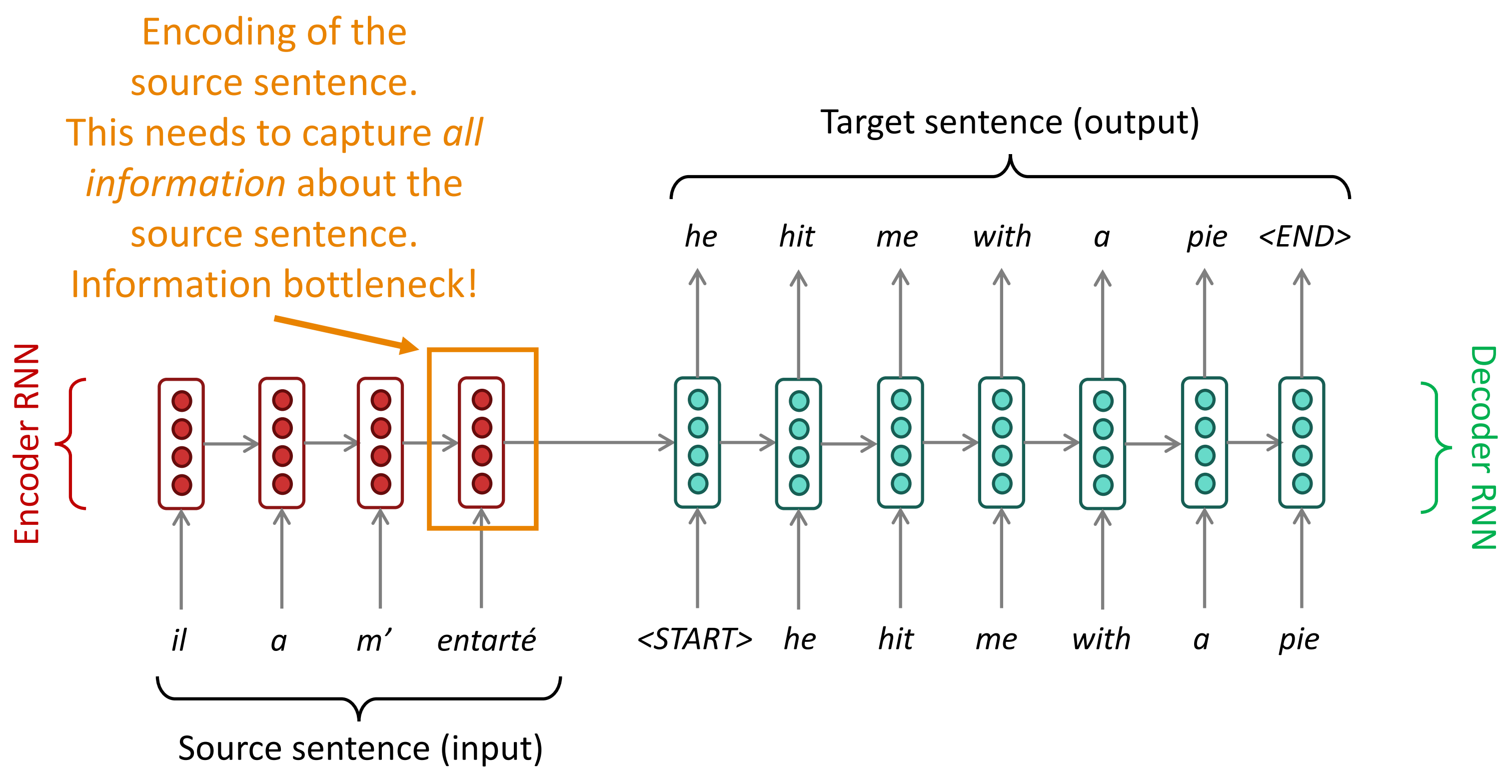


Issues with RNNs: Hard to parallelize

- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



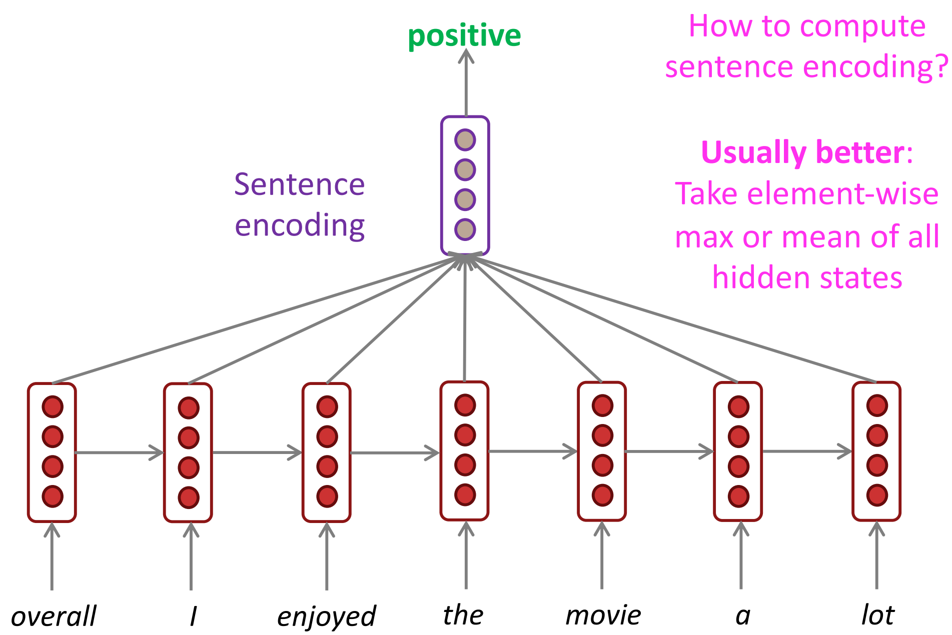
Issues with RNNs: Bottleneck problem



Solution: Attention

- On each step of decoding, use *direct connection to the encoder* to focus on a particular part of the sequence
- A bit like what humans do!
- Attention provides a solution to the bottleneck problem!

Pooling in RNNs

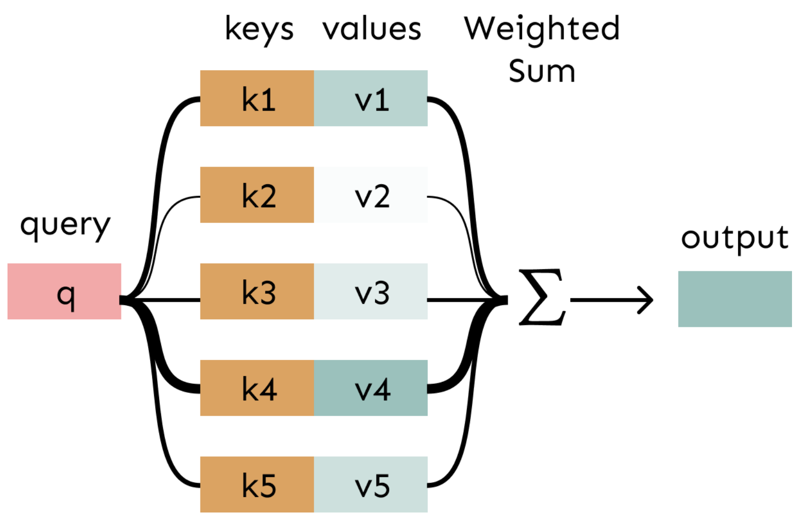


- Starting point: a *very* basic way of 'passing information from the encoder' is to *average*

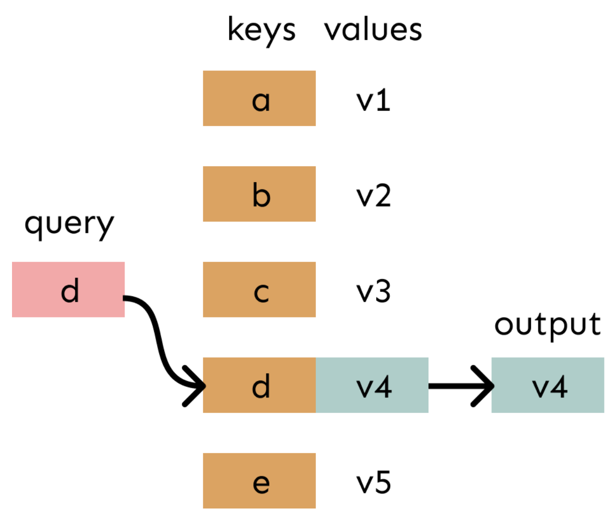
Attention is weighted averaging!

Attention is just a **weighted** average – this is very powerful if the weights are learned!

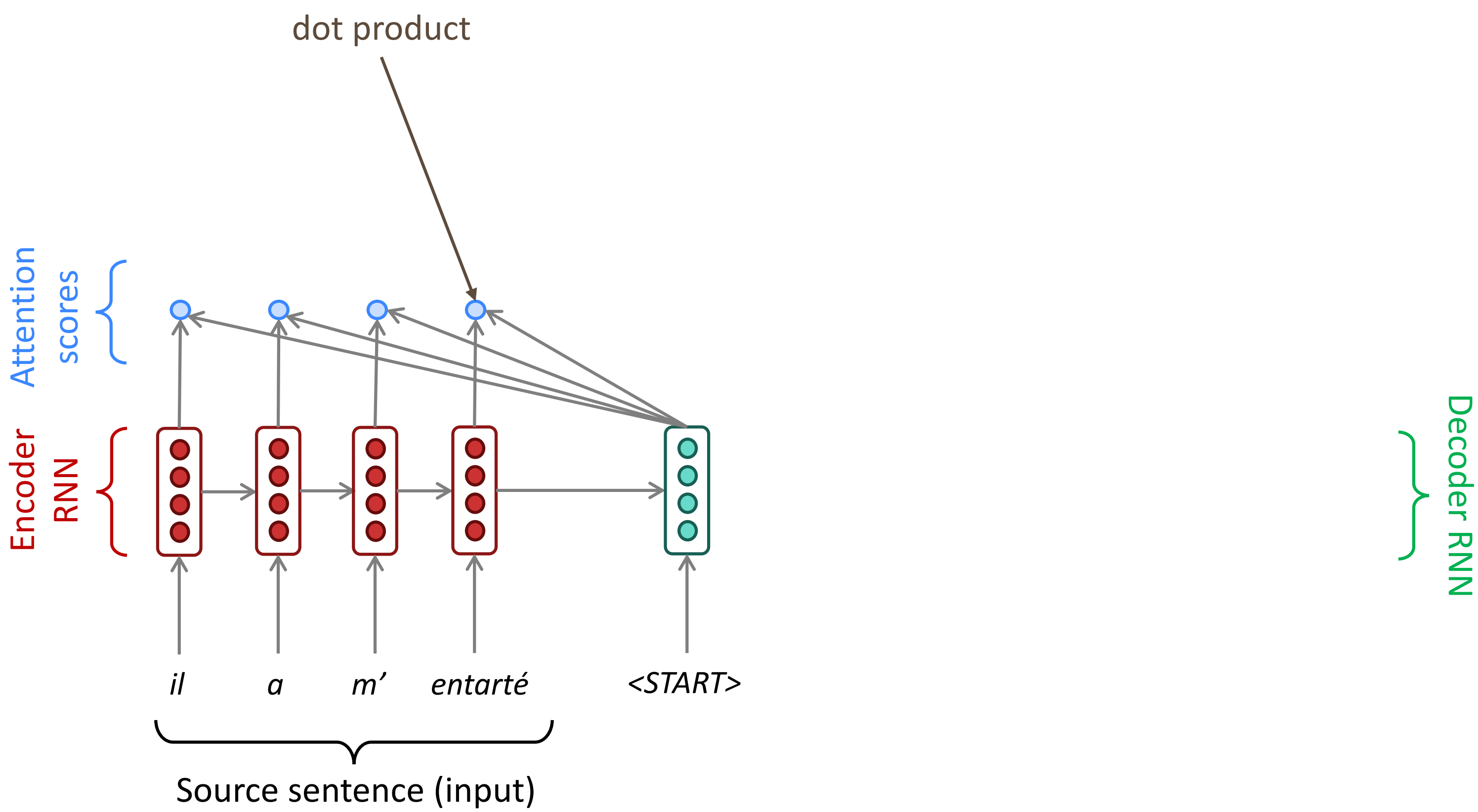
In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



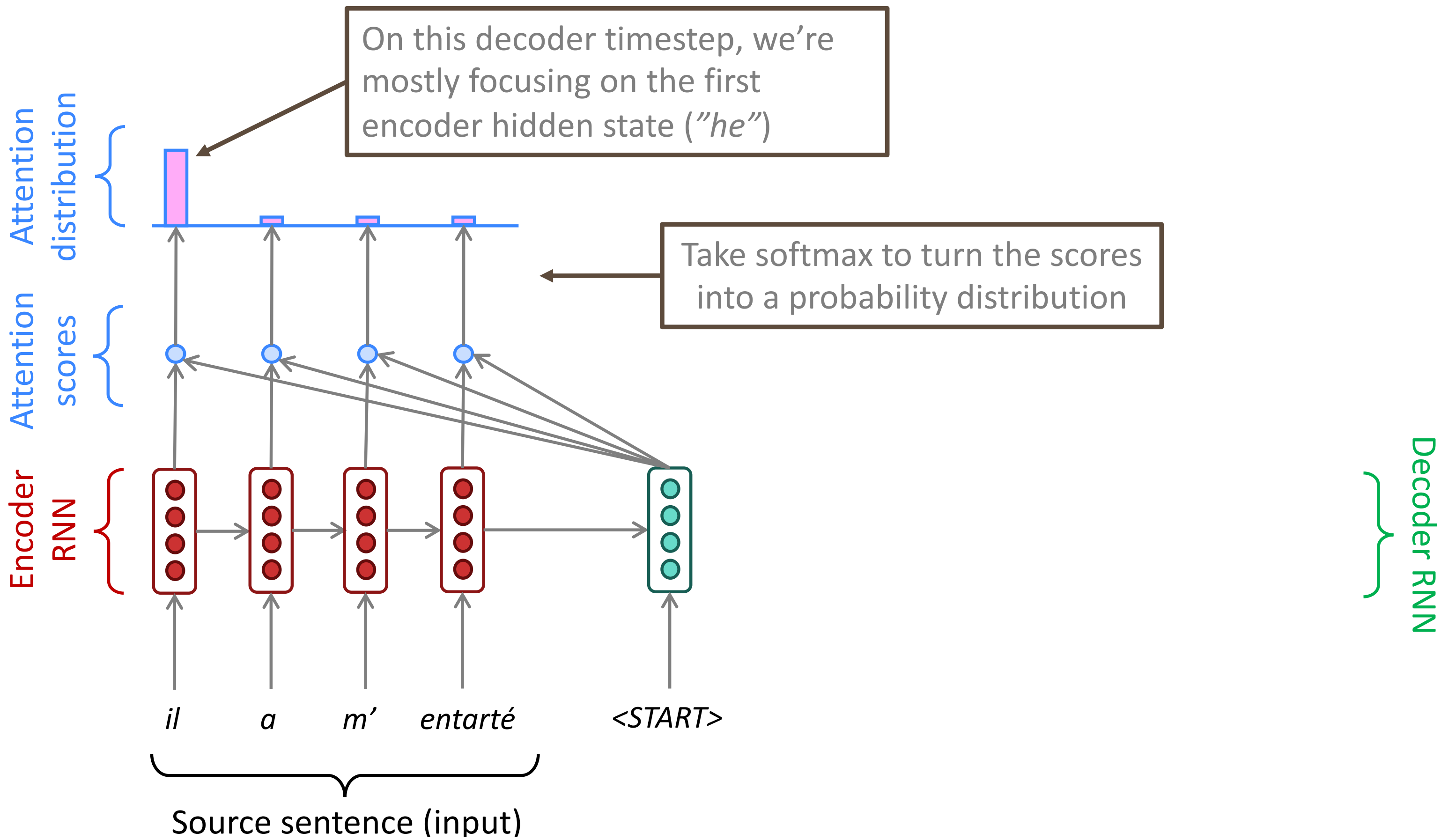
In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



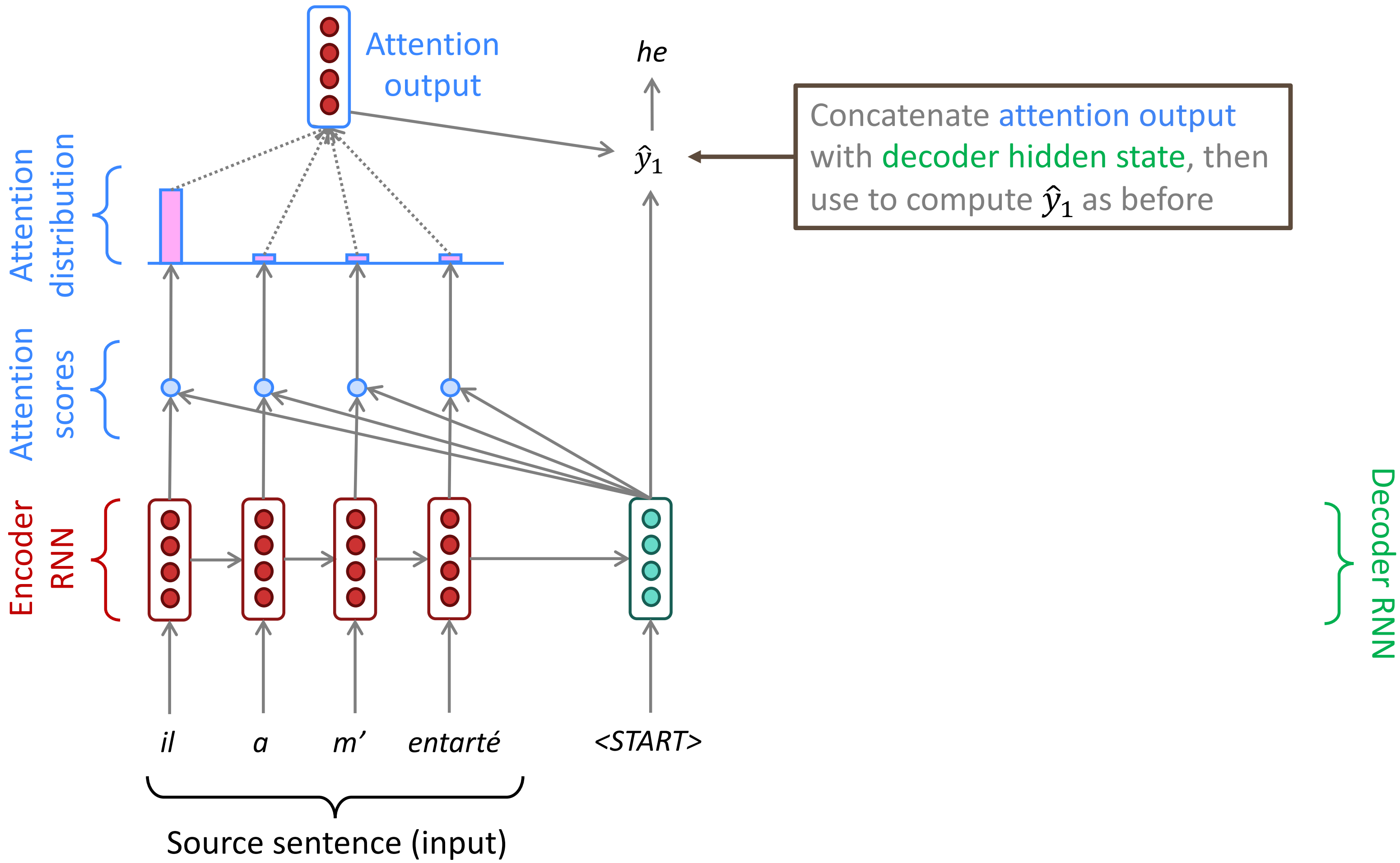
Using dot products



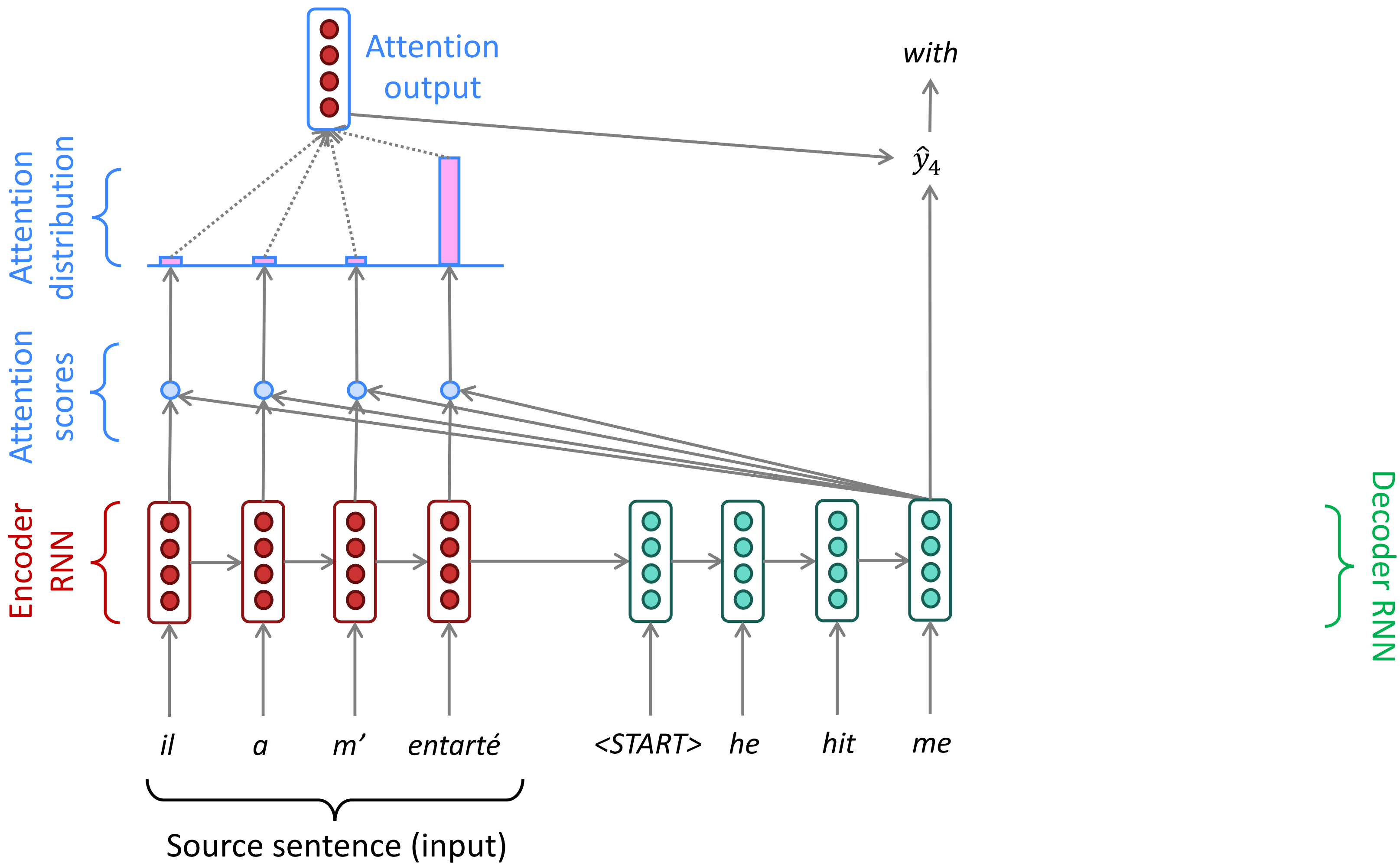
Using softmax for aggregation



Putting it all together



Attention example (continued)



Attention more formally

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \mathrm{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

Attention blueprint

- We have some *values* $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and a *query* $\boldsymbol{s} \in \mathbb{R}^{d_2}$

- Attention always involves:
  1. Computing the *attention scores* $\boldsymbol{e} \in \mathbb{R}^N$
  2. Taking softmax to get *attention distribution* α:

$$\alpha = \mathrm{softmax}(\boldsymbol{e}) \in \mathbb{R}^N$$

  3. Using attention distribution to take weighted sum of values:

$$\boldsymbol{a} = \sum_{i=1}^{N} \alpha_i \boldsymbol{h}_i \in \mathbb{R}^{d_1}$$

  thus obtaining the *attention output* ***a*** (sometimes called the *context vector*)

There are multiple ways to do this

From translation to language generation: Self-attention

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_{\boldsymbol{i}}$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V , each in $\mathbb{R}^{d\times d}$

$\boldsymbol{q}_i = Q\boldsymbol{x}_{\boldsymbol{i}}$ (queries) $\qquad$ $\boldsymbol{k}_i = K\boldsymbol{x}_{\boldsymbol{i}}$ (keys) $\qquad$ $\boldsymbol{v}_i = V\boldsymbol{x}_{\boldsymbol{i}}$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

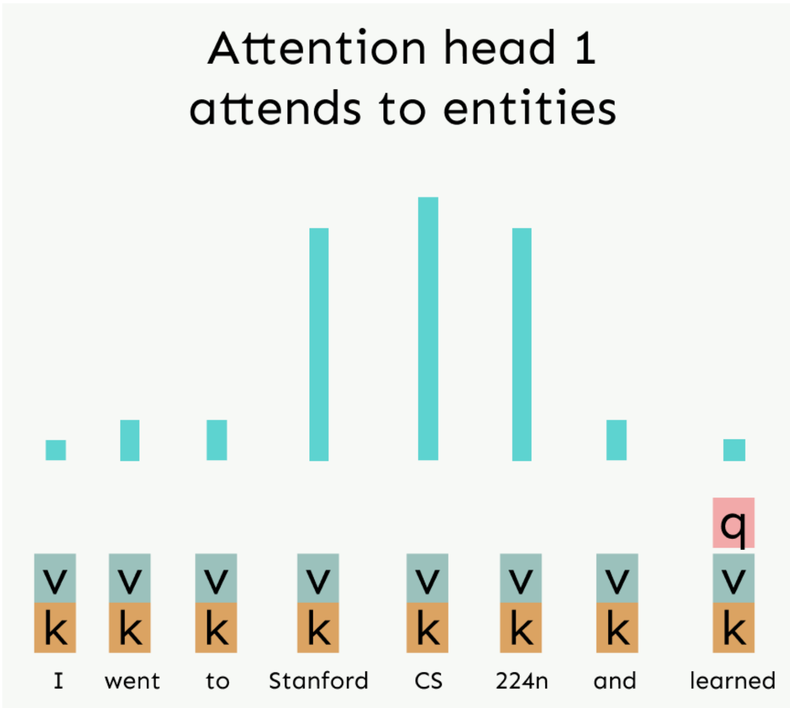
$$\boldsymbol{e}_{ij} = \boldsymbol{q}_{\boldsymbol{i}}^{\top}\boldsymbol{k}_{\boldsymbol{j}} \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'}\exp(\boldsymbol{e}_{ij'})}$$

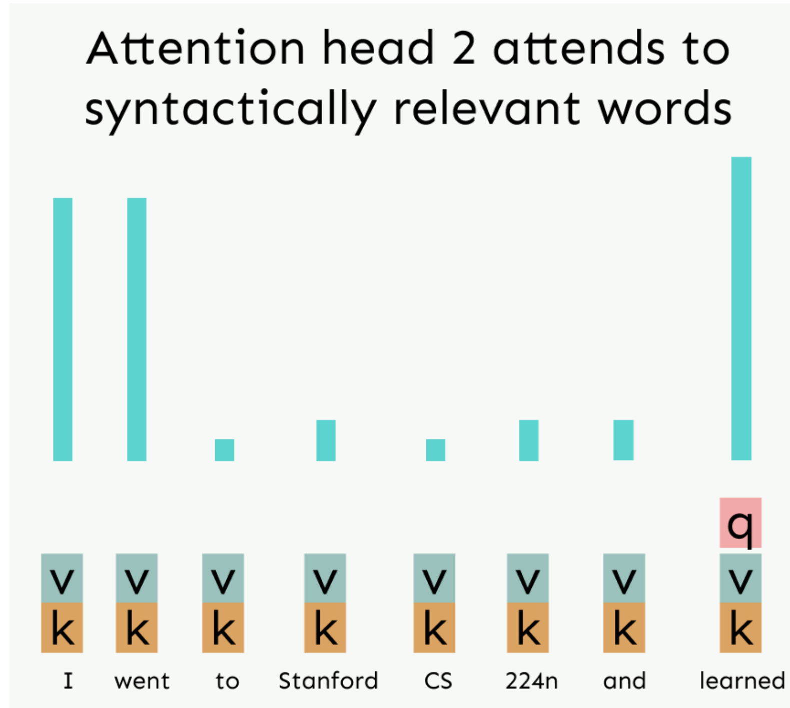
3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_{\boldsymbol{j}} \boldsymbol{\alpha}_{ij}\, \boldsymbol{v}_i$$

$$\boldsymbol{o}_i = \sum_{j} \boldsymbol{\alpha}_{ij}\, \boldsymbol{v}_i$$

Multihead attention

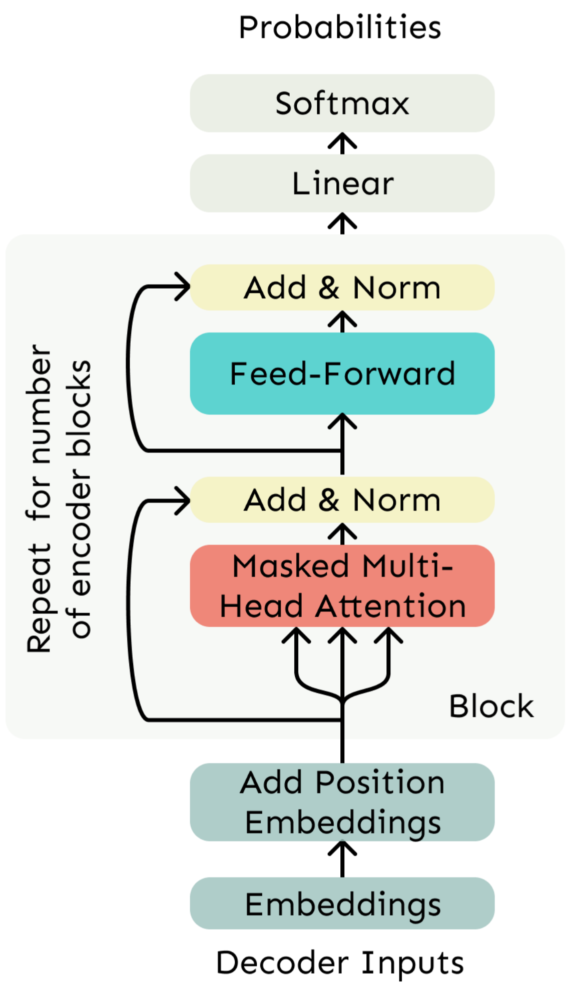




Transformer decoder

The Transformer Decoder

- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.

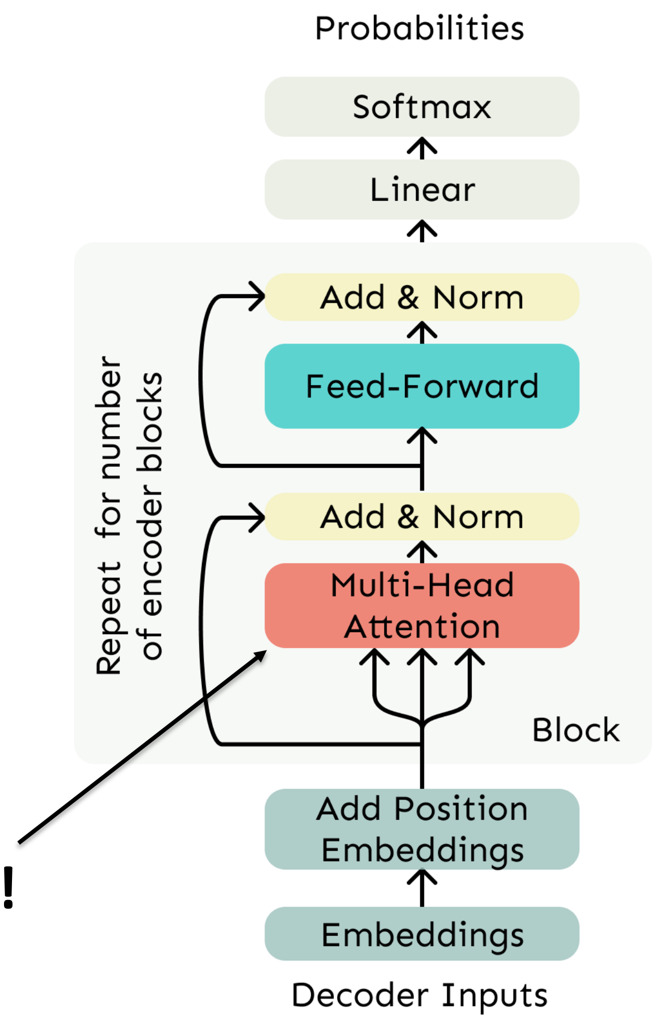


Transformer encoder

The Transformer **Encoder**

- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

No Masking!



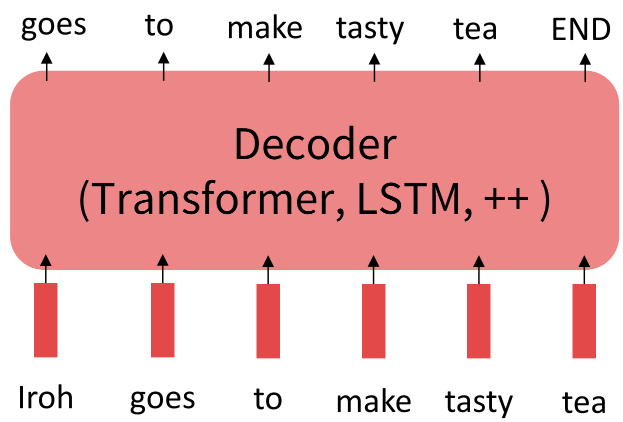
Pretraining

Recall the **language modeling** task:

- Model $p_\theta(w_t|w_{1:t-1})$, the probability distribution over words given their past contexts.
- There's lots of data for this! (In English.)

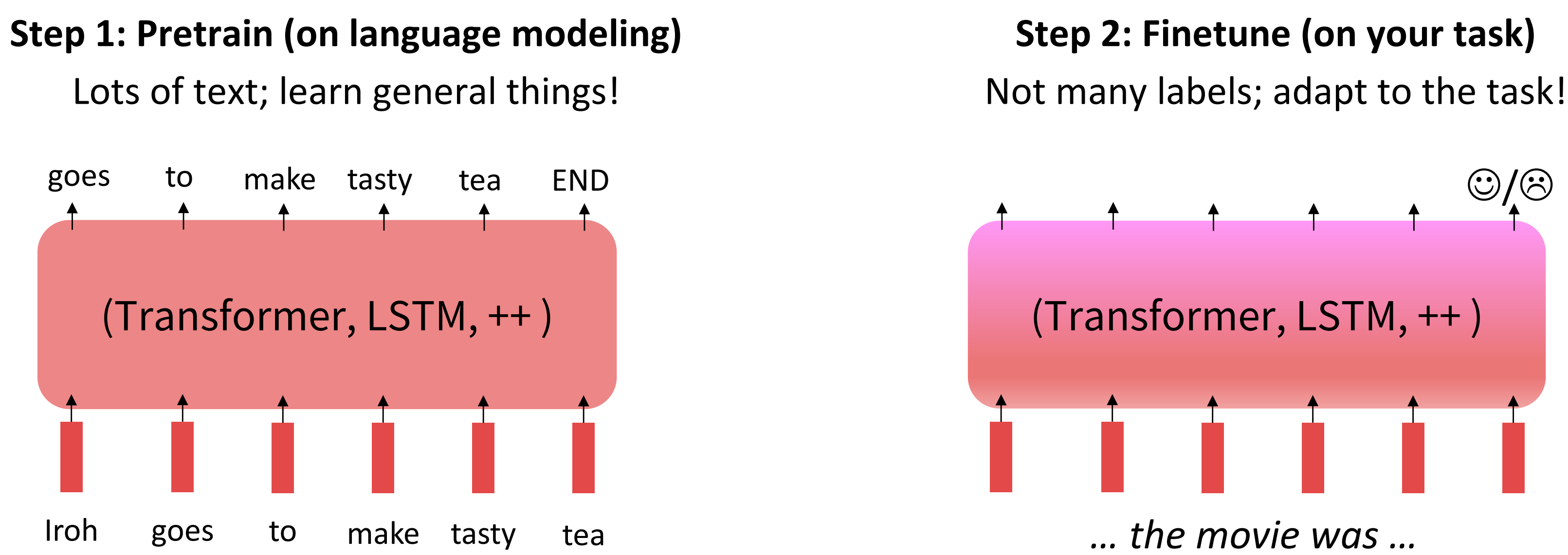
Pretraining through language modeling:

- Train a neural network to perform language modeling on a large amount of text.
- Save the network parameters.

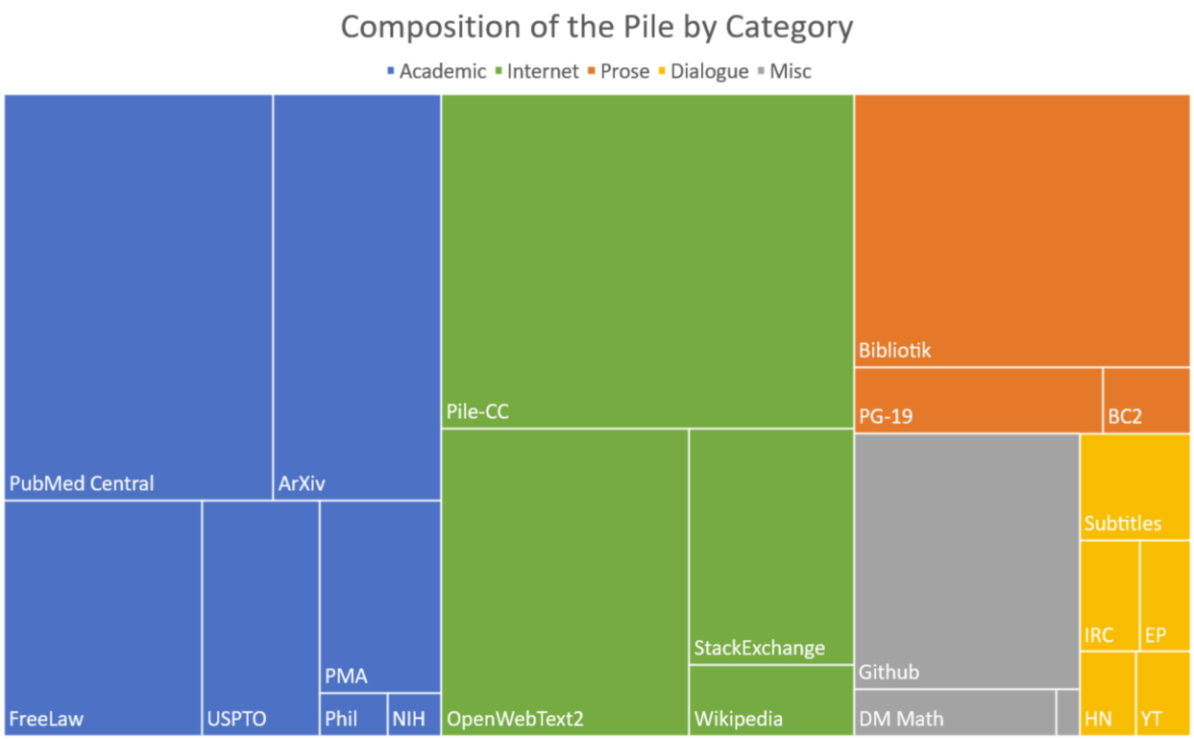


Pretraining / finetuning paradigm

Pretraining can improve NLP applications by serving as parameter initialization.



What data to use?



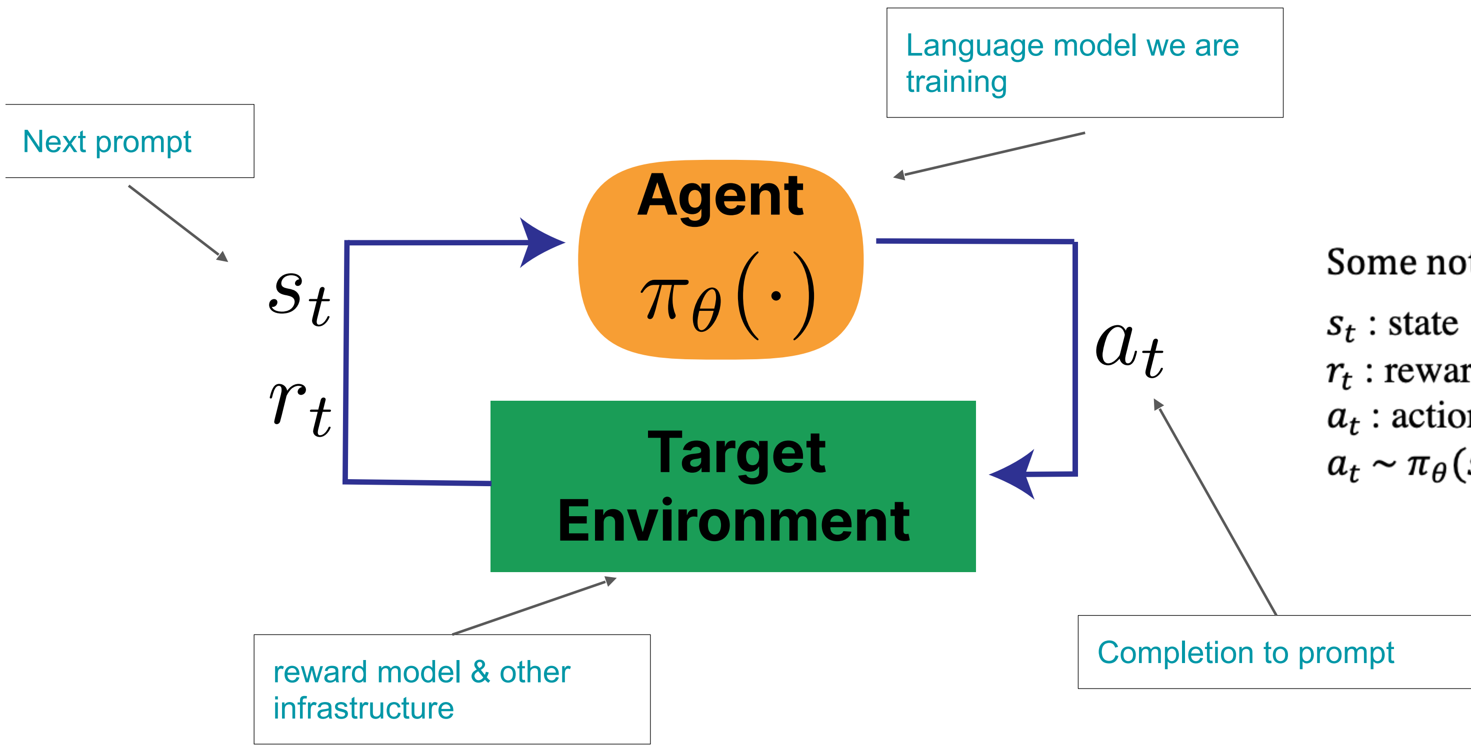
Model	Training Data
BERT	BookCorpus, English Wikipedia
GPT-1	BookCorpus
GPT-3	CommonCrawl, WebText, English Wikipedia, and 2 book databases (“Books 1” and “Books 2”)
GPT-3.5+	Undisclosed

GPT (Devlin et al, 2018)

2018's GPT was a big success in pretraining a decoder!

- Transformer decoder with 12 layers, 117M parameters.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained on BooksCorpus: over 7000 unique books.
    - Contains long spans of contiguous text, for learning long-distance dependencies.
- The acronym "GPT" never showed up in the original paper; it could stand for "Generative PreTraining" or "Generative Pretrained Transformer"

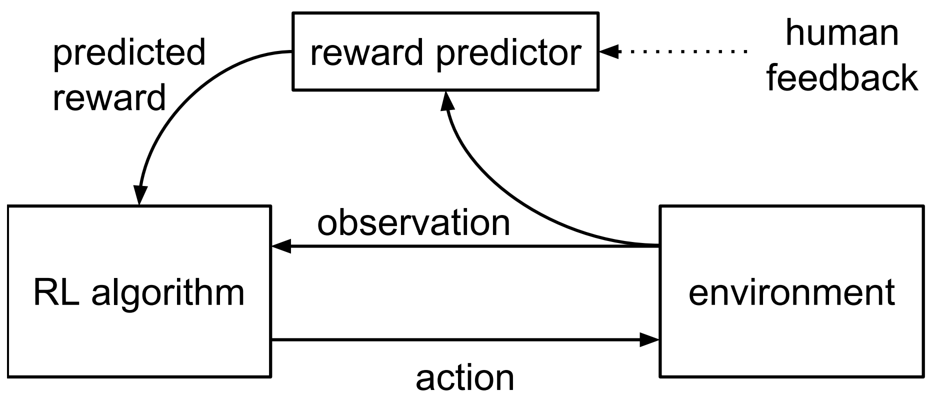
RL comes in the picture!



Some notation:

s_t : state
r_t : reward
a_t : action
$a_t \sim \pi_\theta(s_t)$: policy

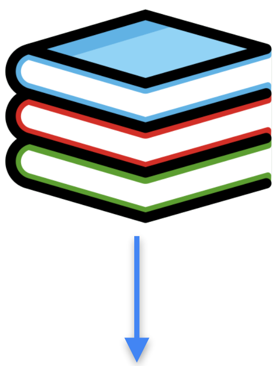
Rcecall: Deep RL from Human Feedback (Christiano et al, 2017)



- People provide a *preference* among two choices
- Assuming there is a latent variable explaining the choice, reward is fit using maximum likelihood (Bradley-Terry model)
- Cf. https://arxiv.org/pdf/1706.03741.pdf

RLHF early attempts

Summarization



“Three pigs defend themselves from a mean wolf”

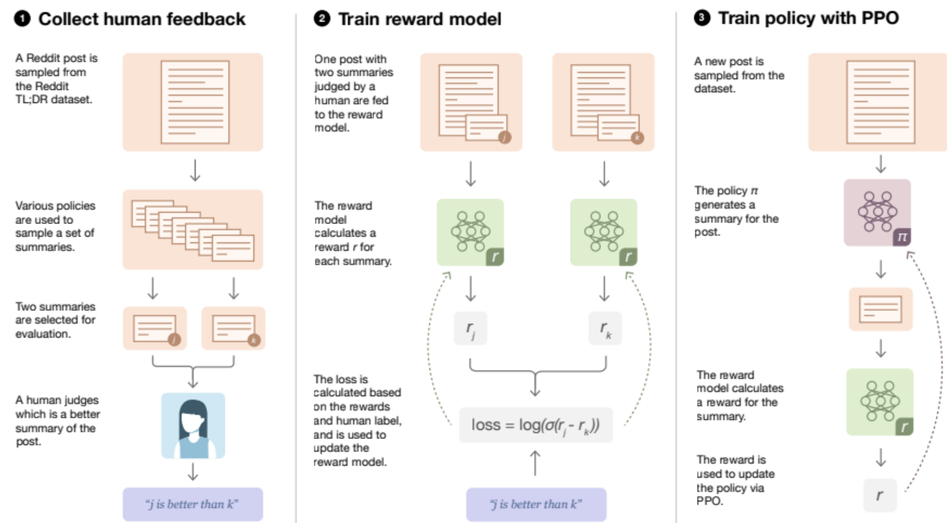
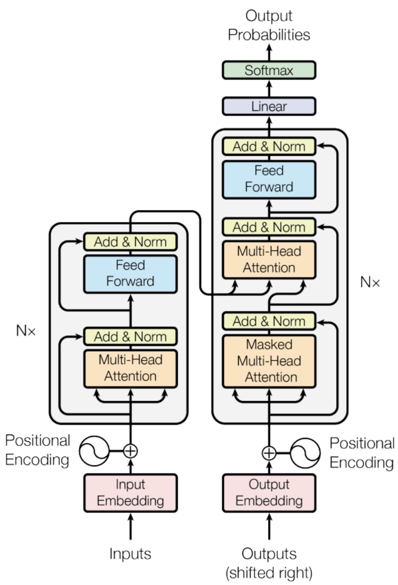


Figure 2: Diagram of our human feedback, reward model training, and policy training procedure.

Stiennon, Nisan, et al. "Learning to summarize with human feedback." *2020.*

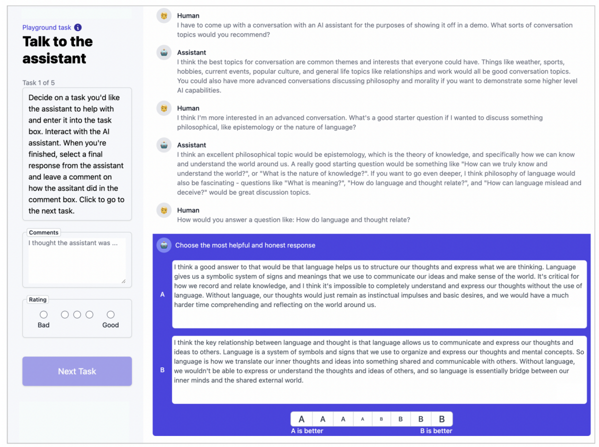
RLHF training phases

base model (instruction, helpful, chatty etc.)

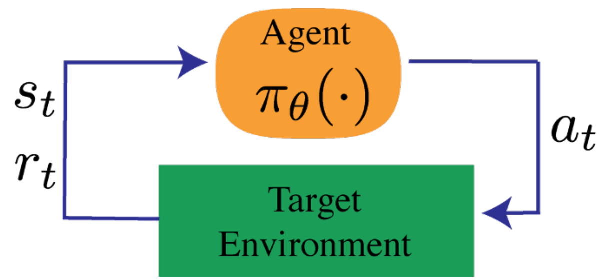


Vaswani et al. 2017

preference collection & training



RL optimization

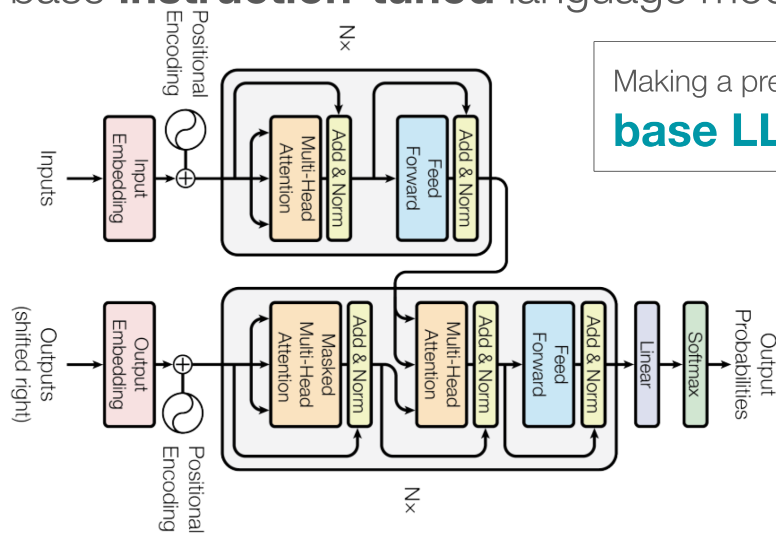


Model structure

starting point: a base **instruction-tuned** language model

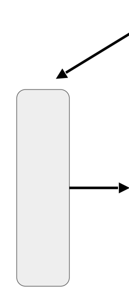
input:

prompt+completion



The Transformer - Vaswani et al. 2017

Making a preference model:
base LLM with new final layer



output:
scalar rewards

Model structure

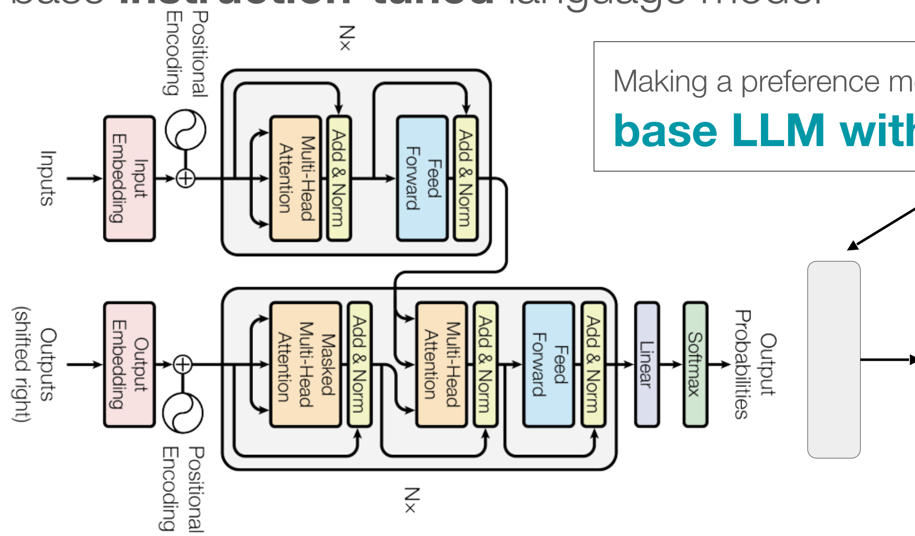
starting point: a base **instruction-tuned** language model

Making a preference model:

base LLM with new final layer

input:

prompt+completion



output:
scalar rewards

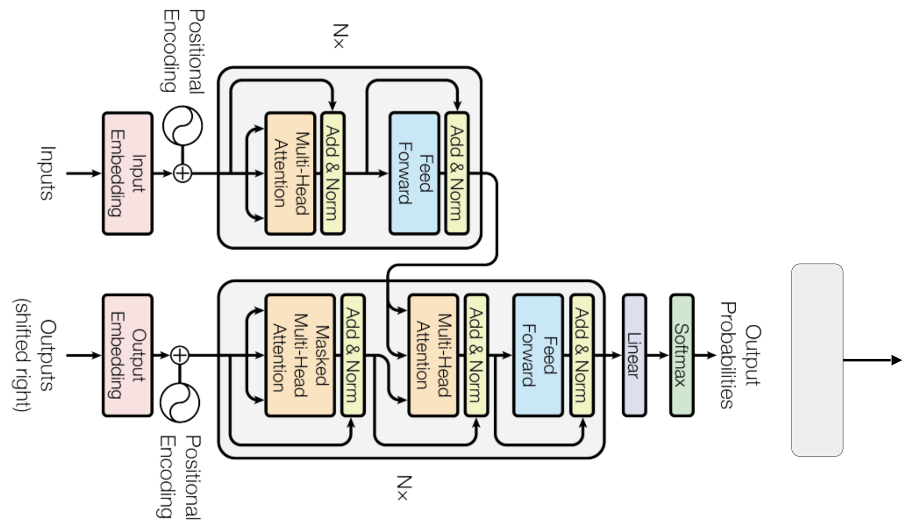
The Transformer - Vaswani et al. 2017

Model training

input pair:

selected prompt +completion

rejected prompt +completion



The Transformer - Vaswani et al. 2017

output:
scalar rewards

loss: increase difference of predicted reward

Recall: Bradely-Terry reward model

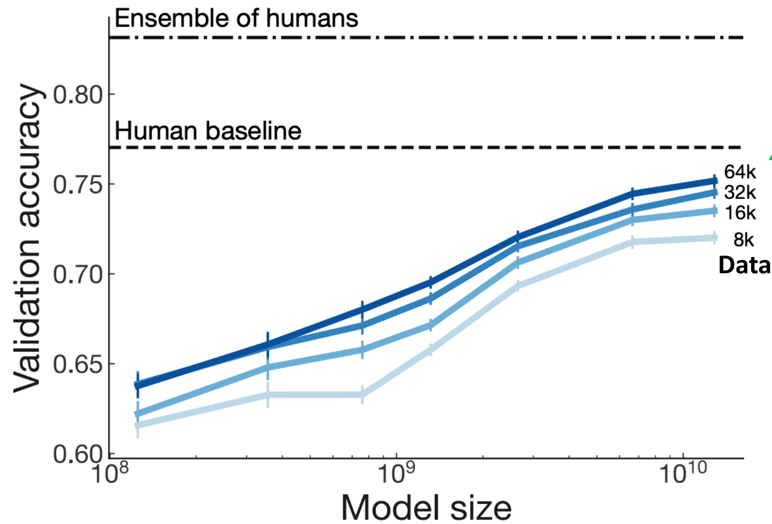
- Collect data from human raters (pairs of y_w, y_l responses to a prompt x)
- Optimize the expected value of:

$$-\log(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l)))$$

  wrt reward parameter vector θ
- Cf. Ouyang et al, InstructGPT
- Corresponds to maximum likelihood fitting of binomial preference function if reward is linear over the variables

Evaluating the reward model

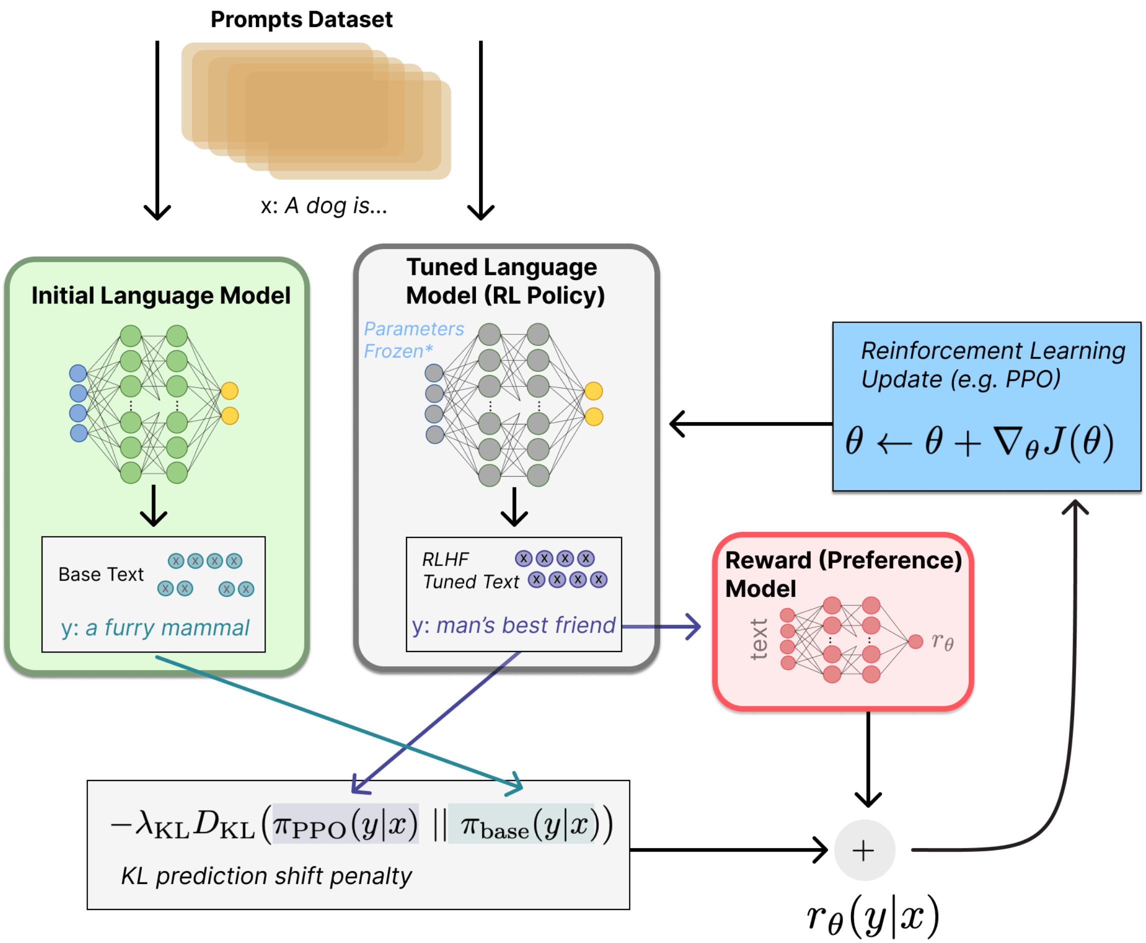
Evaluate RM on predicting outcome of held-out human judgments



Large enough RM trained on enough data approaching single human perf

[Stiennon et al., 2020]

RLHF finetuning



RLHF details

Finally, we have everything we need:

- A pretrained (possibly instruction-finetuned) LM $p^{PT}(s)$
- A reward model $RM_\phi(s)$ that produces scalar rewards for LM outputs, trained on a dataset of human comparisons
- A method for optimizing LM parameters towards an arbitrary reward function.

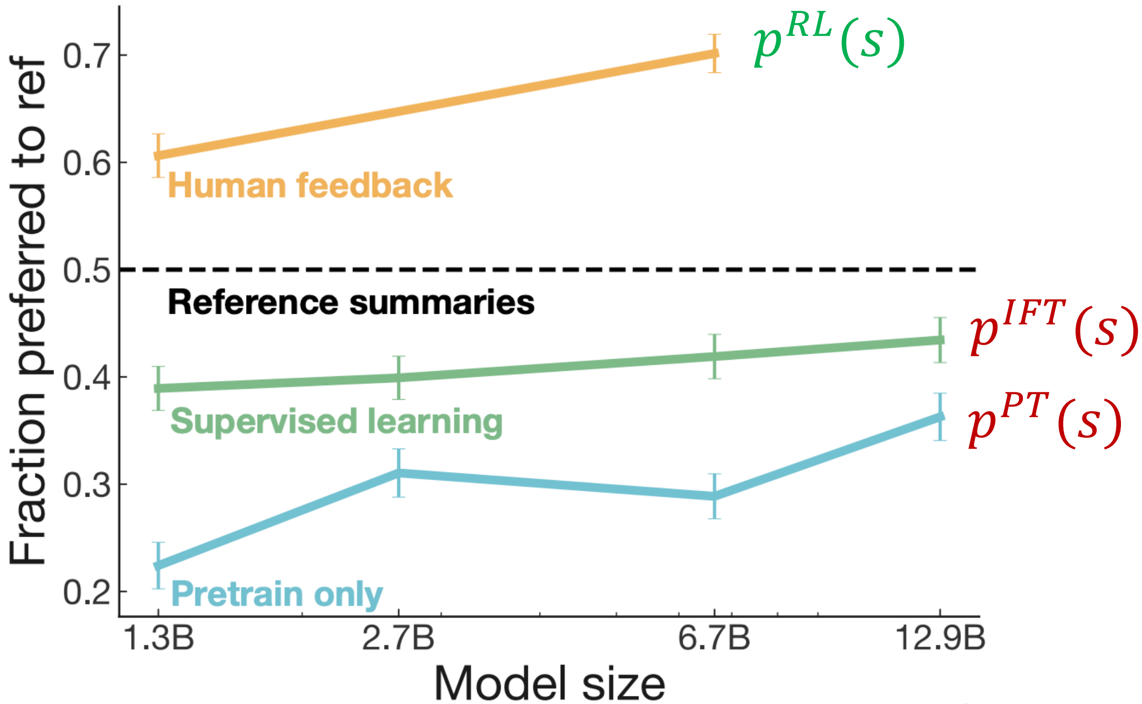
Now to do RLHF:

- Initialize a copy of the model $p_\theta^{RL}(s)$, with parameters θ we would like to optimize
- Optimize the following reward with RL:

$$R(s) = RM_\phi(s) - \underbrace{\beta \log \left(\frac{p_\theta^{RL}(s)}{p^{PT}(s)} \right)}_{\substack{\text{This is a penalty which prevents us from diverging too far from} \\ \text{the pretrained model. In expectation, it is known as the} \\ \textbf{Kullback-Leibler}\text{ (}\textbf{KL}\text{) divergence between } p_\theta^{RL}(s) \text{ and } p^{PT}(s).}}$$

Pay a price when
$p_\theta^{RL}(s) > p^{PT}(s)$

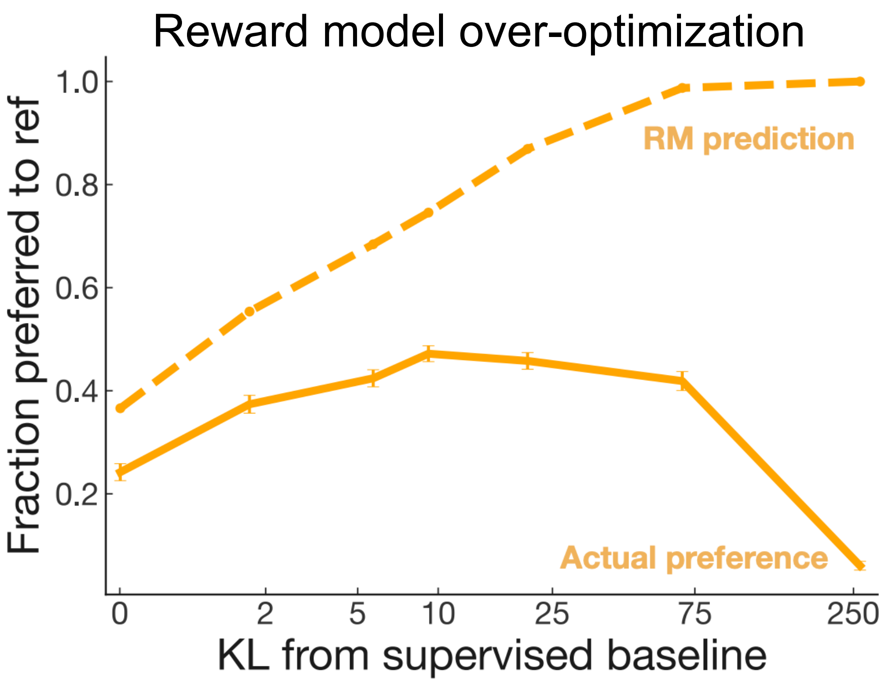
RLHF results



[Stiennon et al., 2020]

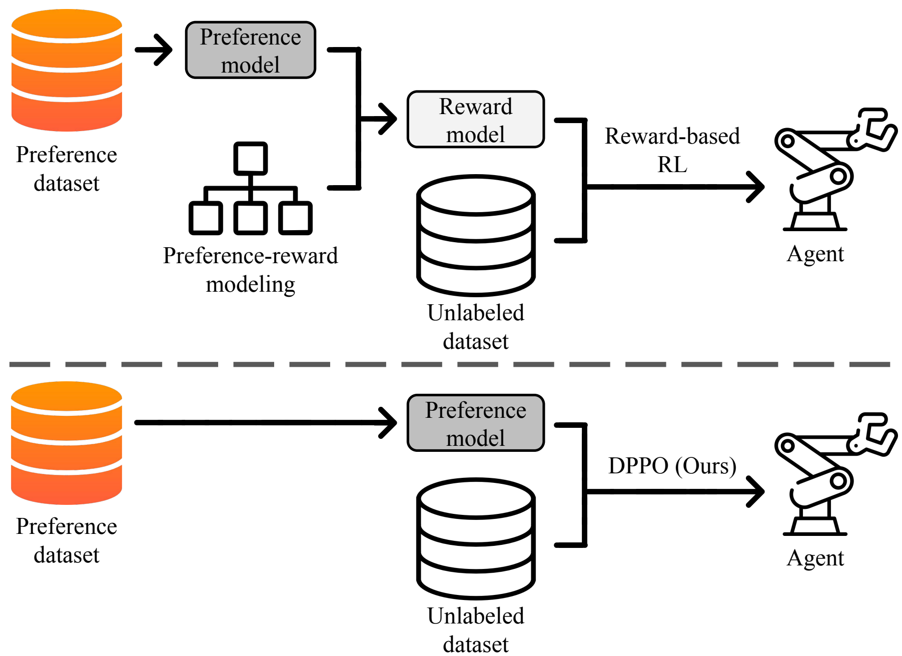
Problem: reward hacking

- Human preferences are unreliable!
  - ”Reward hacking” is a common problem in RL
  - Chatbots are rewarded to produce responses that *seem* authoritative and helpful, *regardless of truth*
  - This can result in making up facts + hallucinations
- **Models** of human preferences are *even more* unreliable!



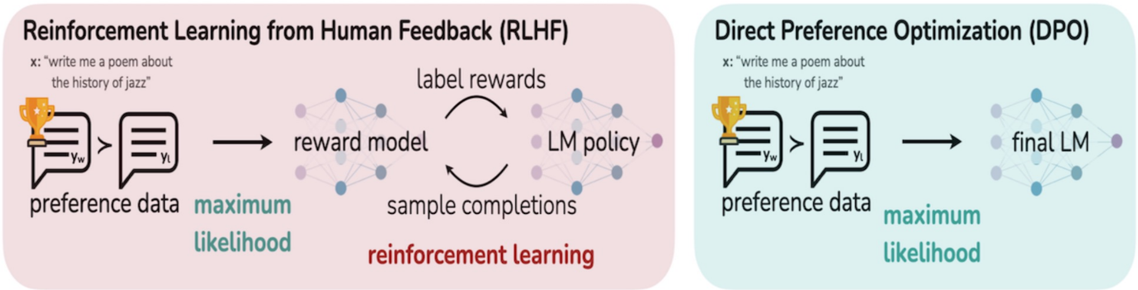
$$R(s) = RM_{\phi}(s) - \beta \log \left(\frac{p_{\theta}^{RL}(s)}{p^{PT}(s)} \right)$$

Direct Preference Optimization



- Cf. An et al, NeurIPS'2023 (https://arxiv.org/pdf/2301.12842.pdf)
- Direct preference optimization (Rafailov et al, NeurIPS'2023, https://arxiv.org/pdf/2305.18290.pdf)
- Several other almost-concurrent papers in this space

Direct Preference Optimization



$$\nabla_\theta \mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) =$$

$$-\beta \mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\Bigg[\underbrace{\sigma(\hat{r}_\theta(x,y_l) - \hat{r}_\theta(x,y_w))}_{\text{higher weight when reward estimate is wrong}}\Bigg[\underbrace{\nabla_\theta \log \pi(y_w \mid x)}_{\text{increase likelihood of } y_w} - \underbrace{\nabla_\theta \log \pi(y_l \mid x)}_{\text{decrease likelihood of } y_l}\Bigg]\Bigg],$$

$$\hat{r}_\theta(x,y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

- You can replace the complex RL part with a very simple weighted MLE objective
- Other variants (KTO, IPO) now emerging too

[Rafailov+ 2023]

Learning with non-transitive preferences: NashLLM

- Objective:find a policy π^* which is preferred over any other policy

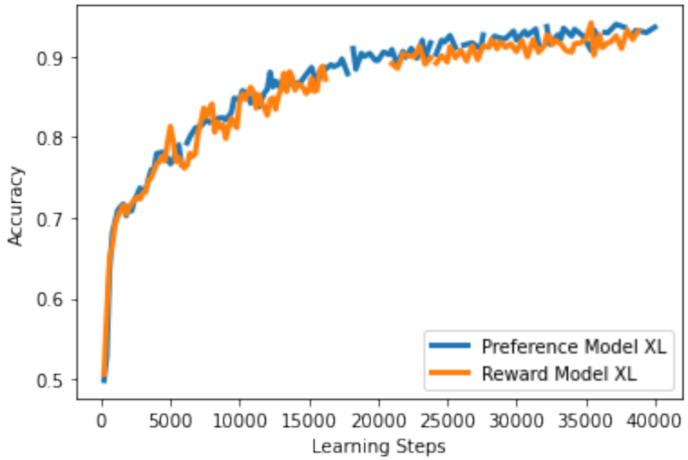
$$\pi^* = \arg\max_{\pi} \min_{\pi'} \mathbb{P}(\pi' \preceq \pi)$$

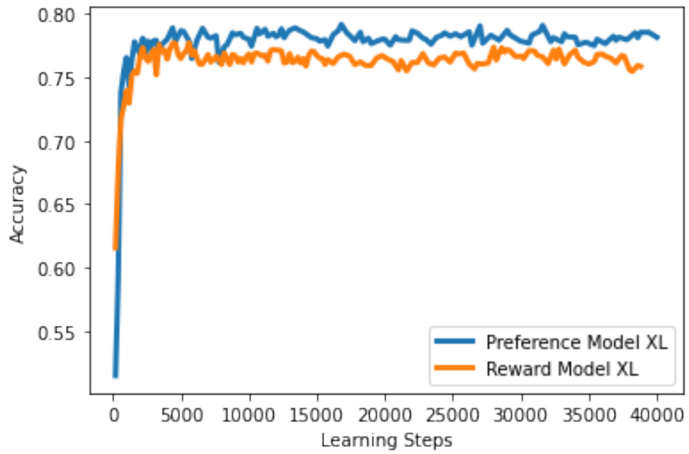
- Think of this as a game: one player picks π the other picks π'
- When both players use π^* this is a *Nash equilibrium* for the game
- For this game an equilibrium exists (even if eg preferences are not transitive)
- Cf. Munos et al, 2024 (https://arxiv.org/pdf/2312.00886.pdf)

NashLLM-style algorithms

- Fit a *two-argument preference function* by supervised learning
- Decide what is the *set of opponent policies*
- Ideally, the max player should play against a mixture of past policies
- *Optimize* using eg online mirror descent, convex-concave optimization...
- A lot of algorithmic variations to explore!

NashLLM results





Using preferences instead of rewards leads to less overfitting

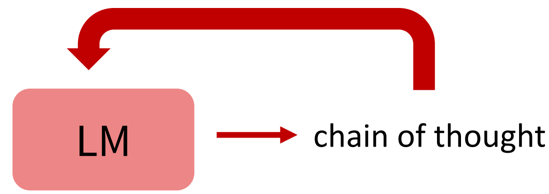
Open directions

- RLHF is still a very underexplored and fast-moving area!
- RLHF gets you further than instruction finetuning, but is (still!) data expensive.
- Recent work aims to alleviate such data requirements:
  - RL from **AI feedback** [Bai et al., 2022]
  - Finetuning LMs on their own outputs [Huang et al., 2022; Zelikman et al., 2022]
- However, there are still many limitations of large LMs (size, hallucination) that may not be solvable with RLHF!

LARGE LANGUAGE MODELS CAN SELF-IMPROVE

Jiaxin Huang[1*] **Shixiang Shane Gu**[2] **Le Hou**[2†] **Yuexin Wu**[2] **Xuezhi Wang**[2]
Hongkun Yu[2] **Jiawei Han**[1]
[1]University of Illinois at Urbana-Champaign [2]Google
[1]{jiaxinh3, hanj}@illinois.edu [2]{shanegu, lehou, crickwu, xuezhiw, hongkuny}@google.com

[Huang et al., 2022]



Self-Taught Reasoner (STaR)
[Zelikman et al., 2022]

More open directions

- Multi-turm
- Exploration
- .....