This note is concerned with getting a language model to do what you want it to do. After the pretraining process, you have a model p_θ parameterized by learned parameters θ which computes the likelihood of all tokens in $w \in \mathcal{V}$ as continuations of a prefix $w_{<t}$, that is, $p_\theta(\cdot \mid w_{<t}) \in \mathbb{R}^{|\mathcal{V}|}$. Now, you have the ambitious goal of constructing a system based on this model that could attempt a broad range of tasks. We'll attempt to cover both the fundamental problems to be solved and core methods.

The language model doesn't know what you want

In this course so far, we've discussed how designing expressive neural networks and training them on large amounts of data can lead to strong **representations**, and a considerable amount of knowledge learned from text. But our interface for interacting with a pretrained language model is, well, a probability distribution. It doesn't know what we want, it just tells us what's likely to come next.

Consider tokenizing the sequence

> What year was Columbia University founded?

with the goal of feeding the tokens to your language model and receiving an answer. Let this be $w_{1:t}$, and consider the distribution $p_\theta(\cdot \mid w_{1:t})$. What are the elements of $\mathcal{V}$ that are likely continuations? Is it 1754, the founding year of Columbia? (Or maybe the tokens 1 or 17 or 175, depending on how the tokenizer tends to tokenize years?) To reason about this, we think, what kind of text was the model trained on? It was trained on a large amount of naturally occuring web text. Surely, the model saw *Columbia University was founded in 1754* at some point, but in this question form it might be more likely that something like the following happens:

> *What year was Columbia University founded?* What year was Princeton University founded? What...

so a likely first new token is *What*. Or it might be likely to generate

> *What year was Columbia University founded?* I've wondered about this for a long time...
> *What year was Columbia University founded?* Great question! In this note, we'll...

The most likely continuation of your sequence isn't necessarily the answer, even if the model **stores knowledge about the answer** in its weights and representations. After all, the model doesn't know what you want, and further questions or follow-up ideas may just be more probable under its estimate of the internet text probability distribution.

Let's look at some methods for making it clearer what we're looking for — leveraging the distribution (and/or parameters) of our pretrained model to make it more useful for our goals.

In-context learning, or, pattern repetition

Consider my question about Columbia University's founding. I can better specify my goals by showing, through the patterns in the input text, the kind of string I want the model to produce. Consider the following updated input:

> *What year was Harvard founded? 1640.*
> *What year was Columbia University founded?*

Before my question about Columbia, I've added what is known as an *in-context example*. It's in-context because this whole prefix we're passing to the model is known as the context, and it's an example of an input (the question) and output (the year of founding) pair that we're looking for. The phrase *in-context learning* refers to the model picking up on the pattern that **likely or probable** continuations to the sequence are years.

Intuitively, by providing in-context examples, we are reducing the *entropy* of the space of outputs, making otherwise-plausible continuations (like following with another related questions) less likely because they don't fit the pattern. This is also often used in, e.g., multiple-choice question answering, to indicate to the model that it should be producing the letter:

Let's set this out in a bit of notation. Consider a dataset of inputs and outputs, $\{(x^{(1)}, y^{(i)})\}_{i=1}^{k}$, and a new input x for which we want to query an answer from our language model. Each input x_i and output y_i is a string over our vocabulary $\mathcal{V}$. If we sample from the language model conditioning on x without using our dataset of examples:

$$\hat{y} \sim p_\theta(y \mid x), \tag{1}$$

we've stated that the problem likely arises that $\hat{y}$ isn't really a good answer to x, but instead just a continuation of x. Recall that this might be done by formatting as:

```
[START] x1 x2 ... xn [SEP]
```

where `[START], [SEP], [END]` are all special tokens in $\mathcal{V}$, and we condition on the sequence up through `[SEP]` and then sample from the model to generate until `[END]`.

Conditioning on some examples from our dataset in-context, also called **few-shot prompting**, is:

$$\hat{y} \sim p_\theta \left(y \mid x, (x^{(1)}, y^{(1)}), \ldots, (x^{(k)}, y^{(k)}) \right), \tag{2}$$

which we might format as:

```
[START] <inp1> x^1_1,... </inp1>
    <out1> y^1_1, ... </out1>
    ...
    <inpk> x^k_1,... </inp1>
    <outk> y^k_1, ... </out1>

    <inpk+1> x1 x2 ... xn </inpk+1>
    <outk+1>
```

So, we've got some extra formatting tags to delimit the start and end of each of the input and the output for the few-shot examples, and we then put the last input—that we're asking our model to answer—and sample a response.

However, one downside of few-shot prompting is that it is computationally expensive at inference time, since the model needs to process long prompts for every input. To address this issue, we can bake in the desired behavior into the model's parameters by adapting them. We call this process **finetuning.**

Supervised finetuning

Finetuning a pretrained model means taking the parameters that were the result of pretraining, θ_p, and performing additional gradient steps on a new loss. For exampe,

from our dataset D of inputs and outputs (perhaps the same one we had for few-shot prompting, but likely much larger), we might perform:

$$\theta_0 = \theta_p \tag{3}$$

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} \left(-\log p_{\theta_{t-1}}(y^{(i)} \mid x^{(i)}) \right), \tag{4}$$

or an Adam update, etc.—the point here is that we start at θ_p, and then take some more gradient steps on our supervised finetuning data.

We call this supervised finetuning because we construct direct examples of what we want the model to do ($y^{(i)}$) for a specific context ($x^{(i)}$). This is *supervision*, like when we made a machine translation system earlier in the class.

Learning from preferences

In our evaluations lecture, we discussed how it's difficult for ahuman to judge the absolute quality of a model output, but easier to choose the better of two outputs. Likewise, when constructing expected outputs for a given input $x^{(i)}$, it is sometimes hard to come up with exactly the right behavior $y^{(i)}$ for that context. It may be easier to have an example of a preferred behavior $y_c^{(i)}$ (called y chosen) and a dispreferred behavior $y_r^{(i)}$ (called y rejected.)

To train our model to be more aligned with our preferences, we might consider maximizing the likelihood of the chosen response, and minimizing the likelihood of the rejected response:

$$\mathcal{L} = - \mathop{\mathbb{E}}_{x, y_w, y_l \sim D} \left[\log p_\theta(y_w \mid x) - \log p_\theta(y_l \mid x) \right]. \tag{5}$$

maximize prob of winning response

minimize prob of losing response

Unfortunately, this won't work, but it does sort of capture the intuition of what we'll end up doing. It won't work in part because this loss isn't bounded below—you can keep reducing the loss by putting less and less probability on the losing response. For example, imagine the winning and losing responses start with 0.0001 probability. Ok, so that's a log-loss of 4 for the winning response and -4 for the losing response, equaling 0. If we put probability close to 1 on the winning response that changes our loss to 0 for that term. But for the losing respone, if I reduce the probability from 0.0001 to 0.00001, that's now -5, and likewise from 10^{-5} to 10^{-6} etc. We keep getting linear improvements in loss for each log-reduction in probability to the losing response, which will eventually just break our model.

One way to fix this (that people don't use, but bear with me) is to just use a logistic function to implement diminishing returns for changing the probabilities:

$$\mathcal{L} = - \mathop{\mathbb{E}}_{x, y_w, y_l \sim D} \left[\log \sigma \left(\log p_\theta(y_w \mid x) - \log p_\theta(y_l \mid x) \right) \right]. \tag{6}$$

maximize prob of winning response

minimize prob of losing response

Note how the logistic function here means eventually as the difference in log-probabilities between these two becomes large, there are diminishing returns to changing them further (the logistic function saturates.)

This brings us to Direct Preference Optimization [Rafailov et al., 2023], a popular algorithm for learning from preferences datasets, and doesn't suffer from this problem. To explain it, we'll start with a few ideas, starting with the idea of modeling the latent preferences behind our preference dataset. The Bradley-Terry model [Bradley and Terry, 1952] is a simple model of preferences in which the probability one thing is preferred over another is the difference in their (potentially latent) intrinsic qualities, squashed to a probability by the logistic function:

$$P_{\text{bt}}(y_w \succ y_l \mid x) = \sigma(r(y_w, x) - r(y_l, x)), \tag{7}$$

where $r(y, x)$ is the quality of a response, and $y_w \succ y_l$ is read out as y_w *is preferred over* y_l *for* x. One aspect to keep in mind is that as the quality of one response grows compared to the other, the probability of preference saturates at 1, so if one response is quality 5 and another is quality 100 vs 300, the winning probabilities aren't that different.

The goal of direct preference optimization is to make a model's probability better match the preferences implicitly behind the dataset that we're optimizing on. We start by positing the quality of each response as the log-ratio of the probability assigned to that response by the preference-trained model divided by the probability assigned before we start preference-training. That is, we take our initial model and call it p_{ref}, and we call the model as we're training it p_θ as usual. The score of some response y for question x is modeled as:

$$r(y, x) = \log \frac{p_\theta(y \mid x)}{p_{\text{ref}}(y \mid x)} \tag{8}$$

These values start at $\log \frac{c}{c} = 0$, since the initial model is the same as the model we're training (before we take any gradient steps.) So, a model implements the notion of considering y_w high score if it assigns higher probability to that response for x than the initial model did.

Now we consider our language model as *implementing* Bradley-Terry. We take Equation 7 and plug in our definition of the score $r(y, x)$ (Equation 8.) We get:

$$P_{\text{bt},\theta}(y_w \succ y_l \mid x) = \sigma\left(\log \frac{p_\theta(y_w \mid x)}{p_{\text{ref}}(y_w \mid x)} - \log \frac{p_\theta(y_l \mid x)}{p_{\text{ref}}(y_l \mid x)}\right). \tag{9}$$

So, our language model is implicitly a Bradley-Terry preference model that starts out (before we do our DPO training) not having any preference for any of the y_w in our dataset over the y_l. We then train the model on the DPO loss, which is just to maximize the log-probability of, in expectation over our dataset, preferring each y_w over each y_l:

$$\mathcal{L}_{\text{DPO}} = - \mathop{\mathbb{E}}_{x, y_w, y_l \sim D} \log \left[\sigma\left(\log \frac{p_\theta(y_w \mid x)}{p_{\text{ref}}(y_w \mid x)} - \log \frac{p_\theta(y_l \mid x)}{p_{\text{ref}}(y_l \mid x)}\right)\right]. \tag{10}$$

Reinforcement Learning from Human Feedback

The main limitation of DPO, SFT, and similar learning strategies that use a fixed dataset of behaviors D is that we're not looking at, and guiding/changing/rewarding/discouraging, what the model would actually do if we sampled from it. Relative to our existing model, the dataset $D = \{x^{(i)}, y_w^{(i)}, y_l^{(i)}\}_{i=1}^M$ may consist of chosen responses that are already worse than what the model is doing. On the other hand, the whole dataset may be too hard (at first)—the model we're training may be *unable* to reliably produce strings like y_w. These are intuitive descriptions of the problems with *off-policy* training.

To use a personal analogy, I can look at the chess moves of grandmaster chess players all day long if I want to—I gain limited insight for my own games, because I'm just not very good at chess. I learn more when I actually *play* games (when I generate moves myself) and get feedback on the quality of those moves. The fact that I'm generating the moves myself means I already have some reasonable probability of ending up in this situation. This is *on-policy* training.

Cool, so that leads us to reinforcement learning from human feedback. The classic citation here for language models is [Ouyang et al., 2022]. It actually came before DPO ([Rafailov et al., 2023]), but I present it afterwards because DPO is in practice just a lot like SFT, and on-policy training is more complex.

Overview. At a high level, our goal is to learn a distribution p_θ that maximizes the quality of responses conditioned on questions:

$$\max_\theta \mathop{\mathbb{E}}_{x\sim D, \hat{y}\sim p_\theta(y|x)} [r(\hat{y}, x)] \tag{11}$$

We're going to do this by (1) sampling from the model, (2) scoring responses with an *estimated* quality $\hat{r}(x, y)$, and then (3) training the model to assign higher likelihood to the higher-quality responses.

Reward Modeling. So, let's start with our estimate of the response quality. It's too slow to have humans judge each pair generated by our model, so we're going to train a model to do it. In particular, we'll specify a model like:

$$h = \text{Transformer}(x, y) \tag{12}$$

$$\hat{r}(y, x) = v^\top h, \tag{13}$$

so, we run a Transformer on the input request and output text, and then take the representation (perhaps from the last token) from that, h. We learn a final parameter vector v, and have the predicted score be $v^\top h$.

We can train this **reward model** on our preference dataset by using it as part of a Bradley-Terry model:

$$\theta_r = \arg\min_\theta \mathbb{E}_{x,y_w,y_l\sim D} \left[-\log \sigma(\hat{r}(y_w, x) - \hat{r}(y_l, x))\right] \tag{14}$$

From now on we'll call this $\hat{r}$ our reward model.

Reinforcement Learning with REINFORCE. REINFORCE [Williams, 1992] is a classic algorithm for learning to maximize reward. Intuitively, we take the maximization problem above (Eqn 11) and note that we can optimize it via sampling and taking gradients (policy gradient). First we write out the expectation as a sum weighted by the probabilities of the model:

$$\mathbb{E}\left[\hat{r}(\hat{y}, x)\right] = \mathbb{E}_{x\sim D}\left[\sum_{\hat{y}\in\mathcal{V}^*} p_\theta(\hat{y} \mid x)\hat{r}(\hat{y}, x)\right], \tag{15}$$

Now we'll take gradients and use what's called the log-derivative trick:

$$\nabla_\theta \mathbb{E}_{x\sim D}\left[\sum_{\hat{y}\in\mathcal{V}^*} p_\theta(\hat{y}\mid x)\hat{r}(\hat{y},x)\right] = \mathbb{E}_{x\sim D}\left[\sum_{\hat{y}\in\mathcal{V}^*} \hat{r}(\hat{y},x)\nabla_\theta p_\theta(\hat{y}\mid x)\right] \tag{16}$$

$$= \mathbb{E}_{x\sim D}\left[\sum_{\hat{y}\in\mathcal{V}^*} \hat{r}(\hat{y},x)p_\theta(\hat{y}\mid x)\nabla_\theta \log p_\theta(\hat{y}\mid x)\right] \tag{17}$$

$$= \mathbb{E}_{x\sim D,\hat{y}\sim p_\theta(\hat{y}|x)}\left[\hat{r}(\hat{y},x)\nabla_\theta \log p_\theta(\hat{y}\mid x)\right], \tag{18}$$

which means all we need to do is, as we said, (1) sample responses from our model, (2) score then with our reward model, and then (3) take the log-likelihood of the sequence under our model and weight it by the reward, backpropagating into the network to train much like in SFT.

A common thing to do is to replace the reward we're trying to maximize with a baselined *advantage*—the improvement in reward over (in this case) what we might expect the reward should have been for x:

$$\hat{A}(\hat{y},x) = r(\hat{y},x) - \hat{r}(x), \tag{19}$$

where $\hat{r}(x)$ is an estimate (made in a similar way to our reward model) of what reward should be achieved for x without knowing the specific response.

Regularizing KL-divergence, and Proximal Policy Optimization. In practice, REINFORCE isn't used to train language models in RLHF. There are two additional pieces of complexity. The first is that we want to penalize how far the reinforcement learning process moves from, say, a good starting point defined by SFT. This is because our reward function necessarily doesn't capture everything we'd like it to—if we blindly maximize reward, the model can **reward hack**—generating outputs that achieve high reward due to flaws in the reward function, not quality in the response. For example, reward models are known to have biases towards longer responses, independent of quality.

To penalize the learned model from deviating too much from a good SFT baseline, we penalize the KL-divergence between the two. We replace our maximization problem in Eqn 11 with

$$\max_\theta \mathop{\mathbb{E}}_{x\sim D,\hat{y}\sim p_\theta(y|x)}[r(\hat{y},x) - \beta \log \frac{p_\theta(\hat{y}\mid x)}{p_{\text{SFT}}(\hat{y}\mid x)}], \tag{20}$$

where p_{SFT} is the SFT-trained distribution, and β is a hyperparameter that controls the strength of the penalty.

One final piece of complexity brings us to *proximal policy optimization* (PPO), the method that kicked off the popularity of RLHF. Intuitively, sampling a lot of $\hat{y} \sim p_\theta(y \mid x)$ is expensive; we'd like to be able to "reuse" each one across multiple gradient steps. But once we've taken a gradient step, the old $\hat{y}$ are no longer drawn from the right (new) distribution; they're *stale*.

So, we now write out our maximization but sample from the stale model. We correct for the difference in probabilities with our current model by defining a ratio $q_\theta(\hat{Y} \mid x) = \frac{p_\theta(\hat{y}|x)}{p_{\theta_{\text{old}}}(\hat{y}|x)}$ (correcting for the probability difference.) We then also make conservative

changes to our network, clipping each ratio

$$q_\theta(\hat{y} \mid x) = \frac{p_\theta(\hat{y} \mid x)}{p_{\theta_{\text{old}}}(\hat{y} \mid x)} \tag{21}$$

$$\max_\theta \mathop{\mathbb{E}}_{x \sim D, \hat{y} \sim p_{\theta_{\text{old}}}(y|x)} \left[\min \left(q_\theta(\hat{y} \mid x) \hat{A}(\hat{y}, x), \text{clip}\left(q_\theta(\hat{y} \mid x), 1-\epsilon, 1+\epsilon\right) \hat{A}(\hat{y}, x) \right) \right]. \tag{22}$$

If the likelihood ratio goes below $1 - \epsilon$ or above 1_ϵ, and the corresponding advantage *would have* caused a step to reduce or increase the probability respectively, the clipping and min stops the step from happening.

References

[Bradley and Terry, 1952] Bradley, R. A. and Terry, M. E. (1952). Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345.

[Ouyang et al., 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744.

[Rafailov et al., 2023] Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., and Finn, C. (2023). Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256.