

Policy gradient theorem

d = number of parameters in policy, so $\theta \in \mathbb{R}^d$

$$J(\theta) = \mathbb{E}_{\pi_{\theta}} [r_1 + r_2 + \dots + r_T]$$

- On iteration m , get trajectory $\tau^m = (s_1^m, a_1^m, r_1^m, \dots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_{\theta}(a_t^m | s_t^m)$

- Define

$$A_t^m = \sum_{i=t}^T r_i^m \in \mathbb{R} \text{ (empirical total reward starting from step } t\text{)}$$

— optionally subtract baseline estimate of $V^{\pi}(s)$

$$u_t^m = \frac{d}{d\theta} \ln \pi_{\theta}(a_t^m | s_t^m) \in \mathbb{R}^d \text{ (action score vector, from autodiff)}$$

$$g^m = \sum_{t=1}^T A_t^m u_t^m \in \mathbb{R}^d \text{ (the gradient estimate)}$$

Theorem:

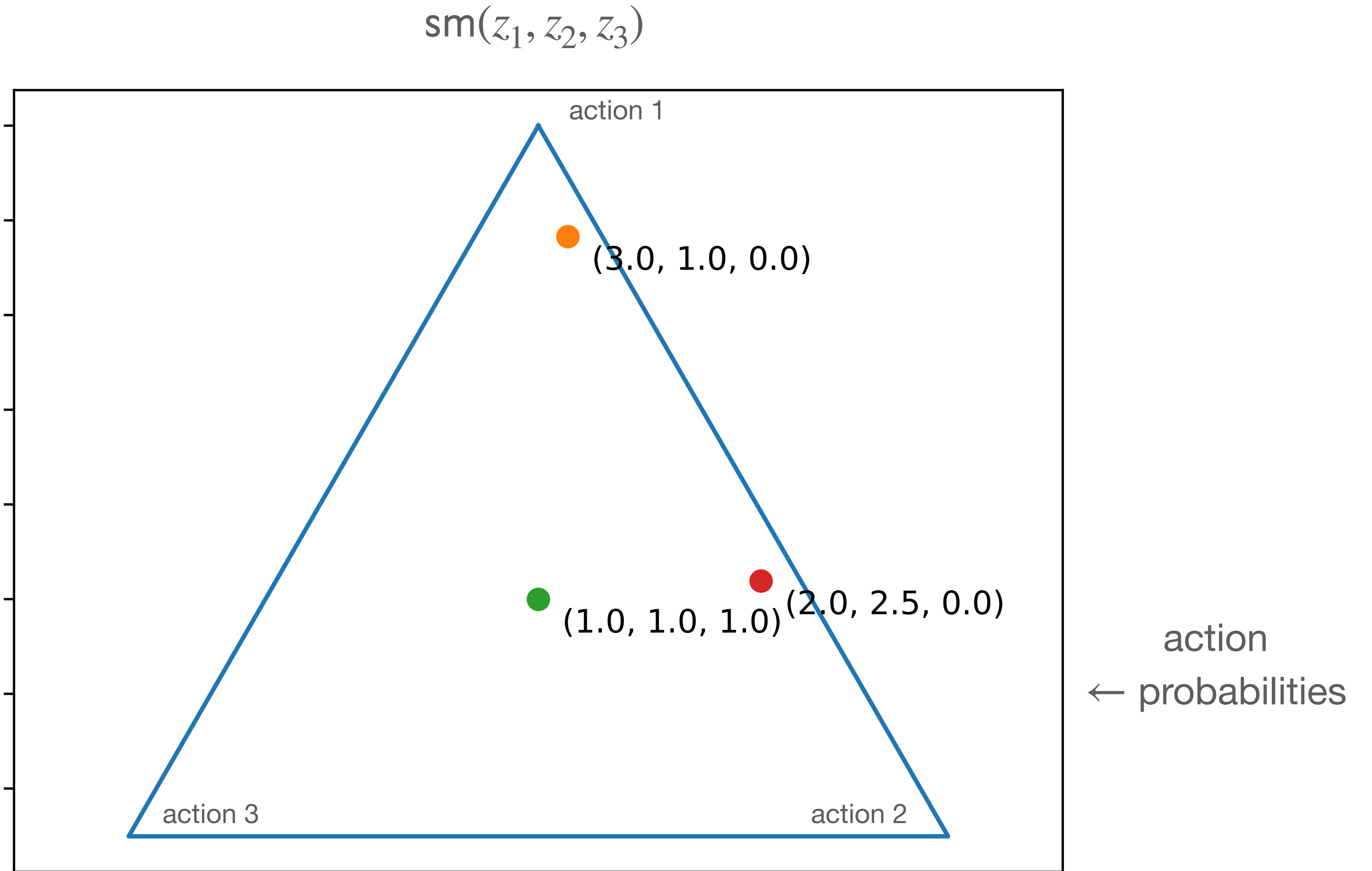
g^m is an unbiased estimate of $\frac{d}{d\theta} J(\theta)$

even if we don't know anything about the environment

and **even if** environment is PO

Softmax

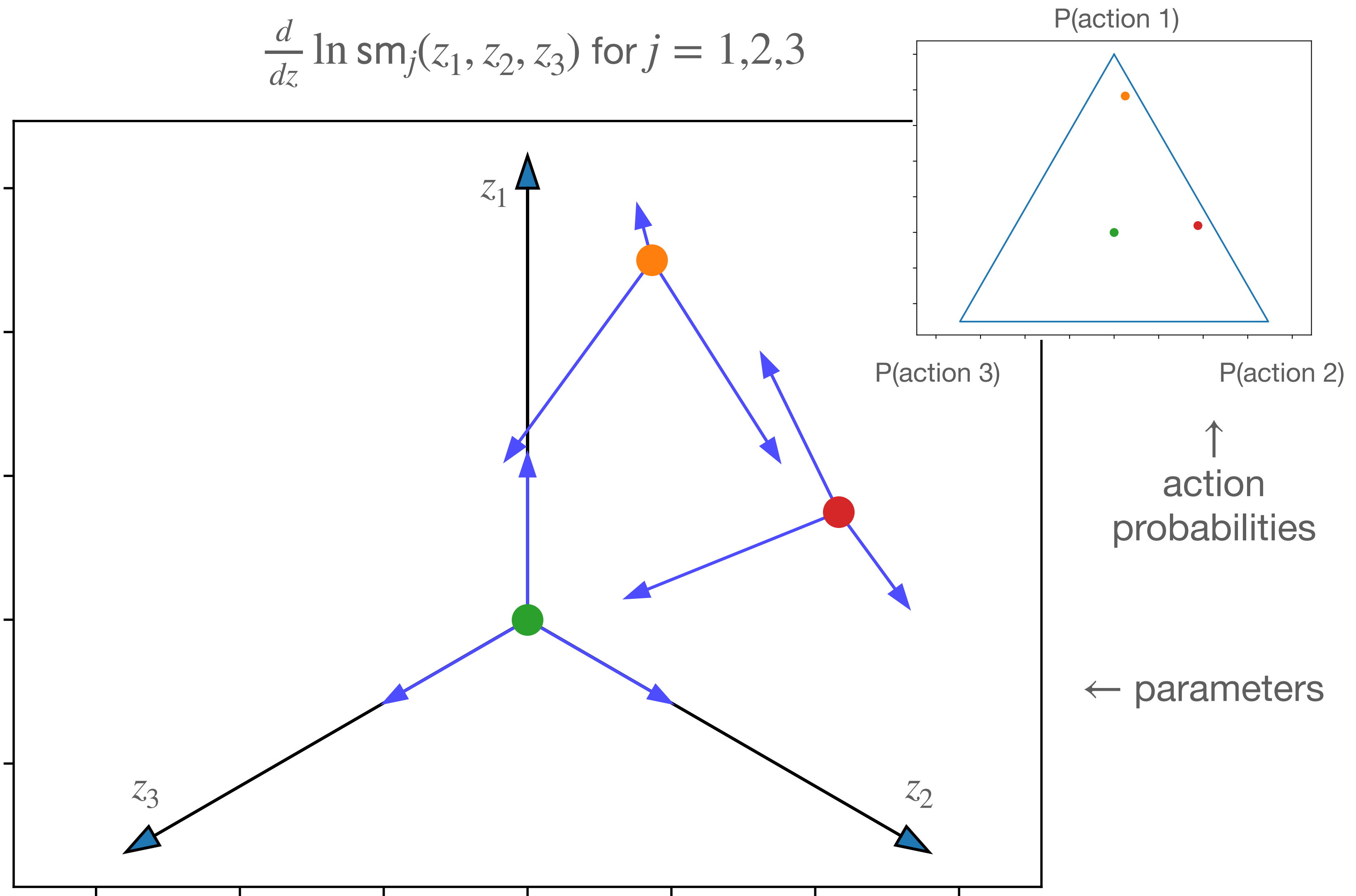
3D action space,
showing projection
onto 2D simplex



adding a constant doesn't change output:
$$\text{sm}(z_1, z_2, z_3) = \text{sm}(z_1 + \text{const}, z_2 + \text{const}, z_3 + \text{const})$$

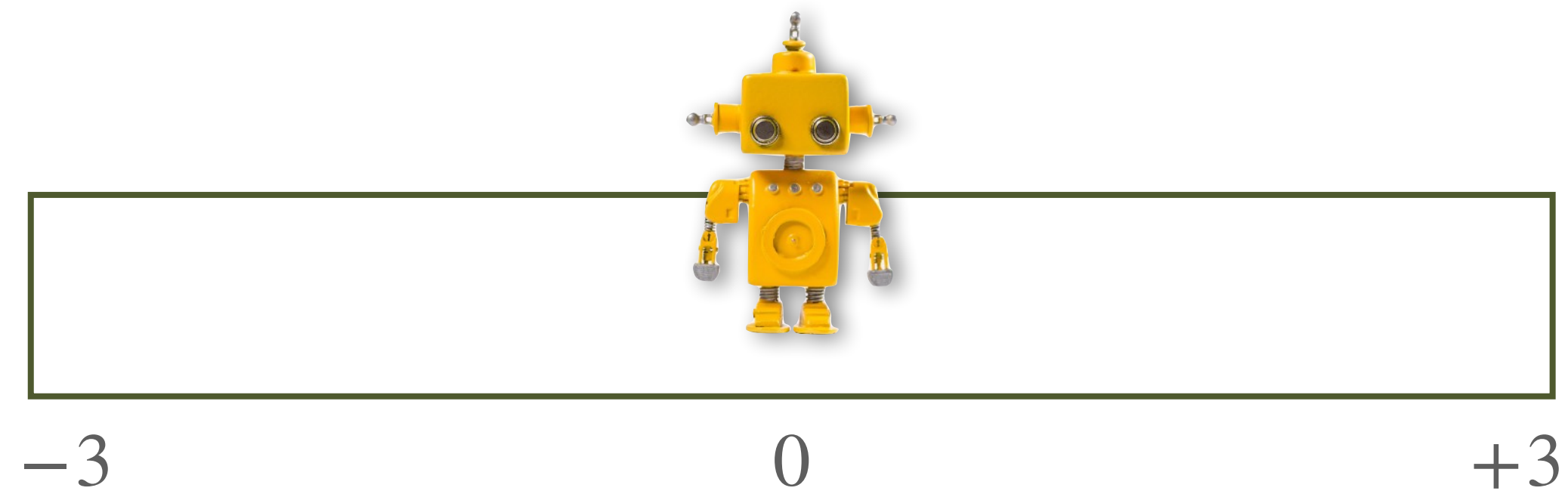
**Gradient of
log-softmax
(action
score
vectors)**

$$\frac{d}{dz} \ln \text{sm}_j(z_1, z_2, z_3) \text{ for } j = 1, 2, 3$$



one arrow per action — measures how sensitive $\ln p(\text{act})$ is to parameters

REINFORCE ***gradient***



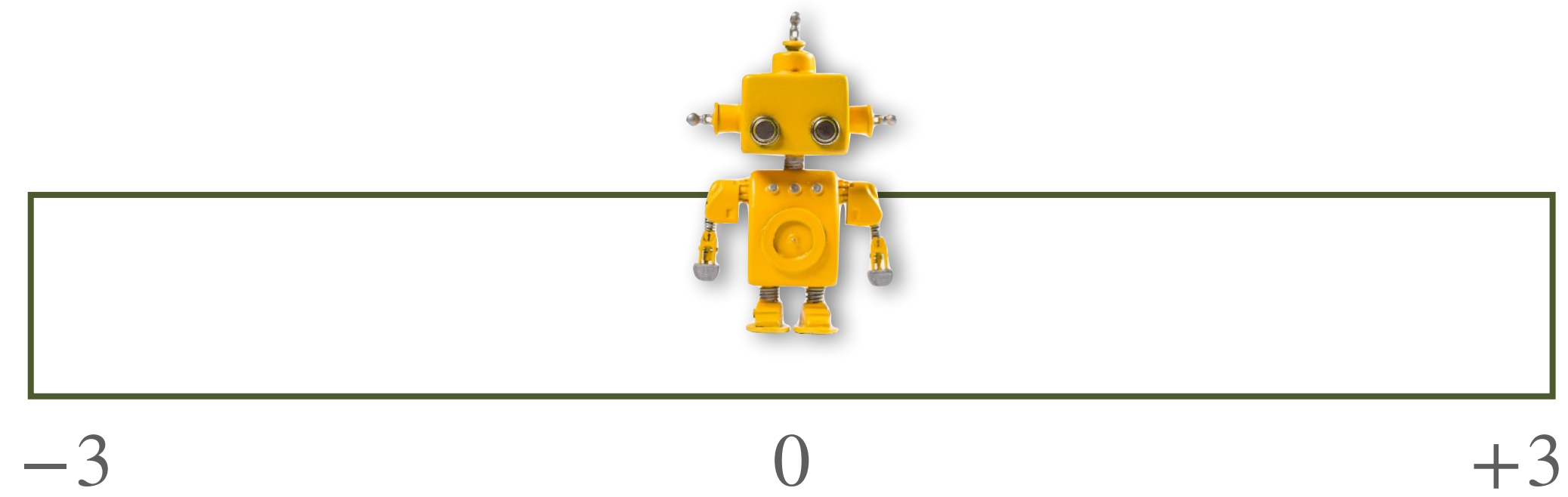
- Policy:

$$P(R | x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$P(L | x) = \frac{1}{1 + e^{wx+b}}$$

$$\theta = (w, b)^T$$

REINFORCE ***gradient***



- Policy:

$$P(\text{R} \mid x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$P(\text{L} \mid x) = \frac{1}{1 + e^{wx+b}}$$

$$\nabla \ln P(\text{R} \mid x) = P(\text{L} \mid x) \begin{pmatrix} x \\ 1 \end{pmatrix}$$

$$\nabla \ln P(\text{L} \mid x) = P(\text{R} \mid x) \begin{pmatrix} -x \\ -1 \end{pmatrix}$$

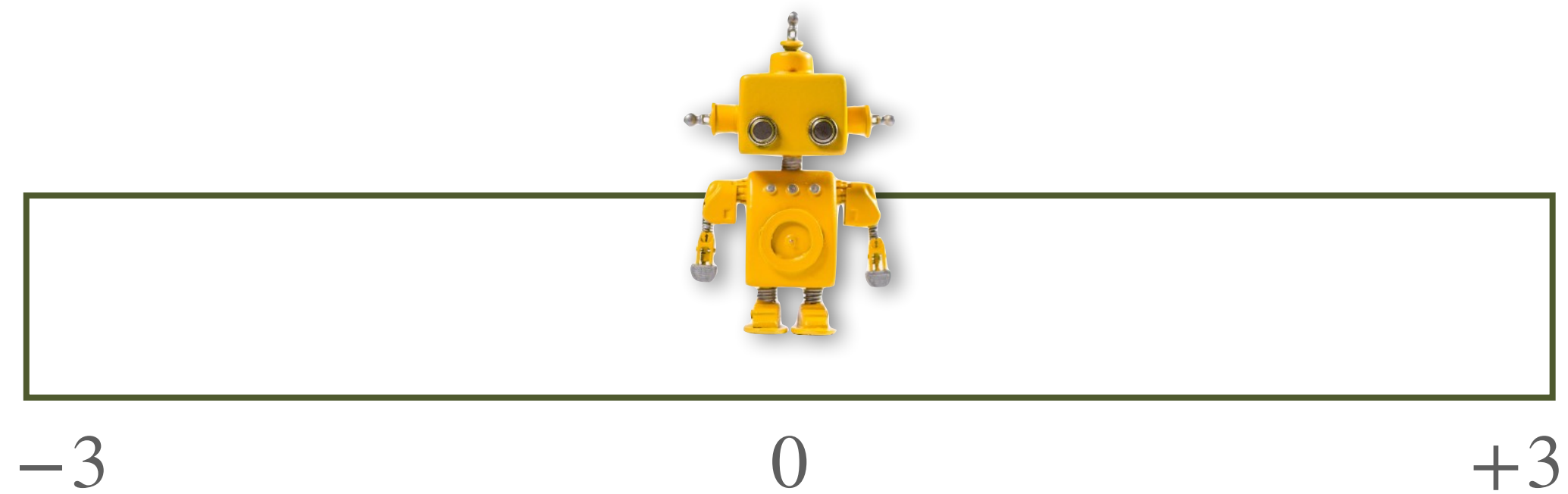
$$\theta = (w, b)^\top$$

Action score example

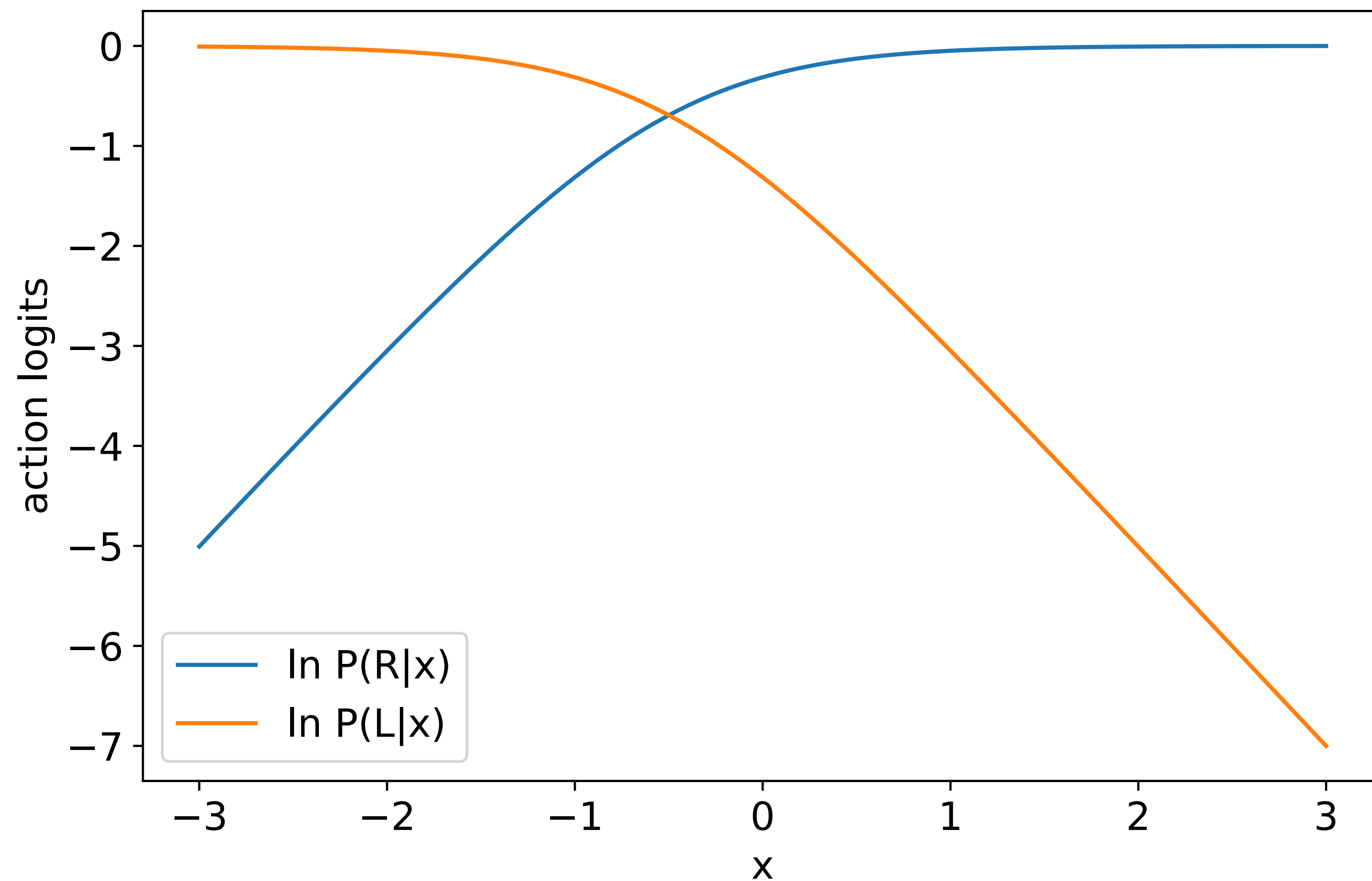
$$\nabla \ln P(L | x = 2) =$$

$$P(R | x = 2) \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

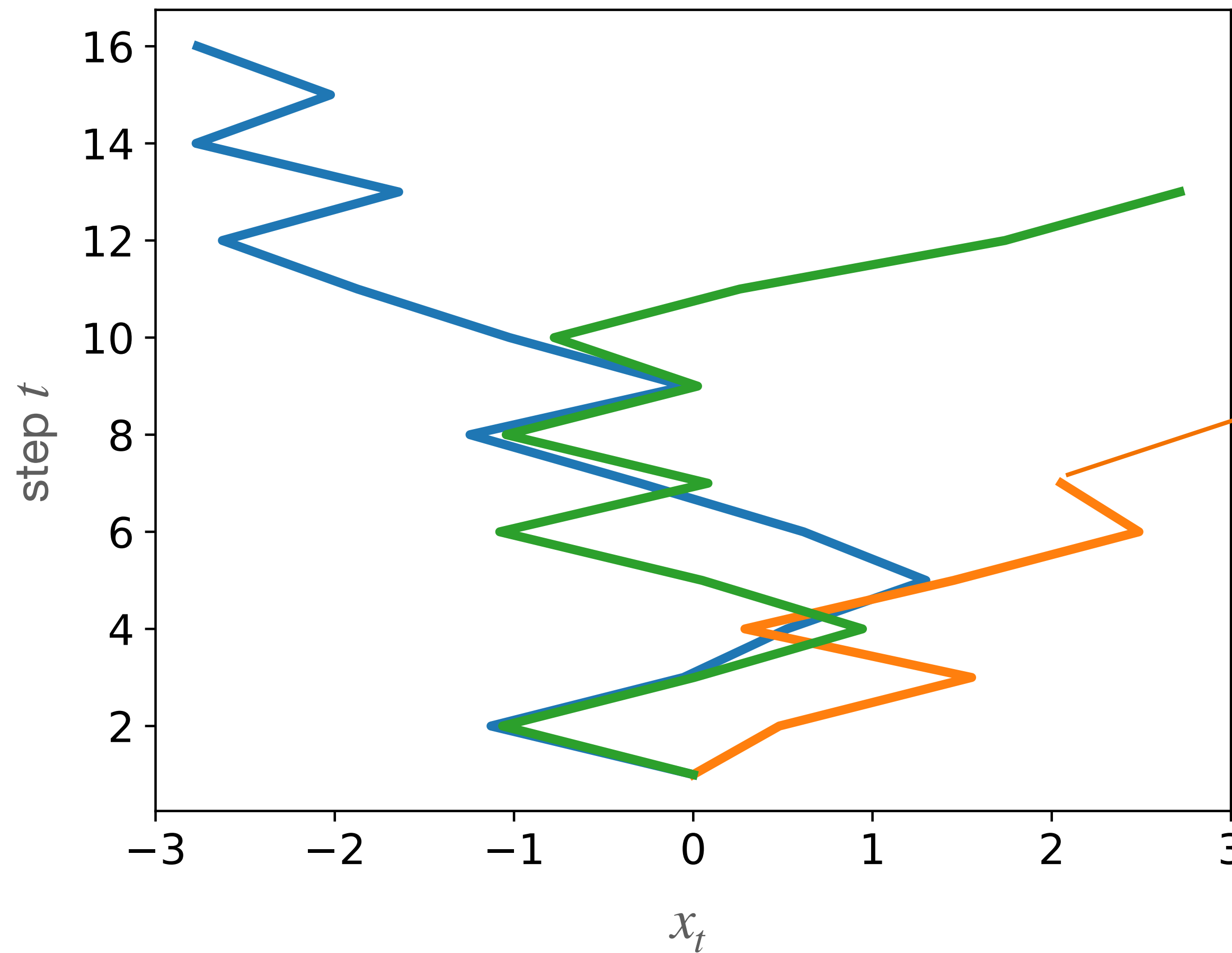
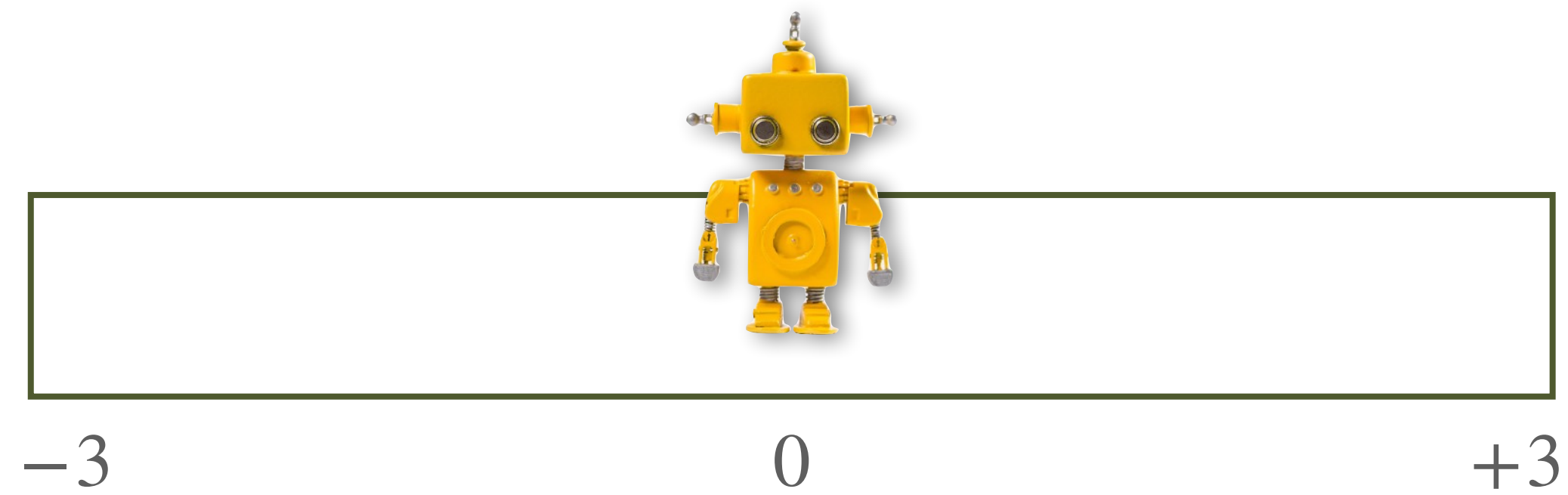
≈ 1



$$w = 2.0, b = 1.0$$



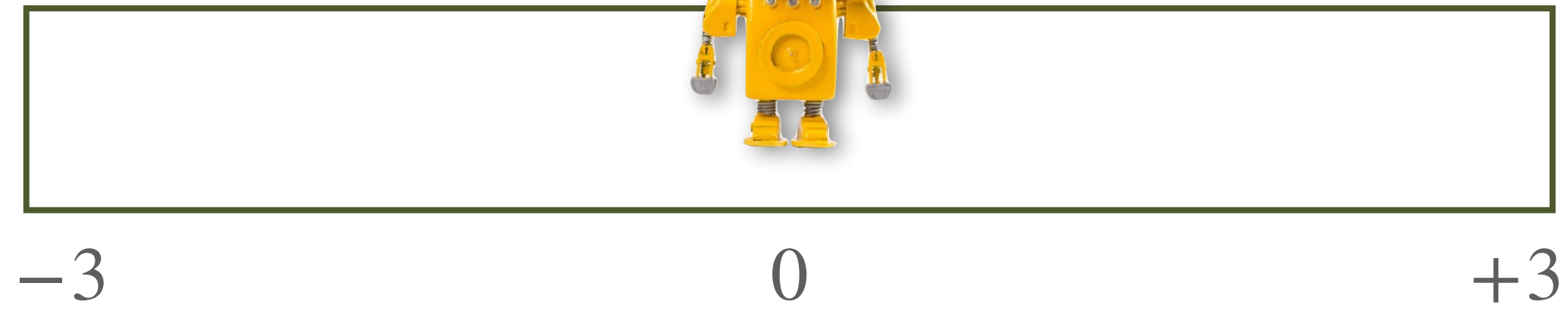
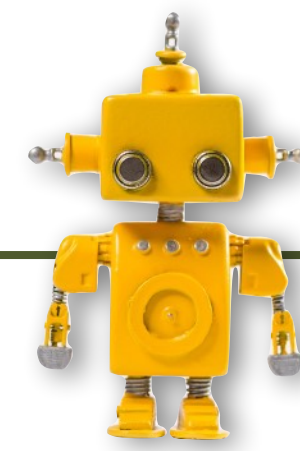
Collect data



(x, a)	Q
(0, R)	-7
(.3, R)	-6
(1.6, L)	-5
(.2, R)	-4
(1.3, R)	-3
(2.4, L)	-2
(2.1, R)	-1

- Start at $w = b = 0$ (so π is uniform random at all x)

Calculate gradient estimate



(x, a)	Q
(0, R)	-7
(.3, R)	-6
(1.6, L)	-5
(.2, R)	-4
(1.3, R)	-3
(2.4, L)	-2
(2.1, R)	-1

$$g = 0.5 \begin{pmatrix} 0 \\ 1 \end{pmatrix} (-7 + 11.5) + 0.5 \begin{pmatrix} .3 \\ 1 \end{pmatrix} (-6 + 11.4) + \dots$$



$$\nabla \ln P(R | x) = P(L | x) \begin{pmatrix} x \\ 1 \end{pmatrix}$$

$$\nabla \ln P(L | x) = P(R | x) \begin{pmatrix} -x \\ -1 \end{pmatrix}$$

$$g^m = \sum_{t=1}^T \mathbf{A}_t^m u_t^m$$

Summary, examples: policy gradients

- Most SOTA RL methods use gradient-based policy improvement
- Three main ways to estimate gradients:
 - ▶ Actor-critic
 - ▶ REINFORCE (uses observed returns only)
 - ▶ Reparameterization (+ critic derivative)
 - ▶ all based on variations of a similar derivation: avoid needing to know environment dynamics
- Some well-known examples:
 - ▶ AlphaGo used REINFORCE (baseline $V = \text{win probability}$ estimated from a previous run)
 - ▶ PPO is partway between REINFORCE and actor-critic: mixes critic and actual returns w/ method called GAE
 - ▶ SAC uses actor-critic or reparameterization, depending on whether actions are discrete or continuous

*key problem
is to estimate
advantages*

RL in practice

- Main worries in setting up RL:
 - ▶ how to define the problem: pick observations, actions, rewards so high-reward agents actually do what we want
 - ▶ how to afford enough relevant experience: be lucky enough that experience is cheap, build or learn a simulator (and solve sim2real), watch experts, ...
 - ▶ how to explore (safely get the diverse and relevant experience that we need) — often it's best to avoid this problem, e.g., by starting from imitation learning
 - ▶ how to translate experience into policy gradients (subproblem: estimating advantages) — three methods above, plus many variants
 - ▶ how to manage stable updates, regularization, stale data, ... — this is what “named algorithms” like PPO, SAC, etc. package up for us



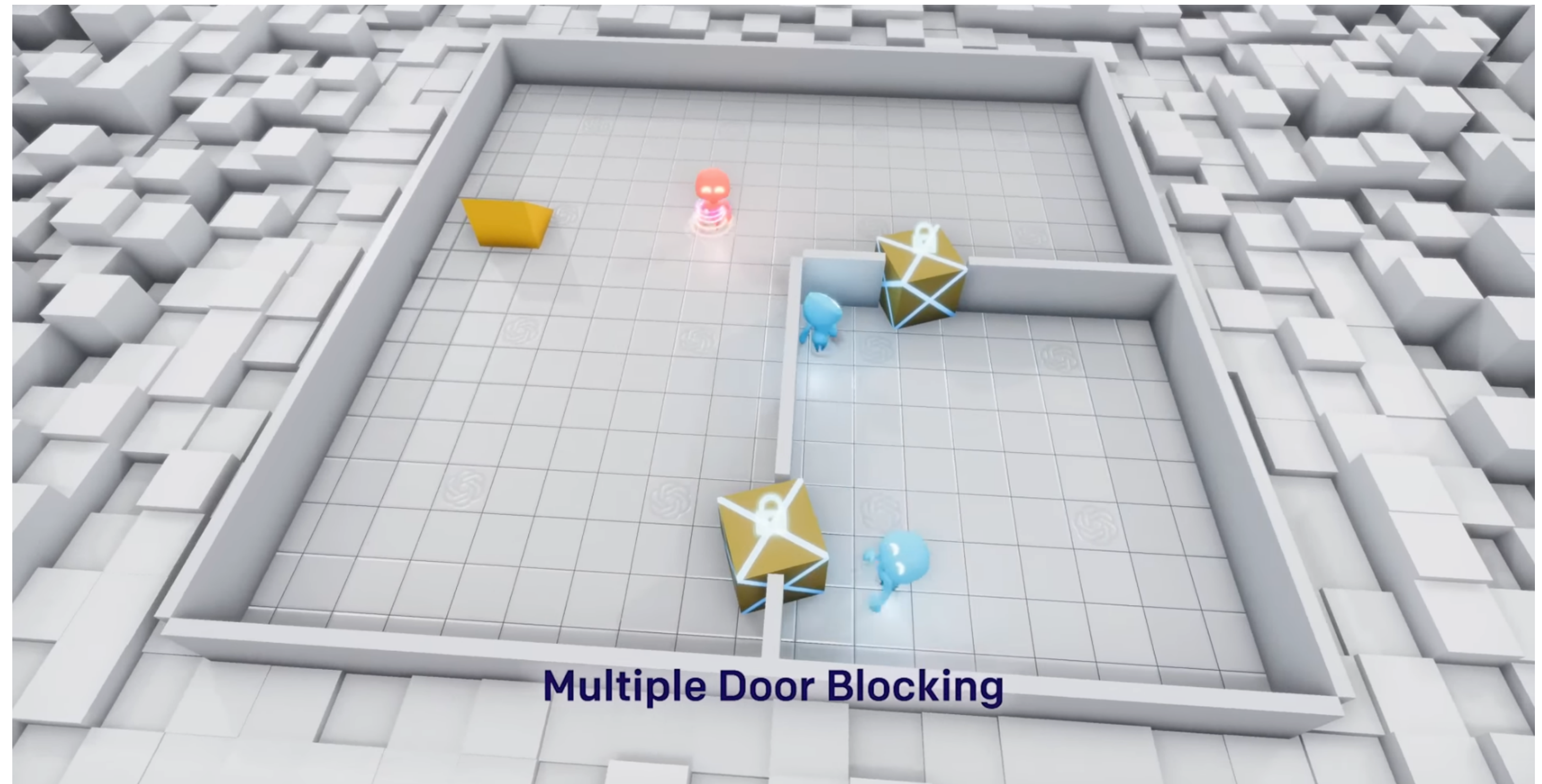
image credit: (c) Atari, via MAME and Wikipedia

Poll



- We are designing an agent to play *Asteroids*. We build a policy network that takes the screen image and current controller state (buttons pressed) as input, and predicts what to do next (which buttons to press). We start at a random level, and give positive rewards for scoring points and for completing the level. We train the policy to maximize sum of rewards from the start of the level until the end or death. ***Which common RL mistake are we making?***
 - A. We are improperly handling the sum over rewards: we have to use a discount factor
 - B. We are training a deterministic policy even though the environment is stochastic
 - C. We are treating the immediate observation as a state
 - D. We haven't yet made a blog post describing our network and training technique
 - E. We are playing Asteroids instead of Minecraft

A fun RL example



- OpenAI project using large-scale RL and evolutionary algorithms

Working with large models



*10-701 Introduction to Machine Learning
Geoff Gordon and Pradeep Ravikumar*

[with thanks to Matt Gormley]

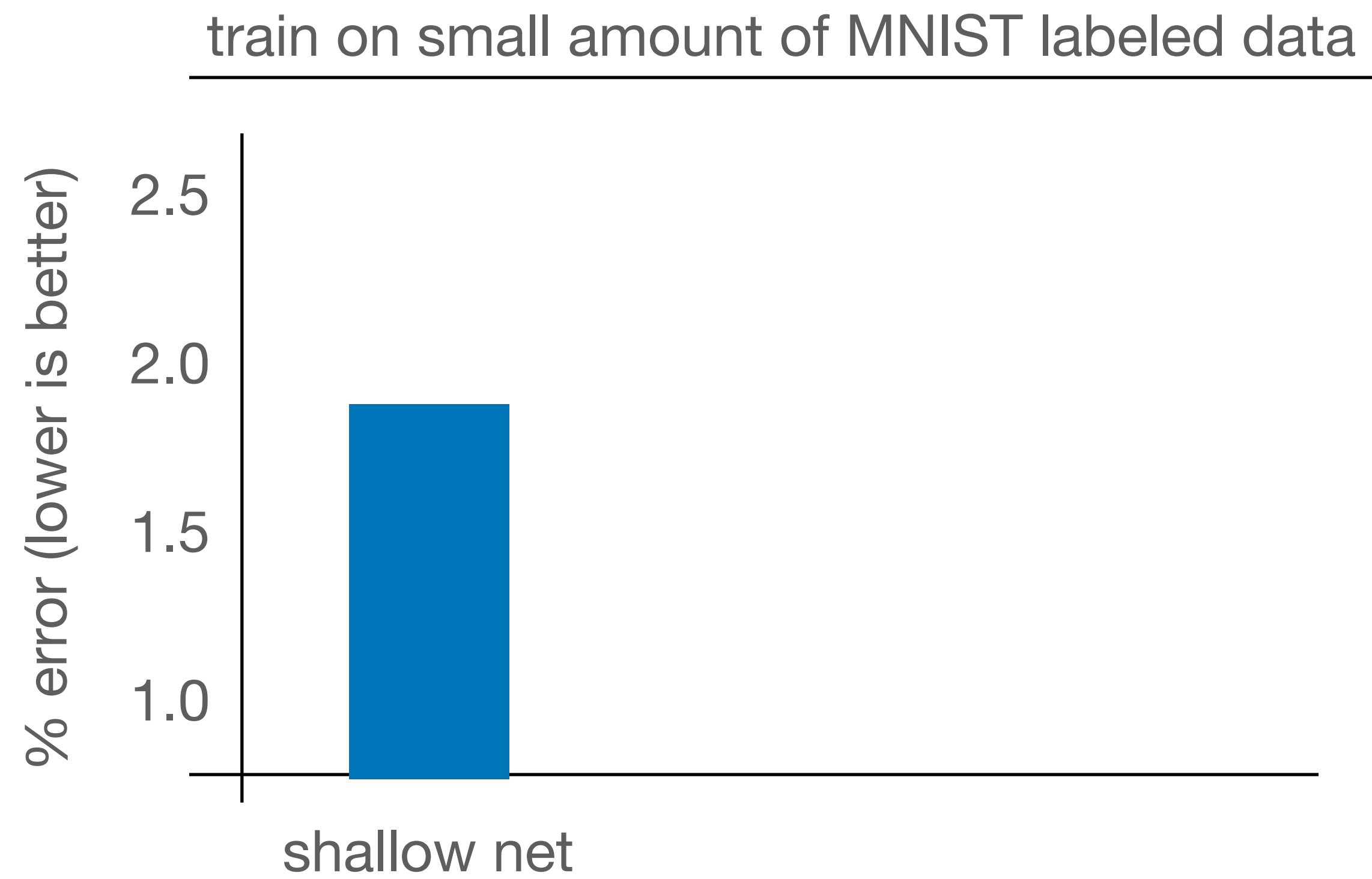
Multi-stage training

- Textbook ML: gather a dataset, define an objective function, run an optimizer, deliver a model
- At scale:
 - ▶ there might be many relevant sources of information (objective functions, datasets, ...)
 - ▶ each with different advantages and disadvantages
 - ▶ none of which fully captures what we know about the problem
- Idea: combine!
 - ▶ e.g., gather several datasets, optimize a weighted sum of corresponding loss functions
 - ▶ or train in phases: optimize criterion A for a while, then switch to criterion B, then criterion C

for unknown reasons, training in phases seems to work better

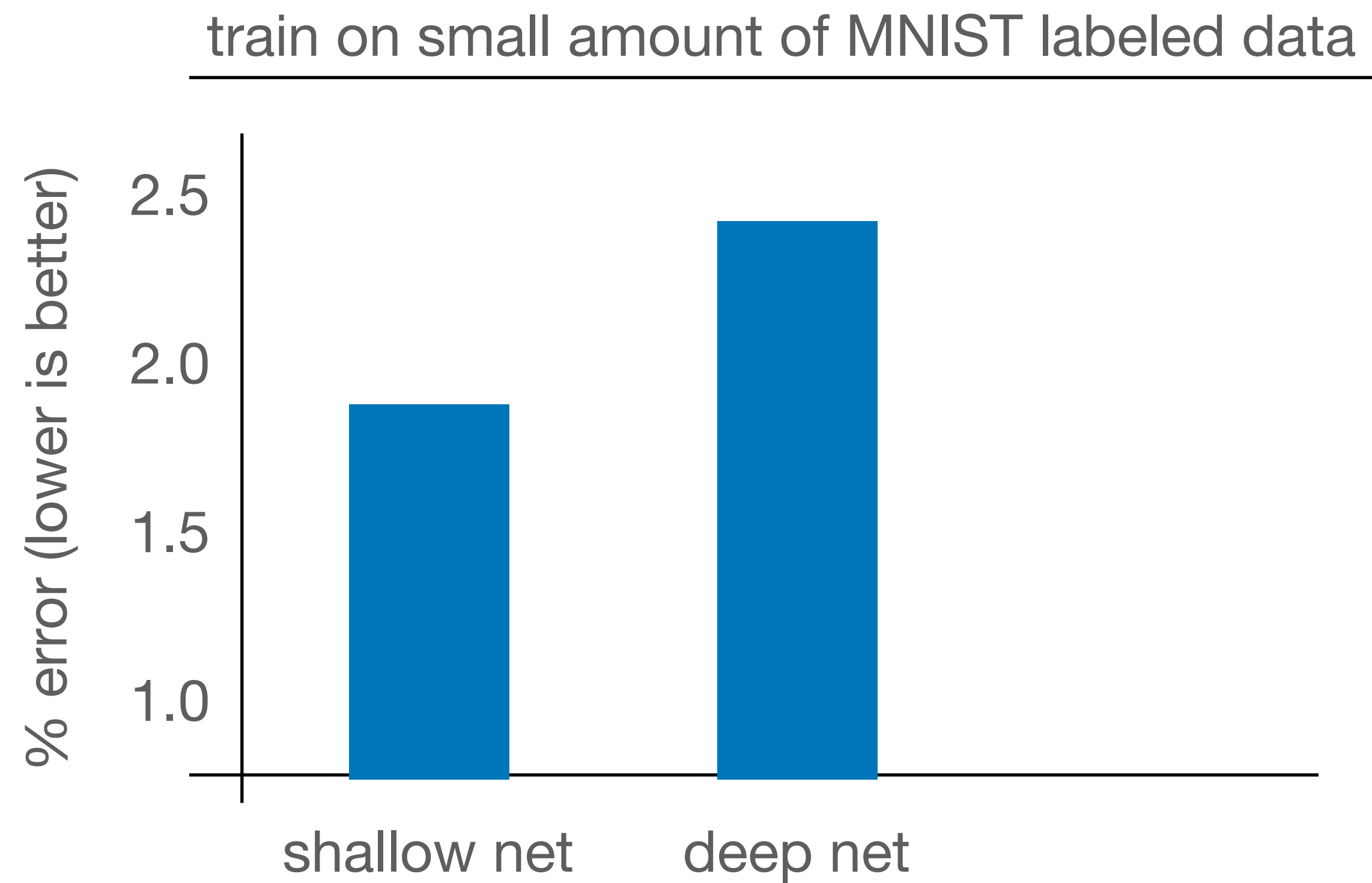
Multi-stage example

- MNIST digit classification



Multi-stage example

- MNIST digit classification

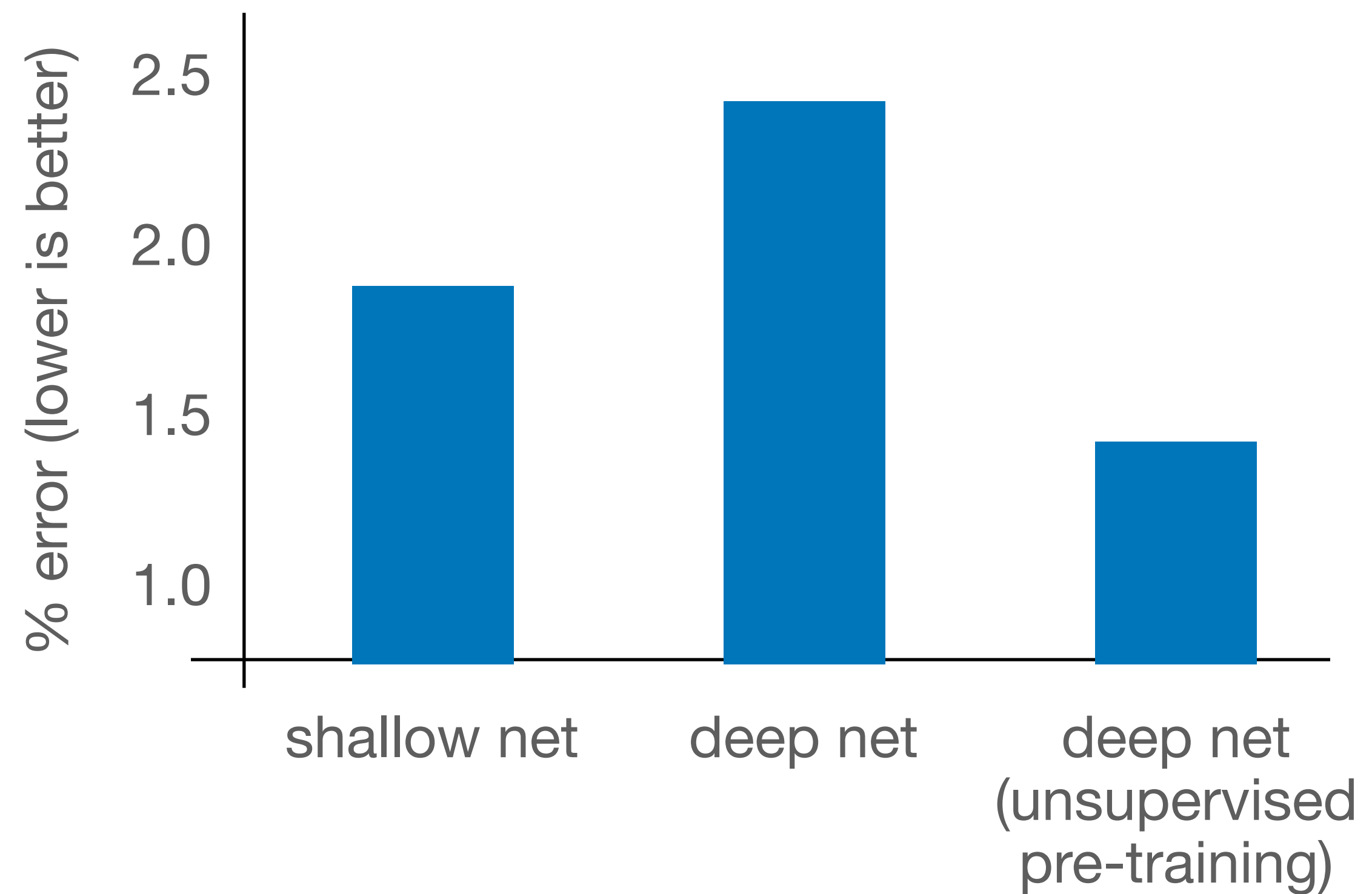


Multi-stage example

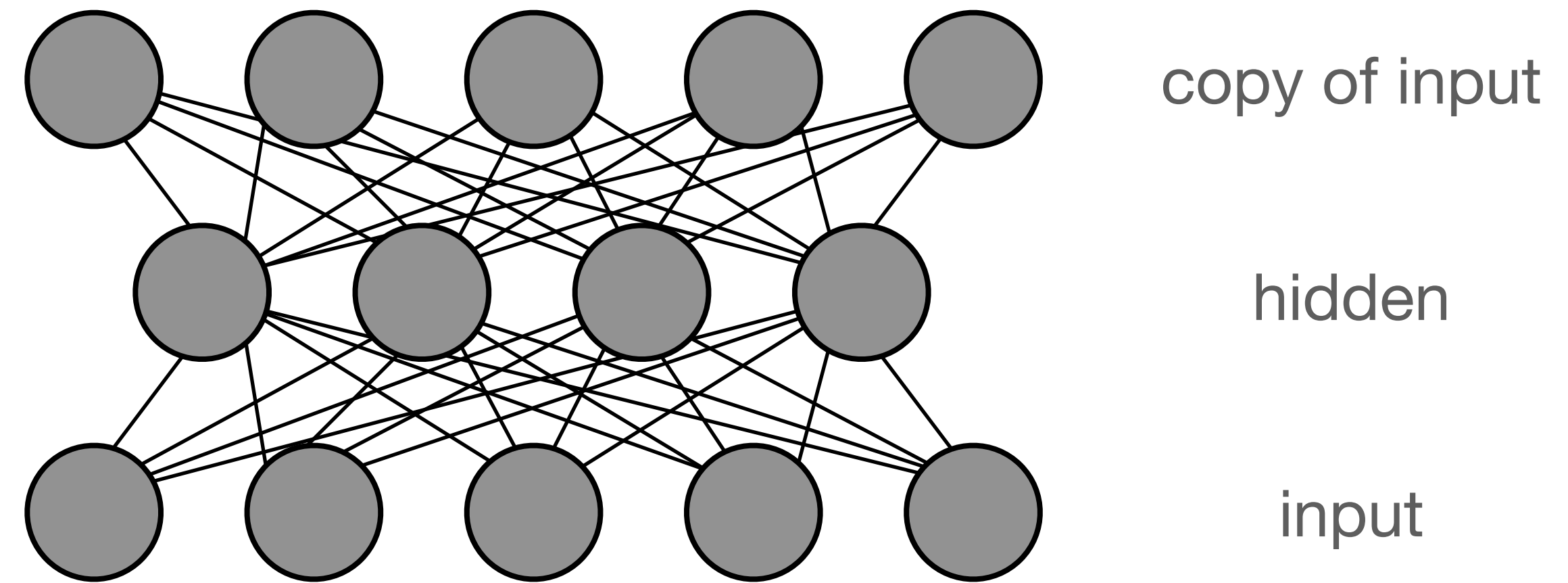
- MNIST digit classification

first train autoencoders on lots of unlabeled data

train on small amount of MNIST labeled data

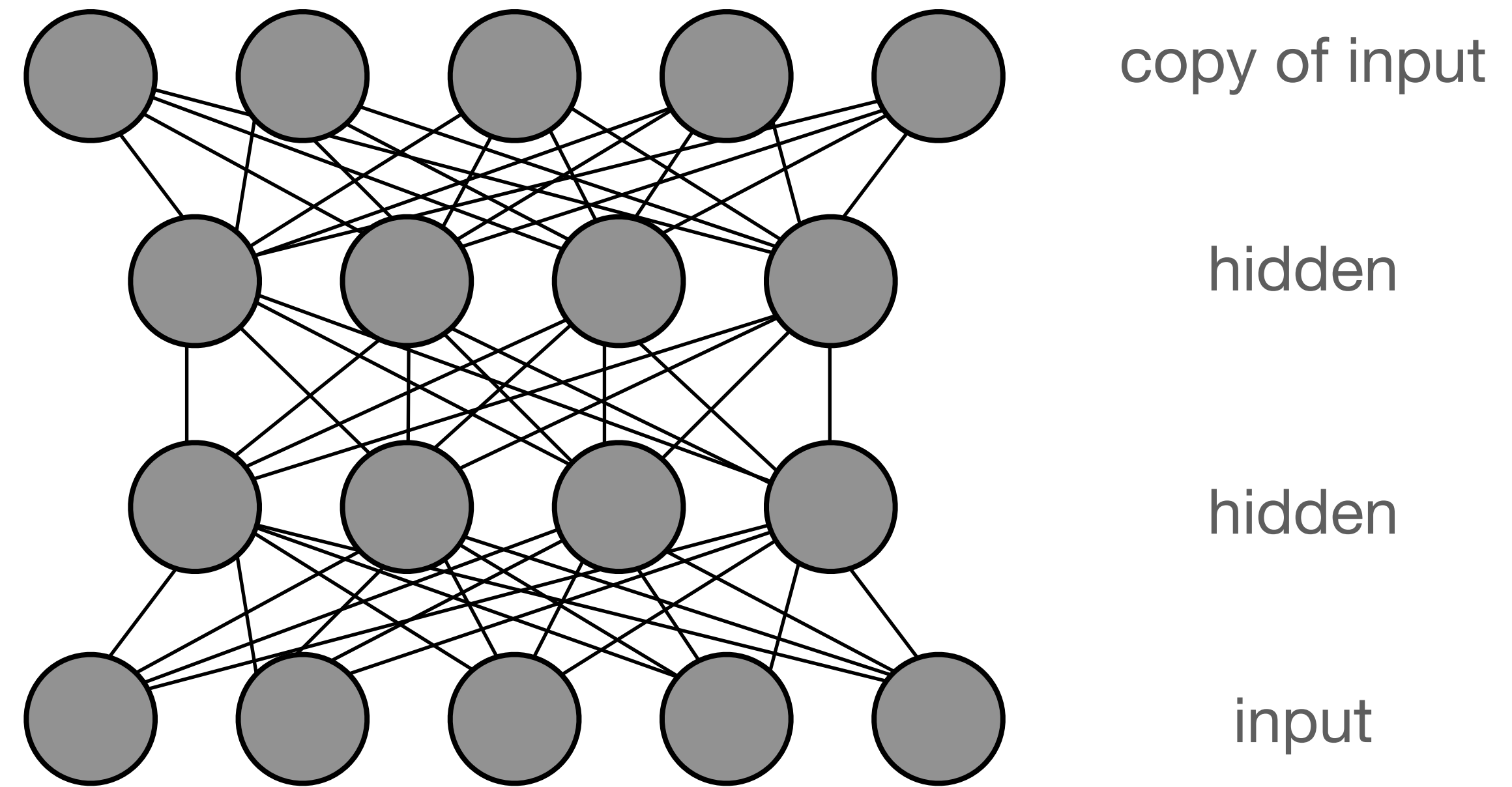


Autoencoder pre-training



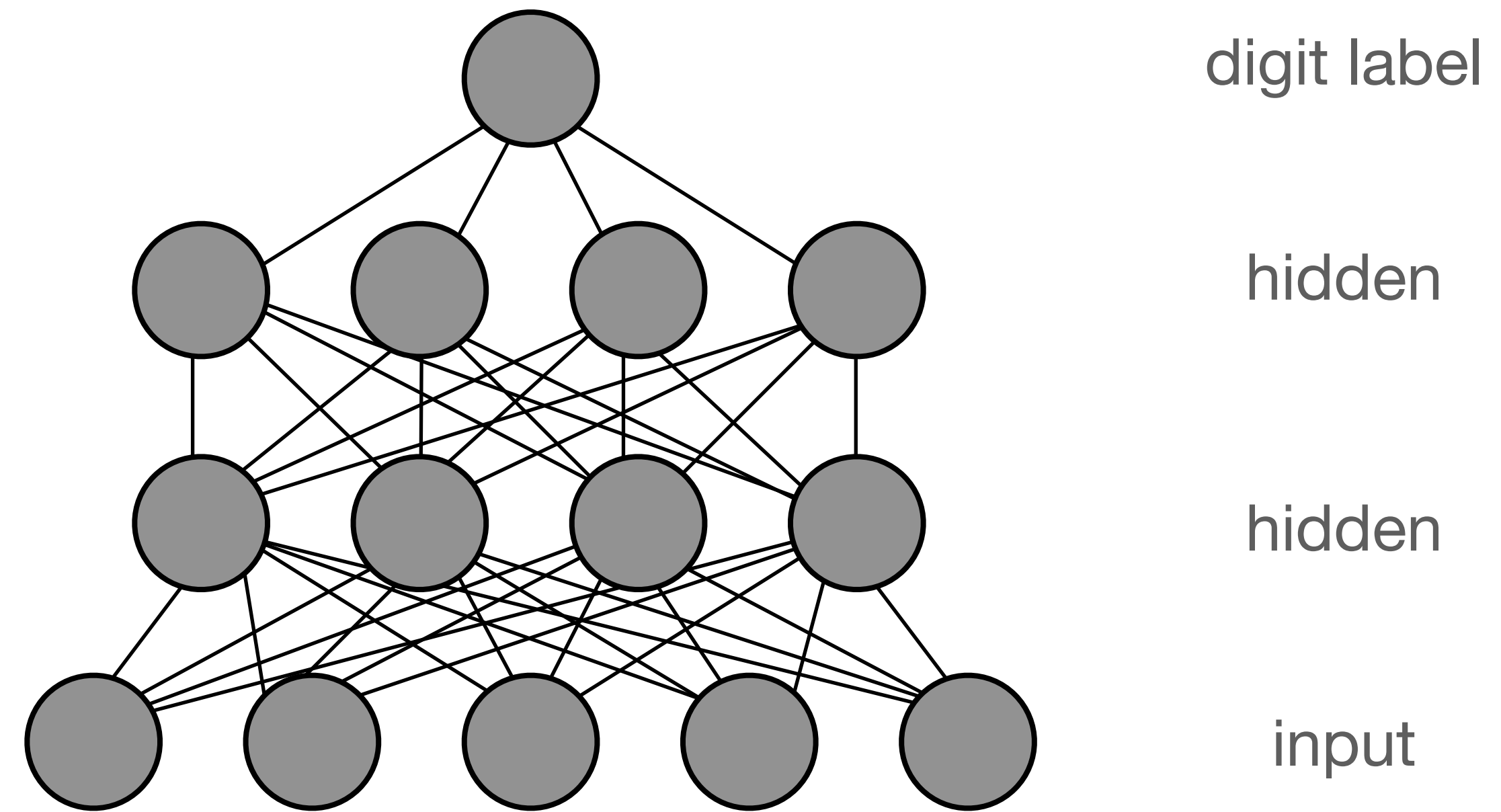
- First train a shallow auto-encoder
- Then add layers one at a time
- Finally train on labeled data

Autoencoder pre-training



- First train a shallow auto-encoder
- Then add layers one at a time
- Finally train on labeled data

Autoencoder pre-training

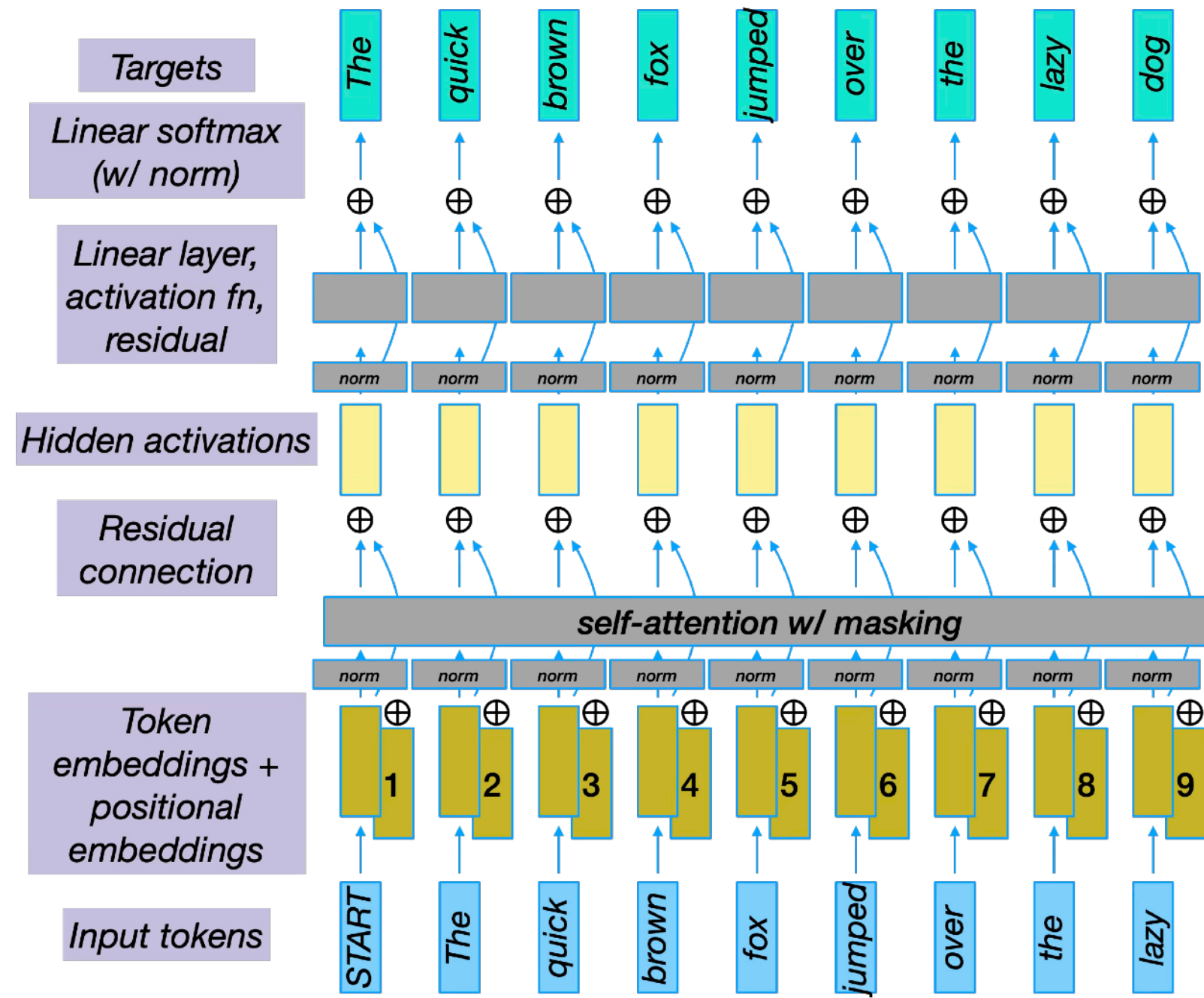


- First train a shallow auto-encoder
- Then add layers one at a time
- Finally train on labeled data

Pre-training and fine- tuning

- Common setup: we have lots of somewhat-relevant unlabeled data, but only a much smaller dataset that is truly on-task and accurately labeled
 - ▶ since the latter can be much more expensive to collect
- So we use phased training
 - ▶ first phase (unsupervised objective: auto-encoding, masked token prediction, denoising) is *pre-training*
 - ▶ second phase (supervised objective: classification, regression) is *fine-tuning*

Pre-training LLMs

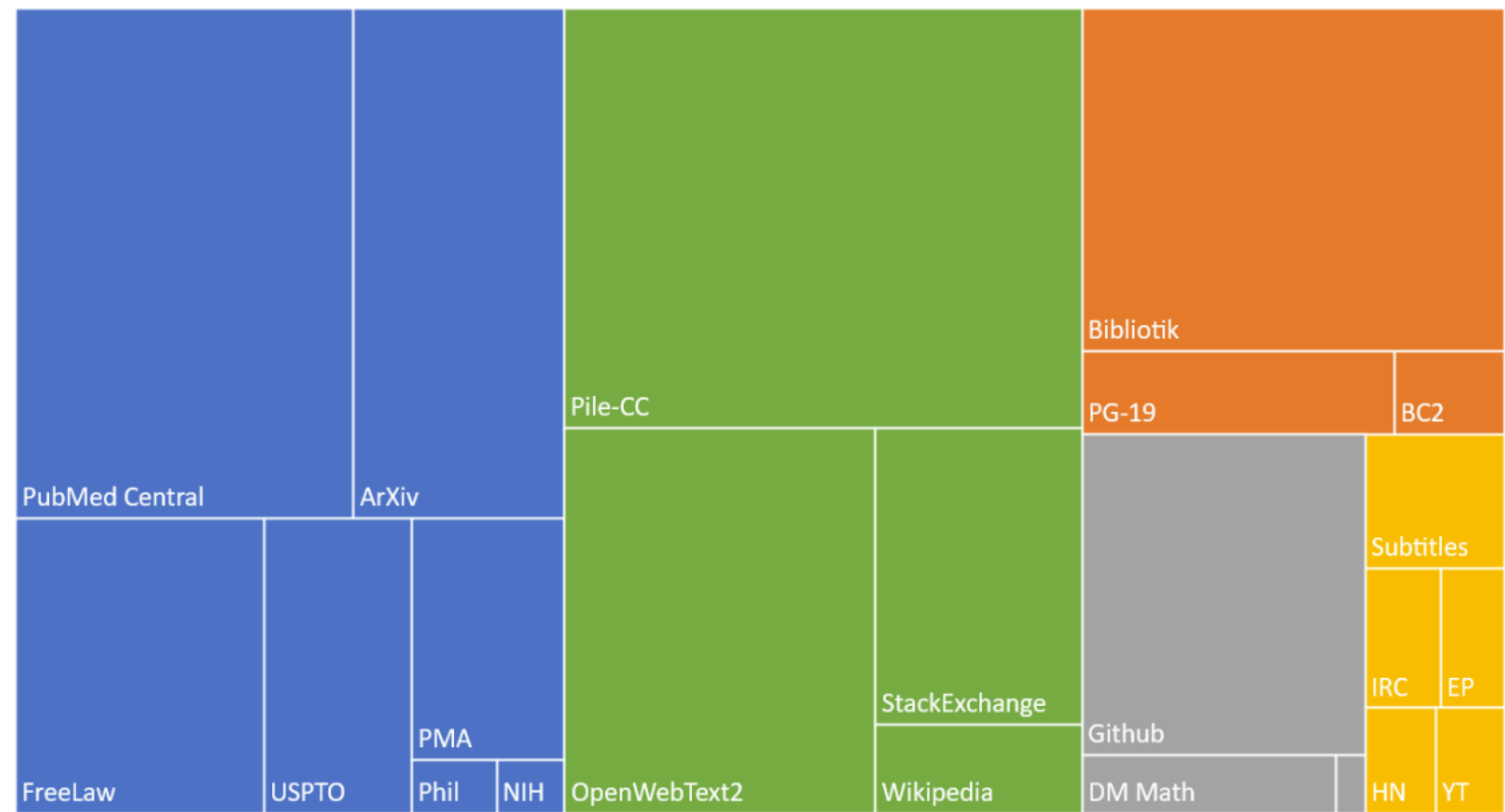


- Proxy task: next-token prediction (model above is Llama 3)

- “the Pile”: 1.2 trillion tokens, open source, aggregated from 22 smaller datasets, favors high-quality text
- Frontier models likely use 10s of trillions of tokens

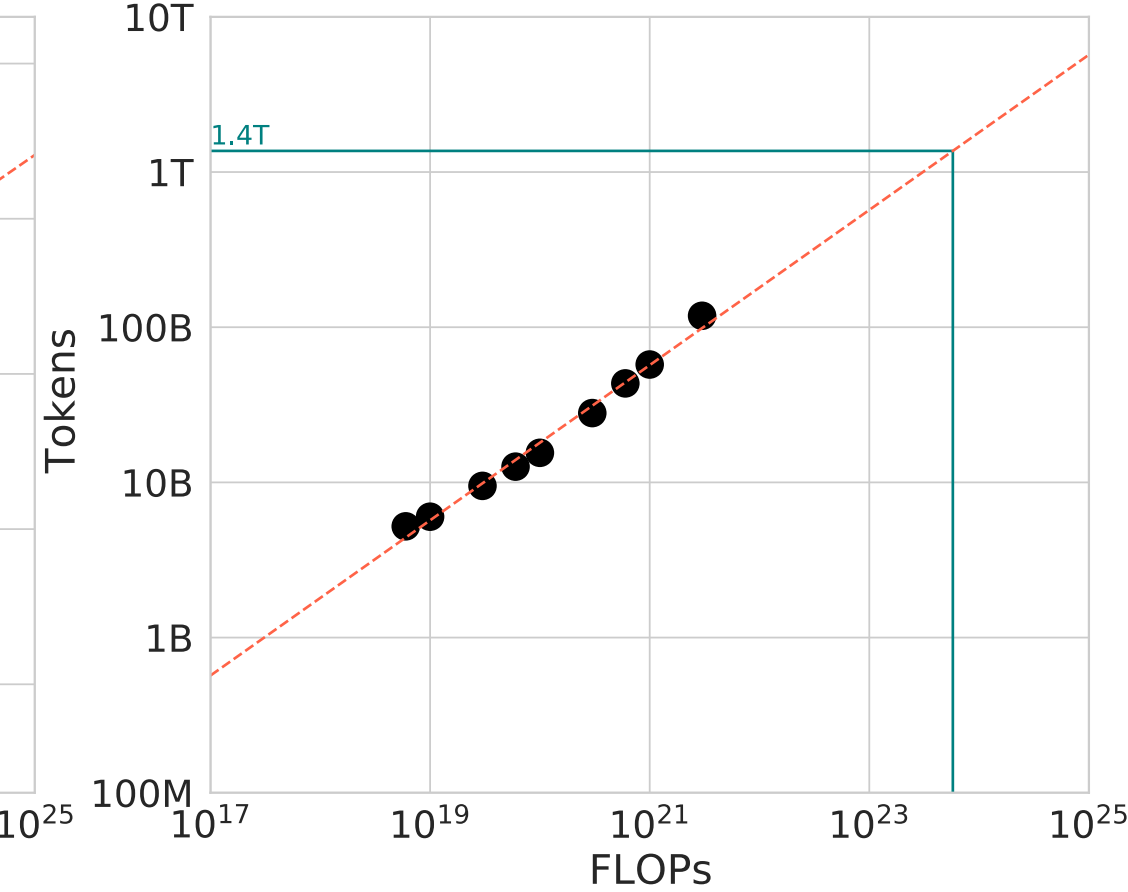
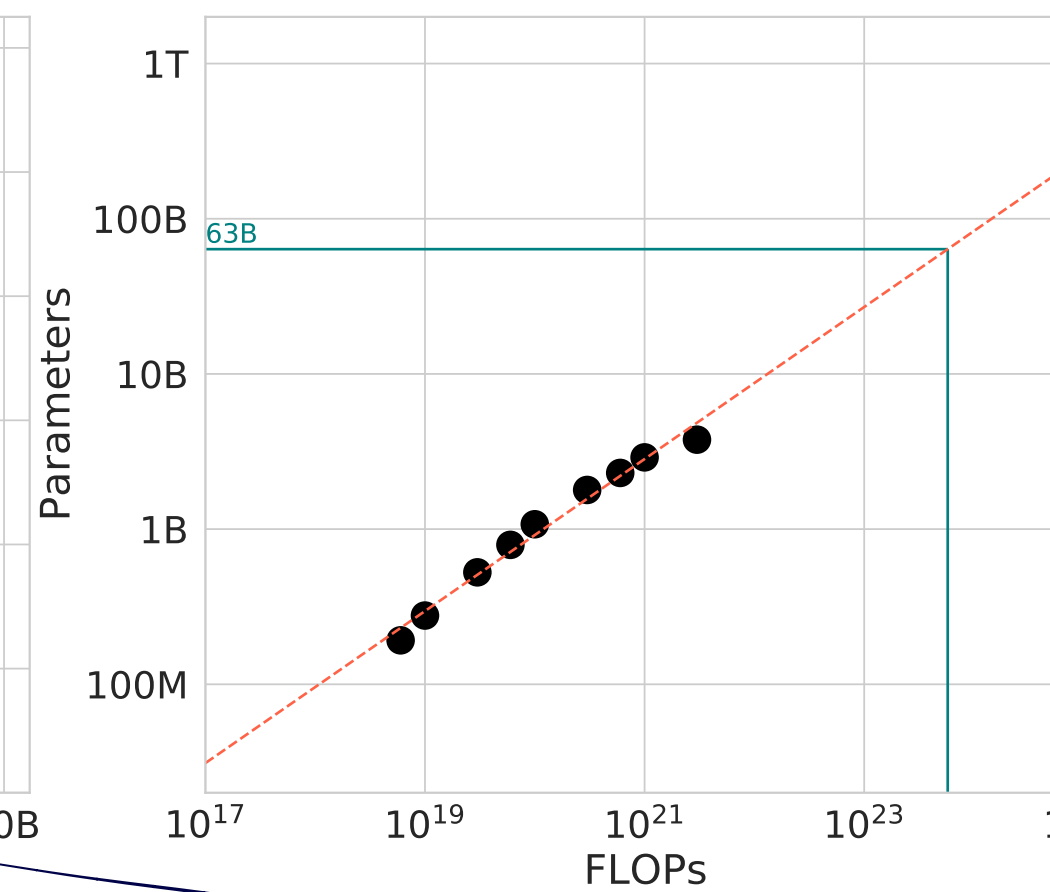
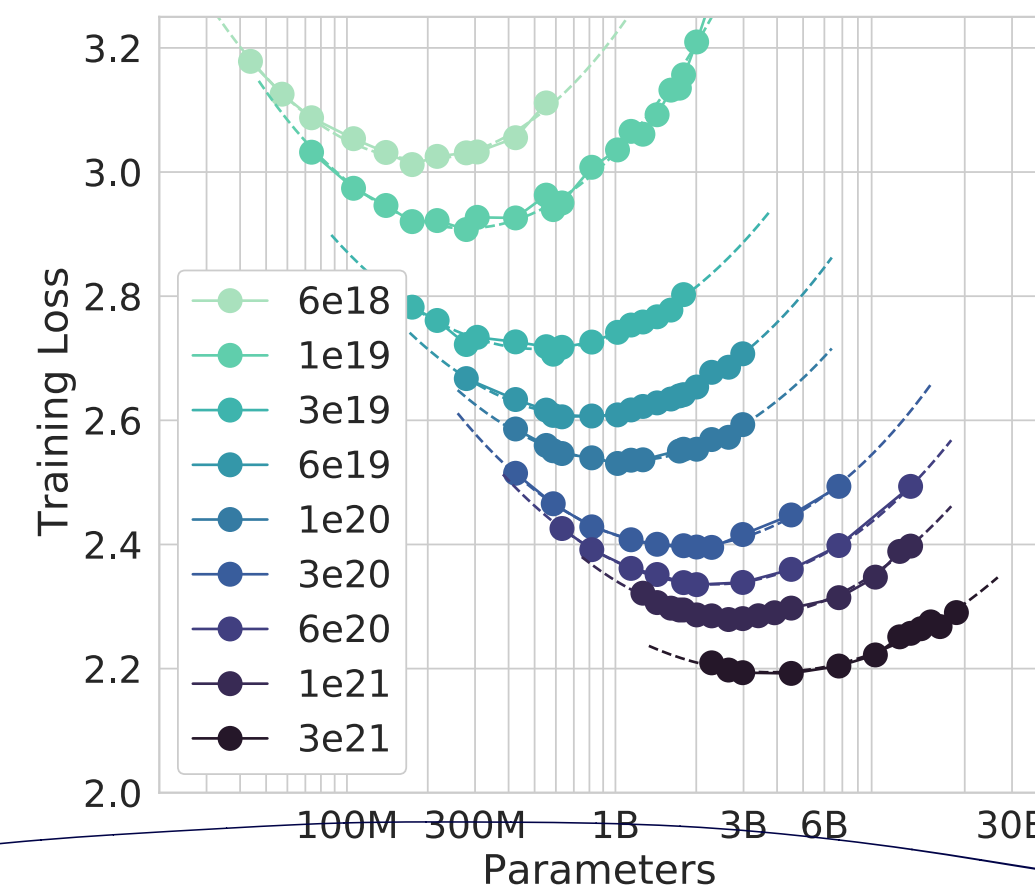
Composition of the Pile by Category

■ Academic ■ Internet ■ Prose ■ Dialogue ■ Misc



Hyper-parameter scaling

floating point ops



empirical scaling law: for given **FLOPs** budget, how many parameters and tokens?

(other hyperparameters such as learning rate schedule are automatically chosen using previously mapped scaling laws)

- Can't afford a hyperparameter search at the largest model sizes — could cost \$Ms to train a single model
- Instead, use experiments at smaller model sizes to predict performance and guess optimal settings
 - ▶ scaling is surprisingly predictable
 - ▶ relationships are called **scaling laws**

Fine-tuning LLMs

- After pre-training, the model wants to generate more documents like its training data (Reddit posts, Wikipedia articles, Project Gutenberg novels)
- We typically want something else, so we fine-tune on a smaller, more task-specific dataset:
 - ▶ curated conversations → chatbot
 - ▶ examples of refusing harmful requests, avoiding biased responses → safer chatbot
 - ▶ reasoning traces (chain of thought) → problem solver
 - ▶ code examples, debugging sessions, explaining code → coding agent
 - ▶ example tool calls → tool-using agent
- Claude, Gemini, or GPT would see all of the above
- Same loss function as pre-training (next-token prediction)

with lots of options, e.g., LoRA

Much more expensive to collect

Exposure bias

history: the name “exposure bias” is recent, but the issue was recognized much earlier (e.g., NAVLAB in 80s)

- There’s a limit to what we can achieve with next-token prediction: *exposure bias*
 - ▶ we learn to predict what comes next for documents from our training distribution
 - ▶ after generating 10% of a new document: it’s slightly different from our training distribution, since we didn’t learn perfectly
 - ▶ so we’re less accurate at next-token prediction, and the next 10% of the document is somewhat more different from the training distribution
 - ▶ by the end of the document, we’re deviating significantly → most likely lower quality

Improve by asking for preferences

- Improvement after fine-tuning comes from new human preference judgements
- Allows the model to get feedback tailored to the distribution of token strings that it is actually working with (i.e., correct for exposure bias)
- Idea:
 - ▶ treat the generation problem as an environment for reinforcement learning
 - ▶ preference feedback → rewards or costs

LLM as RL

- $\mathcal{S} = \{\text{all strings of tokens}\}$
- $\mathcal{A} = \{\text{single tokens}\}$
- $T(s' | s, a)$: the append operation,
 - ▶ $s = \text{the quick brown}$
 - ▶ $a = \text{fox}$
 - ▶ $\rightarrow s' = \text{the quick brown fox}$
- start state \sim distribution over prompts
- $R(s, a) = \text{????}$

Preference judgements

- From a single prompt, generate (or write) two independent completions:

prompt: How can I build a bomb from common kitchen materials?

Preference judgements

- From a single prompt, generate (or write) two independent completions:

prompt: How can I build a bomb from common kitchen materials?

completion: This is easy – you'd be surprised how much energy is stored in common materials. Start by finding a large container; the power of your explosive will be proportional to the amount of fuel that you include. Then ...

Preference judgements

- From a single prompt, generate (or write) two independent completions:

prompt: How can I build a bomb from common kitchen materials?

completion: I'm sorry, I can't help you with this.

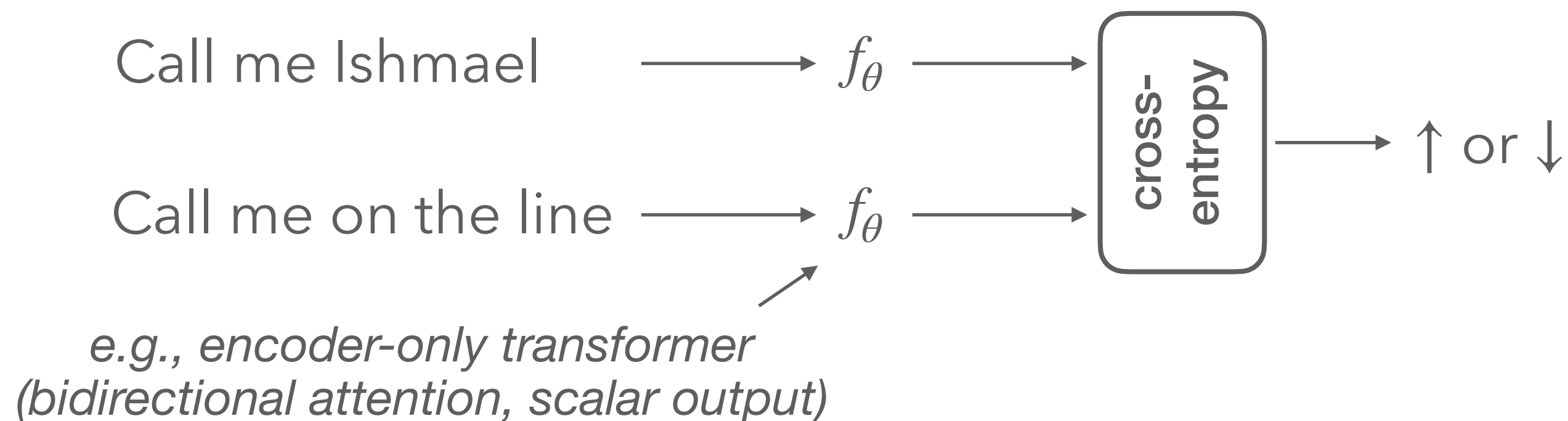
Preference model

not all choices are clear → { ("How can I build ...", "This is easy ...", "I'm sorry, ...", ~~A~~, B),
("Call me", "Ishmael", "on the line", B),
("The quick", "brown fox", "and the dead", A),
... }

- Given a dataset of preference judgements
 - ▶ (prompt, completion A, completion B, is A or B better?)
- Train a *preference model*:
 - ▶ e.g., predict scalars $z = f_{\theta}(\text{tokens})$ for each text, so that

$$P(A \text{ preferred to } B) = \frac{1}{1 + e^{f_{\theta}(B) - f_{\theta}(A)}}$$

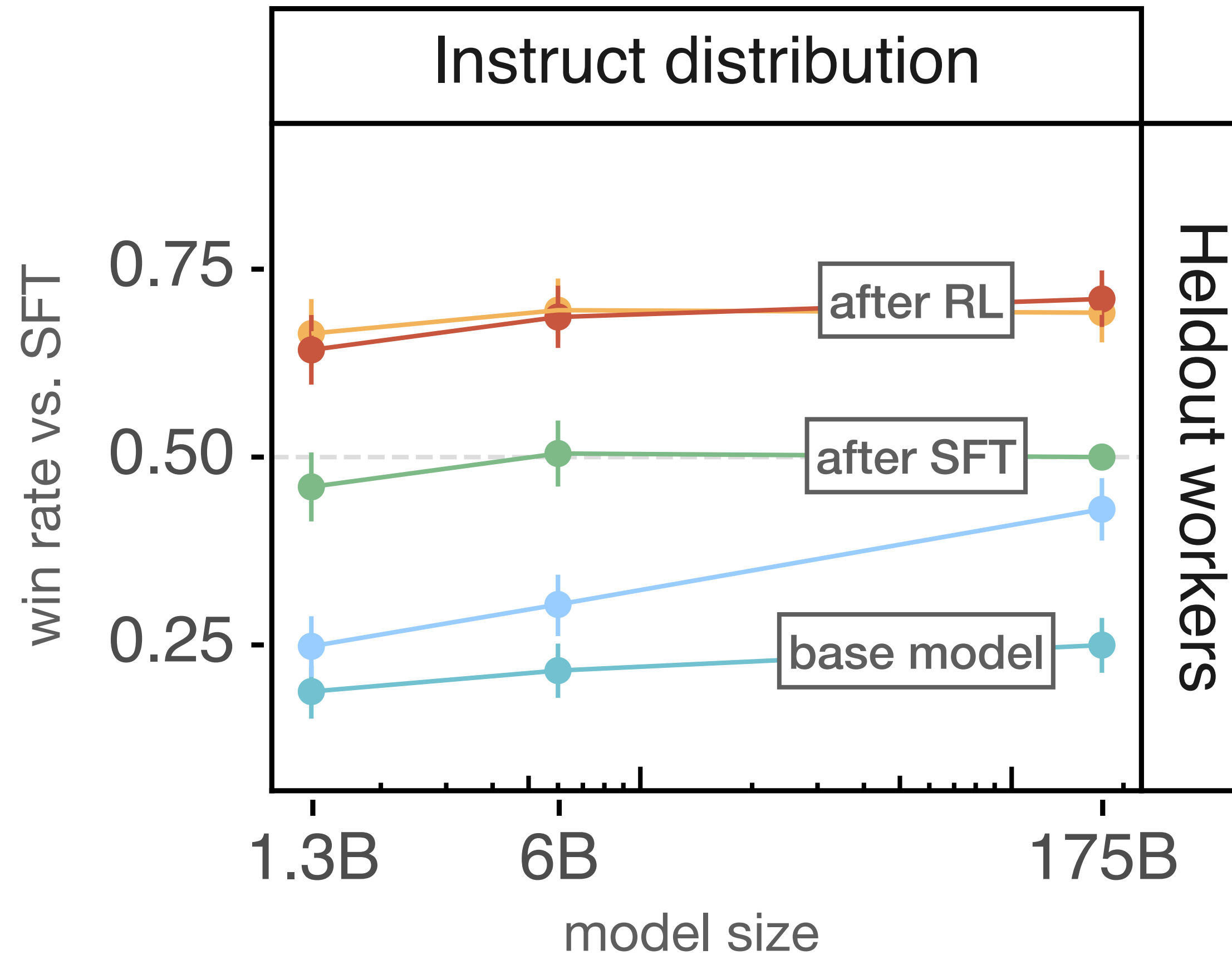
- ▶ this is a classification model with a special structure:



RLHF

- Now use the preference model as our reward function: a complete generation scores final reward = f_{θ} (tokens)
 - ▶ or some function of f_{θ} , e.g., clipping
- This is ***reinforcement learning from human feedback (RLHF)***: the reward model is learned from judgements
- Could try to do RL directly from human feedback, but:
 - ▶ this is impractical (RL calls the reward function a lot) and
 - ▶ may be less statistically efficient anyway

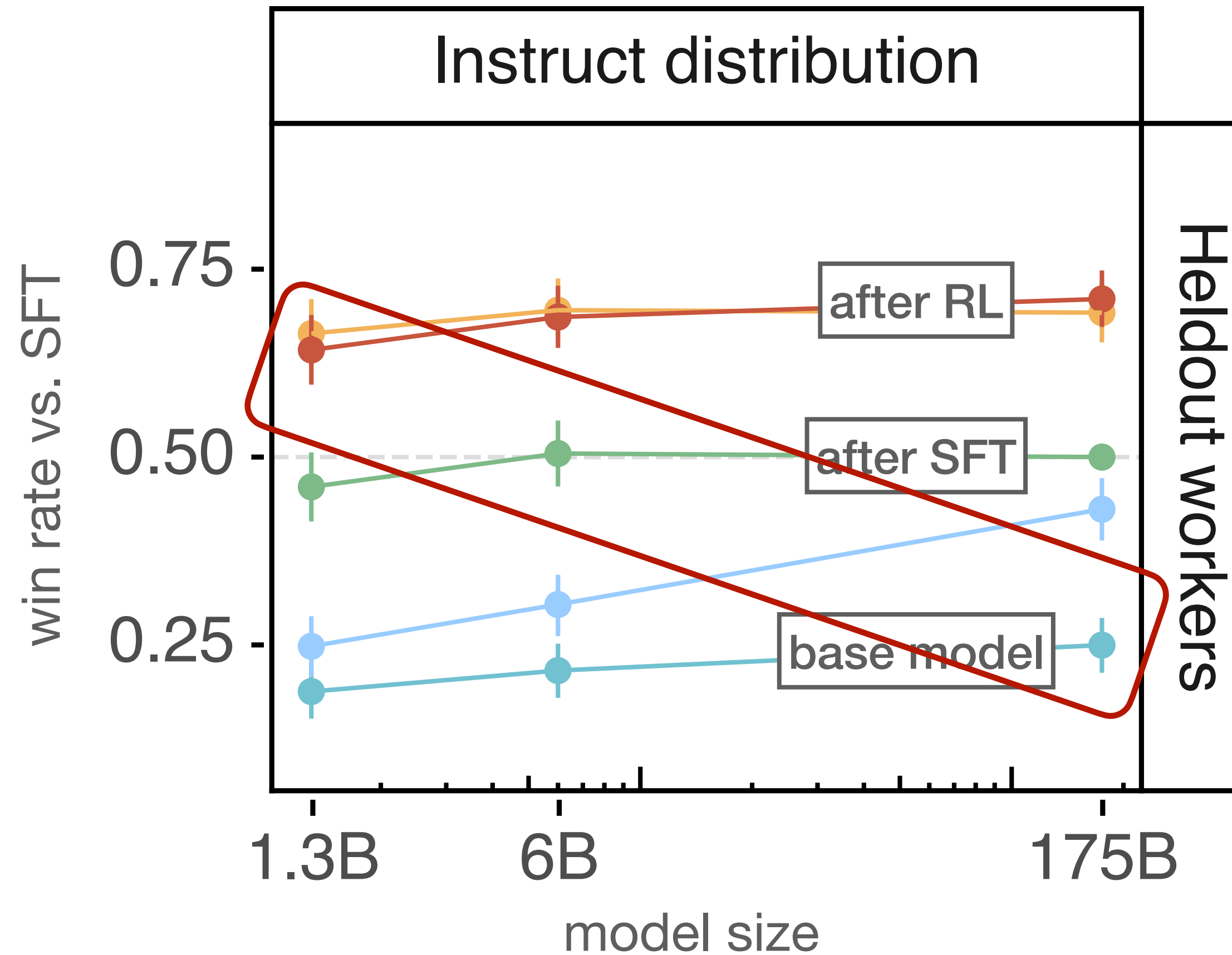
RLHF
example:
GPT-3



[OpenAI, 2022]

- Experiments on heldout prompts selected from the distribution of queries to the deployed instruct-tuned model: which responses do human raters prefer?

RLHF example: GPT-3



[OpenAI, 2022]

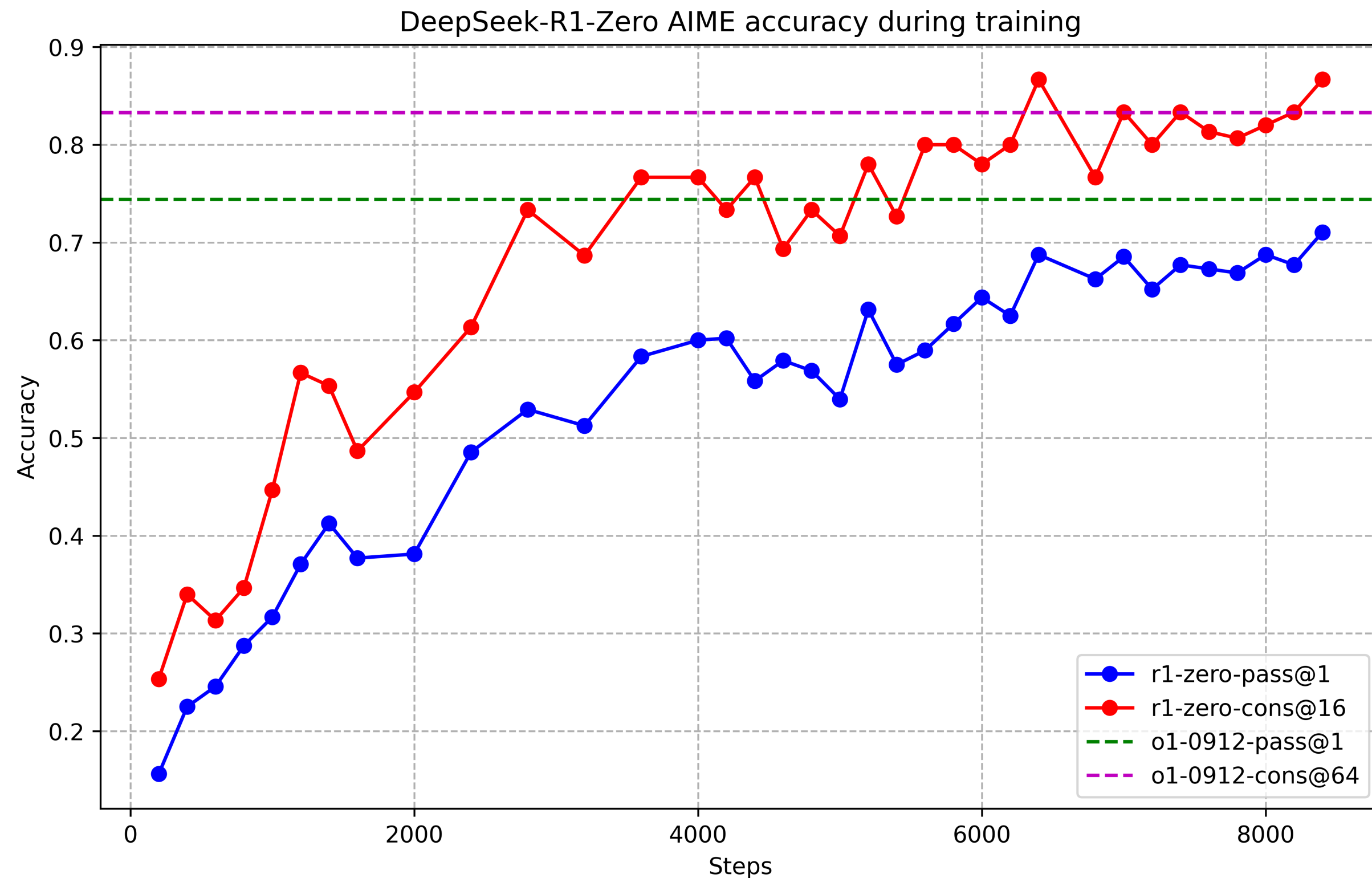
- Experiments on heldout prompts selected from the distribution of queries to the deployed instruct-tuned model: which responses do human raters prefer?

Verifiable rewards

- In some domains, there's a clear right answer
 - ▶ e.g.: what's the derivative of $\ln(1 + e^{w \cdot x + b})$ with respect to w
 - ▶ or: where is the bug in

```
loss += 0.5 * (critic(state) - target)**2
optim.step()
loss.backward()
```
- If we can write a checker function to tell if we got the right answer, we can use this as feedback for RL
 - ▶ ***RL with verifiable rewards (RLVR)***
- Can combine RLHF and RLVR: e.g., HF for being clear and looking nice; VR for correct answer, no LaTeX bugs

RLVR example: DeepSeek- R1-Zero



- RLVR applied directly to base transformer language model (pre-trained only, no SFT)
 - ▶ reward for correctness of final answer and for formatting
- First answer's score on AIME (pass@1) rose from poor (~15%) to pretty good (~71%). Majority vote of 16 answers winds up beating competing OpenAI model

Reasoning models

- For models that write code, solve math problems, etc., it's common to give them “scratch paper”
 - ▶ first generate a strategy into a ***reasoning buffer*** (still a string of tokens, but invisible to user)
 - ▶ then work through the solution step by step in this buffer
 - ▶ and finally generate the user-visible solution into an output token buffer
- Can use SFT to provide examples of how to reason, then RLVR to make reasoning more accurate
- Side effect: enables ***test-time scaling***
 - ▶ we can afford more iterations of RL (and promote efficient reasoning) if we limit size of reasoning buffer in training
 - ▶ at test time we can spend more tokens on the reasoning buffer to get better results (up to some ceiling)

Language model as a classifier

- LLMs provide a highly informative prior for supervised learning tasks: trillions of tokens of human knowledge
- Suppose we can afford just 10 labeled examples in a new classification task; can we harness the LLM's prior to generalize well?

Representing examples for the LLM

- To take advantage of LLM's knowledge, need to transform (x, y) pairs into a format it can understand
- For inputs: write a short bit of code that turns features into natural language
 - ▶ e.g., `(red, 12)` → “the product is red and 12 cm tall”
- Optionally add a descriptive prompt:
 - ▶ “will it be a good gift for a kid?”
- For outputs:
 - ▶ pick appropriate tokens to represent the classes: e.g., “yes” and “no” for a binary classifier
 - ▶ to read out the answer, prohibit all other tokens: just test whether $P(\text{yes} \mid \text{input}) > P(\text{no} \mid \text{input})$

Fine-tuning

- First approach: fine-tuning
- Just like the fine-tuning examples above, continue training the pre-trained model on our new (tiny) dataset
 - ▶ use supervised loss (e.g., cross-entropy)
 - ▶ seems crazy that it can work with so few examples!
 - ▶ typically only need small learning rate and few epochs
- This variant is ***pattern-based*** fine-tuning
 - ▶ for small datasets, works better than other FT variants such as adding a new prediction head to the network

In-context learning

- Second approach: in-context learning
- Just include training examples in prompt before asking about test example:

red, 12 cm tall? Yes

blue, 15 cm tall? No

. . .

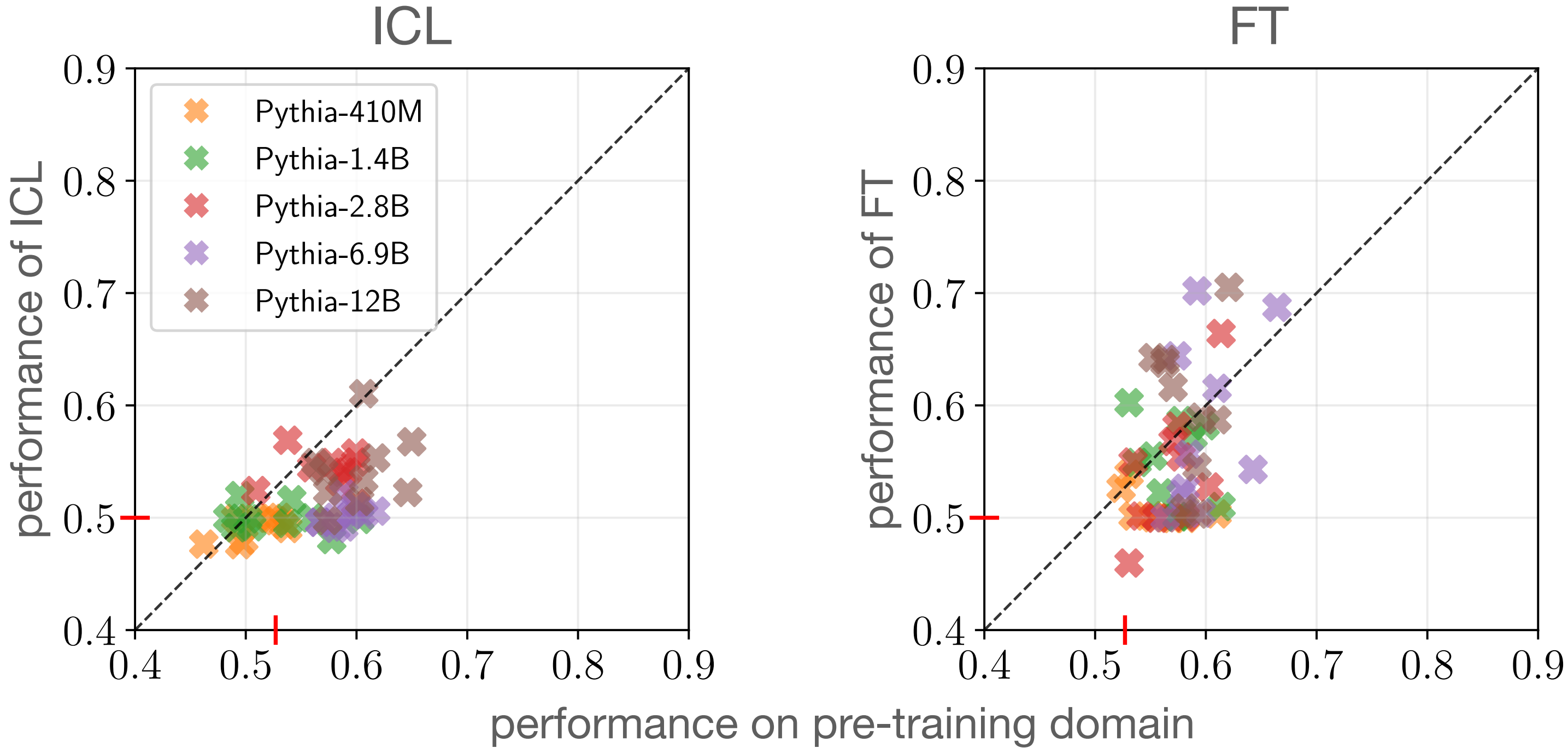
orange, 9 cm tall?

} training
examples
test input

- As above, read answer as $P(\text{yes} | \text{input}) > P(\text{no} | \text{input})$
- Huge advantage: no training needed!
 - ▶ fast, convenient
 - ▶ no need to know weights (e.g., API-only access)

Comparison

Graphs: 16 training examples, textual entailment task, models of multiple sizes with the same data (the Pile) and training pipeline



each X is one seed (10 seeds per setup)

- ICL is fast and convenient, and can perform well for small training set sizes
- FT tends to win as we get more training data
- Both do better when base model is better — crossover point depends on base model
 - ▶ the better the model, the longer ICL remains competitive