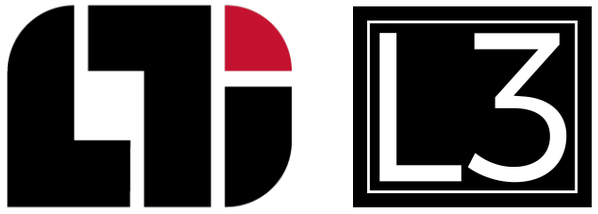
CS11-711 Advanced NLP

Reinforcement Learning

Sean Welleck



https://cmu-l3.github.io/anlp-spring2026/

https://github.com/cmu-l3/anlp-spring2026-code

References: John Schulman's 2016 tutorial, David Silver's 2015 Lectures on RL

Recap: supervised fine-tuning

Example:
(Instruction + input, output)

Recap: maximum likelihood

- Given dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$

- Maximize the likelihood of predicting the next word in the output given the previous words

$$\mathscr{L}(y_{1:T} \mid x) = -\sum_{t} \log p_\theta(y_t \mid y_{<t}, x)$$

- Intuitively, learn to “imitate” behaviors in the provided dataset (and in doing so, generalize to new inputs)

Problem 1: task mismatch

- We typically want a model to perform well at ***tasks***

Language model		**Task criterion**
$p(\text{probable response} \mid \text{prompt})$	$\approx$	Helpful response Non-offensive response
$p(\text{probable solution} \mid \text{problem})$	$\approx$	Correct solution Code that passes test cases

Problem 2: data mismatch

- Data often contains outputs we don’t want
  - Toxic / offensive comments from Reddit
  - Buggy code
- We don’t have much task-specific data
  - Chains of thought while solving problems
  - Helpful responses to all prompts

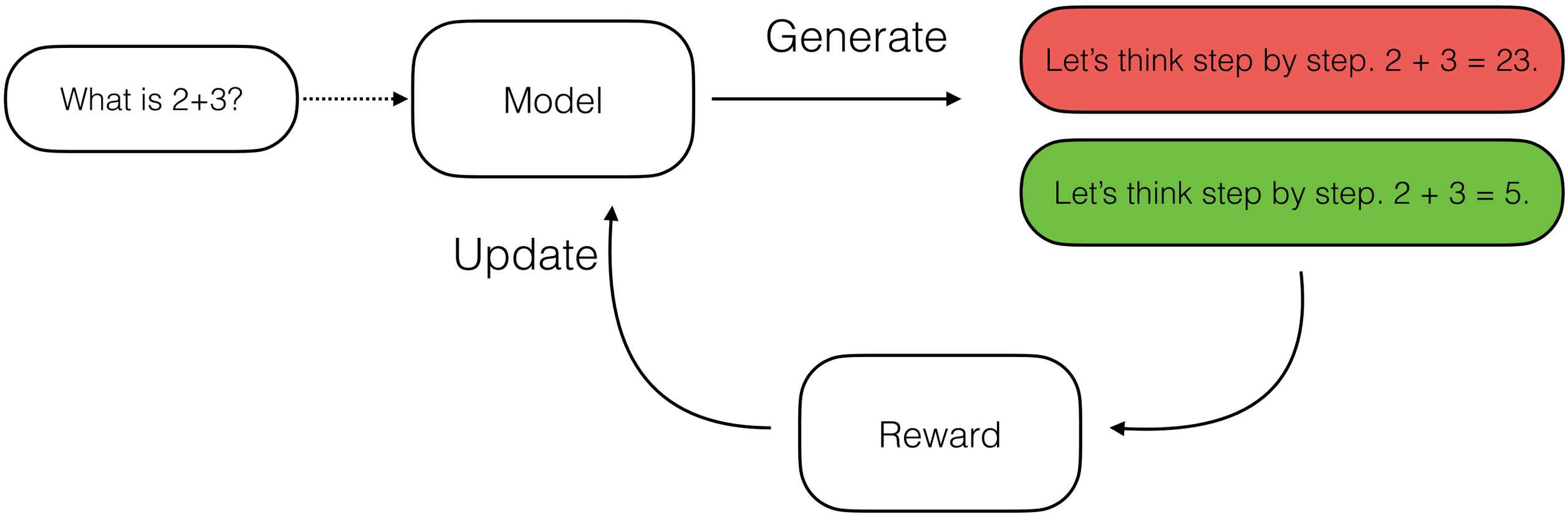
Problem 3: exposure bias

- The model is not exposed to mistakes during training, and cannot deal with them at test-time
  - E.g., make a mistake while solving a problem
  - E.g., click the wrong page while buying something online

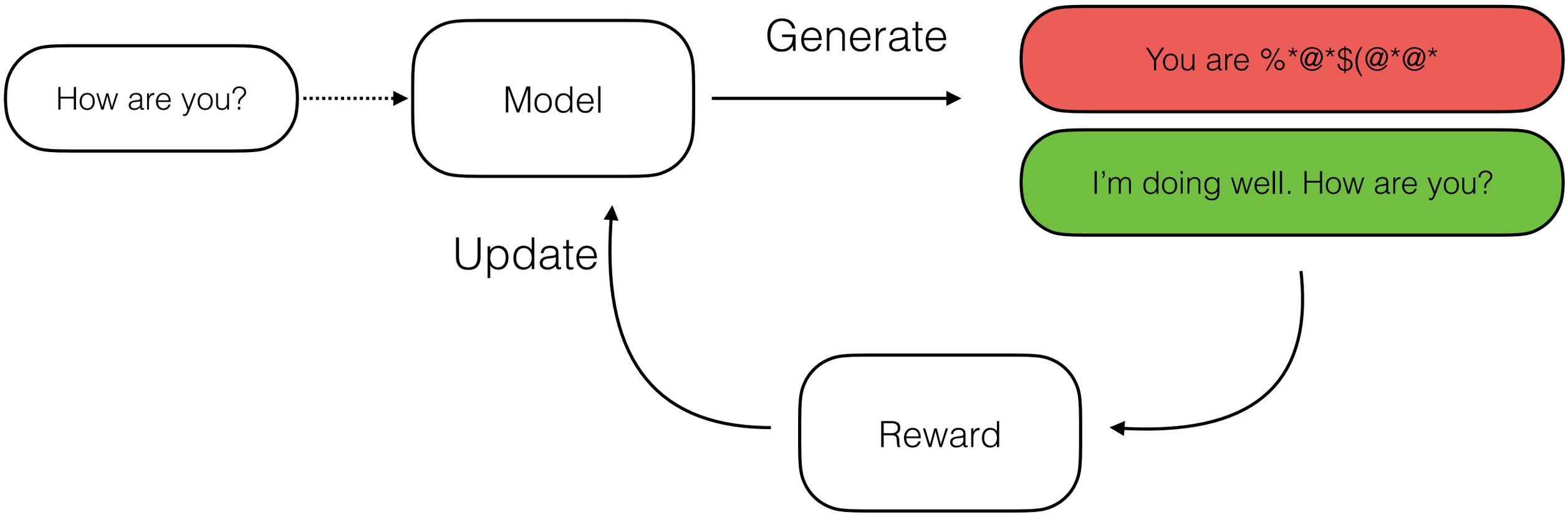
Problem 3: exposure bias

- The model is not exposed to mistakes during training, and cannot deal with them at test-time
  - E.g., make a mistake while solving a problem
  - E.g., click the wrong page while buying something online

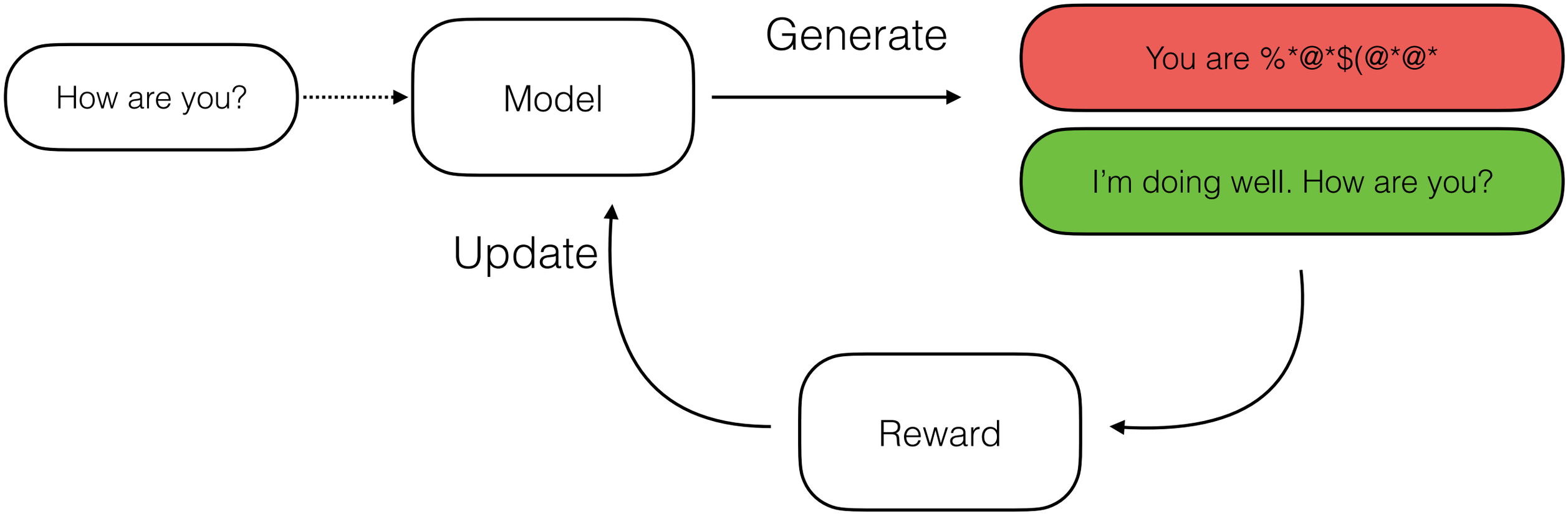
Today: reinforcement learning



Today: reinforcement learning



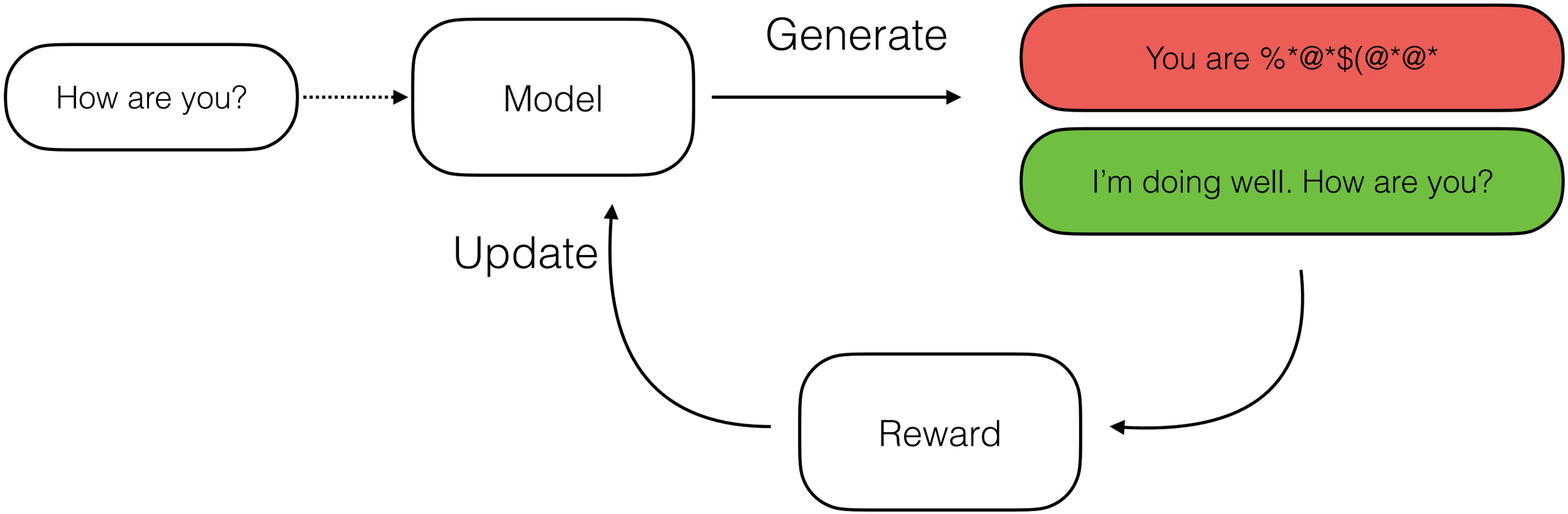
Today: reinforcement learning



Key difference 1:

- The task criteria is now *directly optimized* via the reward

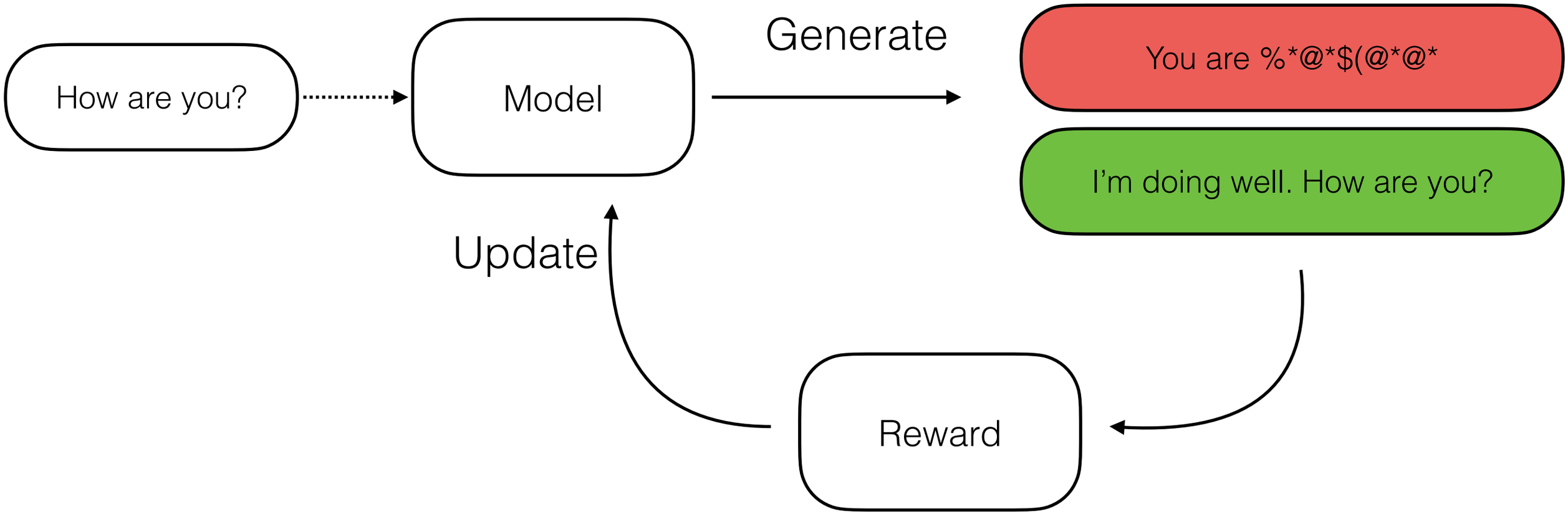
Today: reinforcement learning



Key difference 2:

- Data is generated by the model, and a reward tells us how to use the data for training

Today: reinforcement learning



Key difference 3:

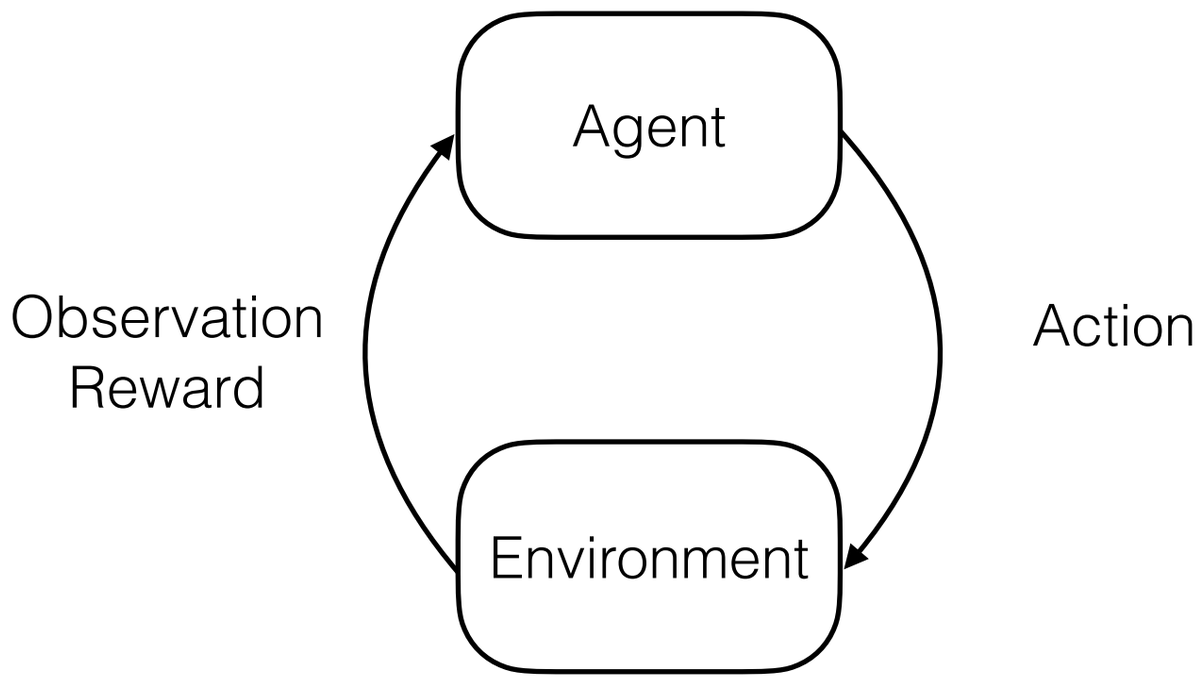
- Models can learn from any trajectory, so long as we can evaluate the trajectory's reward.

Today's lecture

- What is reinforcement learning?
- Policy gradient methods
- Next lecture:
  - Applications in NLP/LLMs
- Next next lecture:
  - Agents

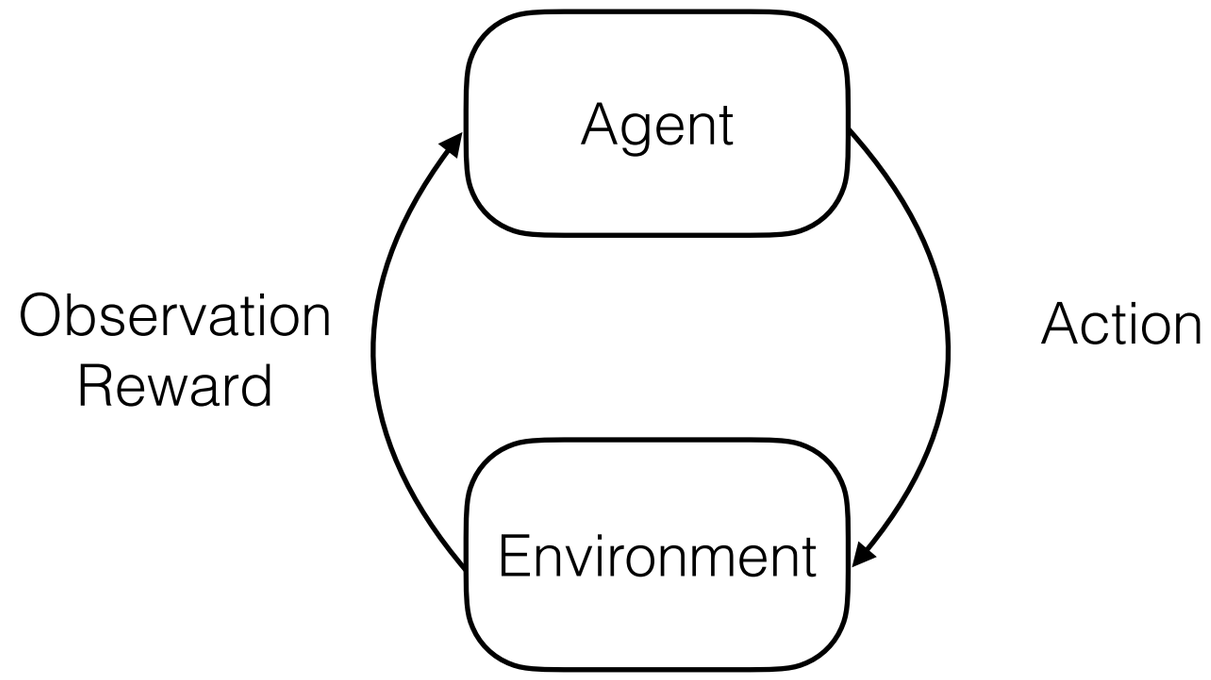
What is reinforcement learning (RL)?

- A branch of machine learning concerned with learning from interaction.
    - An *agent* interacts with an *environment.*
        - The agent learns what to do in order to maximize a reward signal.
- RL: A problem, a class of solutions, and a field that studies the problem/ solutions.

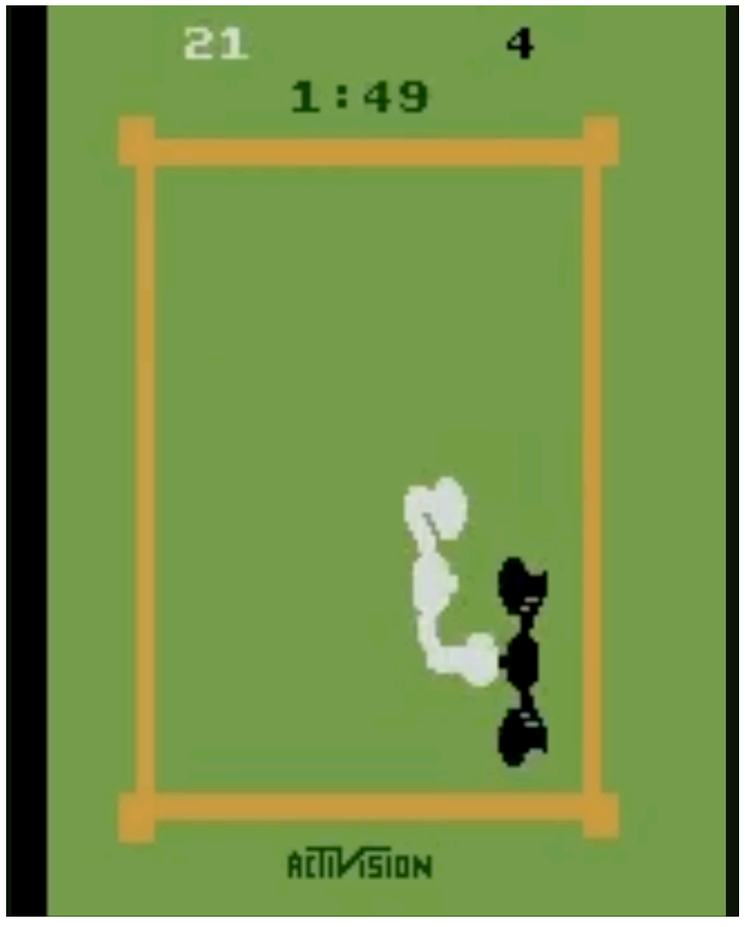


What is reinforcement learning (RL)?

- At each step the agent:
  - Receives observation
  - Receives scalar reward
  - Executes action
- The environment:
  - Receives action
  - Emits observation
  - Emits reward
- Agent tries to maximize reward

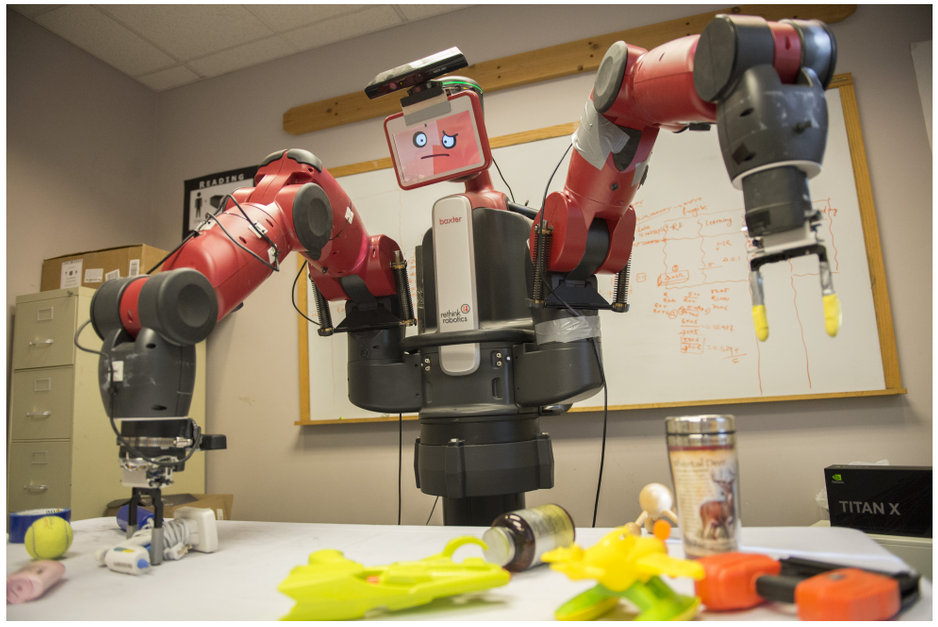


Example: playing Atari games



- Agent: plays the game
- Observation: screen
- Actions: press buttons
- Rewards: score at the end of a game

Example: motor control and robotics



- Agent: the robot
- Observations: camera images, joint angles
- Actions: joint torques
- Rewards: stay balanced, move something to a target location, …

Example: text generation

Question

Find the largest possible real part of $[(75+117i)z+\frac{96+144i}{z}]$ where z is a complex number with $|z|=4$. Let's think step by step and output the final answer within $boxed\{\}$. Think for 3600 tokens.

Language Model

Model Response

Thinking Process (not shown to user)

Okay, so I have this problem: I need to find the largest possible real part of the expression (75 + 117i)z + (96 + 144i)/z, where z is a complex number with $|z| = 4$. Hmm, that seems a bit involved, but let's break it down step by step. First, let me understand what is given. We have a complex number z with magnitude 4, so z can be written in the form $z = 4e^{i\theta}$, where θ is the argument of z. That might be useful because it allows us to express z in terms of its angle, which could simplify the expression.

The expression we need to consider is $(75+117i)z+\frac{96+144i}{z}$. We are asked to find the largest possible real part of this expression. So, let's denote the expression as $E = (75+117i)z+\frac{96+144i}{z}$.

[Reasoning process truncated for brevity ...]

Then, adding $104,976 + 186,624 = 291,600$. $\sqrt{291,600} = \sqrt{2916 * 100} = \sqrt{2916} * 10$. Now, $\sqrt{2916}$: let's see, $54^2 = 2916$, because $50^2 = 2500$, $54^2 = 2916$. So, $\sqrt{2916} = 54$, thus $\sqrt{2916 * 100} = 54 * 10 = 540$. So, that seems correct.

Therefore, the maximum value of $\mathrm{Re}(E)$ is 540.

Therefore, the largest possible real part is 540.

Given the expression $(75+117i)z+\frac{96+144i}{z}$ where z is a complex number with $|z| = 4$, we need to find the largest possible real part of this expression. First, we express z as $4e^{i\theta}$ since $|z| = 4$. Then, we substitute z and $\frac{1}{z}$ into the expression:

$$(75+117i)z+\frac{96+144i}{z}$$

[Output truncated for brevity ...]
Thus, the largest possible real part is:

$$\boxed{540}$$

Correct Answer

$\boxed{540}$

Correctness

✔ **Correct**

Slido: what are the agent, observations, actions, rewards?

- Rewards: is the final answer correct

Formalism: MDPs and Policies

- Formalize the RL setting as a Markov Decision Process (MDP) (S, A, E, R):
  - S: state space
    - E.g. image of a screen
  - A: action space
    - E.g. Atari controller buttons
  - $E(s_{t+1} \mid s_t, a_t)$: environment transitions
    - E.g. Move to next screen after pressing a button
  - $R(s_t, a_t) \to \mathbb{R}$: reward function
    - E.g. Return the game score at each time step
- The agent takes actions using a *policy* π:
  - $\pi(a_t \mid s_t)$

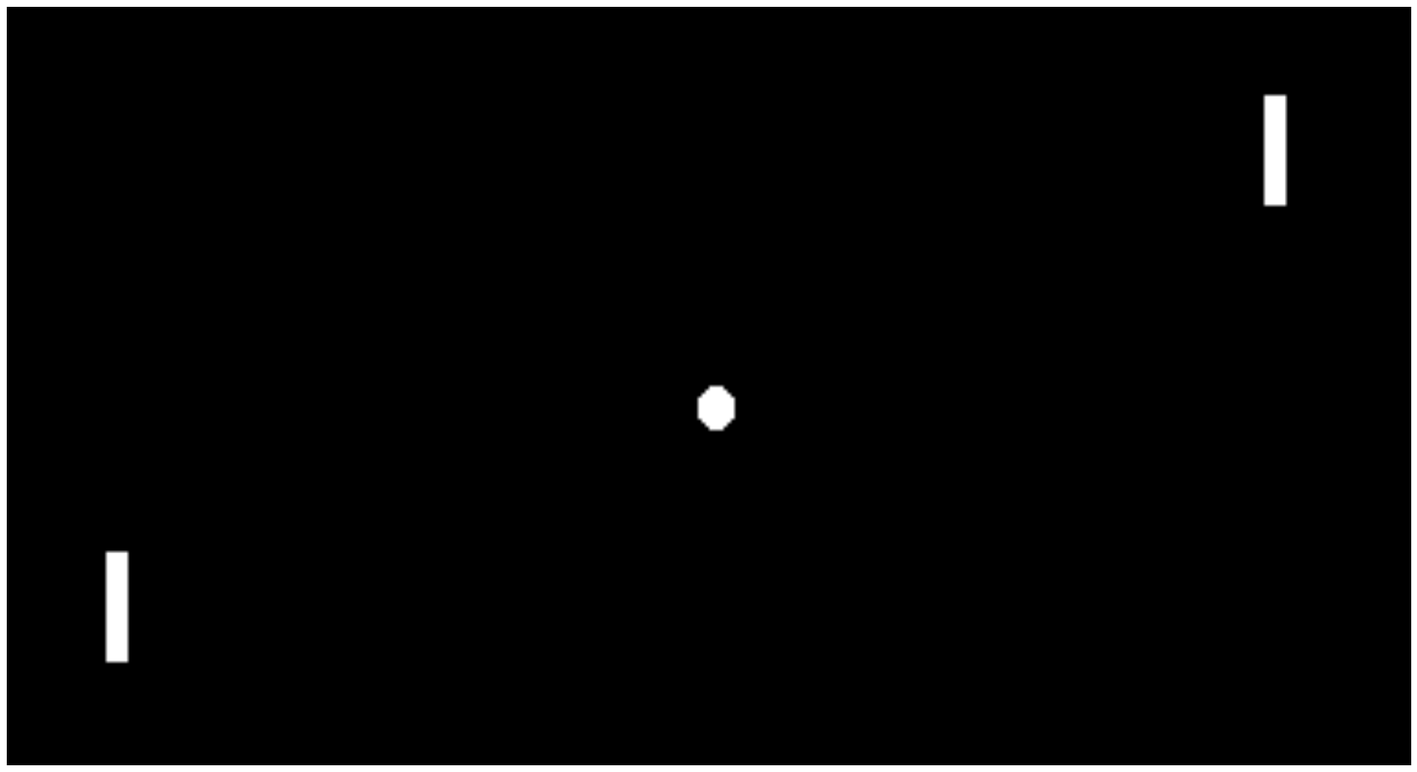
Formalism: MDPs and Policies

- The agent interacts with the environment until a terminal state is reached:
  - $s_0 \sim E$
  - $a_0 \sim \pi(a_0 \mid s_0)$
  - $s_1 \sim E(s_1 \mid s_0, a_0)$
    - $r_1 = R(s_0, a_0)$
  - $a_1 \sim \pi(a_1 \mid s_1)$
  - …
  - s_T, r_T

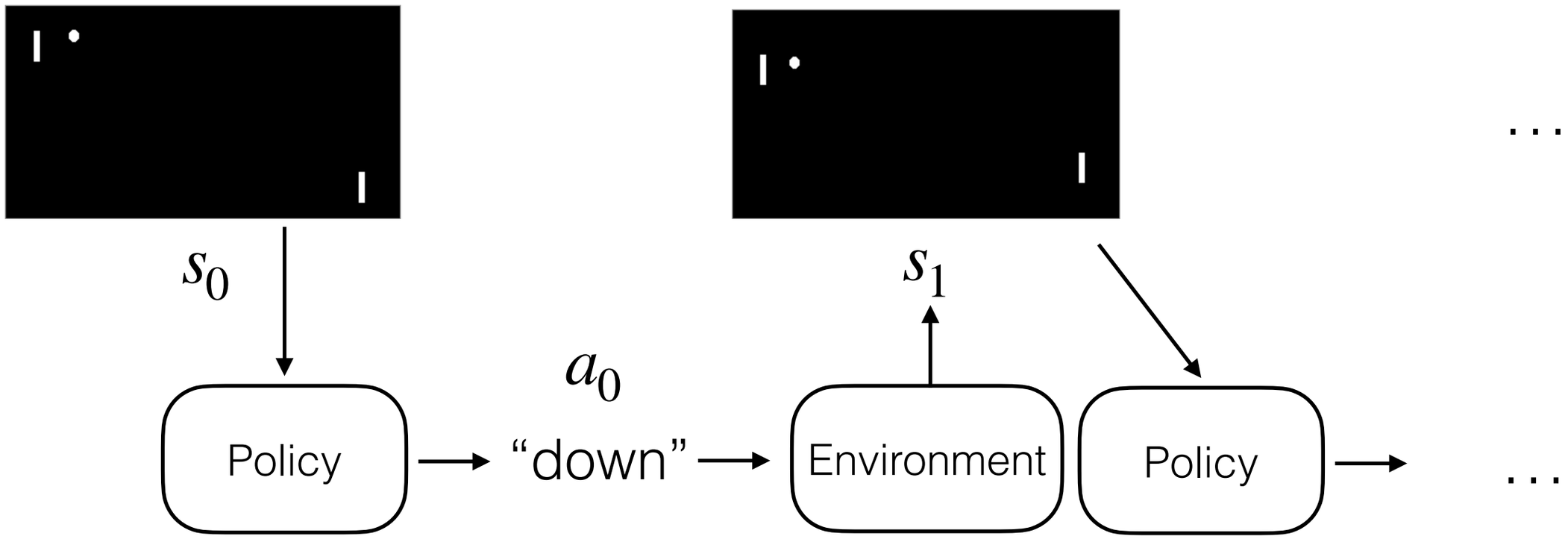
The sequence $\tau_{0:T} = (s_0, a_0, r_1, s_1, \ldots, r_T, s_T)$ is called an *episode* or a *trajectory*.

- Let d_{τ_π} be the distribution of trajectories obtained by using policy π to interact with the environment
- Goal: find a policy that maximizes expected return, $\mathbb{E}_{d_{\tau_\pi}}\left[\sum_{t=1}^{T} r_t\right]$
  - Return: sum of future rewards

Example: Pong



Example: Pong

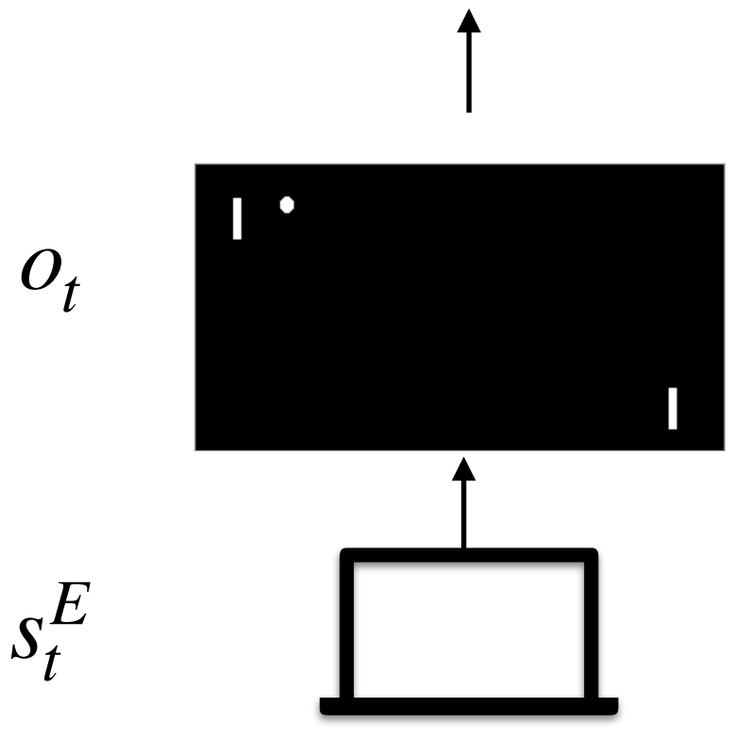


- Play out a trajectory, $(s_0, a_0), (s_1, a_1), \ldots, s_T$
- Reward:
  - 0 for $t < T$
  - $+1$ if s_T is "win", -1 if s_T is "lose"

States and Observations

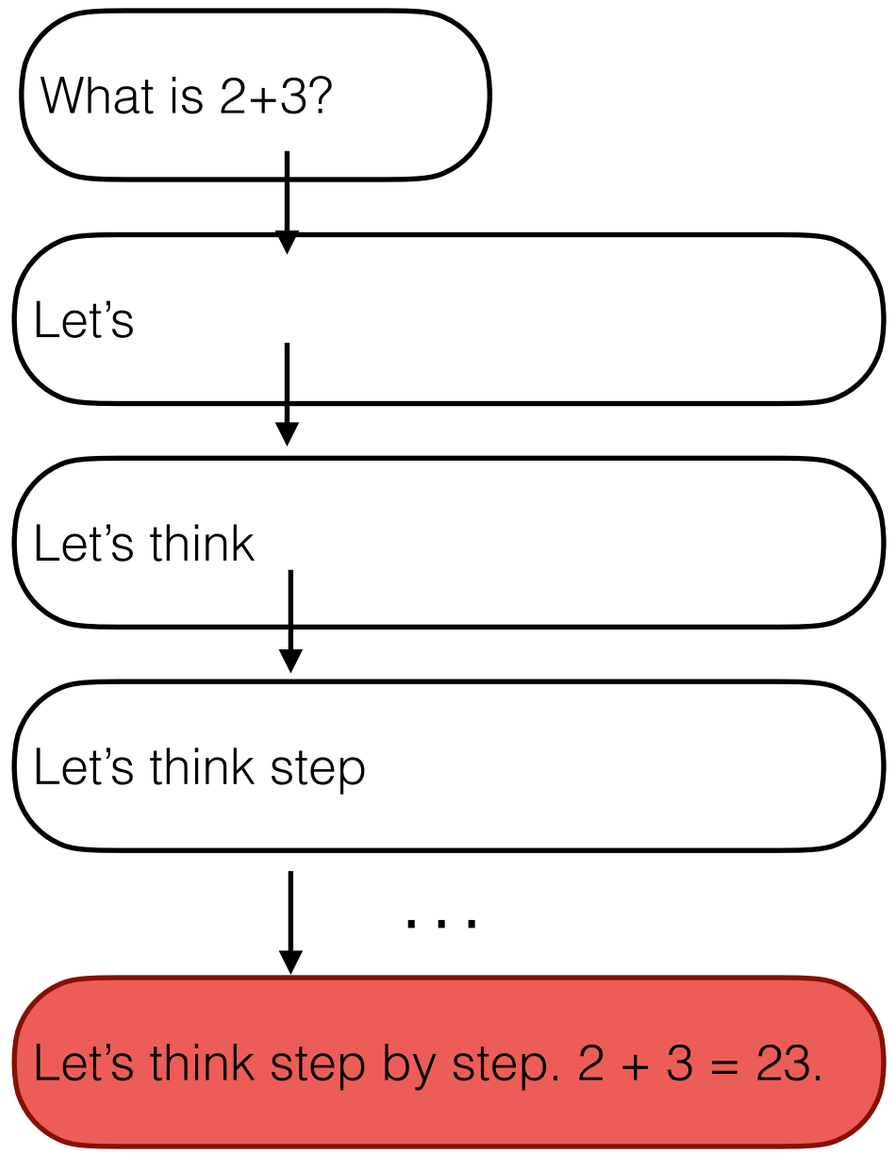
- Environment state s_t^E
  - The environment's private representation
- Observation o_t
  - The information visible to the agent.
- Agent state s_t
  - Whatever information the agent uses to pick the next action
  - Example: just use the current observation

$$s_t = f(o_0, a_0, r_1, \ldots, a_t, o_t)$$



Example 2: math problem solving

- **State**: a prompt and tokens-generated-so far
  - $s_t : (x, y_{<t})$
- **Action**: generate a token
  - $a_t : y_t$
- **Policy**: language model
  - $p_\theta(y_t \mid y_{<t}, x)$
- **Environment**: append token
  - $s_{t+1} : (x, y_{<t} \circ y_t)$
- **Reward**: is the final answer correct
  - $r(x, y_{1:t})$ = 1 if $t = T$ and the answer in $y_{1:T}$ is correct; 0 otherwise.



Types of RL algorithms

- The agent typically includes at least one of the following:
  - Policy: agent's behavior function, e.g.:

$$\pi(a_t \mid s_t)$$

  - Value function: how good is each state (and/or action), e.g.:

$$v_\pi(s_t) = \mathbb{E}\left[\sum_{t'=t+1}^{T} r_{t'}\right]$$

  - Model: agent's representation of the environment, e.g.:

$$e(r_{t+1}, s_{t+1} \mid s_t, a_t)$$

- Policy-based methods: learn a policy π_θ
- Value-based methods: involves learning a value function v_ϕ
- Model-based methods: involves learning a model e_ψ

We will focus on policy gradient methods. Some of them will learn a value function to stabilize training.

Summary: setup

- We have a *Markov decision process* (S, A, E, R)
- Goal: learn a policy that maximizes expected return

Today's lecture

- What is reinforcement learning?

- *Next:* Policy gradient methods

Policy gradient

- Learn a policy that maximizes expected reward

$$\arg\max_{\theta} \underbrace{\mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau)\right]}_{J(\theta)}$$

- Let's use gradient descent?
  - Compute $\nabla_\theta J(\theta)$, then use SGD, Adam, etc…
- More tricky than expected, for instance what is $\nabla_\theta R(\tau)$?
- Solution: *estimate* the gradient using sampled trajectories

Score function gradient estimator
("REINFORCE" [Williams 1992])

- For a distribution $x \sim p$ and a function $f(x)$:

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] = \nabla_\theta \sum_x p_\theta(x) f(x)$$

$$= \sum_x \frac{p_\theta(x)}{p_\theta(x)} \nabla_\theta p_\theta(x) f(x)$$

$$= \sum_x p_\theta(x) \nabla_\theta \log p_\theta(x) f(x)$$

$$= \mathbb{E}_{x\sim p_\theta(x)} \nabla_\theta \log p_\theta(x) f(x)$$

$$\approx \nabla_\theta \log p_\theta(\hat{x}) f(\hat{x}) \text{ where } \hat{x} \sim p_\theta$$

$$\nabla_\theta \log p_\theta(x) = \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}$$

- In summary, $\hat{g}_{\text{REINFORCE}} = f(\hat{x}) \nabla_\theta \log p_\theta(\hat{x})$

Policy gradient/REINFORCE

- Let's apply it to RL! We get:

$$\nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \approx \sum_{t=1}^{T} R_t \nabla_\theta \log \pi_\theta(\hat{a}_t \mid \hat{s}_t)$$

where $R_t = \sum_{t'=t}^{T} r_t$ is the return and $\hat{a}_1, \hat{s}_1, \hat{r}_1, \ldots$ is a trajectory sampled with π_θ

- When R_t is high, push up the probability $\pi_\theta(a_t \mid s_t)$
- When R_t is low, push down the probability $\pi_\theta(a_t \mid s_t)$

Note: this skips
a derivation; see Appendix slides

Putting it all together (Vanilla Policy Gradient / REINFORCE)

For iteration 1, 2, … do:

1. **Collect trajectories**
   - $(s_0, a_0, r_1), (s_1, a_1, r_2), \ldots, (s_{T-1}, a_{T-1}, r_T)$
2. **Compute returns**
   - $R_t = \sum_{t'=t+1}^{T} r_{t'}$
3. **Policy gradient loss**
   - $\mathscr{L}_{PG} = -\sum_t R_t \log \pi_\theta(a_t \mid s_t)$

The gradient of this loss equals the policy gradient

$$\nabla_\theta \mathscr{L}_{PG} = -\sum_t R_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

4. **Update the policy** with back-propagation and SGD/Adam

Example (CartPole)


```
Initial state: [-0.04058227  0.04756223  0.02611397  0.02860643]

Step 1:
  Action: 0 (LEFT)
  Reward: 1.0
  Next state: [-0.03963102 -0.14792429  0.0266861   0.32941288]
  Done: False

Step 2:
  Action: 0 (LEFT)
  Reward: 1.0
  Next state: [-0.04258951 -0.34341577  0.03327436  0.63039047]
  Done: False

Step 3:
  Action: 1 (RIGHT)
  Reward: 1.0
  Next state: [-0.04945782 -0.14877352  0.04588217  0.34836957]
  Done: False
```

```python
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.dropout = nn.Dropout(p=0.6)
        self.affine2 = nn.Linear(128, 2)

    def forward(self, x):
        x = self.affine1(x)
        x = self.dropout(x)
        x = F.relu(x)
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=1)
```

https://github.com/cmu-l3/anlp-spring2026-code/tree/main/16_rl

Example (CartPole)

```python
for i_episode in range(num_episodes):
    state, _ = env.reset()
    ep_reward = 0

    # 1. Collect an episode trajectory
    log_probs, rewards = [], []
    for t in range(1, max_steps_per_episode):
        state = torch.tensor(state).unsqueeze(0)
        probs = policy(state)
        p = Categorical(probs)
        action = p.sample()
        log_prob = p.log_prob(action)

        state, reward, terminated, truncated, _ = env.step(action.item())

        log_probs.append(log_prob)
        rewards.append(reward)
        ep_reward += reward
        if terminated or truncated:
            break
```

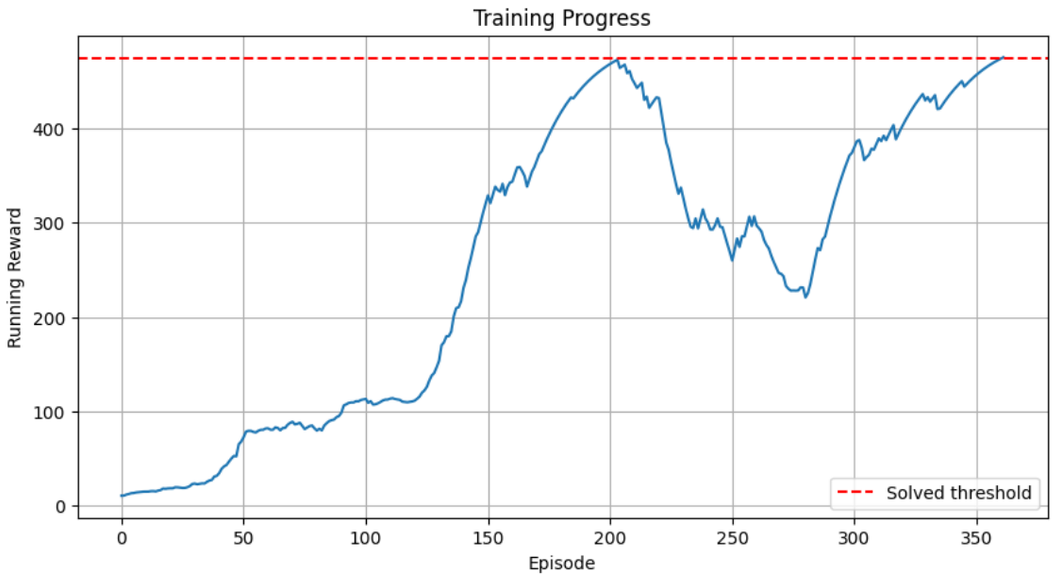
```python
    # 2. Compute returns
    returns = deque()
    R = 0
    for r in rewards[::-1]:
        R = r + discount_gamma * R
        returns.appendleft(R)
    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + EPS)

    # 3. Compute loss
    log_probs = torch.cat(log_probs)
    policy_loss = -(log_probs*returns).sum()

    # 4. Update policy
    optimizer.zero_grad()
    policy_loss.backward()
    optimizer.step()
```

https://github.com/cmu-l3/anlp-spring2026-code/tree/main/16_rl

Example (CartPole)



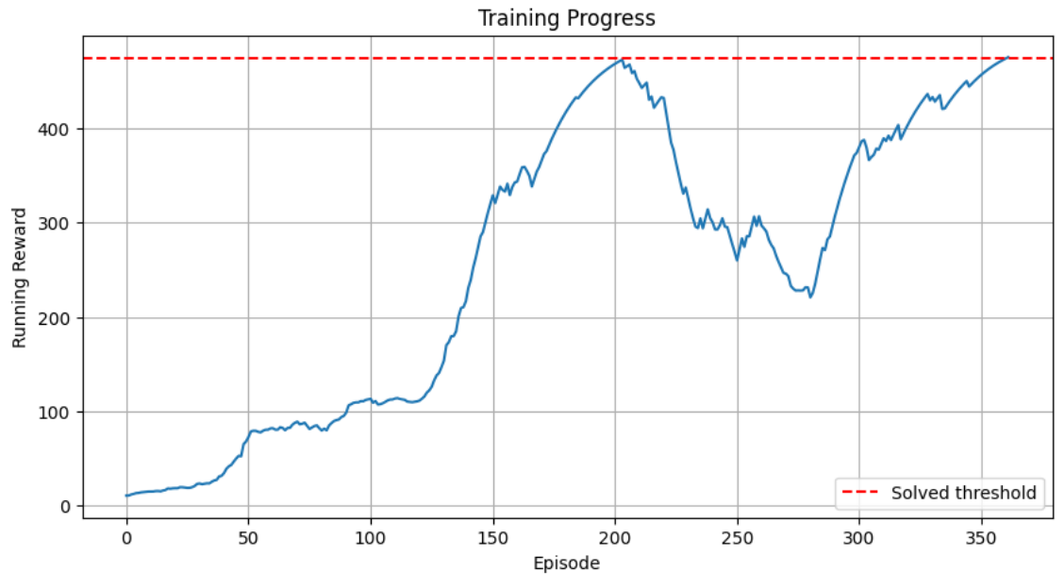
https://github.com/cmu-l3/anlp-spring2026-code/tree/main/16_rl

Today's lecture

- What is reinforcement learning?
- Policy gradient methods
- *Next*: Stabilizing learning

Stabilizing learning

- "Vanilla" policy gradient:
  - $\hat{g}_t = R_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
- Can we do better?
  - Re-weight the gradients
  - Control step size



Discounting

- There are delays between taking actions and receiving reward
  - *Discounted* return: sum of future rewards, discounted by how far it is in the future

$$R_t = \sum_{t'=t+1}^{T} \gamma^{t'-t-1} r_{t'}$$

- Example: suppose we only receive +1 at the end. We discount using a factor of $\gamma = 0.9$:

a_1	a_2	a_3	a_4	a_5	a_6	
0.59	0.66	0.73	0.81	0.9	1	+1

Strategy 1: weight the gradients better

Baselines

- Estimate of the expected reward for a given state.

	Reward	Baseline	B - R
"Summarize this paper: …"	0.8	0.75	0.05
"Summarize this paper: …"	0.3	0.75	-0.45
"Prove this theorem: …"	0.3	0.10	0.20

- Subtracted from the actual reward to determine how good a particular action was relative to what was expected

Strategy 1: weight the gradients better

Baselines

$$\hat{g}_t = (R_t - b(s_t)) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

- A good choice is the expected return:
  - $b(s_t) \approx \mathbb{E}\left[\sum_{t'=t+1}^{T} \gamma^{f(t')} r_{t'}\right]$

Quiz: we saw this function earlier; what is it called?

- Intuition: increase log-prob of action based on *how much better the action is than expected*

Strategy 1: weight the gradients better

Advantages

$$\hat{g}_t = A_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

- Baseline
  - $\hat{A}_t = (R_t - v(s_t))$
- Temporal difference (TD) residual [Sutton & Barto 1998]
  - $\hat{A}_t = \delta_t = r_t + \gamma v(s_{t+1}) - v(s_t)$
- Generalized advantage estimation [Schulman et al 2015]
  - $\hat{A}_t = \sum_l (\gamma\alpha)^l \delta^v_{t+l}$
    - Baselines and TD residual are special cases
- Other techniques covered in the next lecture

Strategy 2: control the step-size better

Large updates

- Updates are noisy, so a large update can derail things
  - Mitigation: don't move the policy too much at once
- Example: Proximal policy optimization (PPO) [Schulman et al 2017]
  - $\text{ratio}(x, y) = \dfrac{p_\theta(y \mid x)}{p_{\theta_{old}}(y \mid x)}$

$$L_{PPO} = \min\left(\text{ratio}(x, y)A(x, y),\ \text{clip}(\text{ratio}(x, y), 1-\epsilon, 1+\epsilon)A(x, y)\right)$$

Putting it all together

For iteration 1, 2, … do:

1. **Collect trajectories**
   - $(s_0, a_0, r_1), (s_1, a_1, r_2), \ldots, (s_{T-1}, a_{T-1}, r_T)$
2. **Compute advantages**
   - May involve a baseline, a value function $v(s_t)$, discounting, etc.
3. **Compute loss**
   - Policy gradient loss, PPO loss
4. **Update the policy** with back-propagation and SGD/Adam

Code example

Baseline: a learned value function $v_\phi(s_t)$

```python
class VF(nn.Module):
    def __init__(self):
        super(VF, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.affine2 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.affine1(x)
        x = F.relu(x)
        v = self.affine2(x)
        return v
```

```python
# 1. Collect an episode trajectory
log_probs, rewards, values = [], [], []
for t in range(1, max_steps_per_episode):
    state = torch.tensor(state).unsqueeze(0)
    probs = policy(state)

    v = vf(state)
```

```python
# 2. Compute returns
returns = deque()
R = 0
for r in rewards[::-1]:
    R = r + discount_gamma * R
    returns.appendleft(R)

returns = torch.tensor(returns)
adv = torch.tensor(returns)
for i, v in enumerate(values):
    adv[i] -= v.item()

# 3. Compute loss
log_probs = torch.cat(log_probs)
policy_loss = -(log_probs*adv).sum()
values = torch.cat(values)
vf_loss = F.mse_loss(values, returns.unsqueeze(1))
```

https://github.com/cmu-l3/anlp-spring2026-code/tree/main/16_rl

Next lecture: RL from human feedback (RLHF)

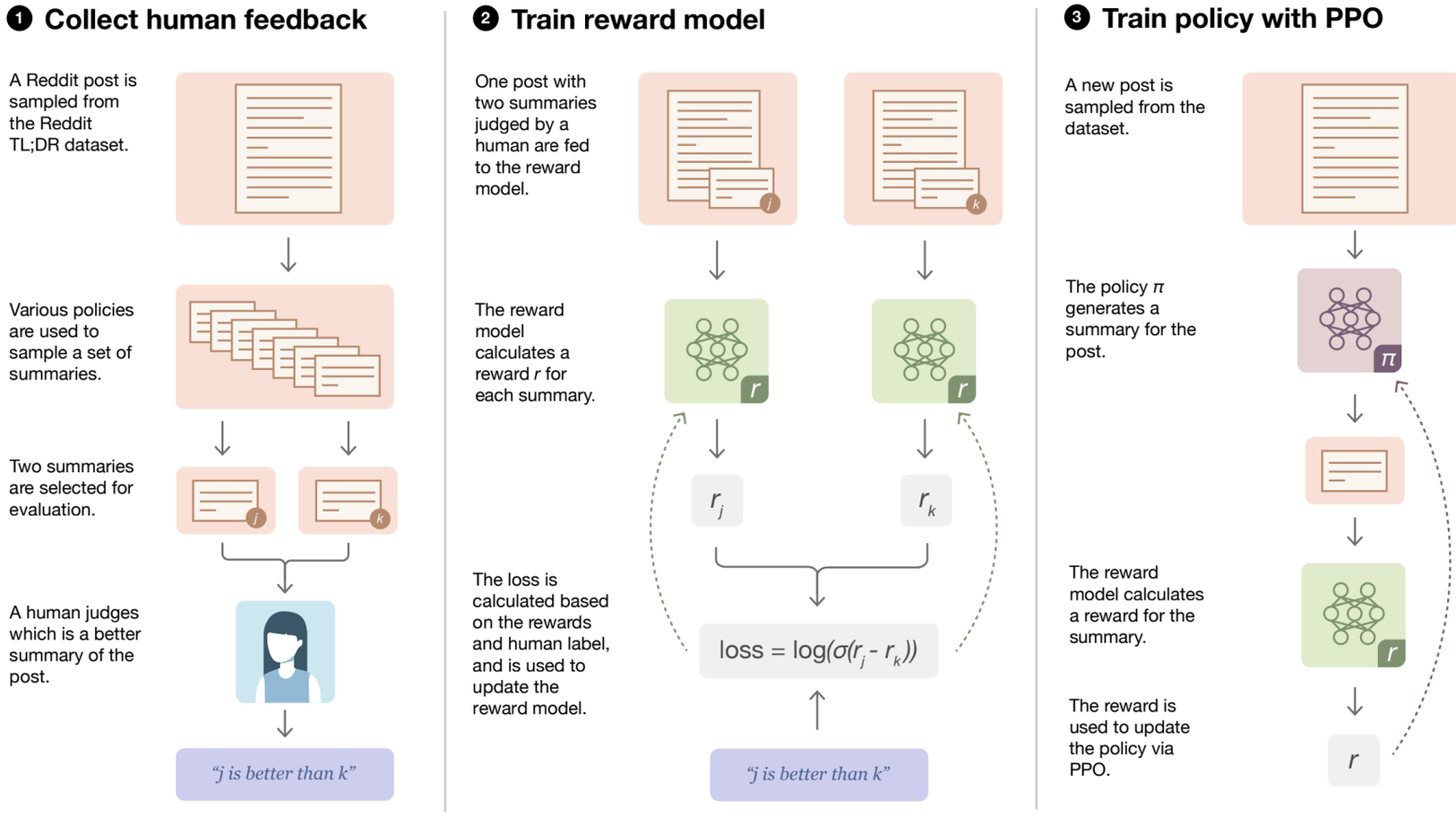
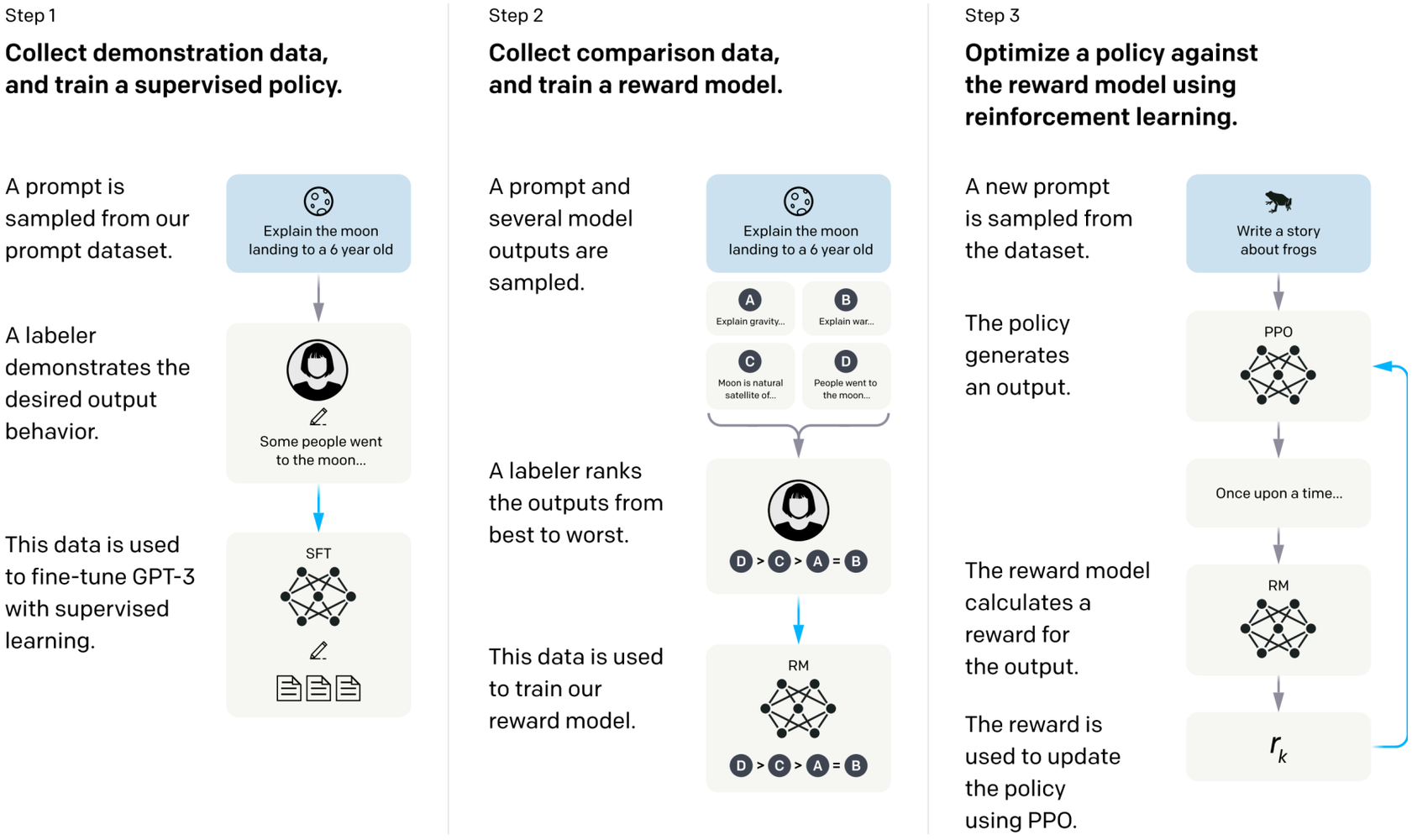


Figure 2: Diagram of our human feedback, reward model training, and policy training procedure.

- Policy: given prompt x, generate response $y_{1:T}$
- Basic MDP, preference reward, PPO

Ziegler et al 2019, Stiennon et al 2020

Next lecture: RL from human feedback (RLHF)

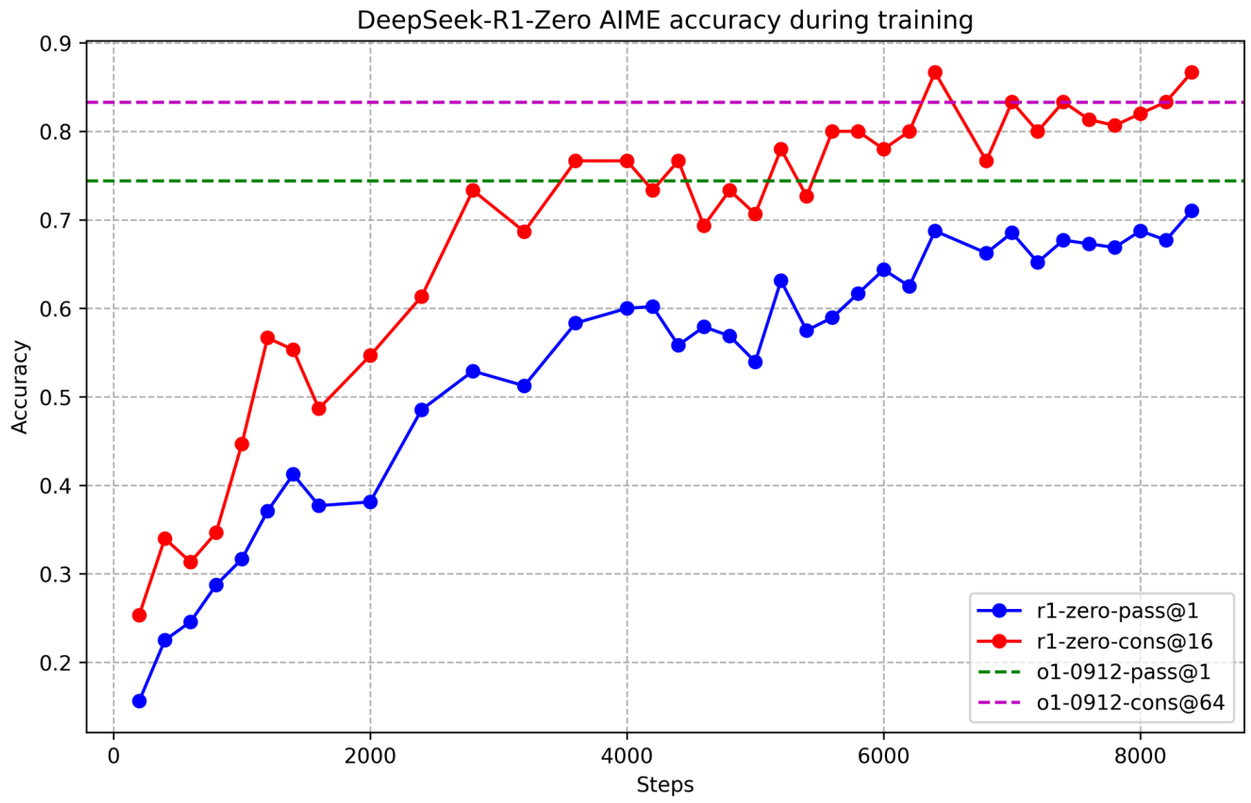


- Policy: given prompt x, generate response $y_{1:T}$
- Basic MDP, preference reward, PPO

Ouyang et al 2022

Next lecture: RL with verifiable rewards (RLVR)

DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning



- Policy: given problem x, generate chain of thought + answer
- 1-step MDP, 0/1 rule-based reward, PPO with output-average baseline ("GRPO")

Thank you

Policy gradient derivation: I

Recall that the score function estimator is:

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] = \mathbb{E}_{x\sim p_\theta(x)} \nabla_\theta \log p_\theta(x) f(x)$$

- Apply it to the distribution $p(\tau \mid \theta)$, defined on the next slide, and the function $R(\tau) = \sum_{t=1}^{T} r_t$:
- $\nabla_\theta \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau)] = \mathbb{E}_\tau \left[\nabla_\theta \log p(\tau \mid \theta) R(\tau) \right]$

From: http://joschu.net/docs/2016-bayareadlschool.pdf

Policy gradient derivation: II

$$\nabla_\theta \mathbb{E}_{\tau \sim p(\tau|\theta)}[R(\tau)] = \mathbb{E}_\tau \left[\nabla_\theta \log p(\tau \,|\, \theta) R(\tau) \right] \; (*)$$

- $$p(\tau \,|\, \theta) = E(s_0) \prod_{t=1}^{T} [\pi_\theta(a_t \,|\, s_t) E(s_{t+1} \,|\, s_t, a_t)]$$
- $$\nabla \log p(\tau \,|\, \theta) = \nabla_\theta \left[\sum_{t=1}^{T} \log \pi_\theta(a_t \,|\, s_t) \right]$$
  - Take the log, the E terms are 0 after taking the gradient
- Plugging this into (*) and moving the gradient inside the sum:
- $$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau \left[R(\tau) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) \right]$$

From: http://joschu.net/docs/2016-bayareadlschool.pdf

Policy gradient derivation: III

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau\left[R(\tau)\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]$$

- Now we want to write this using $R_t = \sum_{t'=t}^{T} r_t$, i.e. per-step returns
- Use a similar argument to get a term for each step:
  - $\nabla_\theta \mathbb{E}[r_t] = \mathbb{E}[r_t \sum_{t'=1}^{t} \log \pi_\theta(a_{t'} \mid s_{t'})]$
- Sum over t and rearrange:
  - $\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) R_t$

From: http://joschu.net/docs/2016-bayareadlschool.pdf