Policy gradient methods

Slides credits

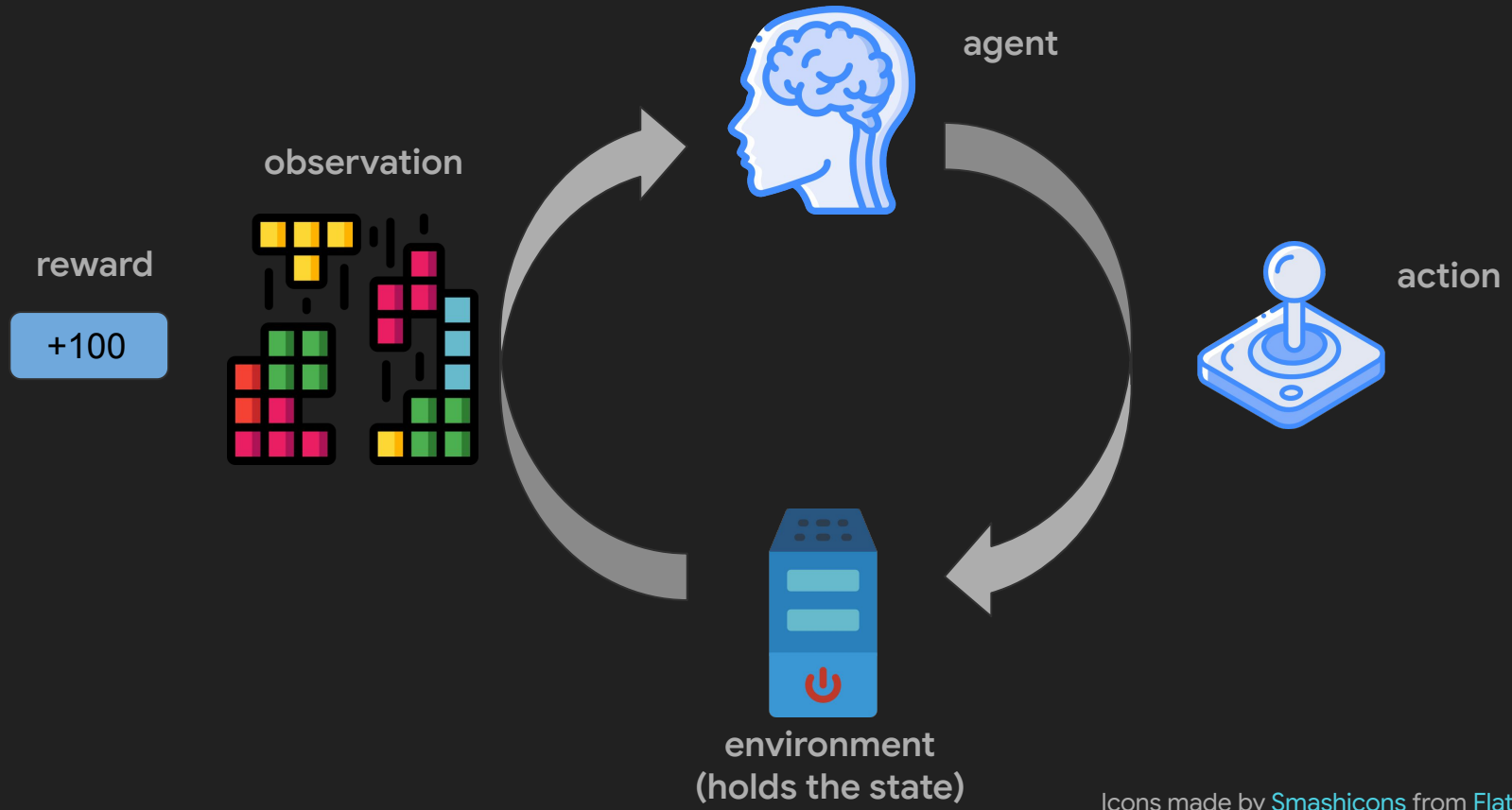
This presentation is based on slides created by Prof. Marek Cygan.

Some slides are also taken from:

- Berkeley Deep Learning course
- Deep RL Bootcamp

Recap

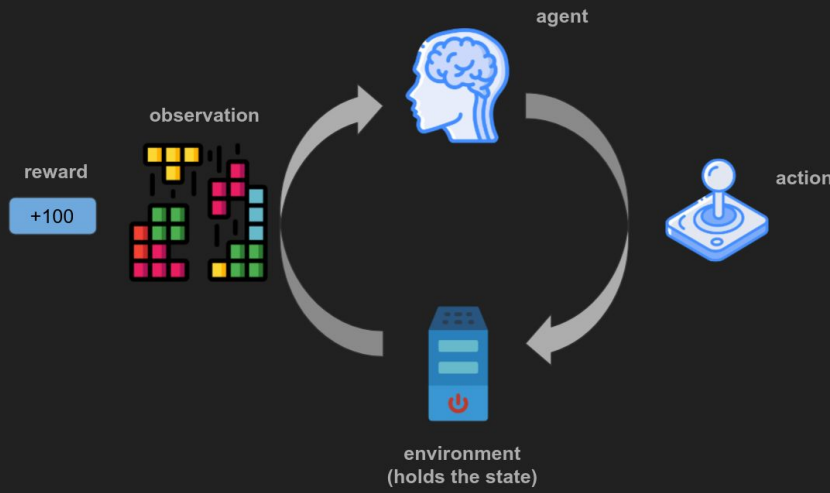
Reinforcement learning



Icons made by Smashicons from Flaticon

What is the problem?

- Data is non-i.i.d.
  (current actions affect future observations)
- Sequence of actions
  (delayed feedback, credit assignment).
- Gradient of an arbitrary objective function
  (no supervision, only reward signal)
- Environment is stochastic.



Value function

For a state s define:

- $V^*(s)$- sum of rewards when starting from state s and acting optimally (how many points can I score from now till the end of the game)
- $V_k^*(s)$- as above but for playing k turns

Current score does not count towards V!

Q-values (action values)

- Q*(s,a) - expected discounted reward, when starting in s, making action a and then playing optimally
- If we have Q-values, we can play the game

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

Approximation by neural networks

Icons made by Smashicons from Flaticon

Basic Q-learning algorithm

Initialize $Q(s, a)$ for all s, a.

Get initial state s

Repeat until convergence

  Sample action a, get next state s'

  If s' is terminal:

    target $= R(s, a, s')$

    Sample new initial state s'

  else:

    target $= R(s, a, s') + \gamma \max_{a'} Q(s', a')$

  $Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha \cdot$ target

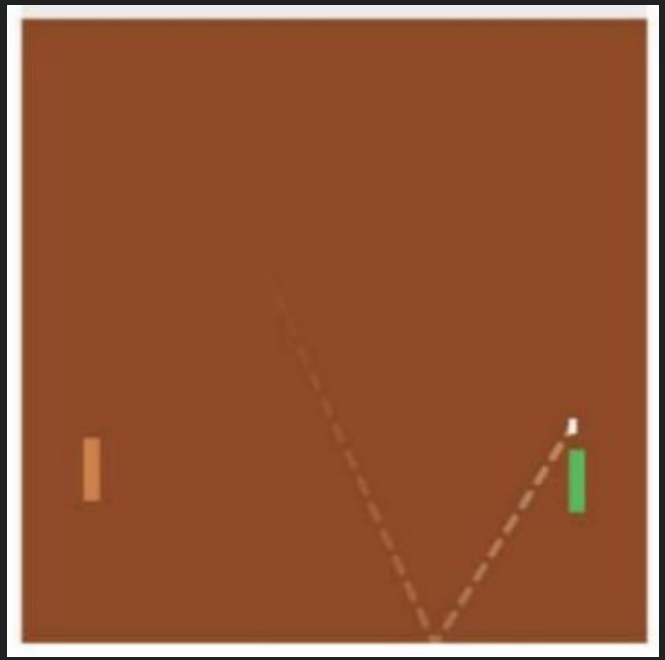
  $s \leftarrow s'$

Policy oriented algorithms

Which question is easier to answer:

- What is the expected value in this state of pong?
- Should I move up or down?

Policy has often a simpler structure than Q or V.



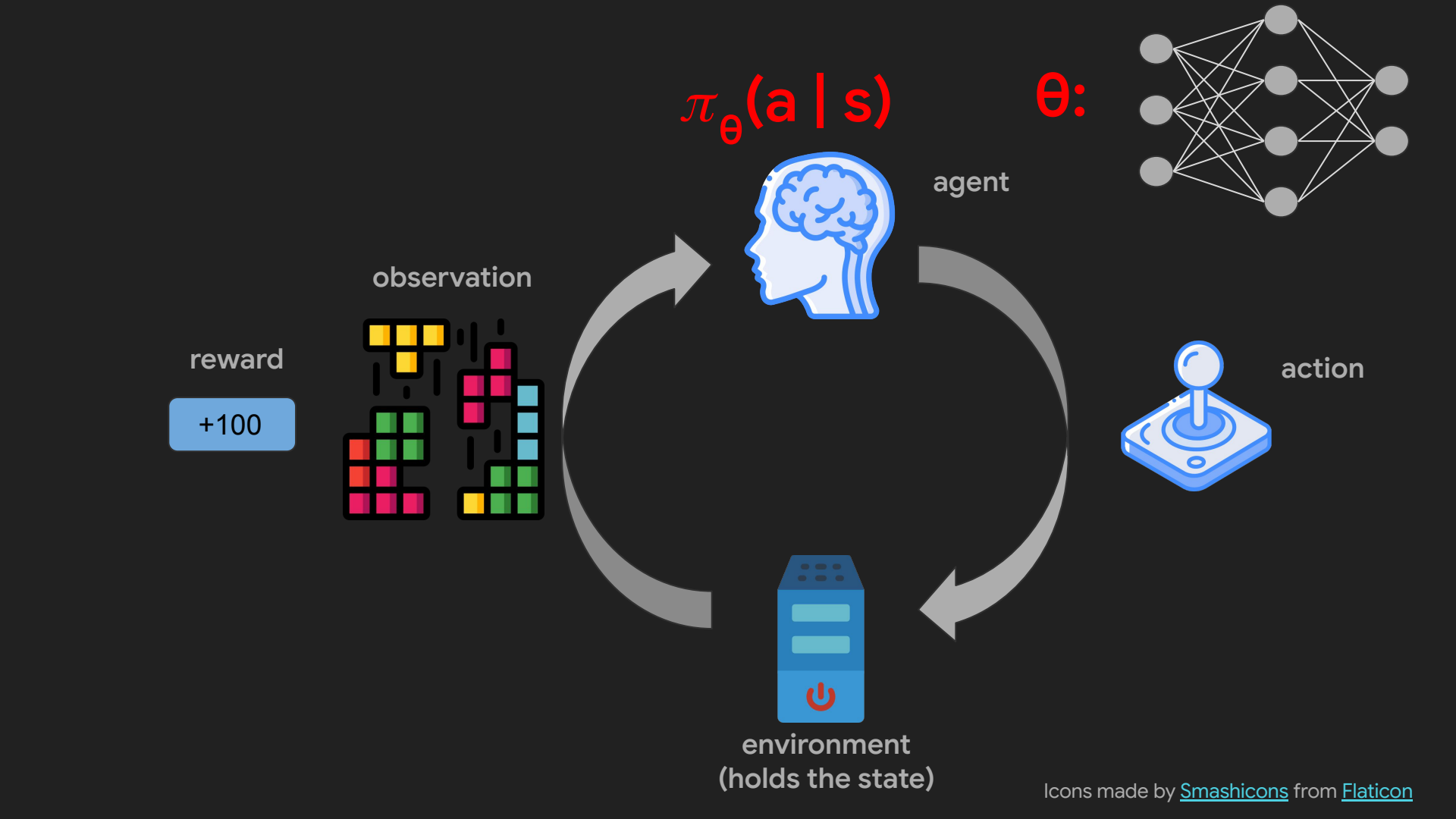
source

Q-values

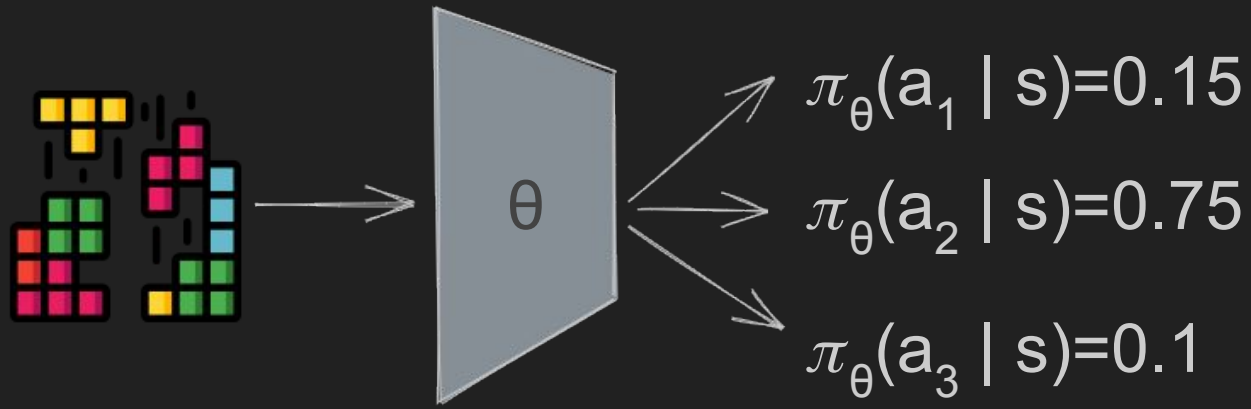
Q*(s,a) - expected discounted reward, when starting in s, making action a and then playing optimally

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

What if the action space is continuous?


$\pi_\theta(a \mid s)$
θ:
agent
observation
reward
+100
action
environment
(holds the state)
Icons made by Smashicons from Flaticon

Network directly represents policy



Randomized policy

- In the Markov Decision Process setting, there is always an optimum deterministic policy.

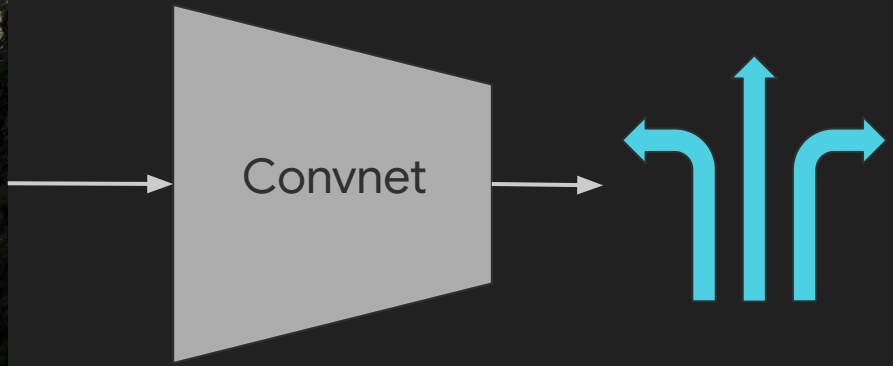
$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

- Randomized policies are much better suited towards optimization facilitation (smoother problem).

$\pi_\theta(a|s)$ - distribution of actions in state s

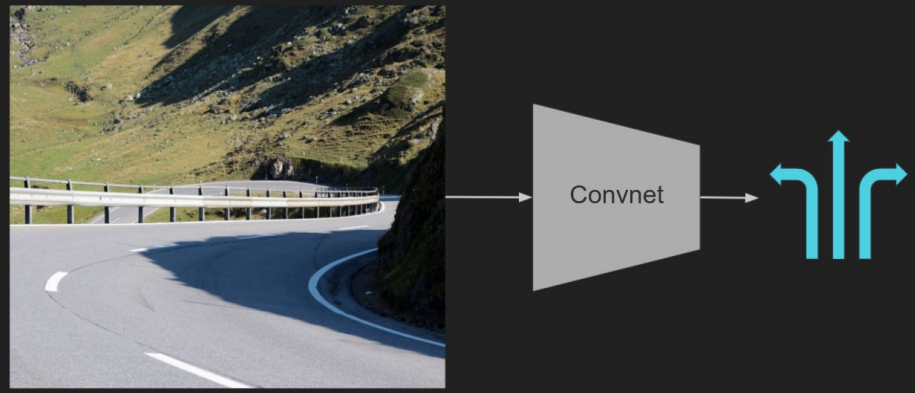
Imitation learning

Behavioural cloning

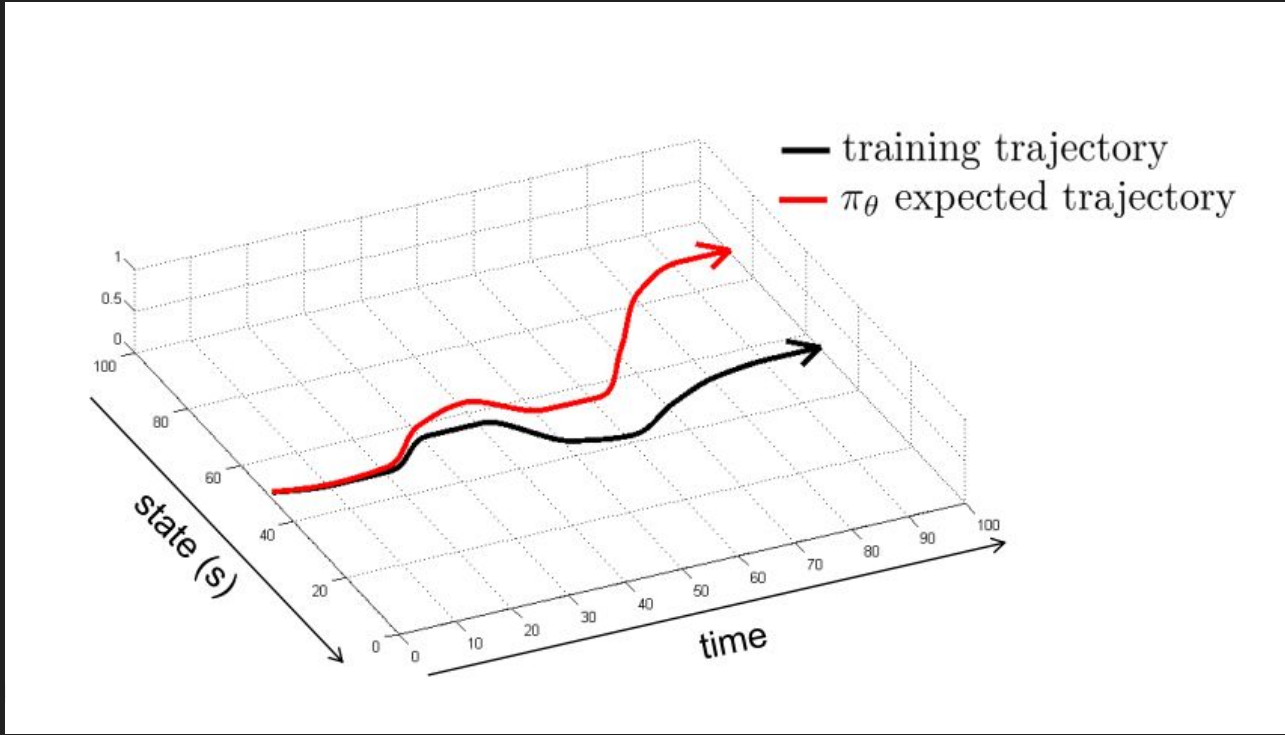


Behavioural cloning

- Collect data from human drivers.
- Use supervised learning to train a policy.

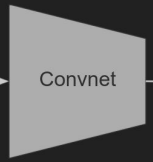


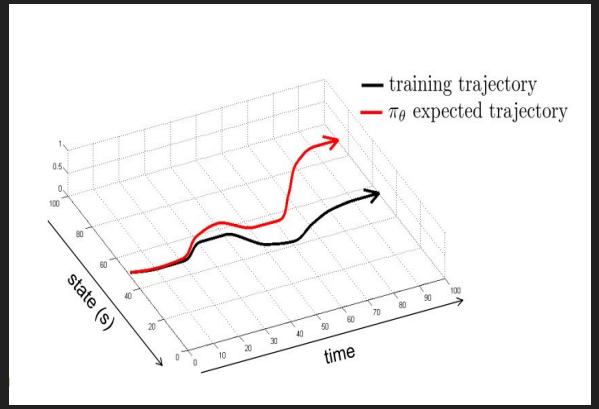
Compounding errors



source

Compounding errors

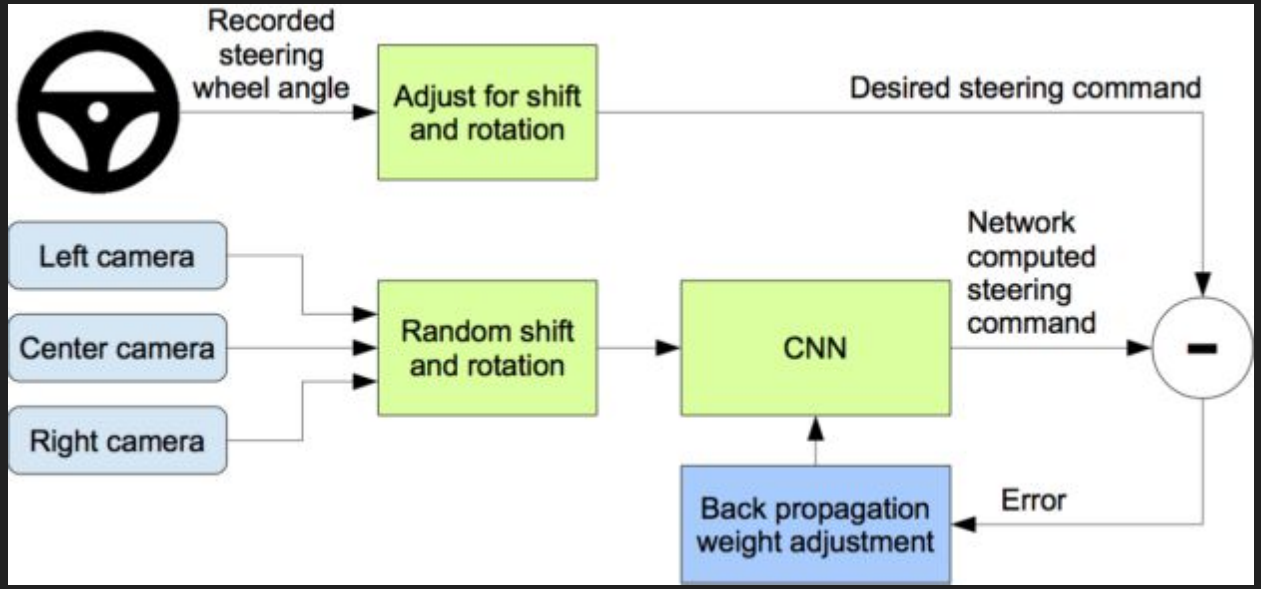


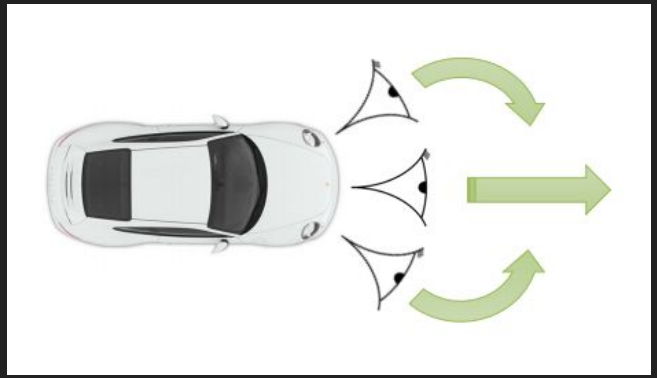


source

source

End-to-End Deep Learning for Self-Driving Cars





End-to-End Deep Learning for Self-Driving Cars, slide

Can we make it work more often?

can we make $p_{\text{data}}(\mathbf{o}_t) = p_{\pi_\theta}(\mathbf{o}_t)$?

idea: instead of being clever about $p_{\pi_\theta}(\mathbf{o}_t)$, be clever about $p_{\text{data}}(\mathbf{o}_t)$!

DAgger: Dataset Aggregation

goal: collect training data from $p_{\pi_\theta}(\mathbf{o}_t)$ instead of $p_{\text{data}}(\mathbf{o}_t)$

how? just run $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$

but need labels $\mathbf{a}_t$!

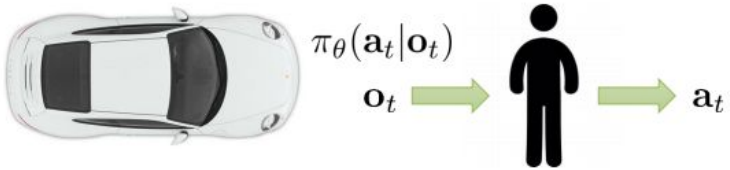
1. train $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_N, \mathbf{a}_N\}$
2. run $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \ldots, \mathbf{o}_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions $\mathbf{a}_t$
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$

Ross et al. '11

slide

What's the problem?

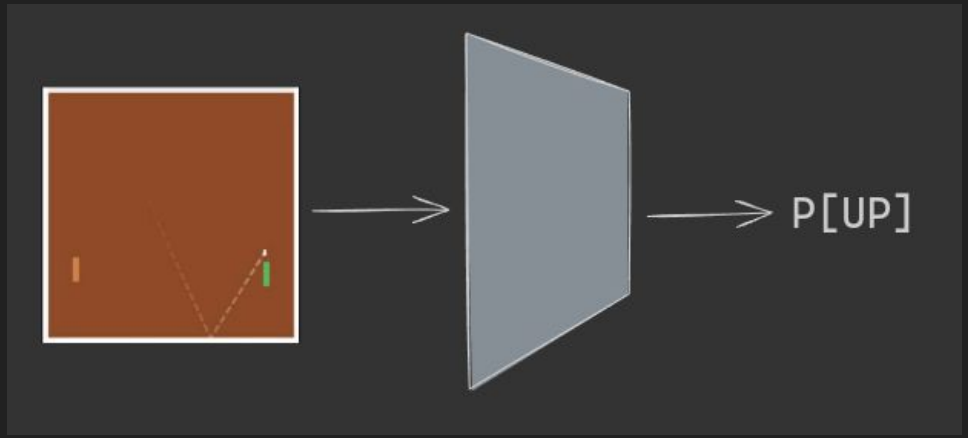
1. train $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_N, \mathbf{a}_N\}$
2. run $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \ldots, \mathbf{o}_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions $\mathbf{a}_t$
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$



Ross et al. '11

Policy gradients - intuition

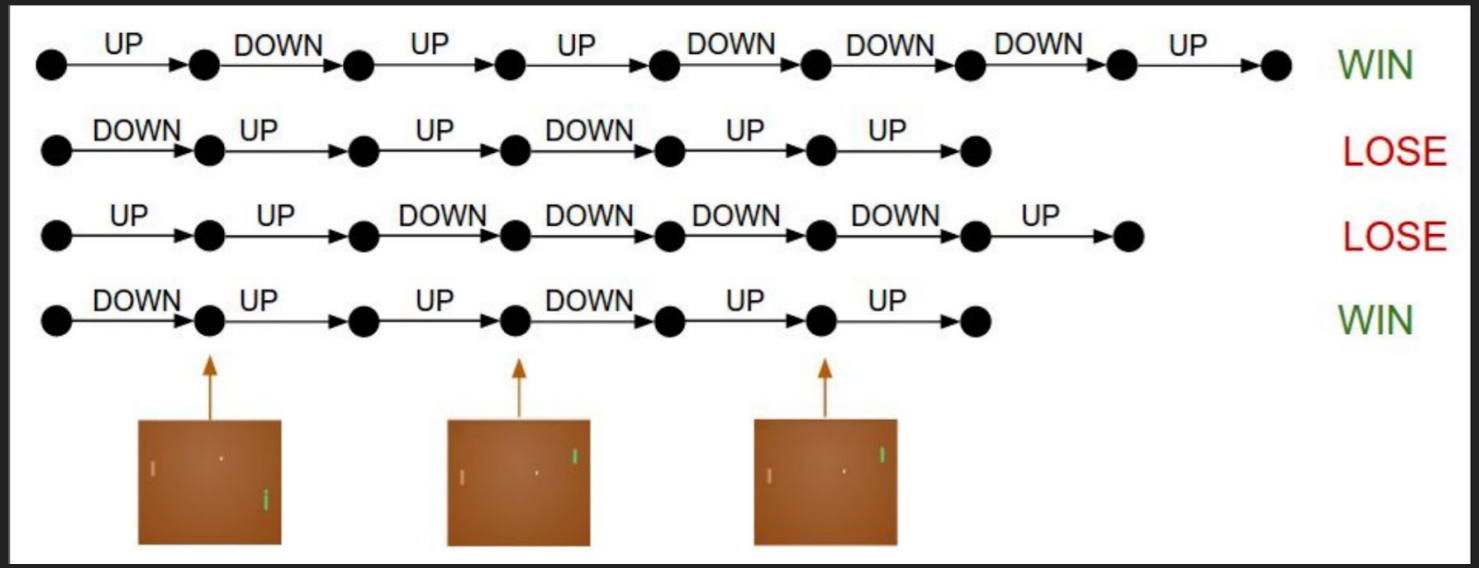
Policy network



- If we had access to an optimum policy oracle, we could use the standard supervised learning
- As we don’t, where can we take the labels from?

source

Collect rollouts



source

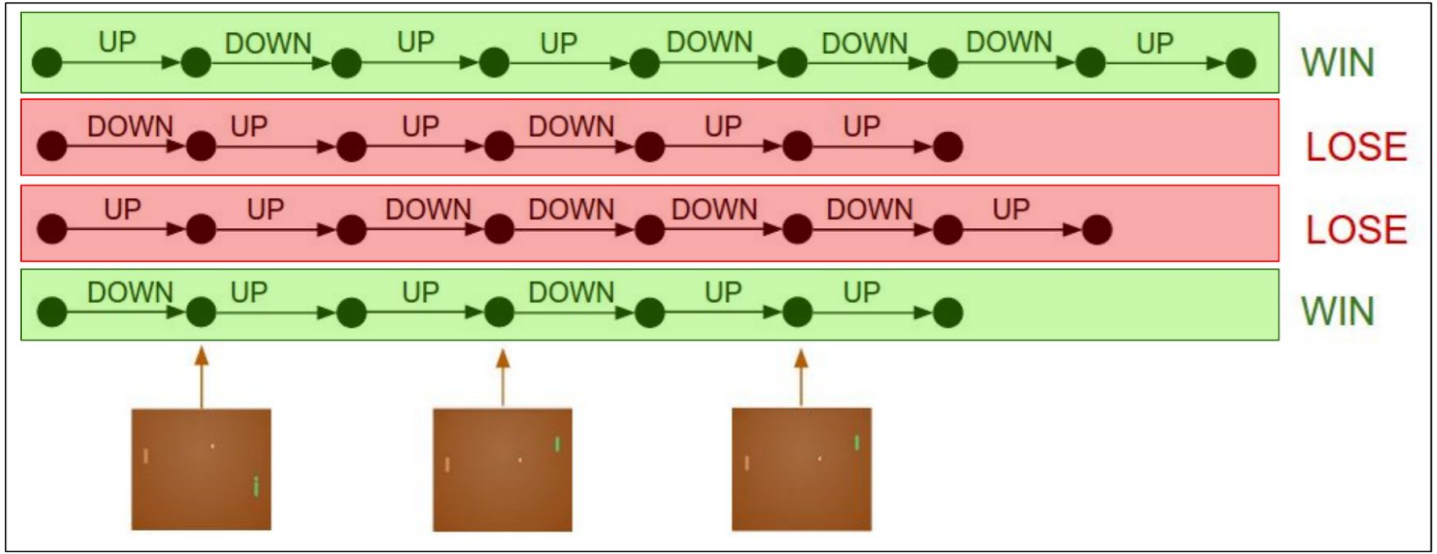
Collect rollouts

Pretend every action we took here was the correct label.

maximize: $\log \pi_\theta(a, s)$

Pretend every action we took here was the wrong label.

maximize: $-1 \cdot \log \pi_\theta(a, s)$



source

Supervised Learning

maximize:

$$\sum_i \log p(y_i | x_i)$$

For images x_i and their labels y_i.

Reinforcement Learning

1) we have no labels so we sample:

$$a \sim \pi_\theta(\cdot, s)$$

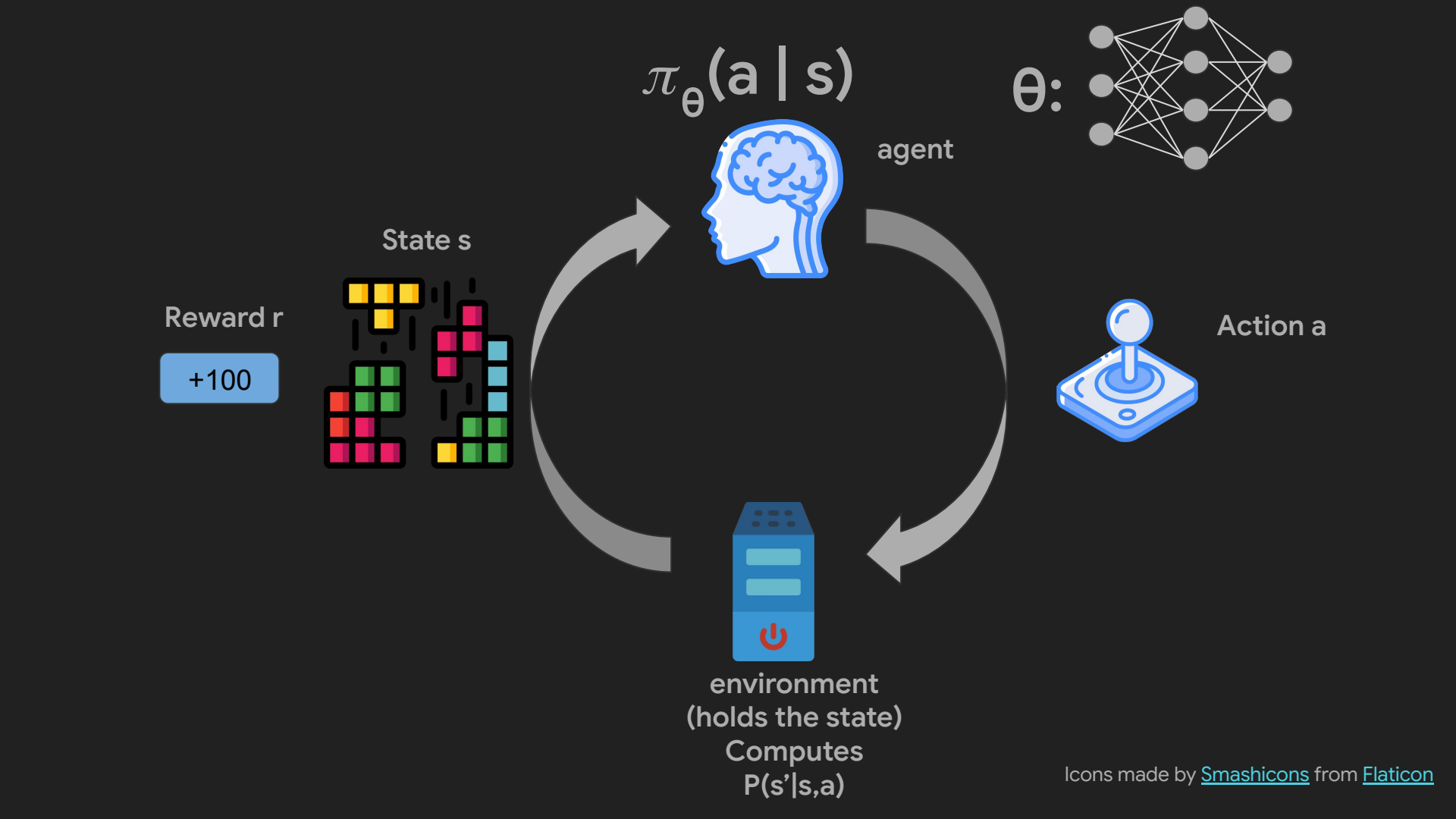
2) once we collect a batch of rollouts:
maximize:

$$\sum_i A_i * \log \pi_\theta(a_i, s_i)$$

+ve advantage will make that action more likely in the future, for that state.
-ve advantage will make that action less likely in the future, for that state.

source

Policy gradients - formally



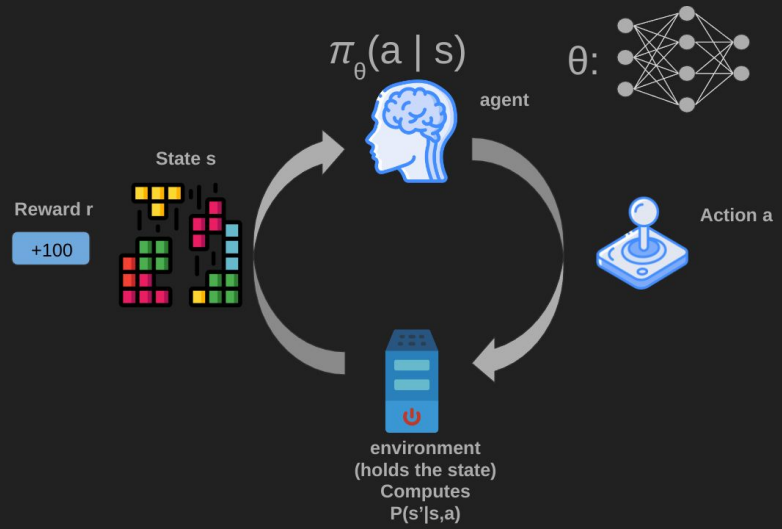
Icons made by Smashicons from Flaticon

Notation

Trajectory $\tau = (s_1, a_1, s_2, a_2, ..., s_T, a_T)$

Policy π_θ

Probability of trajectory $p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$



What do we want to find?

A network: $\theta^* = \arg\max_\theta J(\theta)$

That maximizes this: $J(\theta) = E_{\tau \sim \pi_\theta}\left[\sum_t r(s_t, a_t)\right] \approx \frac{1}{N}\sum_i \sum_t r(s_t^i, a_t^i)$

Trajectories still have to come from the policy!!

Gradient of the objective

$$J(\theta) = E_{\tau \sim \pi_\theta} \underbrace{[r(\tau)]}_{\sum_t r(s_t, a_t)} = \int p_\theta(\tau) r(\tau) d\tau$$

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

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} r(\tau) d\tau =$$

$$= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = E_{\tau \sim \pi_\theta} [\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

Can we compute it in practice?

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\log p_\theta(\tau) = \log p(s_1) + \sum_t \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)$$

$$\nabla_\theta J(\theta) = ... = E_{\tau \sim \pi_\theta} \left[\nabla_\theta \log p_\theta(\tau) r(\tau) \right] =$$

$$= E_{\tau \sim \pi_\theta} \left[\nabla_\theta \left(\log p(s_1) + \sum_t \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right) \cdot r(\tau) \right]$$

These are independent of theta -> Gradient is zero!

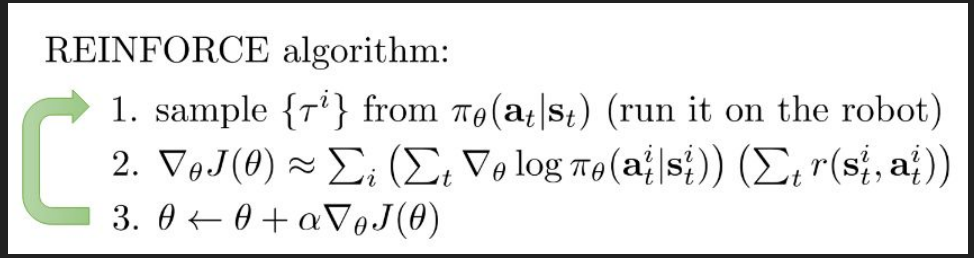
$$= E_{\tau \sim \pi_\theta} \left[\left(\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_t r(s_t, a_t) \right) \right]$$

$$\approx \frac{1}{N} \sum_i \left[\left(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \right) \left(\sum_t r(s_t^i, a_t^i) \right) \right]$$

Pytorch calculates this for us

REINFORCE (Williams, 1992)

Let's apply this in practice:



source

- Only experience from current policy can be used **(on-policy)**
- Can be extremely inefficient **(Monte-Carlo)**

REINFORCE (Williams, 1992)

Let's apply this in practice:

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

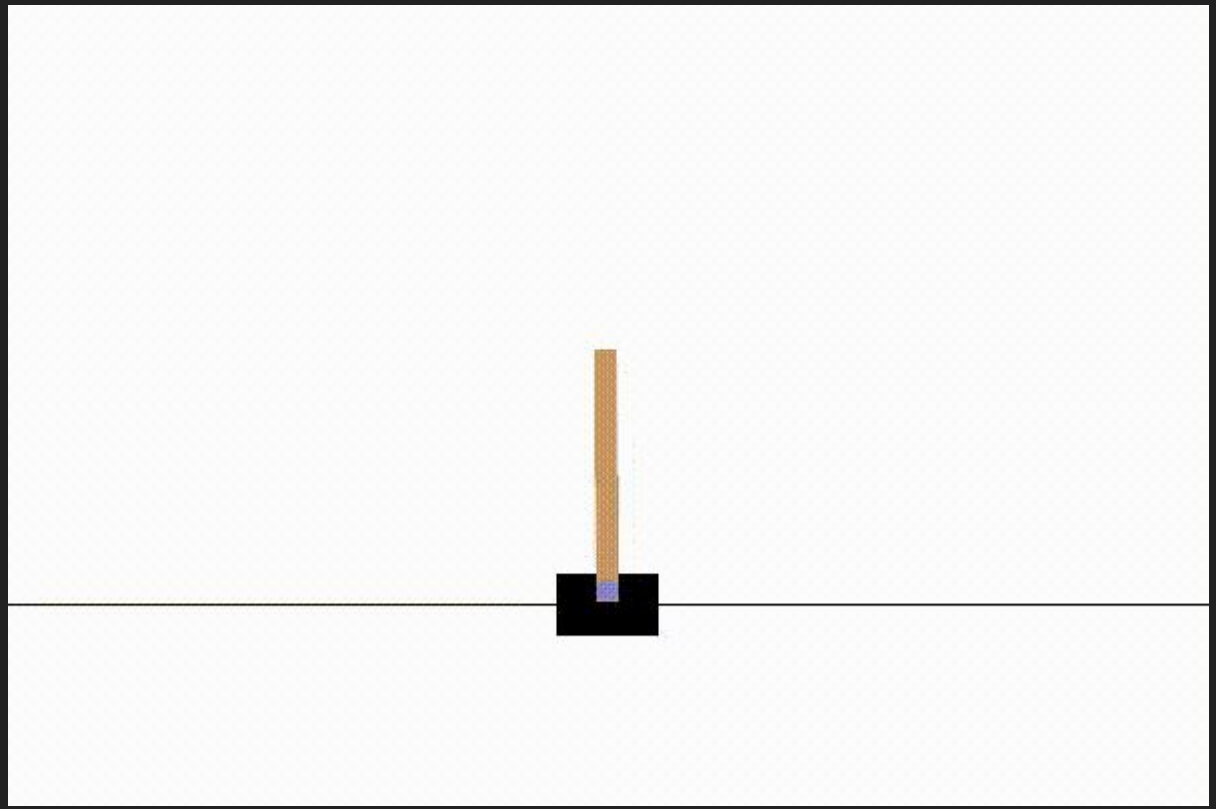
source

- Only experience from current policy can be used **(on-policy)**
- Can be extremely inefficient **(Monte-Carlo)**

Full code in pytorch
(slightly modified)

```
class Policy(nn.Module):
    def __init__(self):
        ...

    def forward(self, x):
        # self.network is a simple MLP with output shape of num_actions
        action_scores = self.network(x)
        return F.softmax(action_scores, dim=1)


def select_action(state):
    state = torch.from_numpy(state).float().unsqueeze(0)
    probs = policy(state)
    distribution = Categorical(probs)
    action = distribution.sample()
    policy.saved_log_probs.append(distribution.log_prob(action))
    return action.item()


def finish_episode():
    reward = sum(policy.rewards)
    policy_loss = []
    for log_prob in policy.saved_log_probs:
        policy_loss.append(-log_prob * reward)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()


def main()
...
    for i_episode in range(num_episodes):
        state, _ = env.reset()
        ep_reward = 0
        for t in range(episode_len):
            action = select_action(state)
            state, reward, terminated, truncated, _ = env.step(action)
            policy.rewards.append(reward)
            if terminated or truncated:
                break

        finish_episode()
...
```

REINFORCE (Williams, 1992)



This took ~10k **episodes**
Or, ~800k **steps**
Or, ~7 mins

This seems a lot for such a simple task.

Can we do better? - Causality

Our target is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i \left[\left(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\right)\left(\sum_t r(s_t^i, a_t^i)\right)\right] =$$

$$= \frac{1}{N}\sum_i \sum_{t=1} \left(\nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \cdot \left(\sum_{t'=1} r(s_{t'}^i, a_{t'}^i)\right)\right)$$

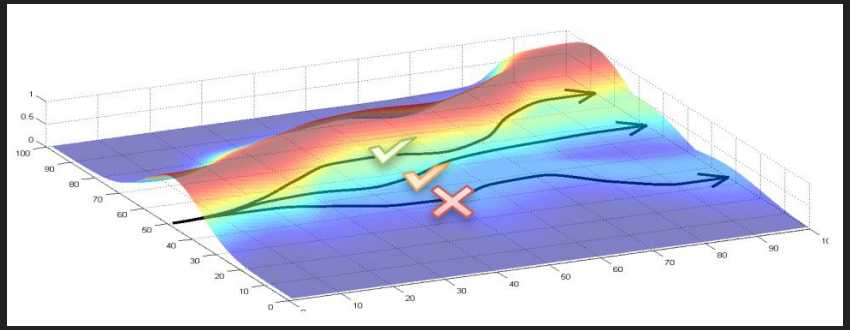
$$= \frac{1}{N}\sum_i \sum_{t=1} \left(\nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \cdot \left(\underbrace{r(s_1^i, a_1^i) + \ldots}_{\text{past}} + \underbrace{r(s_t^i, a_t^i)}_{\text{present}} + \underbrace{r(s_{t+1}^i, a_{t+1}^i) + \ldots}_{\text{future}}\right)\right)$$

We look at past, present and future
But at timestep *t* we don't care about *<t* <- **Causality**

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i \sum_{t=1} \left(\nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \cdot \left(\sum_{t'=t} r(s_t^i, a_t^i)\right)\right)$$

Can we do better? - Baselines

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log p_\theta(\tau^i) \cdot r(\tau^i)$$



source

- If the reward is positive we increase the probability of the **trajectory**
- If the reward is negative we decrease the probability of the **trajectory**

We work on trajectory level not single action level!

But what if all (or most) of the rewards are non-negative? What if they are non-positive?

Can we do better? - Baselines

Idea: Let's subtract *average* reward to change what is a good and a bad trajectory.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log p_\theta(\tau^i) \cdot \Big(r(\tau^i) - b \Big)$$

$$b = \frac{1}{N} \sum_i r(\tau^i)$$

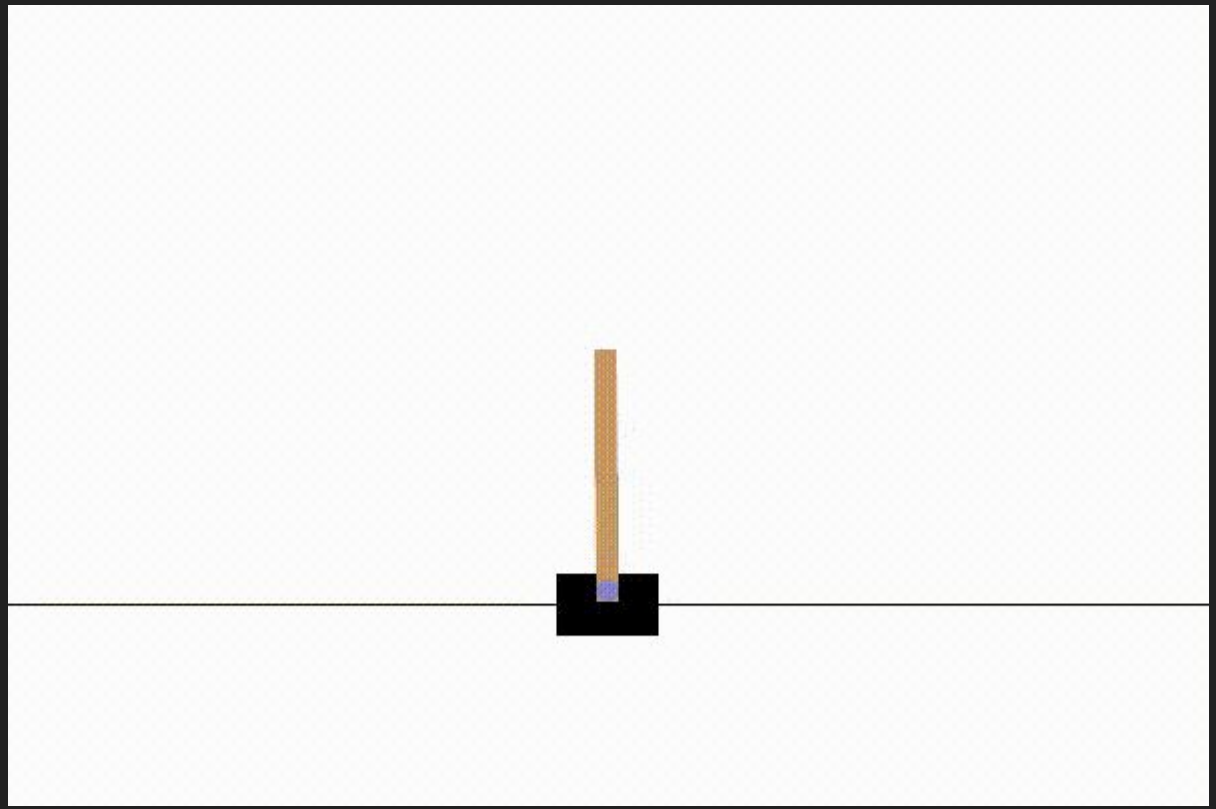
$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)$$

Can we do that (i.e. does the math still work)?

$$E_{\tau \sim \pi_\theta} \left[\nabla_\theta \log p_\theta(\tau) b \right] = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) b \, d\tau = \int \nabla_\theta p_\theta(\tau) b \, d\tau = b \nabla_\theta \int p_\theta(\tau) \, d\tau = b \nabla_\theta 1 = 0$$

Yes! Expectation over average of rewards doesn't add bias.
There are also other, better baselines.

Did we do better?



With causality and baselines it took:
~800 **episodes**
~160k **steps**
~1m20s

In summary about ~5x faster with just 2 small modifications.

(But still seems like a lot of data)

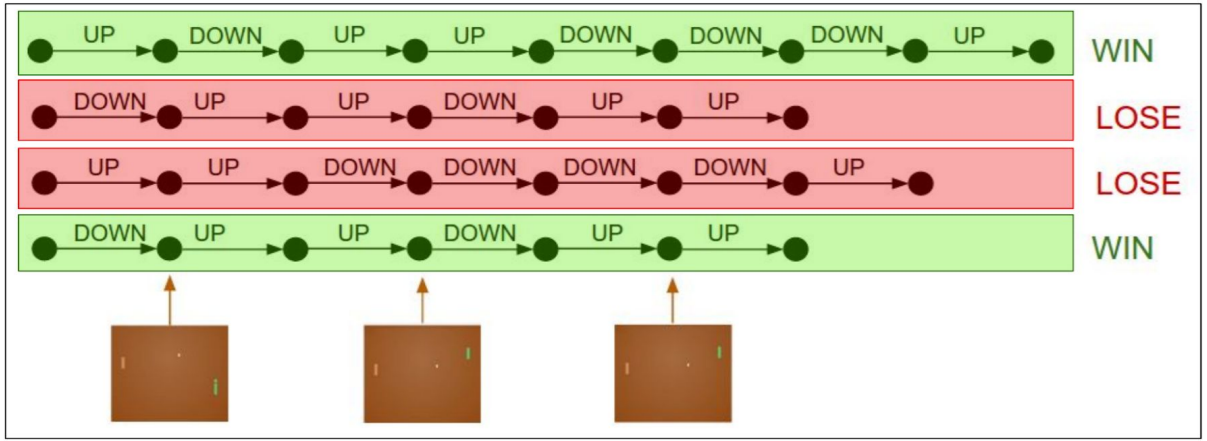
Back to the intuitive explanation

Pretend every action we took here was the correct label.

maximize: $\log \pi_\theta(a, s)$

Pretend every action we took here was the wrong label.

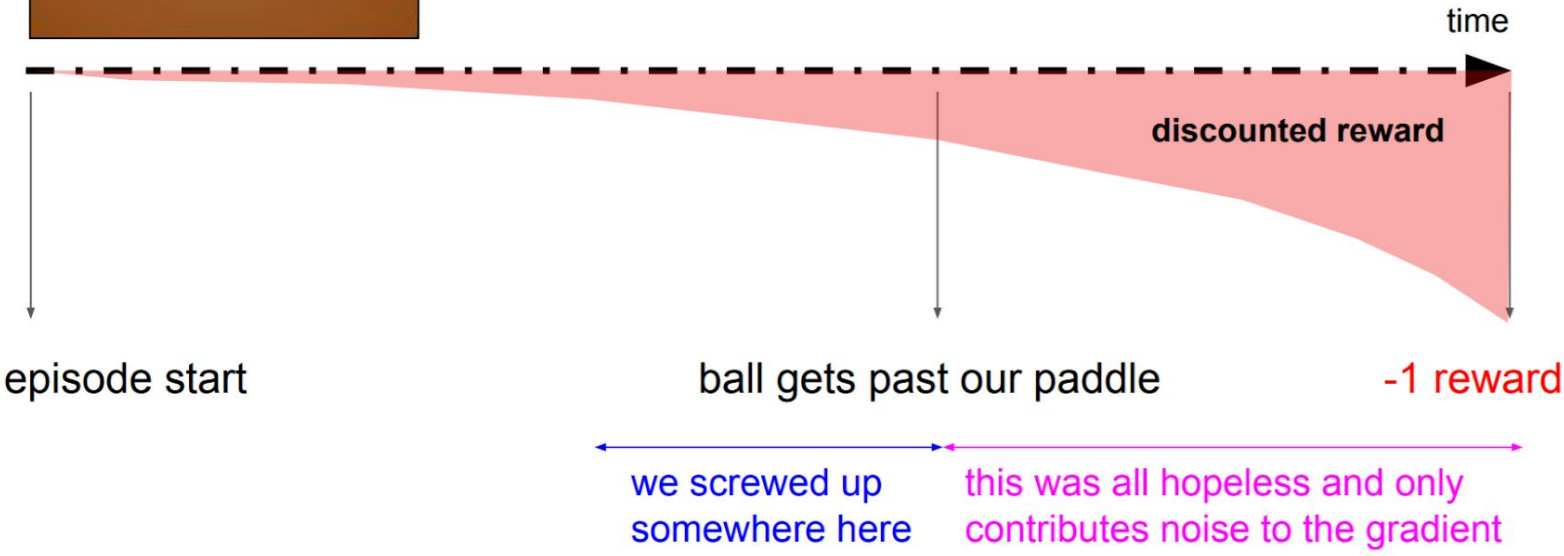
maximize: $-1 \cdot \log \pi_\theta(a, s)$



$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

source

Actions performed in the losing position will be considered bad, no matter what they are. This add a lot of noise to gradient computations



source

Policy optimization

- Optimize directly what you care about.
- Easier to extend (add recurrence, auxiliary objectives).
- Often requires less gradient steps.

Q-learning

- Indirect, exploit the problem structure.
- Can work off-policy (use experience generated by other policies).
- More sample-efficient (requires less samples - interactions with the environment).

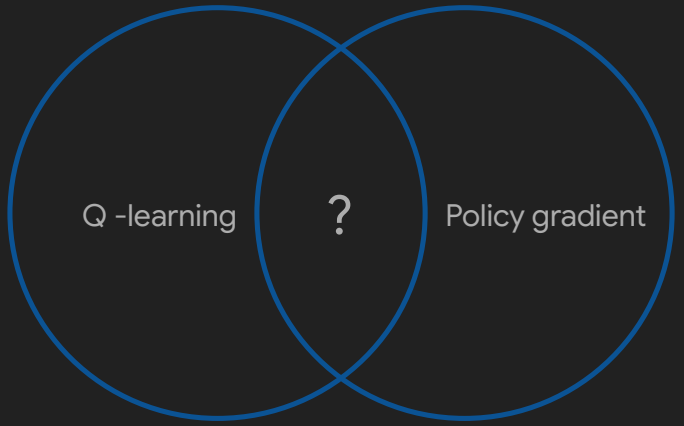
Actor - Critic methods

Best of both worlds:

- Train Q function (**critic**) as in DQN
- Use it to train a policy (**actor**)

Most of the current SOTA are in some way derivatives of this model.

More on that RL course next semester :)



Resources

- Deep RL Bootcamp - Very good slides on Deep RL, but slightly more advanced than our course, going much deeper into most topics
- Berkeley Deep Learning Course - Lectures 14, 15, 16 are similar to our lectures on RL
- Understanding Deep Learning - Very good book on DL, nice chapter on RL, that covers most of this lecture as well, including maths
- Reinforcement Learning: An Introduction - “Bible” of RL, Very detailed, and in depth