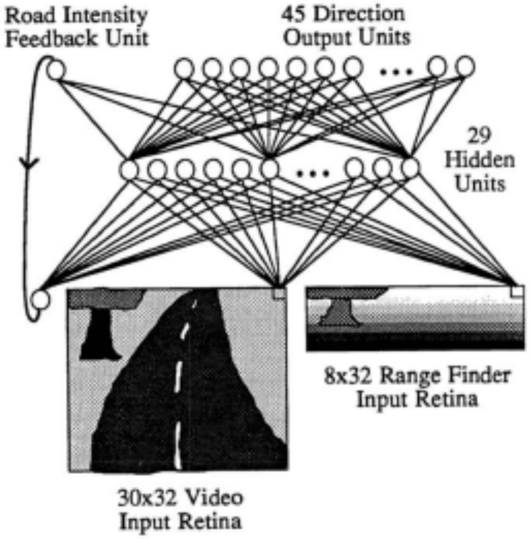
Learning to control

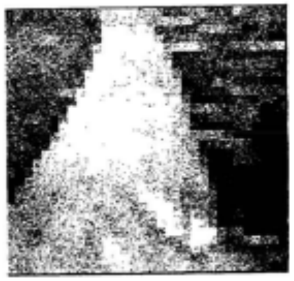
D.A.Forsyth, UIUC

Topics

- Scamper through basic reinforcement learning ideas
- Imitation learning
  - and its variants and problems
  - as structure learning

First learned steering controller


Road Intensity Feedback Unit
45 Direction Output Units
29 Hidden Units
8x32 Range Finder Input Retina
30x32 Video Input Retina



Real Road Image

“ ALVINN:

An autonomous Land vehicle in a neural Network, Pomerleau 1989

Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: transition probability i.e. distribution over next state given (state, action) pair
γ : discount factor

Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
    - Agent selects action a_t
    - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
    - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
    - Agent receives reward r_t and next state s_{t+1}

- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

Fei-Fei+Johnson+Yeung 17

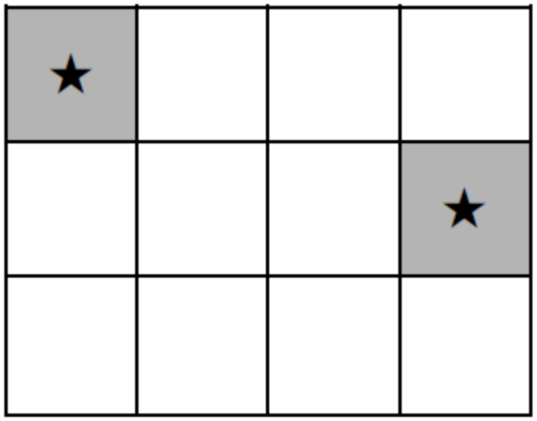
A simple MDP: Grid World

actions = {

1. right →
2. left ←
3. up ↑
4. down ↓

}

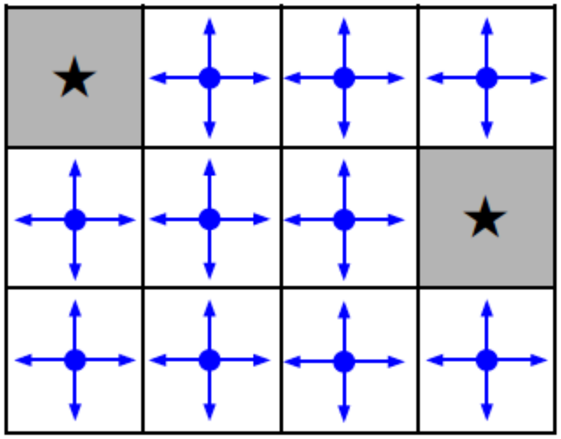
states



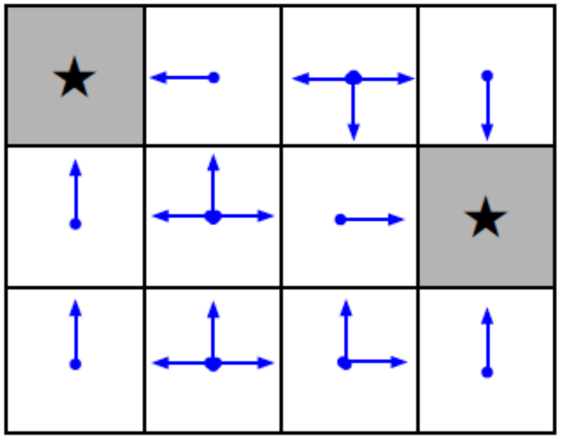
Set a negative "reward" for each transition (e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

The optimal policy **π***

We want to find optimal policy **π*** that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability…)?
Maximize the **expected sum of rewards!**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman equation

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$$

Intuition: if the optimal state-action values for the next time-step Q*(s’,a’) are known, then the optimal strategy is to take the action that maximizes the expected value of

$r + \gamma Q^*(s',a')$

The optimal policy **π*** corresponds to taking the best action in any state as specified by Q*

Fei-Fei+Johnson+Yeung 17

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')|s,a\right]$$

Q_i will converge to Q* as i -> infinity

What's the problem with this?
Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy **π***)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates => greater data efficiency

Policy Gradients

What is a problem with Q-learning?
The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair

But the policy can be much simpler: just close your hand
Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Fei-Fei+Johnson+Yeung 17

Policy Gradients

Formally, let's define a class of parametrized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

How can we do this?

Gradient ascent on policy parameters!

Fei-Fei+Johnson+Yeung 17

REINFORCE algorithm

Mathematically, we can write:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}\left[r(\tau)\right] = \int_{\tau} r(\tau)p(\tau;\theta)\mathrm{d}\tau$$

Where r(τ) is the reward of a trajectory $\tau = (s_0, a_0, r_0, s_1, \ldots)$

Fei-Fei+Johnson+Yeung 17

REINFORCE algorithm

Expected reward: $$J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} \left[r(\tau) \right] = \int_{\tau} r(\tau) p(\tau;\theta) \mathrm{d}\tau$$

Now let's differentiate this: $$\nabla_{\theta} J(\theta) = \int_{\tau} r(\tau) \nabla_{\theta} p(\tau;\theta) \mathrm{d}\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $$\nabla_{\theta} p(\tau;\theta) = p(\tau;\theta) \frac{\nabla_{\theta} p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta) \nabla_{\theta} \log p(\tau;\theta)$$

If we inject this back:

$$\nabla_{\theta} J(\theta) = \int_{\tau} \left(r(\tau) \nabla_{\theta} \log p(\tau;\theta) \right) p(\tau;\theta) \mathrm{d}\tau = \mathbb{E}_{\tau \sim p(\tau;\theta)} \left[r(\tau) \nabla_{\theta} \log p(\tau;\theta) \right]$$

Can estimate with Monte Carlo sampling

REINFORCE algorithm

$$\nabla_\theta J(\theta) = \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau$$
$$= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t,a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on transition probabilities!

Therefore when sampling a trajectory τ, we can estimate J(θ) with

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Intuition

Gradient estimator:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Interpretation:

- If r(τ) is high, push up the probabilities of the actions seen
- If r(τ) is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. But in expectation, it averages out!

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Variance reduction

Gradient estimator: $\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t} r_{t'}\right)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'}\right)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Fei-Fei+Johnson+Yeung 17

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

How to choose the baseline?

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

Variance reduction techniques seen so far are typically used in “Vanilla REINFORCE”

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action a_t in a state s_t if $Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ is large. On the contrary, we are unhappy with an action if it's small.

Using this, we get the estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$

Fei-Fei+Johnson+Yeung 17

Actor-Critic Algorithm

Problem: we don’t know Q and V. Can we learn them?

Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

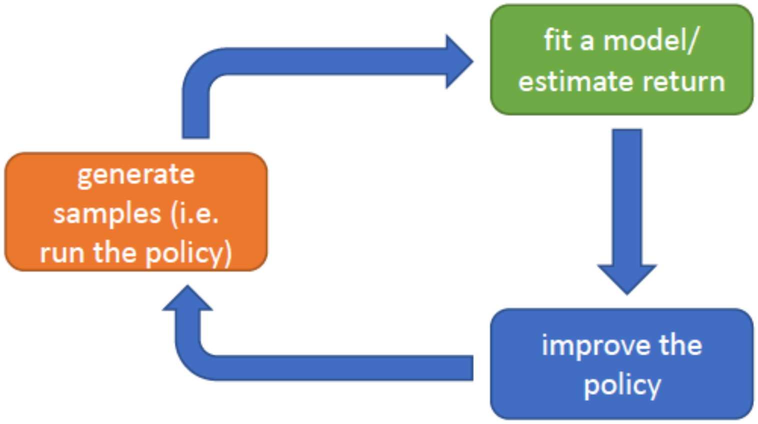
- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Fei-Fei+Johnson+Yeung 17

Why so many RL algorithms?

- Different tradeoffs
  - Sample efficiency
  - Stability & ease of use
- Different assumptions
  - Stochastic or deterministic?
  - Continuous or discrete?
  - Episodic or infinite horizon?
- Different things are easy or hard in different settings
  - Easier to represent the policy?
  - Easier to represent the model?


fit a model/ estimate return
generate samples (i.e. run the policy)
improve the policy

Levine, ND

Blog post entitled: “Why deep reinforcement learning doesn’t work”

https://www.alexirpan.com/2018/02/14/rl-hard.html

Reinforcement Learning: Learning policies guided by sparse rewards, e.g., win or not the game.

- Good: simplest, cheapest form of supervision
- Bad: High sample complexity

Where is it successful so far?

- in simulation, where we can afford a lot of trials, easy to parallelize
- not in robotic systems:
  1. action execution takes long
  2. we cannot afford to fail
  3. safety concerns

Crusher robot

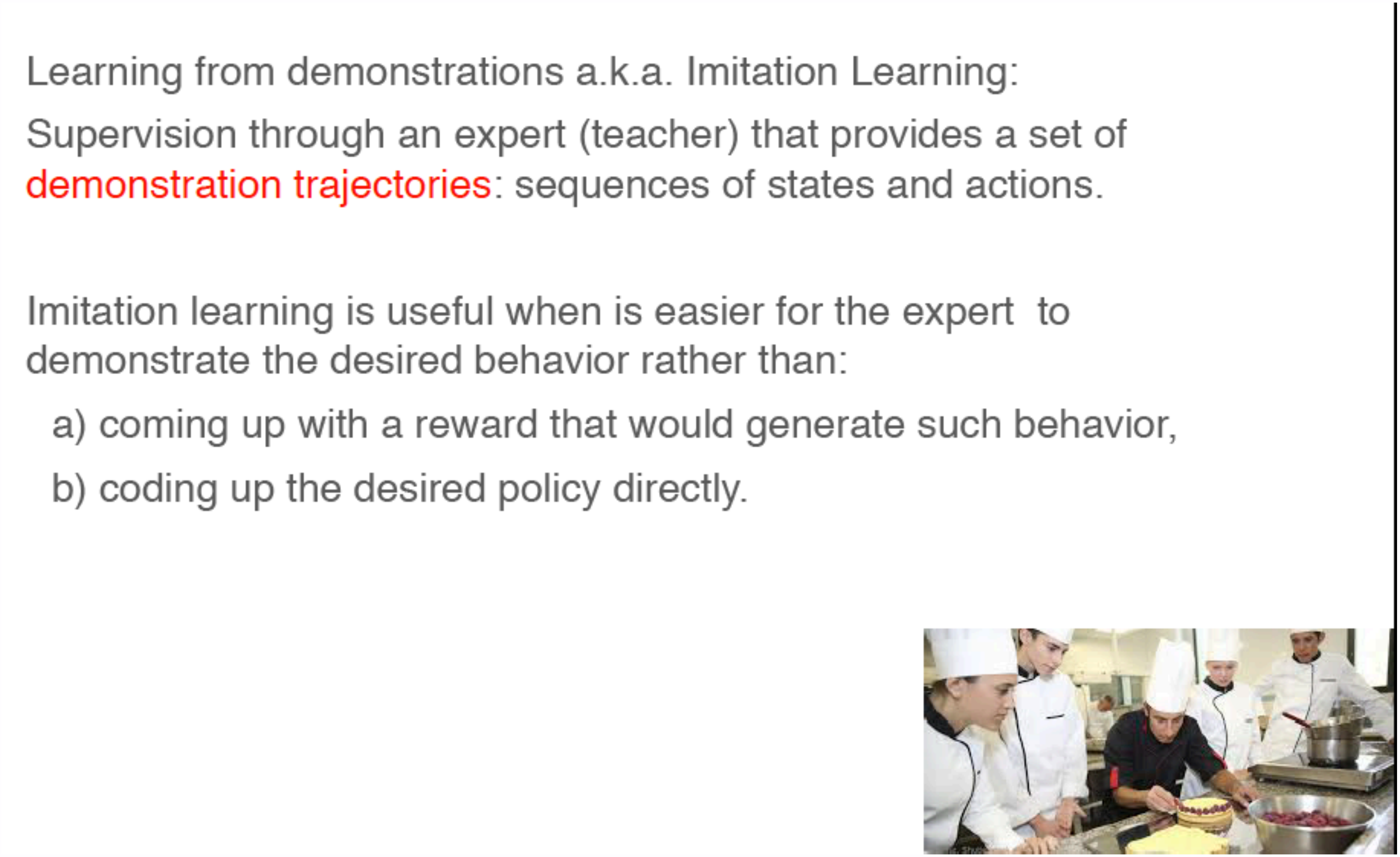
Fragkiadaki, ND

Ideally we want dense in time rewards to closely guide the agent closely along the way.

Who will supply those shaped rewards?

1. We will manually design them: *“cost function design by hand remains one of the ’black arts’ of mobile robotics, and has been applied to untold numbers of robotic systems”*
2. We will learn them from demonstrations: *“rather than having a human expert tune a system to achieve desired behavior, the expert can demonstrate desired behavior and the robot can tune itself to match the demonstration”*

Fragkiadaki, ND



Learning from demonstrations a.k.a. Imitation Learning:

Supervision through an expert (teacher) that provides a set of demonstration trajectories: sequences of states and actions.

Imitation learning is useful when is easier for the expert to demonstrate the desired behavior rather than:

a) coming up with a reward that would generate such behavior,

b) coding up the desired policy directly.

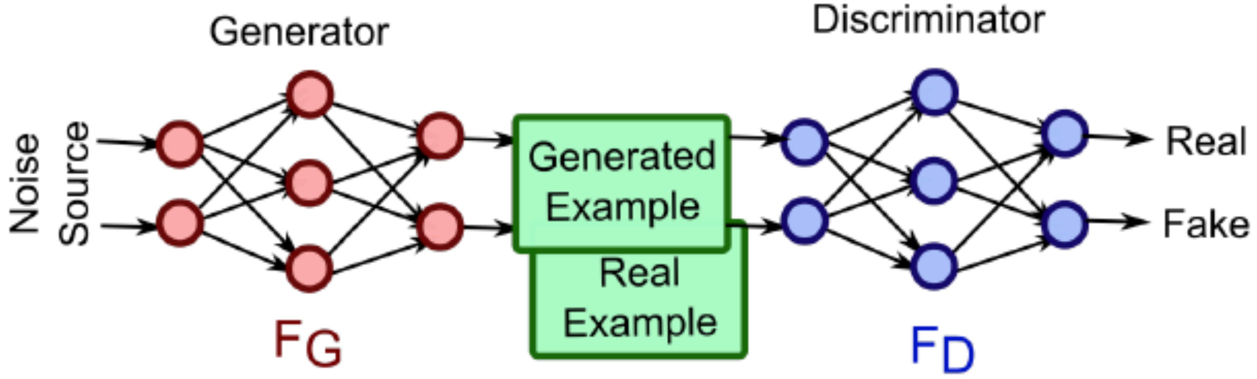
Fragkiadaki, ND

The Imitation Learning problem

The agent (learner) needs to come up with a policy whose resulting state, action trajectory distribution matches the expert trajectory distribution.

Does this remind us of something…?

GANs! Generative Adversarial Networks (on state-action trajectories)



Generative Adversarial Networks, Goodfellow et al. 2014

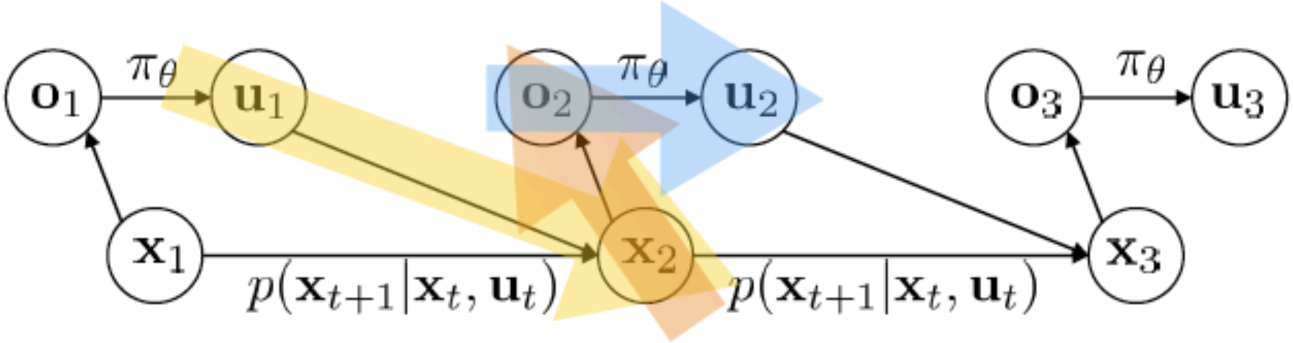
Fragkiadaki, ND

The Imitation Learning problem: Challenge

Actions along the trajectories are interdependent, as actions determine state transitions and thus states and actions down the road.

interdependent labels -> structure prediction

Action interdependence in time:



Algorithms developed in Robotics for imitation learning found applications in structured predictions problems, such as, sequence generation/labelling e.g. parsing.

Fragkiadaki, ND

Imitation Learning

For taking this structure into account, numerous formulations have been proposed:

- Direct: Supervised learning for policy (mapping states to actions) using the demonstration trajectories as ground-truth(a.k.a. behavior cloning) + ways to handle the neglect of action interdependence.

- Indirect: Learning the latent rewards/goals of the teacher and planning under those rewards to get the policy, a.k.a. Inverse Reinforcement Learning (next lecture)

Experts can be:

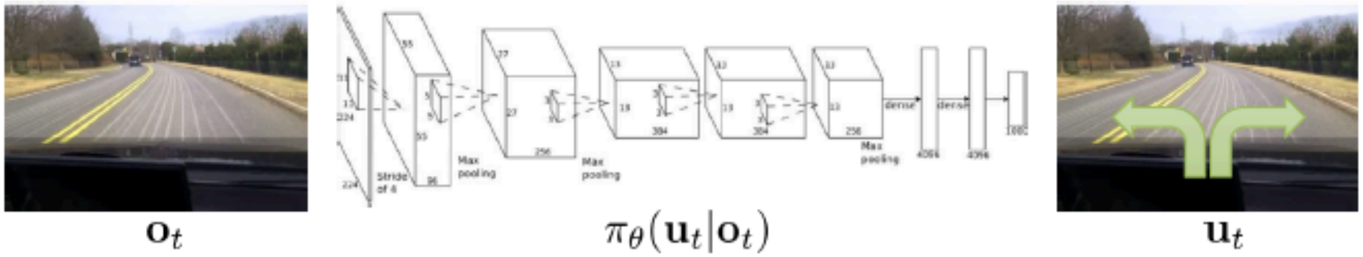
- Humans

- Optimal or near Optimal Planners/Controllers

Fragkiadaki, ND

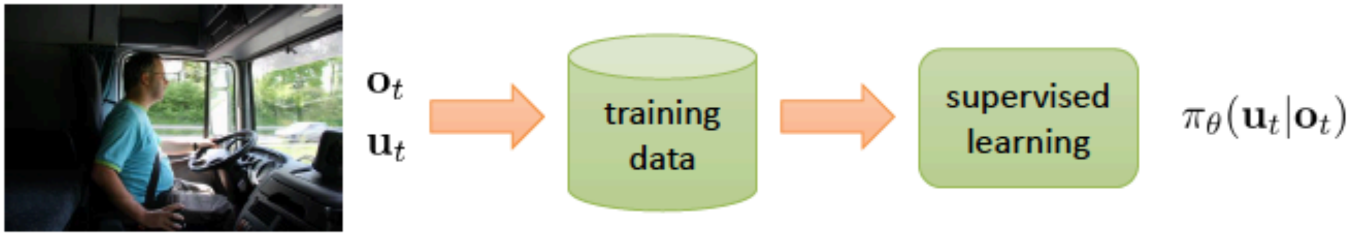
Imitation Learning as Supervised Learning

Driving policy: a mapping from (history of) observations to steering wheel angles



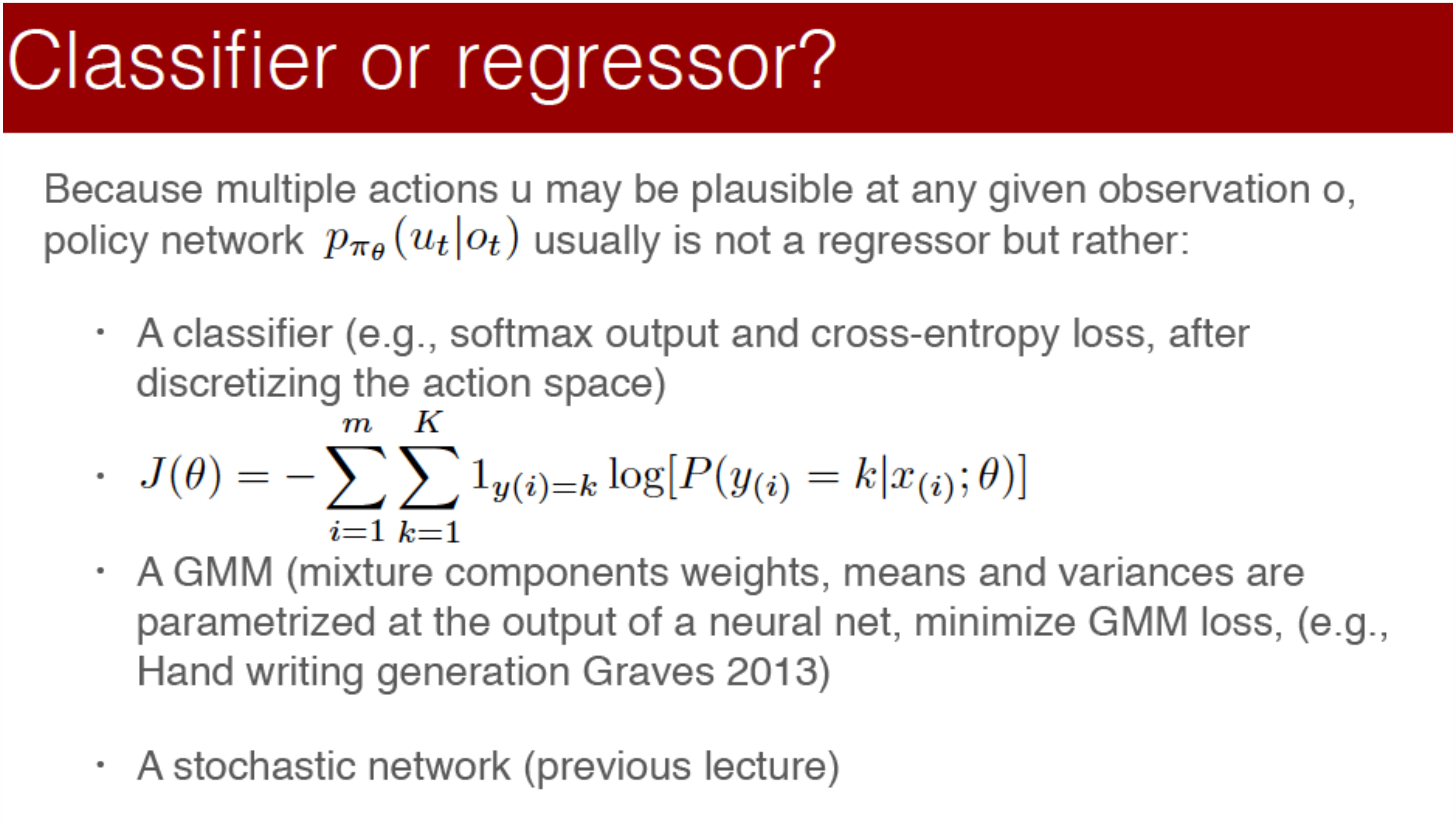
Behavior Cloning=Imitation Learning as Supervised learning

- Assume actions in the expert trajectories are i.i.d.
- Train a classifier or regressor to map observations to actions at each time step of the trajectory.



Fragkiadaki, ND

End to End Learning for Self-Driving Cars, Bojarski et al. 2016



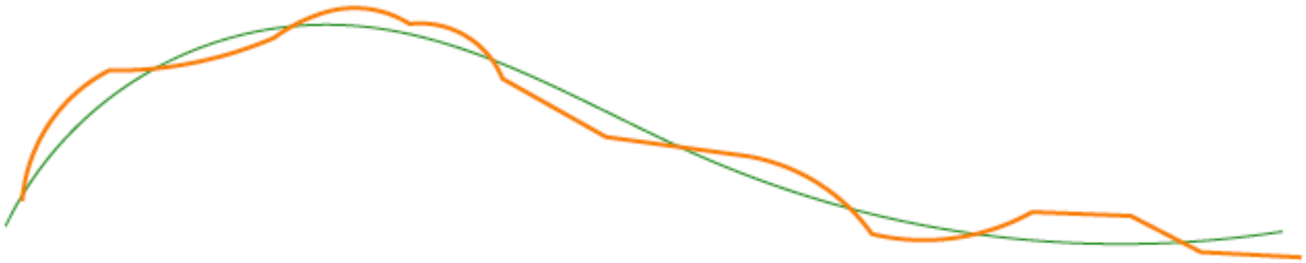
Classifier or regressor?

Because multiple actions u may be plausible at any given observation o, policy network $p_{\pi_\theta}(u_t|o_t)$ usually is not a regressor but rather:

- A classifier (e.g., softmax output and cross-entropy loss, after discretizing the action space)
- $J(\theta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1_{y(i)=k} \log[P(y_{(i)} = k|x_{(i)};\theta)]$
- A GMM (mixture components weights, means and variances are parametrized at the output of a neural net, minimize GMM loss, (e.g., Hand writing generation Graves 2013)
- A stochastic network (previous lecture)

Fragkiadaki, ND

Independent in time errors

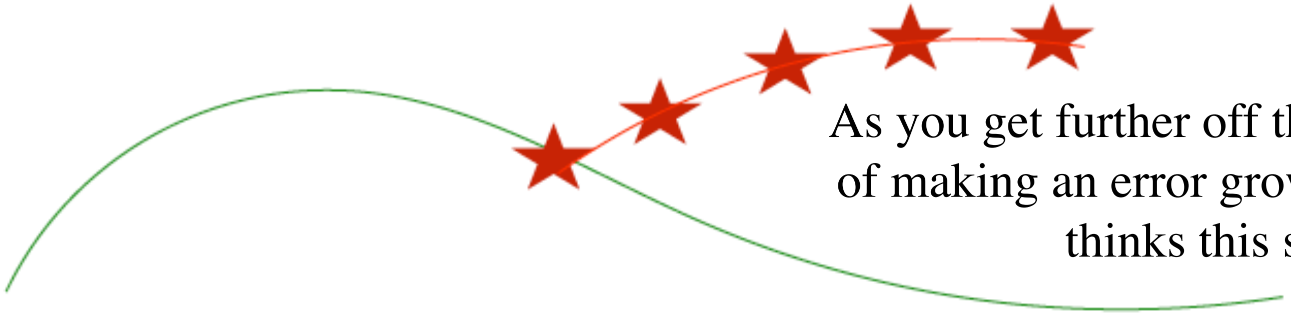


error at time t with probability ε

E[Total errors] ≲ εT

Fragkiadaki, ND

Compounding Errors

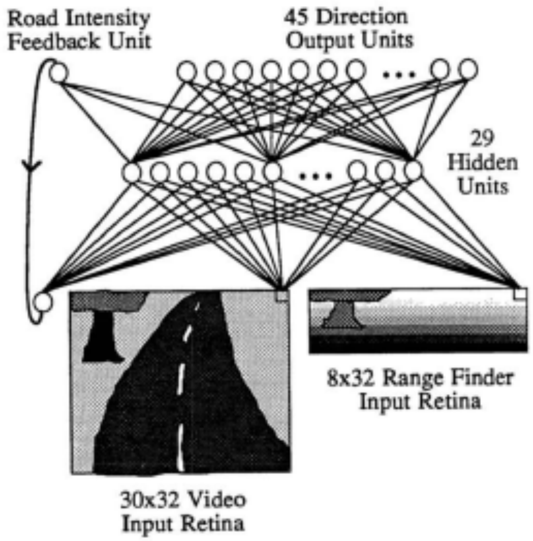


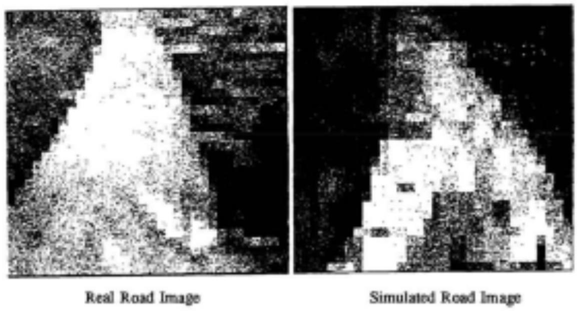
As you get further off the path, the probability of making an error grows, cause the classifier thinks this state is rare

error at time t with probability ε

$$E[\text{Total errors}] \lesssim \varepsilon(T + (T-1) + (T-2) + \ldots + 1) \propto \varepsilon T^2$$

Fragkiadaki, ND A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning, Ross et al. 2011


Road Intensity Feedback Unit
45 Direction Output Units
29 Hidden Units
8x32 Range Finder Input Retina
30x32 Video Input Retina

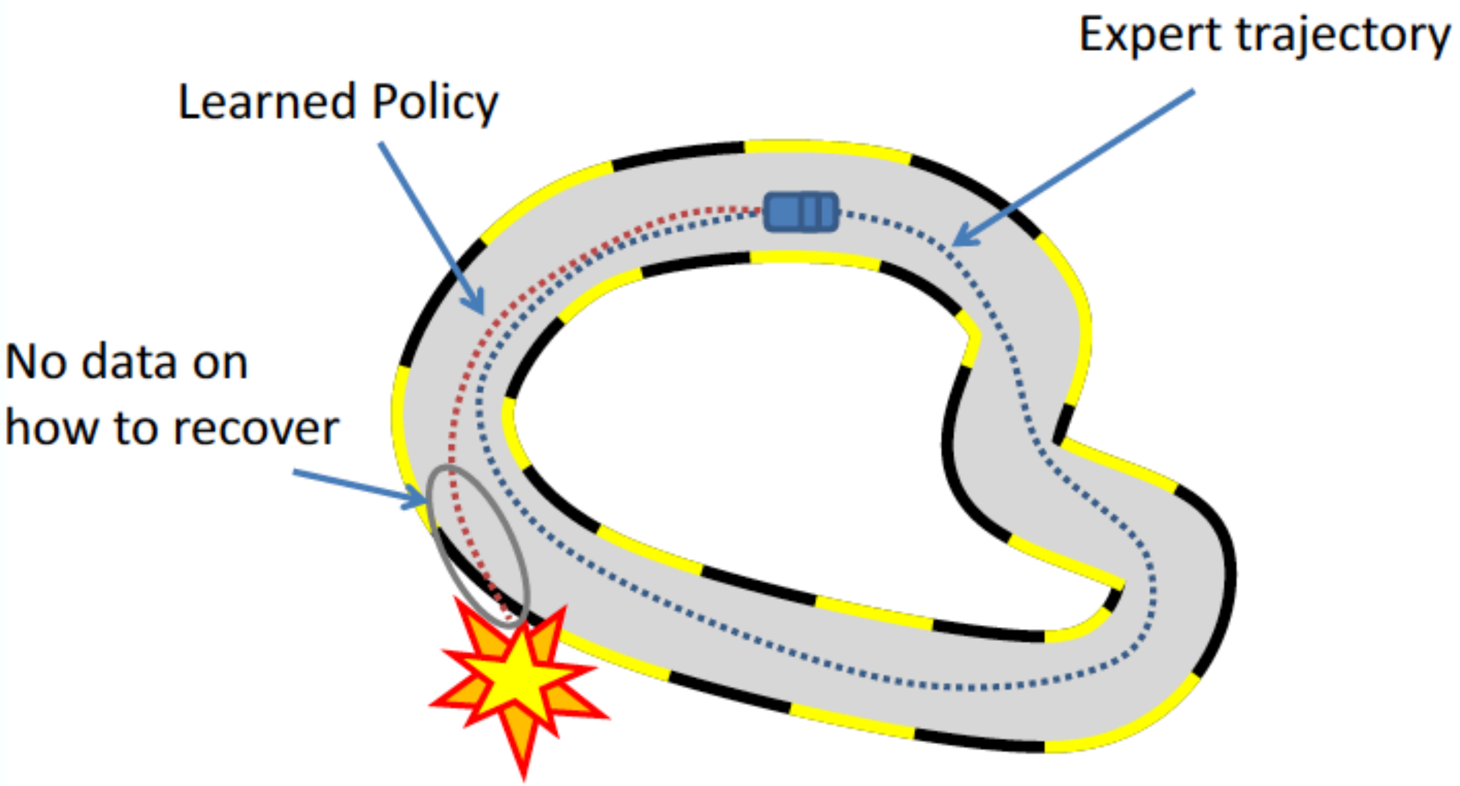


"In addition, the network must not solely be shown examples of accurate driving, but also how to recover (i.e. return to the road center) once a mistake has been made. Partial initial training on a variety of simulated road images should help eliminate these difficulties and facilitate better performance. " ALVINN: An autonomous Land vehicle in a neural Network, Pomerleau 1989

Fragkiadaki, ND

Data Distribution Mismatch!

$$p_{\pi^*}(o_t) \neq p_{\pi_\theta}(o_t)$$



Fragkiadaki, ND

Data Distribution Mismatch!

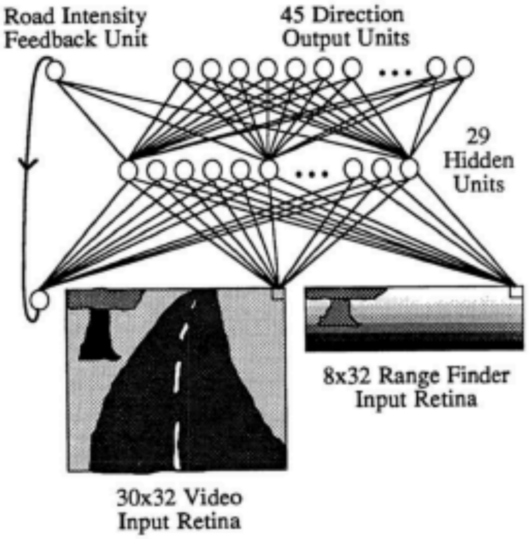
	supervised learning	supervised learning + control (NAIVE)
train	$(x,y) \sim D$	$s \sim d_{\pi^*}$
test	$(x,y) \sim D$	$s \sim d_{\pi}$

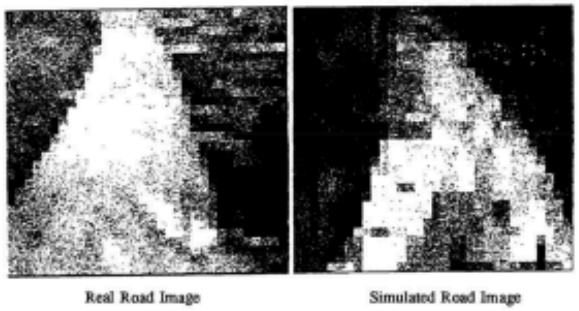
SL succeeds when training and test data distributions match, that is a fundamental assumption.

Fragkiadaki, ND

Demonstration Augmentation: ALVINN 1989

Road follower



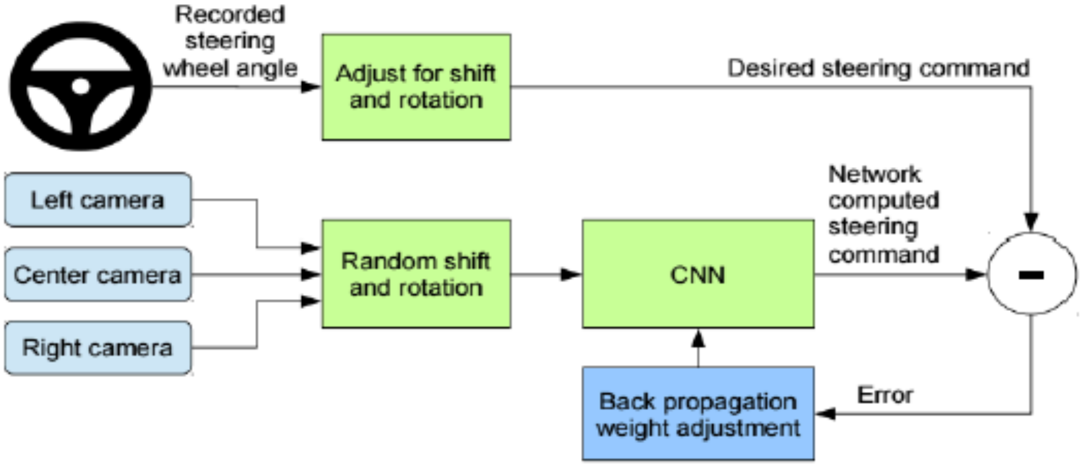


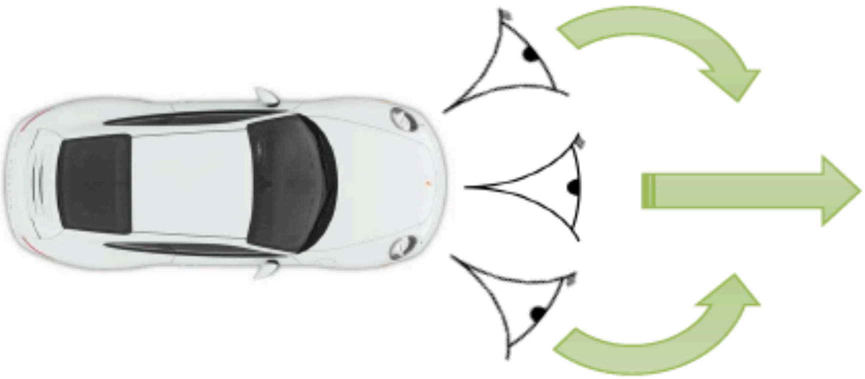
- Using graphics simulator for road images and corresponding steering angle ground-truth
- Online adaptation to human driver steering angle control
- 3 layers, fully connected layers, very low resolution input from camera and lidar..

"In addition, the network must not solely be shown examples of accurate driving, but also how to recover (i.e. return to the road center) once a mistake has been made. Partial initial training on a variety of simulated road images should help eliminate these difficulties and facilitate better performance. " ALVINN: An autonomous Land vehicle in a neural Network, Pomerleau 1989

Fragkiadaki, ND

Demonstration Augmentation: NVIDIA 2016


Recorded steering wheel angle
Adjust for shift and rotation
Desired steering command
Left camera
Center camera
Right camera
Random shift and rotation
CNN
Network computed steering command
-
Back propagation weight adjustment
Error



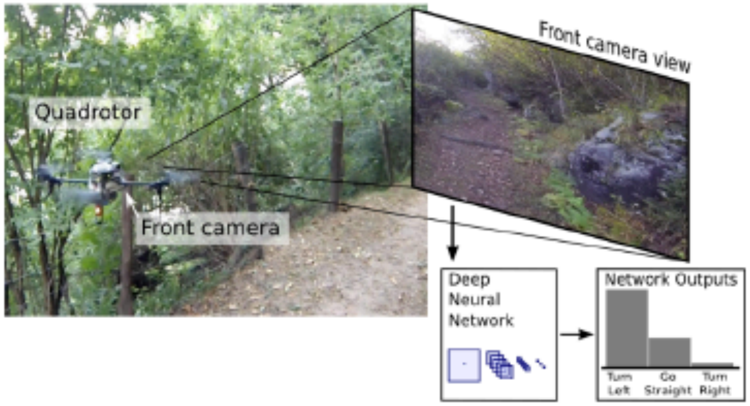
Additional, left and right cameras with automatic grant-truth labels to recover from mistakes

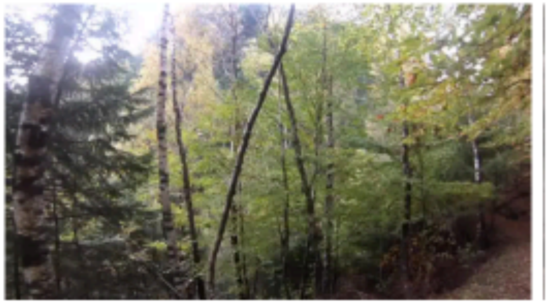
“*DAVE-2 was inspired by the pioneering work of Pomerleau [6] who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system*. Training with data from only the human driver is not sufficient. The network must learn how to recover from mistakes. …”,

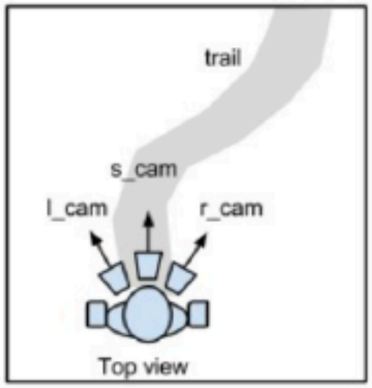
End to End Learning for Self-Driving Cars , Bojarski et al. 2016

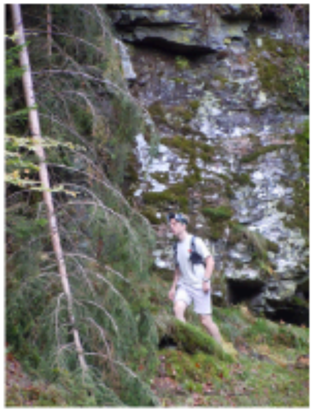
Fragkiadaki, ND

Data Augmentation (3): Trails 2015







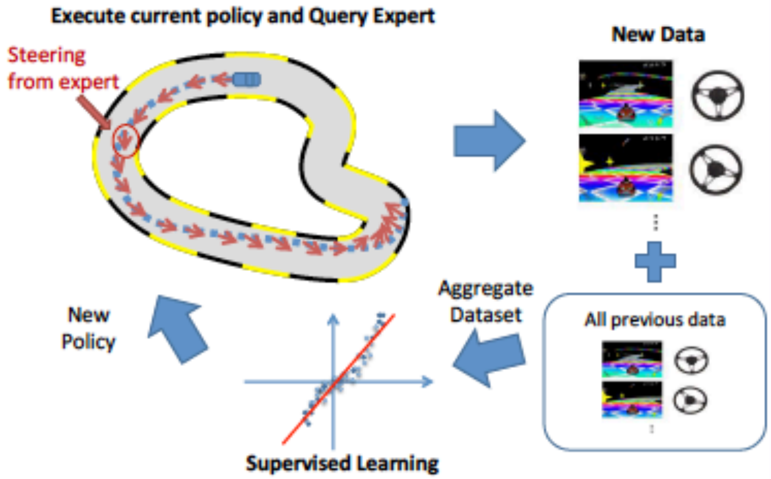


A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots Giusti et al.

DAGGER (in simulation)

Dataset AGGregation: bring learner's and expert's trajectory distributions closer by labelling additional data points resulting from applying the current policy

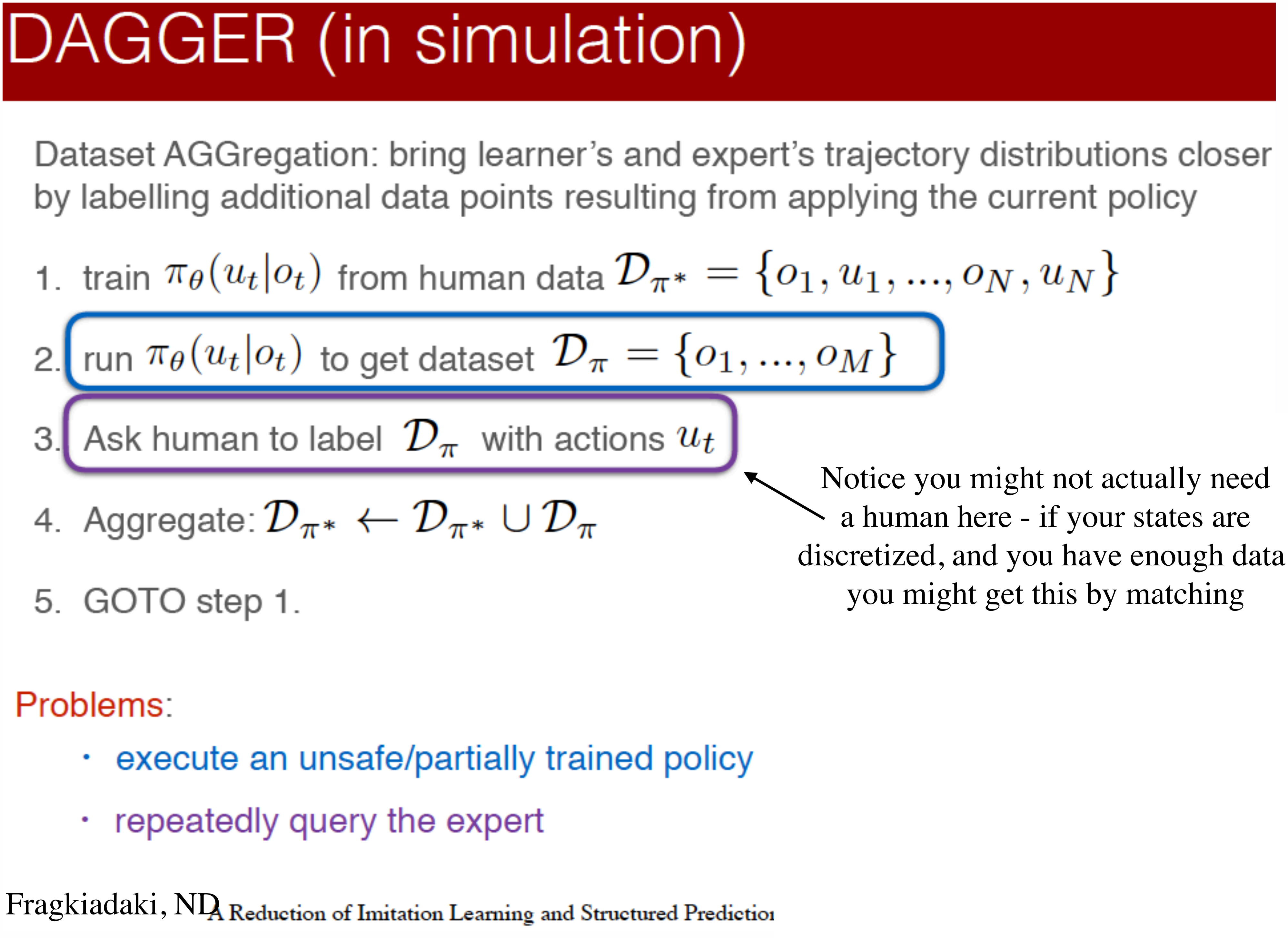
1. train $\pi_\theta(u_t|o_t)$ from human data $\mathcal{D}_{\pi^*} = \{o_1, u_1, ..., o_N, u_N\}$
2. run $\pi_\theta(u_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., o_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions u_t
4. Aggregate: $\mathcal{D}_{\pi^*} \leftarrow \mathcal{D}_{\pi^*} \cup \mathcal{D}_\pi$
5. GOTO step 1.



Problems:

- execute an unsafe/partially trained policy
- repeatedly query the expert

A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning, Ross et al. 2011



DAGGER (in simulation)

Dataset AGGregation: bring learner's and expert's trajectory distributions closer by labelling additional data points resulting from applying the current policy

1. train $\pi_\theta(u_t|o_t)$ from human data $\mathcal{D}_{\pi^*} = \{o_1, u_1, ..., o_N, u_N\}$
2. run $\pi_\theta(u_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., o_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions u_t
4. Aggregate: $\mathcal{D}_{\pi^*} \leftarrow \mathcal{D}_{\pi^*} \cup \mathcal{D}_\pi$
5. GOTO step 1.

Problems:

- execute an unsafe/partially trained policy
- repeatedly query the expert

Notice you might not actually need a human here - if your states are discretized, and you have enough data, you might get this by matching

Fragkiadaki, ND

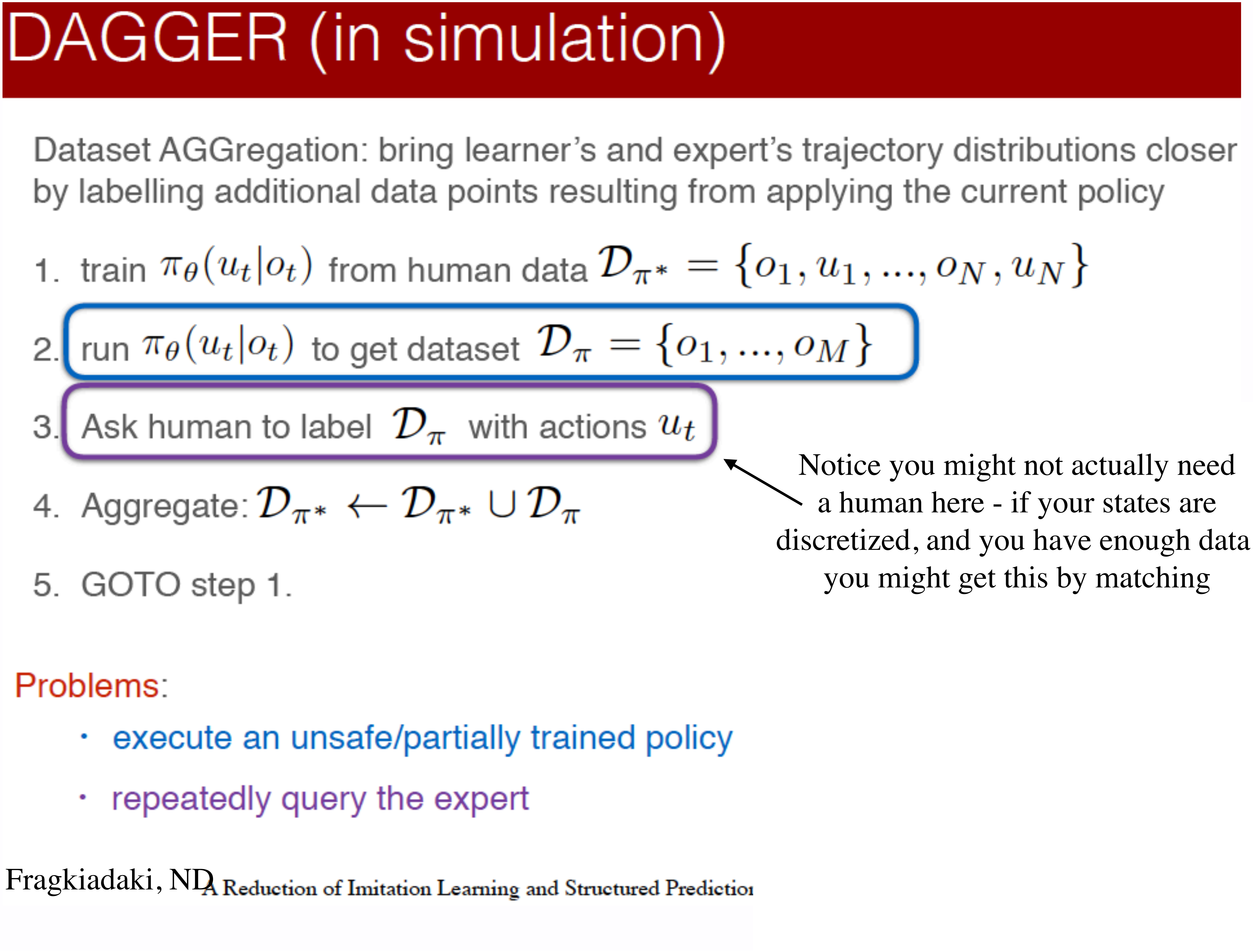
A Reduction of Imitation Learning and Structured Predictio

```
Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in $\Pi$.
**for** $i = 1$ **to** $N$ **do**
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
    Sample $T$-step trajectories using $\pi_i$.
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by $\pi_i$
    and actions given by expert.
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
    Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
**end for**
**Return** best $\hat{\pi}_i$ on validation.
```

Algorithm 3.1: DAGGER Algorithm.

Variants of DAGGER

- AGGRAVATE
- AGGRAVATED



DAGGER (in simulation)

Dataset AGGregation: bring learner's and expert's trajectory distributions closer by labelling additional data points resulting from applying the current policy

1. train $\pi_\theta(u_t|o_t)$ from human data $\mathcal{D}_{\pi^*} = \{o_1, u_1, ..., o_N, u_N\}$
2. run $\pi_\theta(u_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., o_M\}$
3. Ask human to label $\mathcal{D}_\pi$ with actions u_t
4. Aggregate: $\mathcal{D}_{\pi^*} \leftarrow \mathcal{D}_{\pi^*} \cup \mathcal{D}_\pi$
5. GOTO step 1.

Notice you might not actually need a human here - if your states are discretized, and you have enough data, you might get this by matching

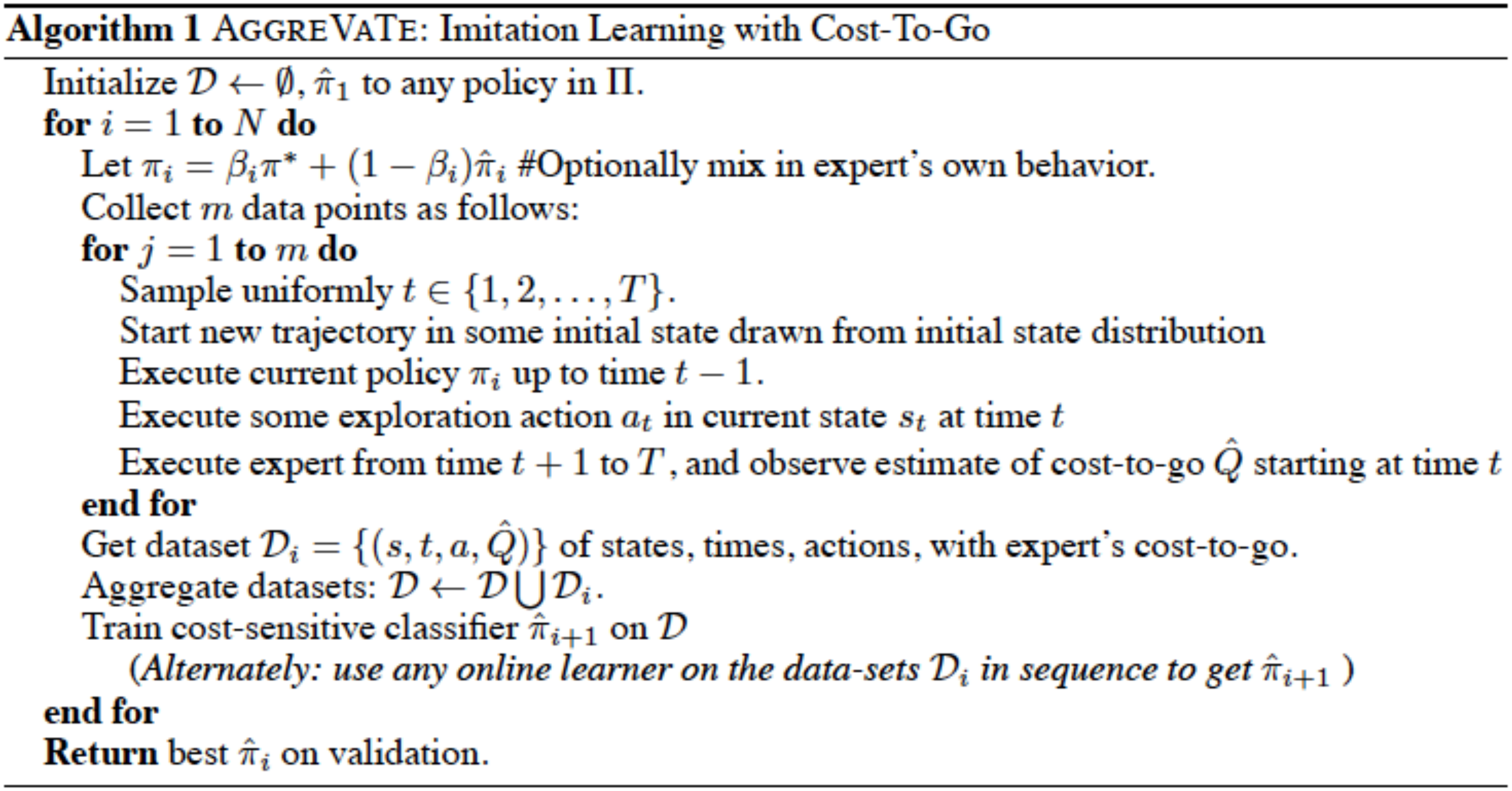
Problems:

- execute an unsafe/partially trained policy
- repeatedly query the expert

Fragkiadaki, ND

A Reduction of Imitation Learning and Structured Predictioı

Aggrevate

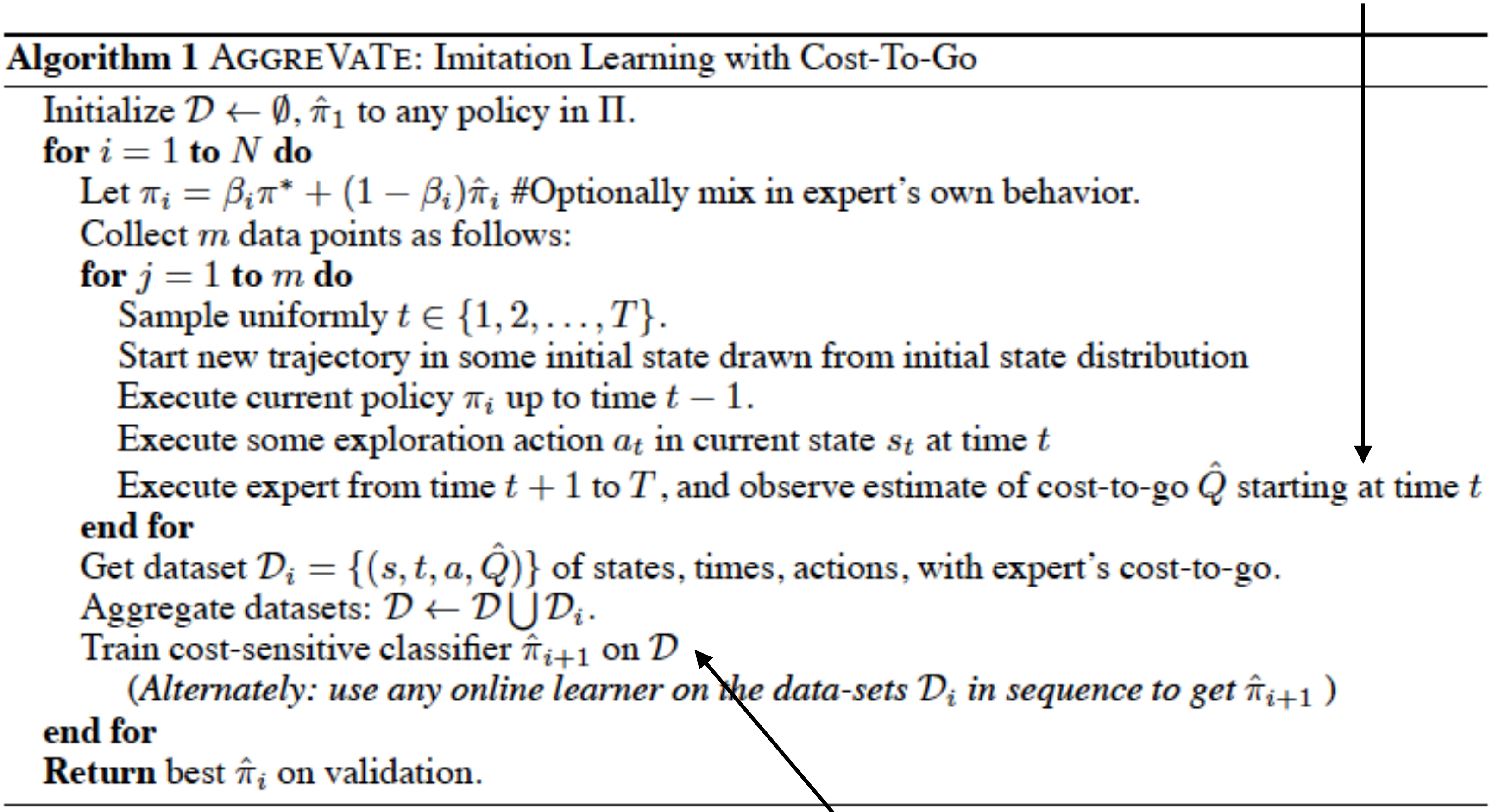


Algorithm 1 AGGREVATE: Imitation Learning with Cost-To-Go

```
Initialize $\mathcal{D} \leftarrow \emptyset$, $\hat{\pi}_1$ to any policy in $\Pi$.
for $i = 1$ to $N$ do
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$ #Optionally mix in expert's own behavior.
    Collect $m$ data points as follows:
    for $j = 1$ to $m$ do
        Sample uniformly $t \in \{1, 2, \ldots, T\}$.
        Start new trajectory in some initial state drawn from initial state distribution
        Execute current policy $\pi_i$ up to time $t - 1$.
        Execute some exploration action $a_t$ in current state $s_t$ at time $t$
        Execute expert from time $t + 1$ to $T$, and observe estimate of cost-to-go $\hat{Q}$ starting at time $t$
    end for
    Get dataset $\mathcal{D}_i = \{(s, t, a, \hat{Q})\}$ of states, times, actions, with expert's cost-to-go.
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
    Train cost-sensitive classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$
        (Alternately: use any online learner on the data-sets $\mathcal{D}_i$ in sequence to get $\hat{\pi}_{i+1}$ )
end for
Return best $\hat{\pi}_i$ on validation.
```

Aggrevate

Notice you might not actually need a human here - if your states are discretized, and you have enough data you might get this by matching



Algorithm 1 AGGREVATE: Imitation Learning with Cost-To-Go

```
Initialize $\mathcal{D} \leftarrow \emptyset$, $\hat{\pi}_1$ to any policy in $\Pi$.
for $i = 1$ to $N$ do
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$ #Optionally mix in expert's own behavior.
    Collect $m$ data points as follows:
    for $j = 1$ to $m$ do
        Sample uniformly $t \in \{1, 2, \ldots, T\}$.
        Start new trajectory in some initial state drawn from initial state distribution
        Execute current policy $\pi_i$ up to time $t - 1$.
        Execute some exploration action $a_t$ in current state $s_t$ at time $t$
        Execute expert from time $t + 1$ to $T$, and observe estimate of cost-to-go $\hat{Q}$ starting at time $t$
    end for
    Get dataset $\mathcal{D}_i = \{(s, t, a, \hat{Q})\}$ of states, times, actions, with expert's cost-to-go.
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
    Train cost-sensitive classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$
        (Alternately: use any online learner on the data-sets $\mathcal{D}_i$ in sequence to get $\hat{\pi}_{i+1}$ )
end for
Return best $\hat{\pi}_i$ on validation.
```

i.e. classifier minimizes sum of costs, not zero-one loss

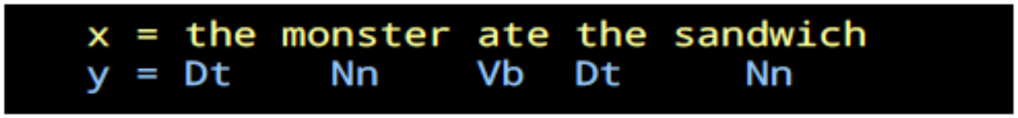
Aggrevated

- With properly chosen policy, can differentiate loss
  - from aggrevate
  - wrt parameters
  - typically, policy is deep network
- Details
  - paper

Structured prediction

Structured prediction: a learner makes predictions over a set of interdependent output variables and observes a joint loss.

Example: part of speech tagging



```
x = the monster ate the sandwich
y = Dt    Nn      Vb  Dt     Nn
```

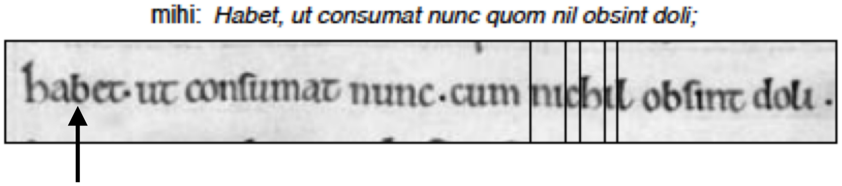
A structured prediction problem consists of an *input space* $\mathcal{X}$, an *output space* $\mathcal{Y}$, a fixed but unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, and a non-negative *loss function* $l(y^*, \hat{y}) \rightarrow \mathbb{R}^{\geq 0}$ which measures the distance between the true $y*$ and predicted $\hat{y}$ outputs. The goal of structured learning is to use N samples $(x_i, y_i)_{i=1}^{N}$ to learn a mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$ that minimizes the expected structured loss under $\mathcal{D}$.

Fragkiadaki, ND

Structured prediction examples

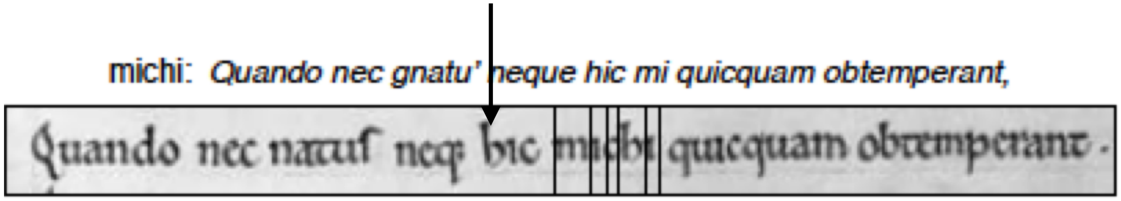
- Label a sequence of words with part of speech tags
- Predict line of text from a line of ink

- In each case:
  - We must map a sequence to a sequence
    - just like in choice of steering angle
  - The “future” affects current decisions
    - just like in choice of steering angle
  - We have a bunch of labelled examples
    - just like in choice of steering angle


mihi: Habet, ut consumat nunc quom nil obsint doli;
habet· ut consumat nunc· cum michi obsint doli ·

A ‘b’

Not a ‘b’


michi: Quando nec gnatu' neque hic mi quicquam obtemperant,
Quando nec natus neq hic michi quicquam obtemperant ·

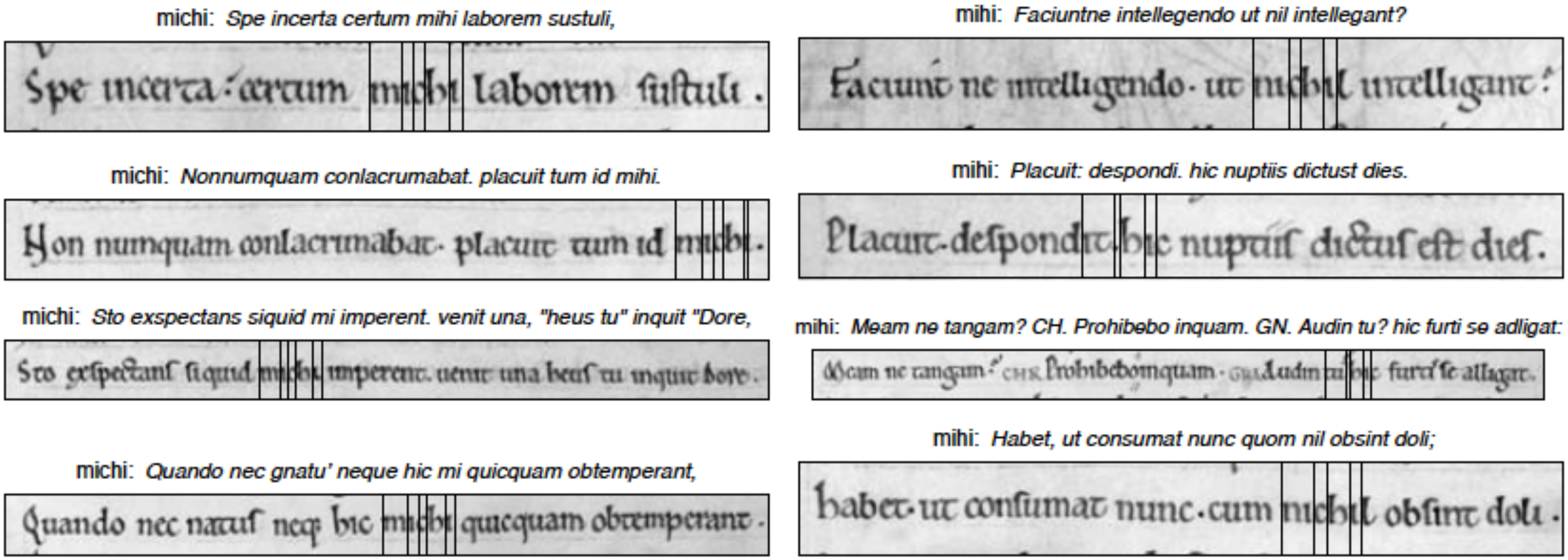

michi: Spe incerta certum mihi laborem sustuli,
michi: Nonnumquam conlacrumabat. placuit tum id mihi.
michi: Sto exspectans siquid mi imperent. venit una, "heus tu" inquit "Dore,
michi: Quando nec gnatu' neque hic mi quicquam obtemperant,
mihi: Faciuntne intellegendo ut nil intellegant?
mihi: Placuit: despondi. hic nuptiis dictust dies.
mihi: Meam ne tangam? CH. Prohibebo inquam. GN. Audin tu? hic furti se adligat:
mihi: Habet, ut consumat nunc quom nil obsint doli;

Figure 7: *The handwritten text does not fully correspond to the transcribed version; for example, scribes commonly write “michi” for the standard “mihi”. Our search process reflects the ink fairly faithfully, however.* **Left** *the first four lines returned for a search on the string “michi”;* **right** *the first four lines returned for a search on the string “mihi”, which does not appear in the document. Note that our search process can offer scholars access to the ink in a particular document, useful for studying variations in transcription, etc.*

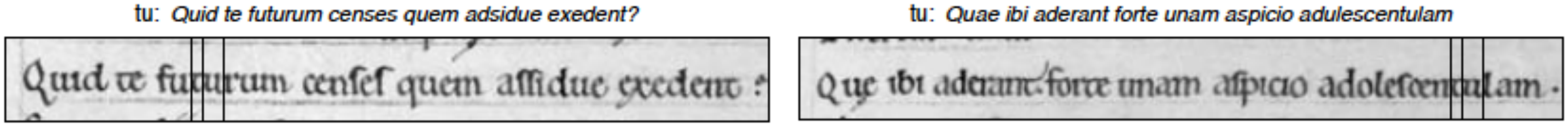

tu: Quid te futurum censes quem adsidue exedent?
tu: Quae ibi aderant forte unam aspicio adulescentulam

Figure 8: *Searches on short strings produce substrings of words as well as words (we show the first two lines returned from a search for "tu").*

Strategy - I

Ink (known)

Frame (known)

- Construct a parametric cost function

$$H(\mathcal{X}, \mathcal{Y}; \theta)$$

String (unknown)

Steering (unknown)

- Inference:
  - Choose the best string

$$Y = \underset{\mathcal{Y}}{\arg\min}\ \mathcal{H}(\mathcal{X}, \mathcal{Y}; \theta)$$

For sequences

- Some natural choices
    - cost function has form

$$V(\overset{\substack{\text{Ink (known)}\\ \downarrow}}{x_1}, \underset{\substack{\uparrow \\ \text{Text (unknown)}}}{y_1}; \theta) + E(y_1, y_2; \theta) + V(x_2, y_2; \theta) + E(y_2, y_3; \theta) + \ldots$$

Notice this term reflects the “effect of the future”

For sequences

- Natural, because inference is easy
  - dynamic programming

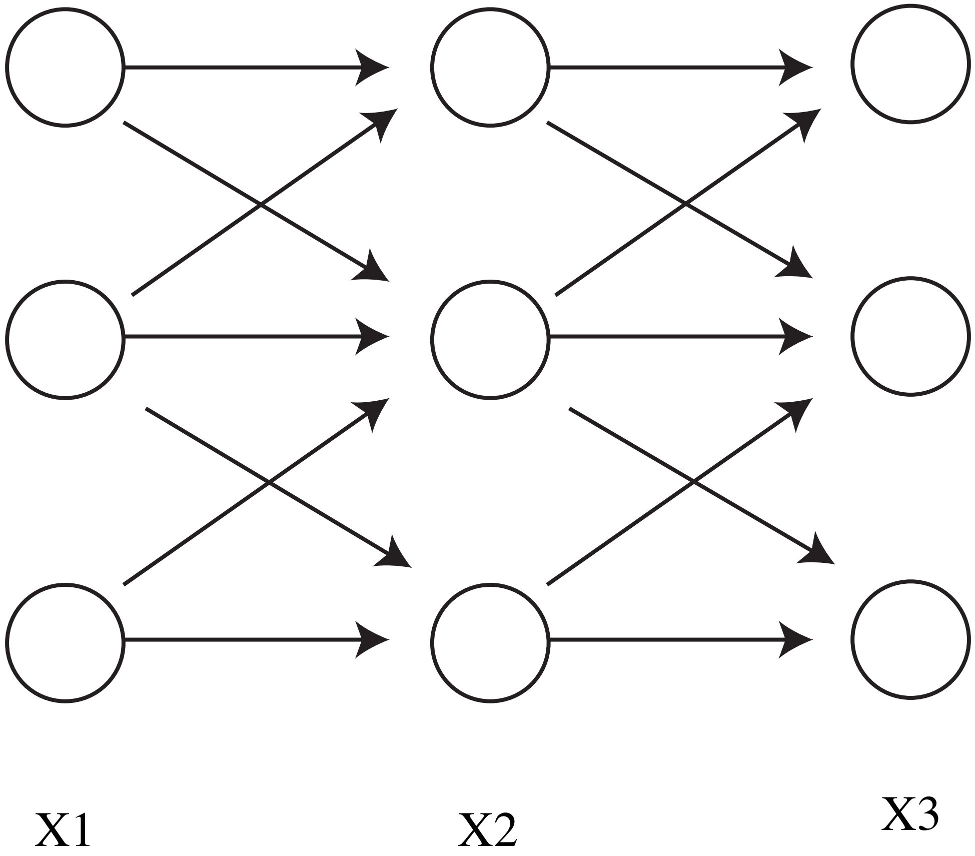
Ink (known)

$$V(x_1, y_1; \theta) + E(y_1, y_2; \theta) + V(x_2, y_2; \theta) + E(y_2, y_3; \theta) + \ldots$$

Text (unknown)

Notice this term reflects the “effect of the future”

$$V(x_1, y_1; \theta) + E(y_1, y_2; \theta) + V(x_2, y_2; \theta) + E(y_2, y_3; \theta) + \ldots$$



But we don’t know V, E !

- We do have examples (X*, Y*)
- Idea:
  - Choose V, E so that:
    - Inference on X* yields Y*

Strategy for structured prediction

- Construct a parametric cost function $H(\mathcal{X}, \mathcal{Y}; \theta)$

- So that, for training X*

$$\underset{\mathcal{Y}}{\text{argmin}} \; H(\mathcal{X}^*, \mathcal{Y}; \theta)$$

- is close to correct Y*
  - (see movies for some details on construction)

Fragkiadaki, ND

For sequences

- Some natural choices
  - cost function:

$$V(x_1, y_1; \theta) + E(y_1, y_2; \theta) + V(x_2, y_2; \theta) + E(y_2, y_3; \theta) + \ldots$$

  - Make V, E linear in theta
    - might involve complicated feature constructions
    - BUT simplifies learning

This yields

- The cost function has the form

$$\mathcal{H}(x, y; \theta) = \theta^T G(x, y)$$

- Choose theta so that for all training pairs x*, y*

$$\theta^T G(x^*, y^*) \leq \theta^T G(x^*, y)$$

- Note
  - this isn’t one inequality - it’s one inequality per possible y!
  - also, likely not feasible
  - also, doesn’t prefer y’s that are “close” to y*

So rearrange inequalities

- Force G(x*, y) to grow:

$$\theta^T G(x^*, y^*) + \epsilon D(y, y*) \leq \theta^T G(x^*, y)$$

- Rearrange, slack variable, and deal with many y:

$$\xi = (\max(0, \ \underset{y}{\max} \ \theta^T (G(x^*, y^*) - G(x^*, y)) + \epsilon D(y, y^*)$$

And now solve optimization problem

Don’t choose large theta - this helps generalization

$$\frac{1}{2}\theta^T\theta + \sum_i \xi_i$$

$$\xi_i = (\max(0, \ \max_y \ \theta^T(G(x^*{}_i, y^*{}_i) - G(x^*{}_i, y)) + \epsilon D(y, y^*{}_i)$$

Which is much nastier than it looks

Don’t choose large theta - this helps generalization

$$\frac{1}{2}\theta^T\theta + \sum_i \xi_i$$

$$\xi_i = (\max(0, \ \max_y \ \theta^T(G({x^*}_i, {y^*}_i) - G({x^*}_i, y)) + \epsilon D(y, {y^*}_i)$$

To take a step, we’ll need to know the sequence that maximizes this

Strategy

- Subgradient descent
  - slacks aren’t differentiable, but it doesn’t really matter (piecewise linear)
  - when you know the maximising y, the slacks are linear in theta
- Repeat
  - pass through data, computing maximizing y
    - can be brutally expensive
  - this gives slacks as linear function of theta
  - differentiate, take a gradient step

Applying this to predicting angle

- Simplest case:

$$\mathcal{H}(\mathcal{X},\mathcal{Y};\theta) = \sum_{i} F(X_i, Y_i; \theta)$$

Index gives frame

- We've actually done this
  - minimizing == choose the best angle for the current frame
  - and this has problems because it doesn't take future into account

Applying this to predicting angle

- Simplest case:

$$\mathcal{H}(\mathcal{X},\mathcal{Y};\theta) = \sum_{\underset{\uparrow}{i}} F(X_i, Y_{i-1}, Y_i;\theta)$$

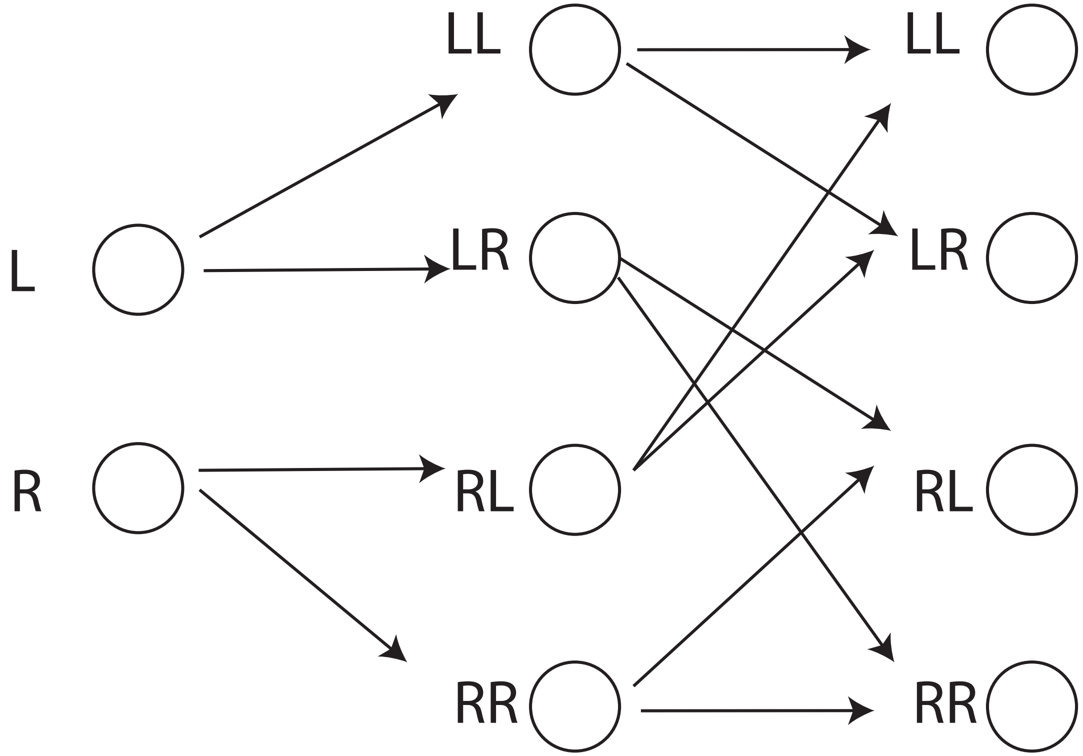
Index gives frame

- Now things get interesting
  - chosen angle depends on previous angle
  - inference
    - dynamic programming still works (delicate)
  - learning
    - as above
  - Q: what about more previous angles?

Dynamic programming

$$\mathcal{H}(\mathcal{X},\mathcal{Y};\theta) = \sum_i F(X_i, Y_{i-1}, Y_i;\theta)$$

Make “stacked” states



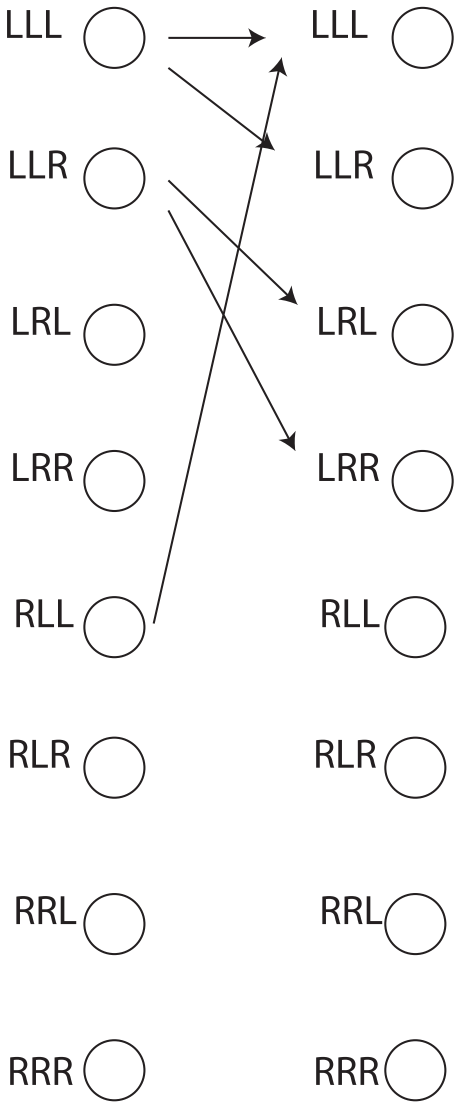
More previous angles…

- Q: Can you do this?
  - Yes - as per stacked state argument
- Q: Should you do this?
  - Likely yes - roads have long scale structure, so should be able to smooth

Dynamic programming

$$\mathcal{H}(\mathcal{X},\mathcal{Y};\theta) = \sum_i F(X_i, Y_{i-2}, Y_{i-1}, Y_i; \theta)$$

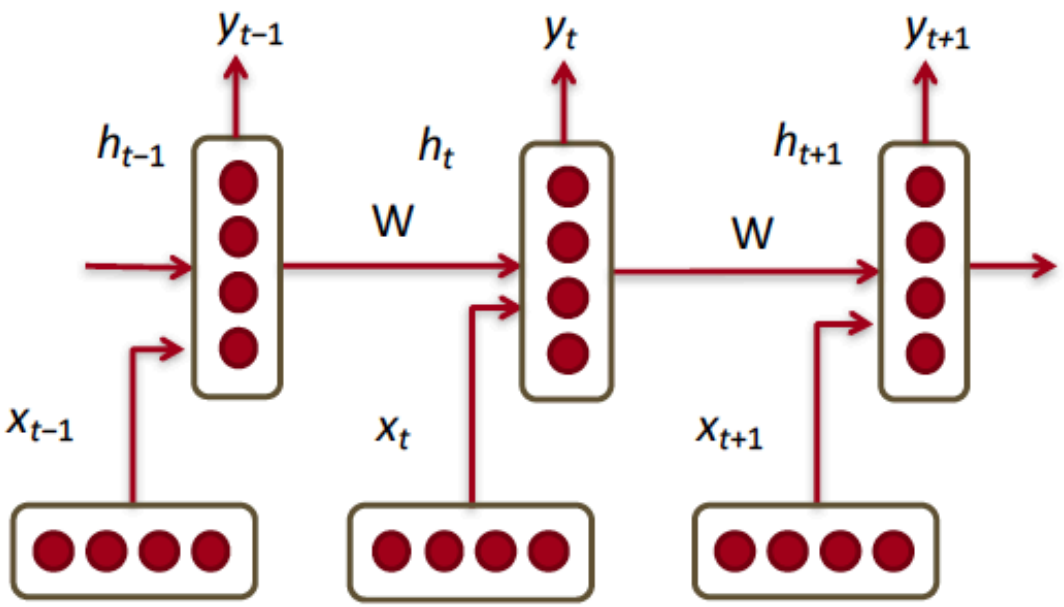
Make “stacked” states



Recurrent Neural Networks

- RNNs tie the weights at each time step

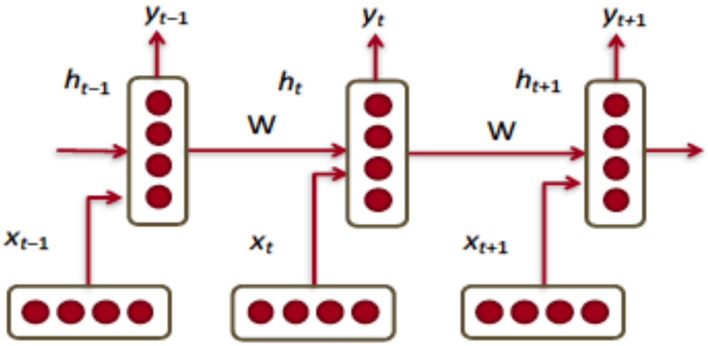
- Condition the neural network on all previous inputs
- In principle, any interdependencies can be modeled between inputs and outputs, as well as between output labels.
- In practice, limitations from SGD training, capacity, initialization etc.



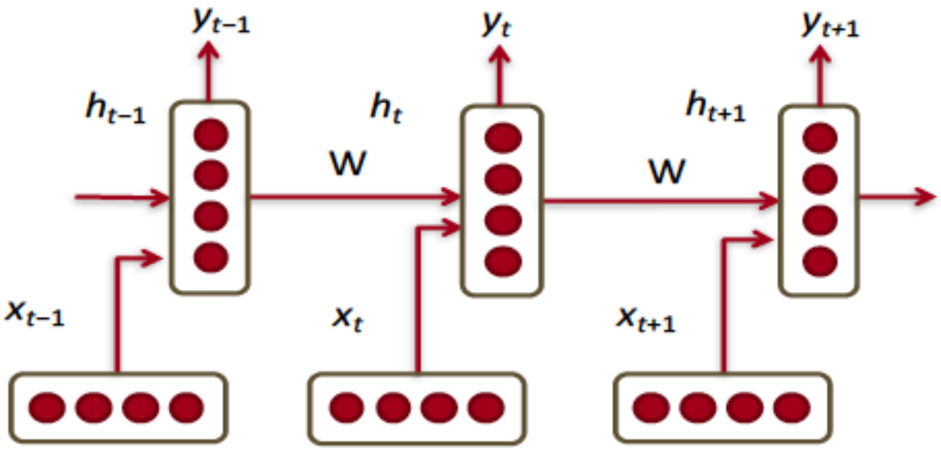
Fragkiadaki, ND

Recurrent Neural Networks

For sequence labelling problems, actions of the labelling policies are y_t, e.g., part of speech tags



For sequence generation, actions of the labelling policies are $y_t = x_{t+1}$, e.g., word in answer generation $\hat{P}(x_{t+1} = v_j | x_t, ..., x_1) = \hat{y}_{t,j}$



Fragkiadaki, ND

Recurrent Neural Networks

The network is typically trained to maximize the log-likelihood of the output sequences given the input sequences of a training set $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$:

$$\theta^* = \arg\max_\theta \log \sum_{(x^{(i)},y^{(i)})\in\mathcal{D}} P_\theta(y^{(i)}, x^{(i)})$$

If the likelihood of an example decomposes over individual time steps:

$$\log P_\theta(y|x) = \sum_t \log P_\theta(y_t|h_t)$$

Else loss is computed at the end of the sequence and is back propagated through time.

A learned policy is the inference function of the model:

$$\hat{\theta}(h_t) = \arg\max_y P(y_t = y|h_t; \theta)$$

The reference policy is the policy that always outputs the true labels:

$$\theta^*(h_t) = y_t$$

Fragkiadaki, ND

Recurrent Neural Networks

The regular training procedure of RNNs treat true labels y_t as actions while making forward passes. Hence, the learning agent follows trajectories generated by the reference policy rather than the learned policy. In other words, it learns:

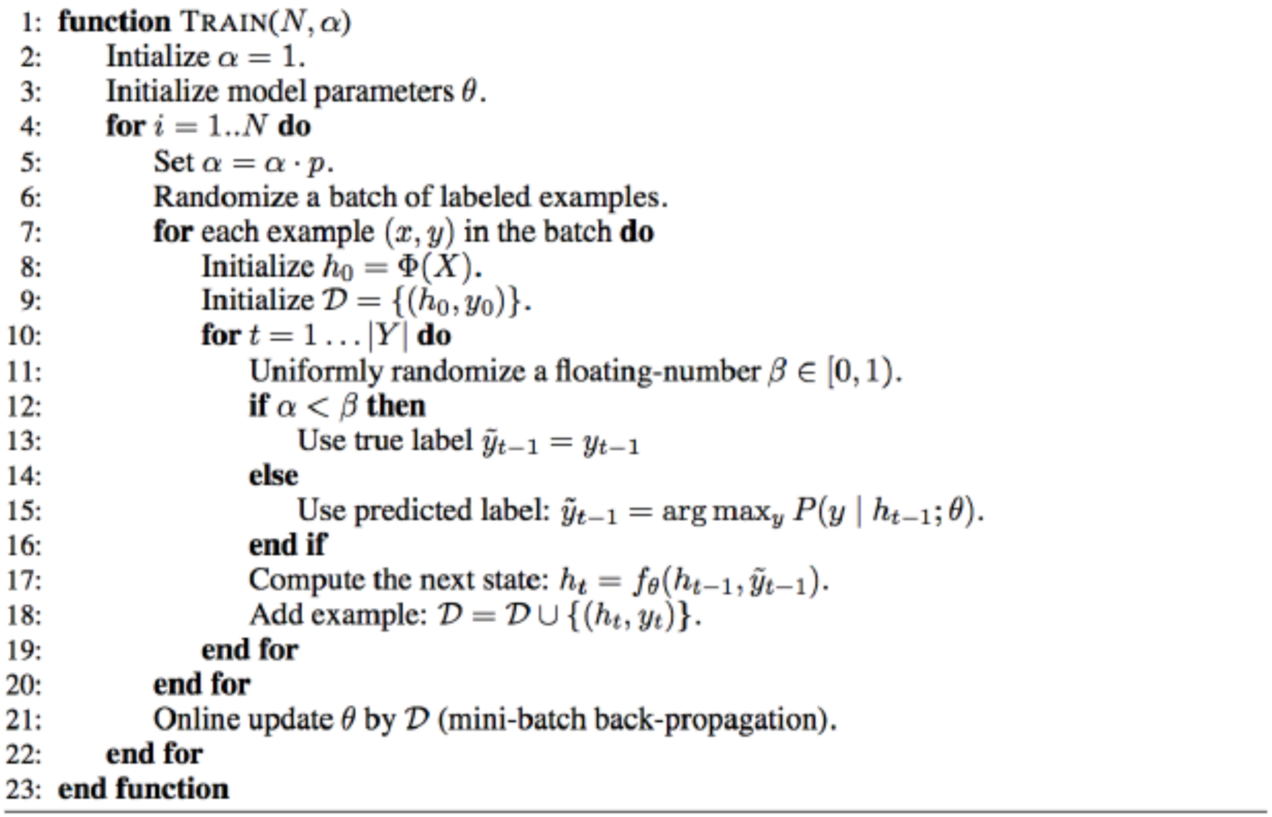
$$\hat{\theta}^{sup} = \arg\min_{\theta} \mathbb{E}_{h \sim d_{\pi *}} [l_\theta(h)]$$

However, our true goal is to learn a policy that minimizes error under its own induced state distribution:

$$\hat{\theta} = \arg\min_{\theta} \mathbb{E}_{h \sim d_{\theta}} [l_\theta(h)]$$

Imitation Learning with Recurrent Neural Networks, Nyuyen 2016

DAGGER for sequence labelling/generation with RNNs



```
function TRAIN($N, \alpha$)
    Intialize $\alpha = 1$.
    Initialize model parameters $\theta$.
    for $i = 1..N$ do
        Set $\alpha = \alpha \cdot p$.
        Randomize a batch of labeled examples.
        for each example $(x, y)$ in the batch do
            Initialize $h_0 = \Phi(X)$.
            Initialize $\mathcal{D} = \{(h_0, y_0)\}$.
            for $t = 1 \ldots |Y|$ do
                Uniformly randomize a floating-number $\beta \in [0, 1)$.
                if $\alpha < \beta$ then
                    Use true label $\tilde{y}_{t-1} = y_{t-1}$
                else
                    Use predicted label: $\tilde{y}_{t-1} = \arg\max_y P(y \mid h_{t-1}; \theta)$.
                end if
                Compute the next state: $h_t = f_\theta(h_{t-1}, \tilde{y}_{t-1})$.
                Add example: $\mathcal{D} = \mathcal{D} \cup \{(h_t, y_t)\}$.
            end for
        end for
        Online update $\theta$ by $\mathcal{D}$ (mini-batch back-propagation).
    end for
end function
```

Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks, Bengio(Samy) et al.

Imitation Learning with Recurrent Neural Networks, Nyuyen 2016

Fragkiadaki, ND

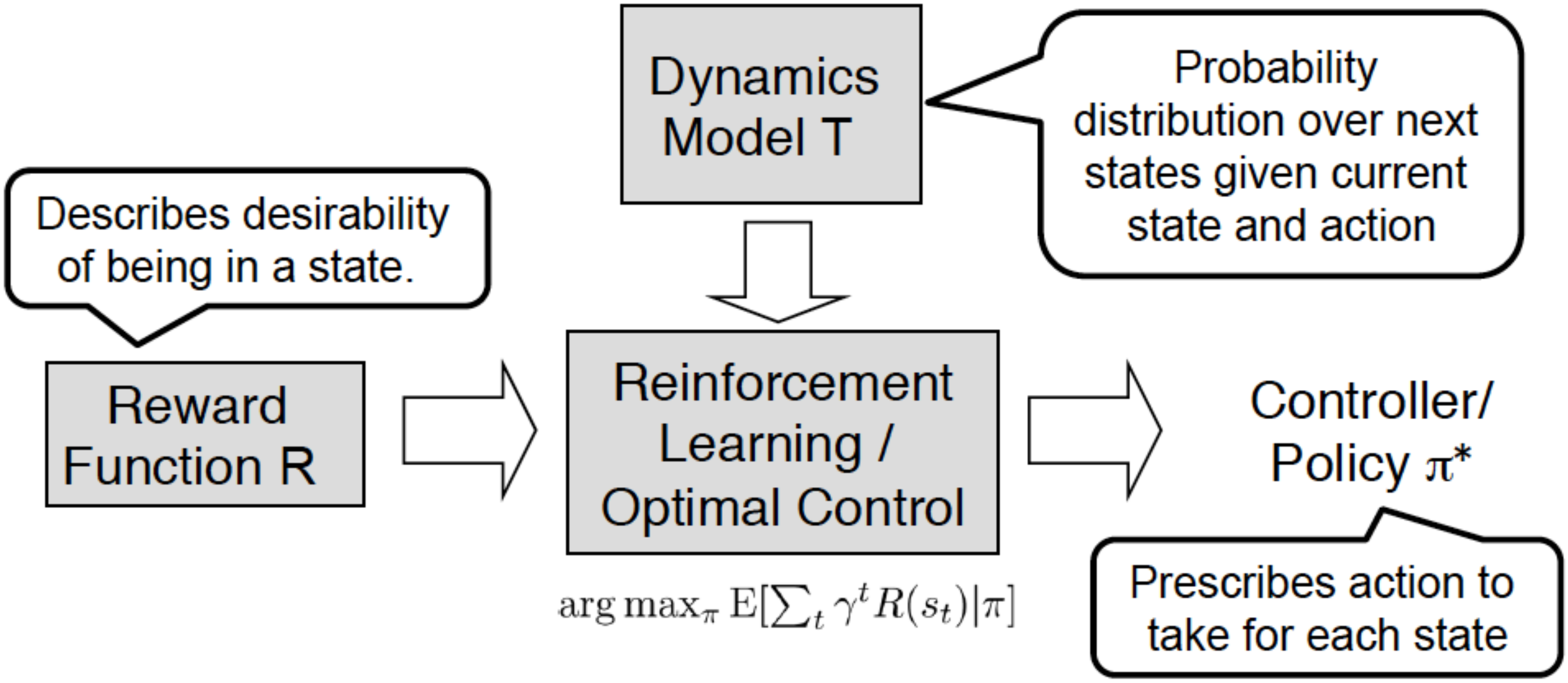
Imitation Learning

Two broad approaches :

- Direct: Supervised training of policy (mapping states to actions) using the demonstration trajectories as ground-truth (a.k.a. behavior cloning)
- **Indirect**: Learn the unknown reward function/goal of the teacher, and derive the policy from these, a.k.a. **Inverse Reinforcement Learning**

Mitchell, via Fragkiadaki

Inverse Reinforcement Learning



Dynamics Model T

Probability distribution over next states given current state and action

Describes desirability of being in a state.

Reward Function R

Reinforcement Learning / Optimal Control

$\arg\max_\pi \mathrm{E}[\sum_t \gamma^t R(s_t)|\pi]$

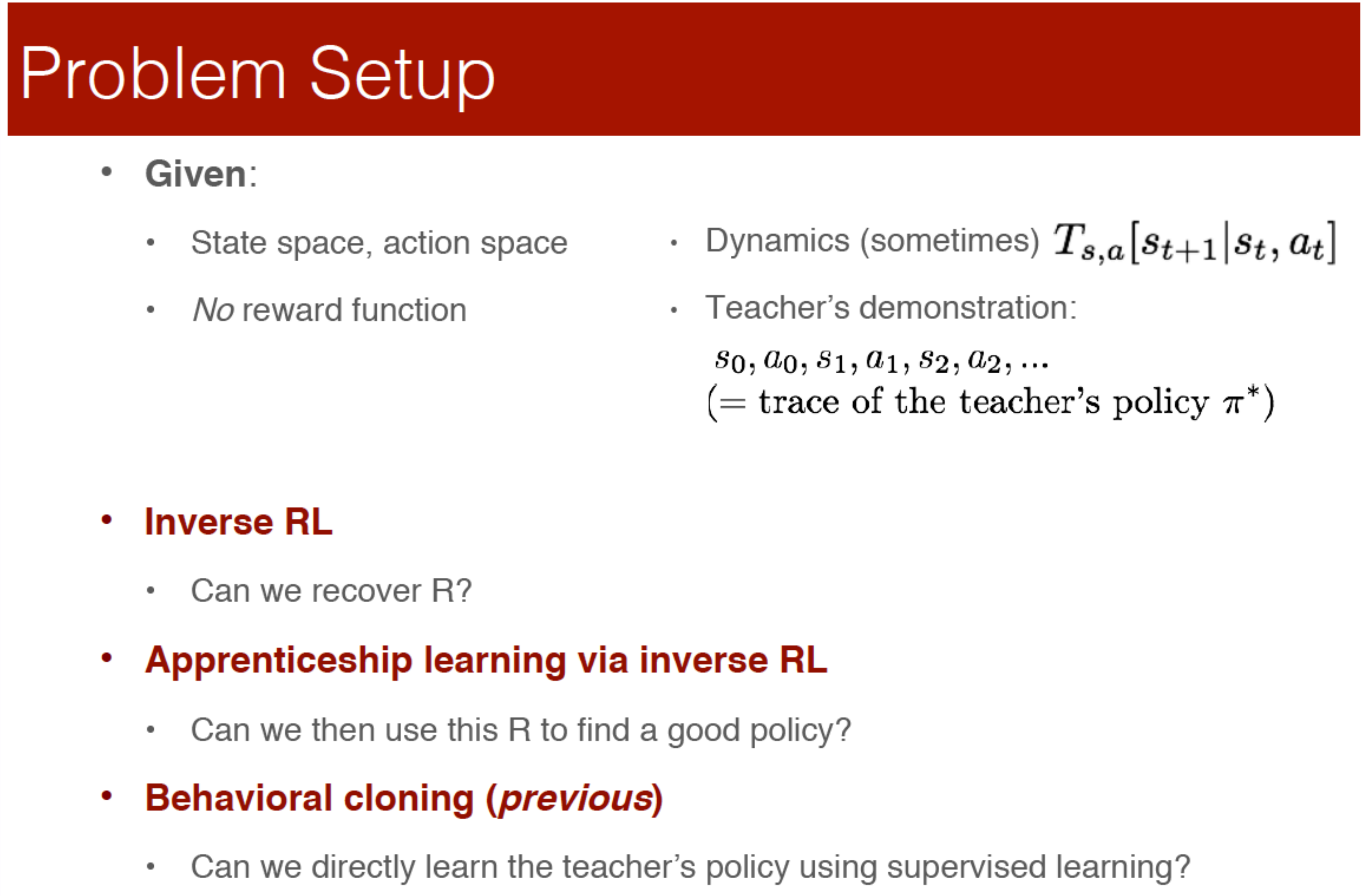
Controller/ Policy π*

Prescribes action to take for each state

Diagram: Pieter Abbeel

Actually, we don't really have pi; we have observations of what happens under pi, which is not quite the same thing

Given π, let's recover R!



Problem Setup

- **Given**:
  - State space, action space
  - *No* reward function
  - Dynamics (sometimes) $T_{s,a}[s_{t+1}|s_t, a_t]$
  - Teacher's demonstration:

    $s_0, a_0, s_1, a_1, s_2, a_2, \ldots$
    $(= \text{trace of the teacher's policy } \pi^*)$

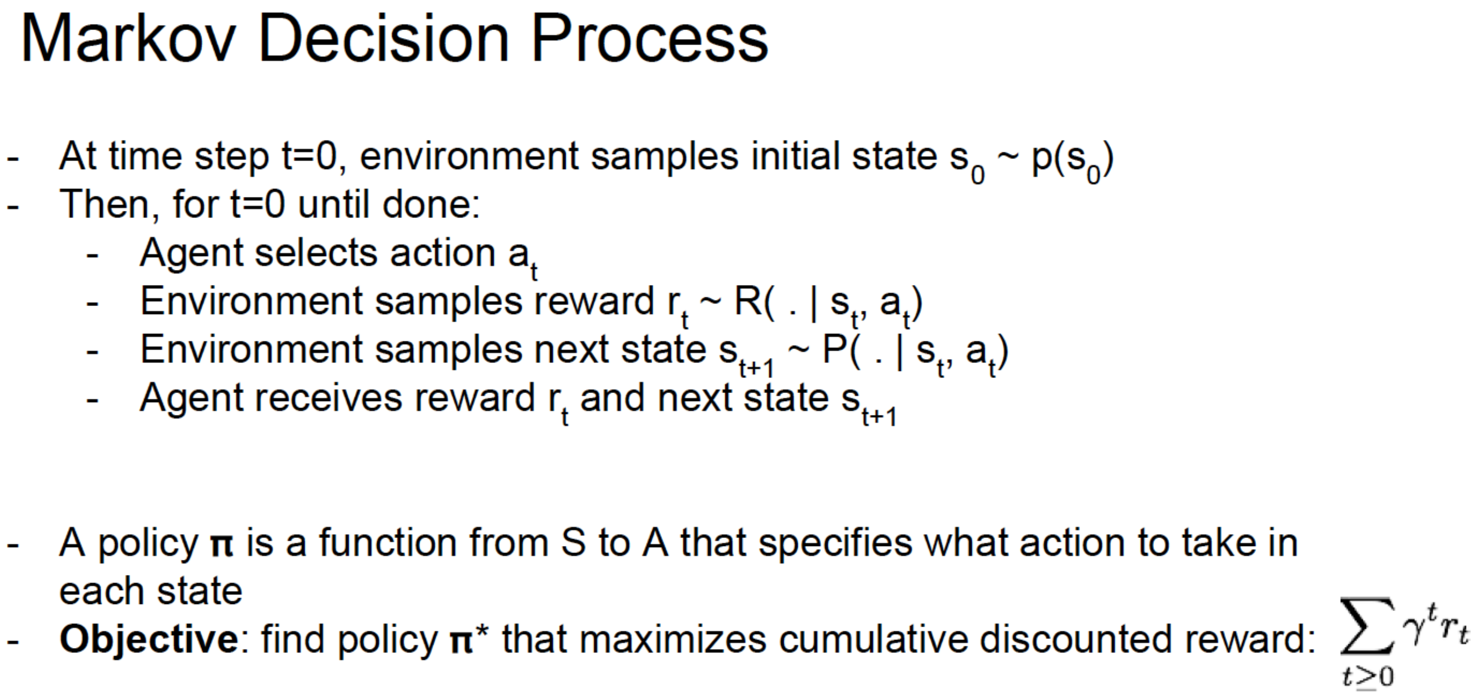
- **Inverse RL**
  - Can we recover R?
- **Apprenticeship learning via inverse RL**
  - Can we then use this R to find a good policy?
- **Behavioral cloning (*previous*)**
  - Can we directly learn the teacher's policy using supervised learning?

This is really like structured prediction

LEARCH=IRL via structured prediction

- Adopt dual representation of policies in MDP
- Then it all boils down to what we’ve seen

Dual representation in MDPs



Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
    - Agent selects action a_t
    - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
    - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
    - Agent receives reward r_t and next state s_{t+1}

- A policy **π** is a function from S to A that specifies what action to take in each state
- **Objective**: find policy **π*** that maximizes cumulative discounted reward: $\sum_{t \geq 0} \gamma^t r_t$

You can represent a policy by the distribution state-action pairs that arises, via a fairly fiddly duality argument. Such representations are constrained.

Dual representations of policies

- We’re interested in policies that go from start to goal
  - and are deterministic and acyclic
  - you can represent these with an indicator vector for each state action pair
    - 1 if in that state you do that action
    - 0 otherwise
- Big point
  - assume we have a cost for each state action pair
    - write c
  - then cost of policy mu is

$$\mu^T \mathbf{c}$$

Cost is some function of state, action

- Assume that it is linear in features
  - so

$$\mathbf{c} = w^T \mathcal{F}$$

This is a matrix of features, and it may change from instance to instance (obstacles in different places, etc.)

- We know features for any instance,
  - but we don't know w
  - we assume that w is the same across instances
  - and we have seen experts

The cost incurred by an expert

- Cost for instance i takes the form

Unknown

$$\mathbf{c}^T \mu_i = w^T \mathcal{F} \mu_i$$

Known

- Assume each time the expert does the optimal thing

$$\mathbf{c}^T \mu_i \leq \mathbf{c}^T \mu$$

For ANY other mu!

But this is what we saw before…

- Details in Learch paper