Learning to Search

CS 6355: Structured Prediction

Some slides adapted from Daumé and Ross

Inference

- What is inference?
  - An overview of what we have seen before
  - Combinatorial optimization
  - Different views of inference
- Graph algorithms
  - Dynamic programming, greedy algorithms, search
- Integer programming
- Heuristics for inference
  - Sampling
- Learning to search

Learning to Search

We have seen that inference as graph search

- Iteratively construct a series of partial structures
- Find the highest scoring structure in this fashion

Can we learn a model that is designed with such inference in mind?

- Learning to search is a way of formulating structured prediction problems as a search problem
- Integrates learning and prediction into a unified framework

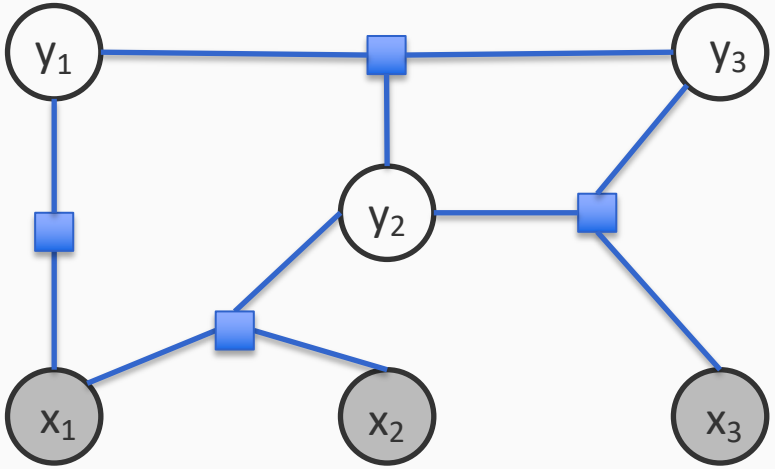
Overview

1. Preliminaries
   - Learning to minimize costs
   - Search problems and a generic search algorithm
2. Learning to search: A general formulation
3. LaSO: Learning as Search Optimization
4. SEARN: Search and Learning
5. DAgger: Dataset Aggregation

Overview

1. Preliminaries
   - Learning to minimize costs
   - Search problems and a generic search algorithm
2. Learning to search: A general formulation
3. LaSO: Learning as Search Optimization
4. SEARN: Search and Learning
5. DAgger: Dataset Aggregation

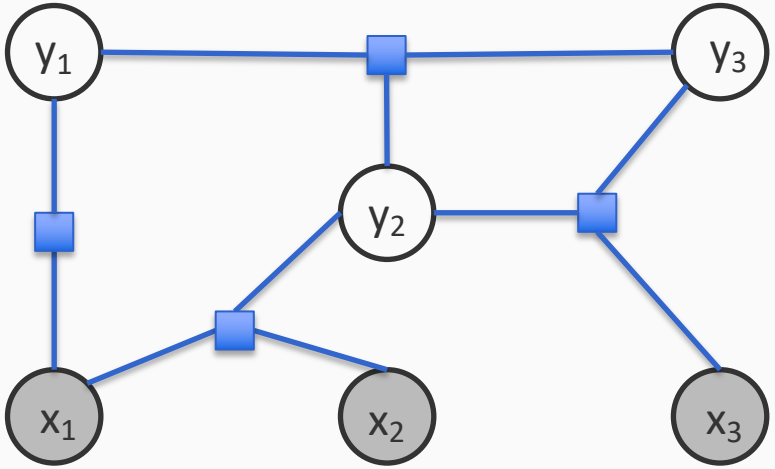
Learning to minimize prediction cost



$$\mathbf{y} = (y_1, y_2, y_3)$$

Suppose each y can be one of A, B or C, and the true label is $(y_1 = \mathrm{A}, y_2 = \mathrm{B}, y_3 = \mathrm{C})$

Learning to minimize prediction cost



$$\mathbf{y} = (y_1, y_2, y_3)$$

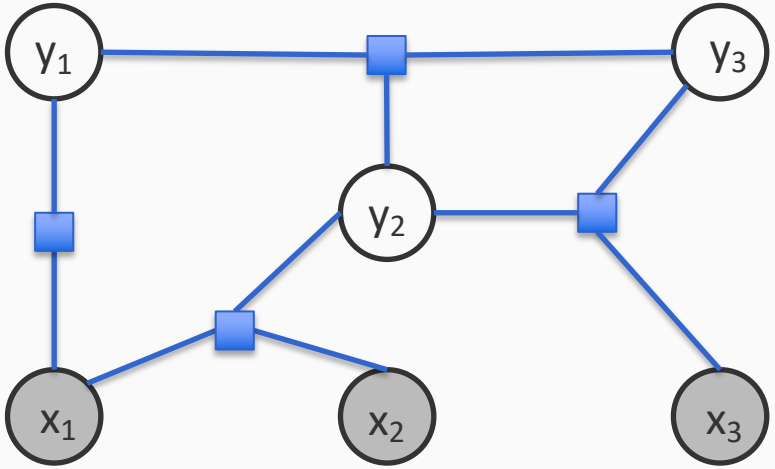
Suppose each y can be one of A, B or C, and the true label is $(y_1 = \mathrm{A}, y_2 = \mathrm{B}, y_3 = \mathrm{C})$

The goal: Learn a classifier that has lowest cost

The cost vector for this input x can be:

$$c(A, A, A) = 1$$
$$c(A, A, B) = 1$$
$$c(A, A, C) = 1$$
$$\ldots$$
$$c(A, B, C) = 0$$
$$\ldots$$
$$c(C, C, B) = 1$$
$$c(C, C, C) = 1$$

Learning to minimize prediction cost



$$\mathbf{y} = (y_1, y_2, y_3)$$

Suppose each y can be one of A, B or C, and the true label is $(y_1 = \mathrm{A}, y_2 = \mathrm{B}, y_3 = \mathrm{C})$

The goal: Learn a classifier that has lowest cost

The cost vector for this input x can be:

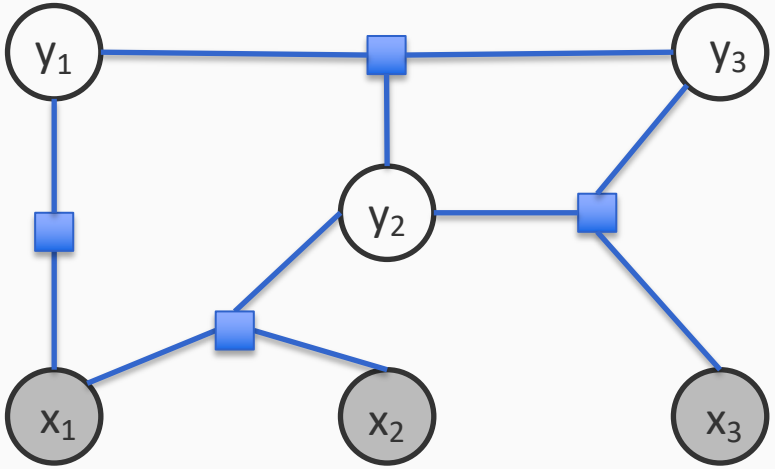
$$\begin{aligned} c(A,A,A) &= 1 \\ c(A,A,B) &= 1 \\ c(A,A,C) &= 1 \\ &\dots \\ c(A,B,C) &= 0 \\ &\dots \\ c(C,C,B) &= 1 \\ c(C,C,C) &= 1 \end{aligned}$$

or

$$\begin{aligned} c(A,A,A) &= 2 \\ c(A,A,B) &= 2 \\ c(A,A,C) &= 1 \\ &\dots \\ c(A,B,C) &= 0 \\ &\dots \\ c(C,C,B) &= 3 \\ c(C,C,C) &= 2 \end{aligned}$$

Hamming Distance

Learning to minimize prediction cost



$$\mathbf{y} = (y_1, y_2, y_3)$$

Suppose each y can be one of A, B or C, and the true label is $(y_1 = \mathrm{A}, y_2 = \mathrm{B}, y_3 = \mathrm{C})$

The goal: Learn a classifier that has lowest cost

The cost vector for this input x can be:

$$c(A,A,A) = 1$$
$$c(A,A,B) = 1$$
$$c(A,A,C) = 1$$
$$\dots$$
$$c(A,B,C) = 0$$
$$\dots$$
$$c(C,C,B) = 1$$
$$c(C,C,C) = 1$$

or

$$c(A,A,A) = 2$$
$$c(A,A,B) = 2$$
$$c(A,A,C) = 1$$
$$\dots$$
$$c(A,B,C) = 0$$
$$\dots$$
$$c(C,C,B) = 3$$
$$c(C,C,C) = 2$$

Hamming Distance

What is the dimensionality of the cost vector c?

A Structured Prediction Problem

Learn a mapping $h(\mathbf{x})$ from inputs $\mathbf{x}$ to outputs $\mathbf{y}$

- Each $\mathbf{y}$ decomposes into decisions/labels $(y_1, y_2, \ldots, y_T)$
- Each $\mathbf{x}$ is associated with a cost vector $\mathbf{c}$
  - $\mathbf{c}$ has 2^{T} components if the y_i's are binary
    - Each component specifies the cost of the corresponding full assignment $\mathbf{y}$
  - Sometimes thought of as a function
- The goal is to minimize $L(h) = E[c_{h(x)}]$

Overview

1. Preliminaries
   - Learning to minimize costs
   - Search problems and a generic search algorithm
2. Learning to search: A general formulation
3. LaSO: Learning as Search Optimization
4. SEARN: Search and Learning
5. DAgger: Dataset Aggregation

Formalizing search problems

- Initial state: denoted by s_0
  - The starting point for the search
- Actions: Actions(s)
  - The set of actions that can be performed at a state
- Transition: Result(s, a)
  - "Applies" an action a to a state s to produce the next state
- Goal test: A check for whether the search is complete or not
- Path cost/score: A score for the path from the start state to any state

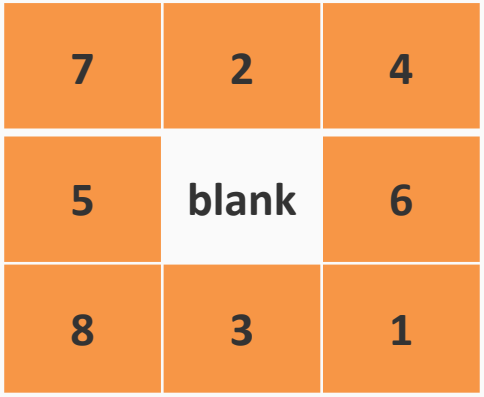
Formalizing search problems

- Initial state: denoted by s_0
  - The starting point for the search
- Actions: Actions(s)
  - The set of actions that can be performed at a state
- Transition: Result(s, a)
  - "Applies" an action a to a state s to produce the next state
- Goal test: A check for whether the search is complete or not
- Path cost/score: A score for the path from the start state to any state

A *solution* is an action sequence that leads from initial state to a goal state.

An *optimal solution* has the lowest path cost or highest score.

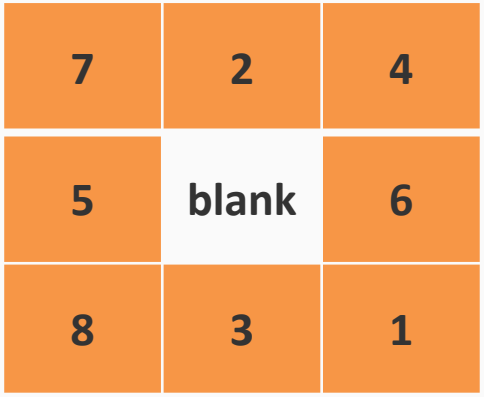
Example Search Problem: 8-puzzle



Initial State

Goal State

Example Search Problem: 8-puzzle



Initial State

Goal State

What are these five components for 8-puzzle?

Initial state: s_0
Actions: Actions(s)
Transition model: Result(s, a)
Goal test
Path cost / score

Generic Search Algorithm

How do we solve a search problem?

Answer: By starting at the initial state, and navigating the state space till we get to an answer

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

    if GoalTest(*node*) then **return** *node*

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

    if GoalTest(*node*) then **return** *node*

    next = Result(*node*, Actions(*node*))

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

    if GoalTest(*node*) then **return** *node*

    next = Result(*node*, Actions(*node*))

    nodes = *enqueue*(*problem, nodes, next*)

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

    if GoalTest(*node*) then **return** *node*

    next = Result(*node*, Actions(*node*))

    nodes = *enqueue*(*problem, nodes, next*)

return failure

Generic Search Algorithm

Algo Search(*problem, initial, enqueue*):

nodes = MakeQueue(MakeNode(*problem, initial*))

while *nodes* is not empty:

    node = Pop(*nodes*)

    if GoalTest(*node*) then **return** *node*

    next = Result(*node*, Actions(*node*))

    nodes = *enqueue*(*problem, nodes, next*)

return failure

All magic happens in the *enqueue* function (BFS, DFS, beam, A*)

Or is there any magic?

Overview

1. Preliminaries
   - Learning to minimize costs
   - Search problems and a generic search algorithm
2. Learning to search: A general formulation
3. LaSO: Learning as Search Optimization
4. SEARN: Search and Learning
5. DAgger: Dataset Aggregation

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

The high level idea:

- Frame the problem of structured prediction as a generic search problem

- Learn to enqueue nodes so that "good" states are explored first, and we get to a solution easily.

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$
- Initial state: Empty assignments $(-, -, \ldots, -)$

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$
- Initial state: Empty assignments $(-, -, \ldots, -)$
- Actions: Pick a y_i component and assign a label to it

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$
- Initial state: Empty assignments $(-, -, \ldots, -)$
- Actions: Pick a y_i component and assign a label to it
- Transition model: Move from one partial structure to another

Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$
- Initial state: Empty assignments $(-, -, \ldots, -)$
- Actions: Pick a y_i component and assign a label to it
- Transition model: Move from one partial structure to another
- Goal test: Whether all y components are assigned
  - A goal state does *not* need to be optimal

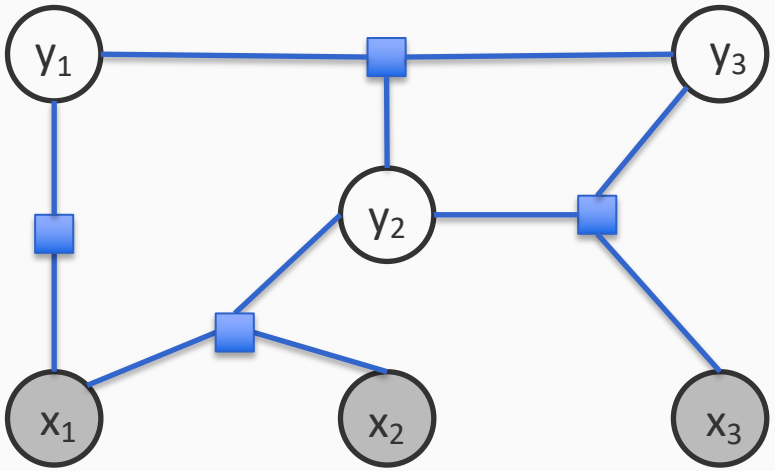
Learning to search: General setting

Predicting an output $\mathbf{y}$ as a sequence of decisions

General data structures

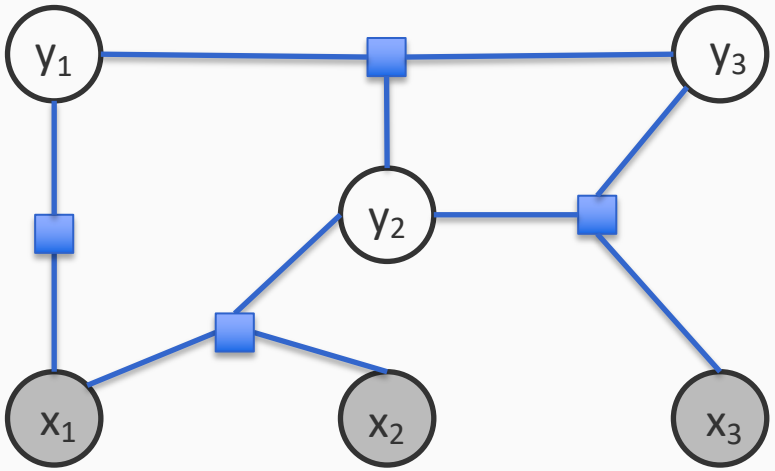
- State: Partial assignments to $(y_1, y_2, \ldots, y_T)$
- Initial state: Empty assignments $(-, -, \ldots, -)$
- Actions: Pick a y_i component and assign a label to it
- Transition model: Move from one partial structure to another
- Goal test: Whether all y components are assigned
  - A goal state does *not* need to be optimal
- Path cost/score function: $\mathbf{w}^T \phi(\mathbf{x}, \text{node})$
  - or, a neural network that depends on the $\mathbf{x}$ and the node
  - A node contains the current state and the back pointer to trace back the search path

Example



Suppose each y can be one of A, B or C

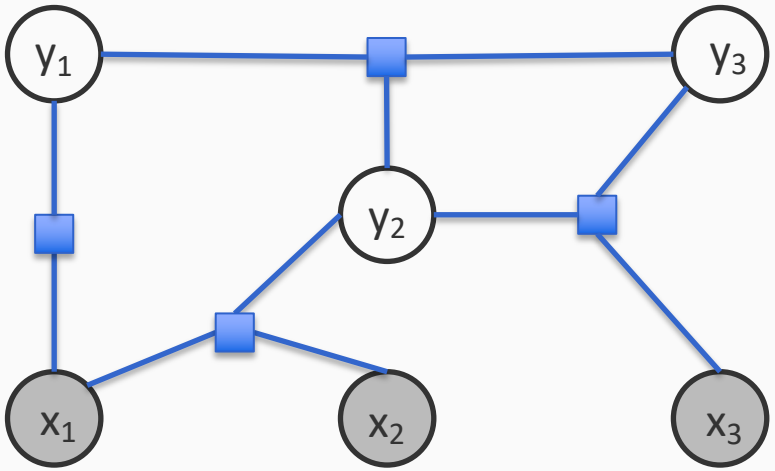
Example



Suppose each y can be one of A, B or C

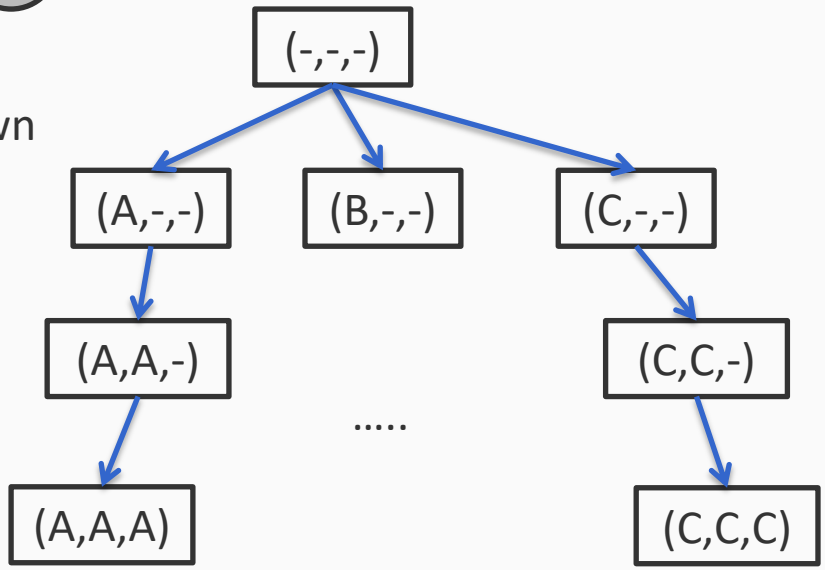
- State: Triples (y_1, y_2, y_3) all possibly unknown
  - (A, -, -), (-, A, A), (-, -, -),…
- Transition: Fill in one of the unknowns
- Start state: (-,-,-)
- End state: All three y's are assigned

Example



Suppose each y can be one of A, B or C

- State: Triples (y_1, y_2, y_3) all possibly unknown
  - (A, -, -), (-, A, A), (-, -, -),…
- Transition: Fill in one of the unknowns
- Start state: (-,-,-)
- End state: All three y's are assigned



1st Framework:

LaSO: Learning as Search Optimization

[Hal Daumé III and Daniel Marcu, ICML 2005]

The *enqueue* function in LaSO

The *enqueue* function in LaSO

- The goal of learning is to produce an *enqueue* function that
  - places good hypotheses high on the queue
  - places bad hypotheses low on the queue

The *enqueue* function in LaSO

- The goal of learning is to produce an *enqueue* function that
  - places good hypotheses high on the queue
  - places bad hypotheses low on the queue
- LaSO assumes enqueue is based on two components g + h

The *enqueue* function in LaSO

- The goal of learning is to produce an *enqueue* function that
  - places good hypotheses high on the queue
  - places bad hypotheses low on the queue
- LaSO assumes enqueue is based on two components g + h
  - g: path component. ($g = w^T \phi(x, node)$)

The *enqueue* function in LaSO

- The goal of learning is to produce an *enqueue* function that
  - places good hypotheses high on the queue
  - places bad hypotheses low on the queue
- LaSO assumes enqueue is based on two components g + h
  - g: path component. ($g = w^T \phi(x, node)$)
  - h: heuristic component. (h is given)
    - A^* if h is admissible, heuristic search if h is not admissible, best first search if h = 0, beam search if queue is limited.

The *enqueue* function in LaSO

- The goal of learning is to produce an *enqueue* function that
  - places good hypotheses high on the queue
  - places bad hypotheses low on the queue
- LaSO assumes enqueue is based on two components g + h
  - g: path component. ($g = w^T \phi(x, node)$)
  - h: heuristic component. (h is given)
    - A^* if h is admissible, heuristic search if h is not admissible, best first search if h = 0, beam search if queue is limited.

The goal is to learn w. How?

“y-good” node

“y-good” node

Assumption: for any given node s and an ***gold*** output y, we can tell whether s can or cannot lead to y.

“y-good” node

Assumption: for any given node s and an ***gold*** output y, we can tell whether s can or cannot lead to y.

Definition: The node s is y-good if s can lead to y

“y-good” node

Assumption: for any given node s and an ***gold*** output y, we can tell whether s can or cannot lead to y.

Definition: The node s is y-good if s can lead to y

$y = (y_1, y_2, y_3)$

Suppose each y can be one of A, B or C, and the true label is (y_1=A, y_2=B, y_3=C)

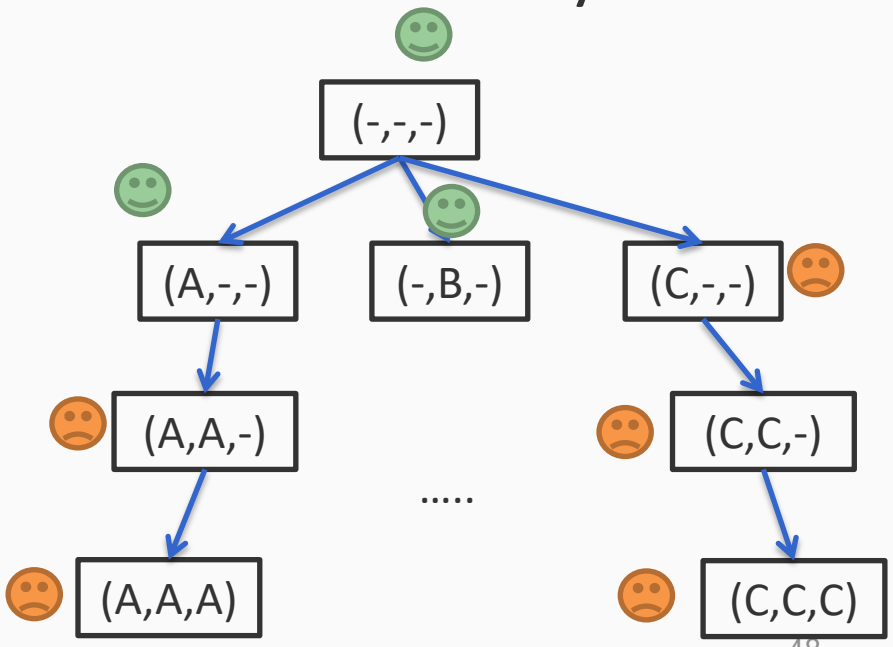
“y-good” node

Assumption: for any given node s and an ***gold*** output y, we can tell whether s can or cannot lead to y.

Definition: The node s is y-good if s can lead to y

$y = (y_1, y_2, y_3)$

Suppose each y can be one of A, B or C, and the true label is (y_1=A, y_2=B, y_3=C)



Learning in LaSO

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:
  - update $\boldsymbol{w}$

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:
  - update $\boldsymbol{w}$
  - clear the queue and insert all the correct moves

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:
  - update $\boldsymbol{w}$
  - clear the queue and insert all the correct moves
- Two kinds of errors:

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:
  - update $\boldsymbol{w}$
  - clear the queue and insert all the correct moves
- Two kinds of errors:
  - Error type 1: none of the queue is y-good

Learning in LaSO

- Search as if in the prediction phase, but when an error is made:
  - update $\boldsymbol{w}$
  - clear the queue and insert all the correct moves
- Two kinds of errors:
  - Error type 1: none of the queue is y-good
  - Error type 2: the goal state is not y-good

Learning Algorithm in LaSO

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error

    else

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error

    else
        if GoalTest(*node*) **then return** *w*

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error

    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error

    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error
        step 1:
            update w
    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

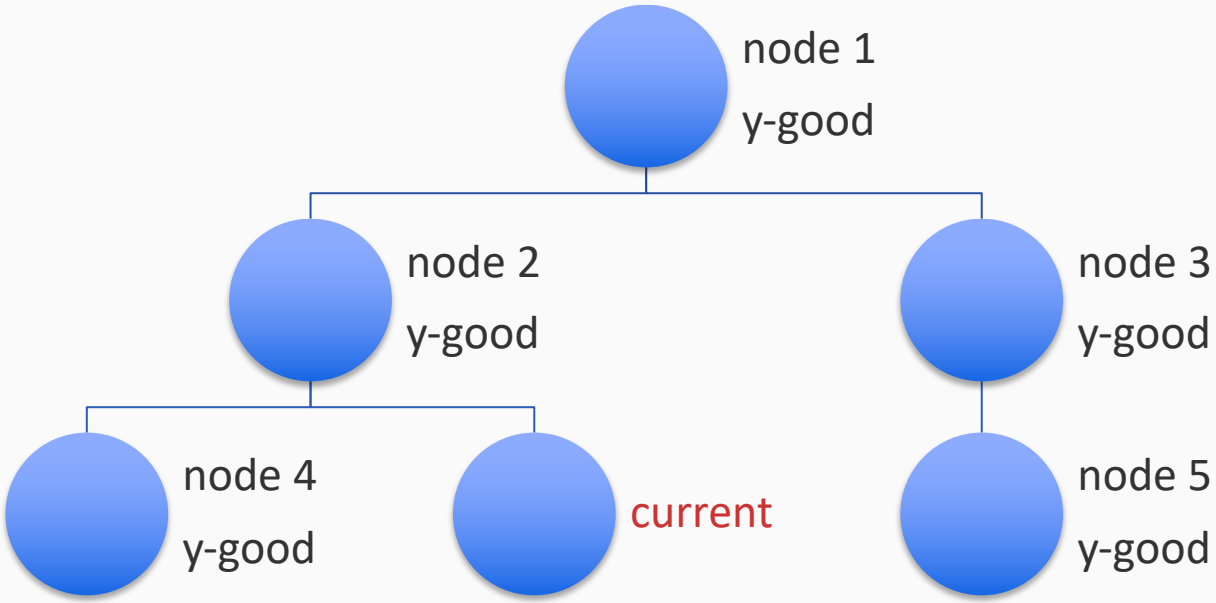
Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error
        step 1:
            update w
        step 2:
            refresh queue
    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

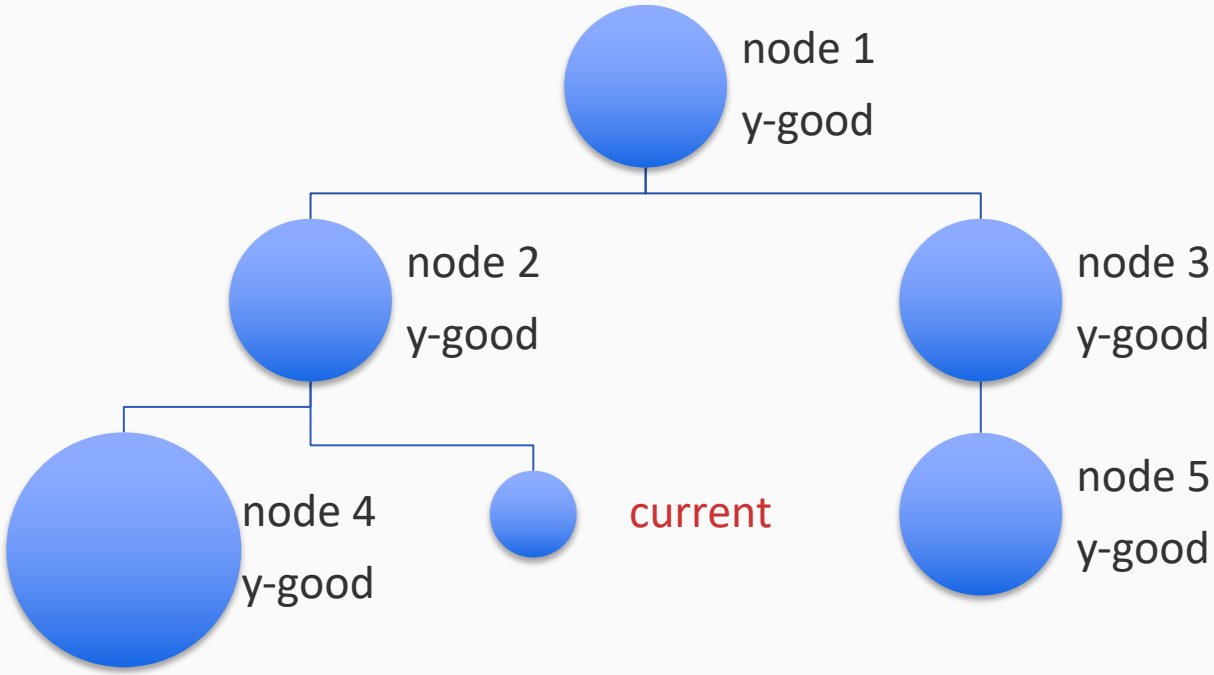
Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if error
        step 1:
            update w
        step 2:
            refresh queue
    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

What should learning do?



Let's say we found an error (of either type) at the current node, then we should have made the choice of node 4 instead of the current node

What should learning do?



Let's say we found an error (of either type) at the current node, then we should have made the choice of node 4 instead of the current node

Node 4 is the y-good *sibling* of the current node

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if

    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if none of (*node* + *nodes*) is y-good or
        GoalTest(*node*) and *node* is not y-good **then**
    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if none of (*node* + *nodes*) is y-good or
        GoalTest(*node*) and *node* is not y-good **then**
        sibs = siblings(*node, y*)

    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if none of (*node* + *nodes*) is y-good or
        GoalTest(*node*) and *node* is not y-good **then**
        sibs = siblings(*node, y*)
        w = update(*w, x, sibs, {node, nodes}*)

    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Learning Algorithm in LaSO

Algo Learn(*problem, initial, enqueue, w, x, y*)
nodes = MakeQueue(MakeNode(*problem, initial*))
while *nodes* is not empty:
    node = Pop(*nodes*)
    if none of (*node* + *nodes*) is y-good or
        GoalTest(*node*) and *node* is not y-good **then**
        sibs = siblings(*node, y*)
        w = update(*w, x, sibs, {node, nodes}*)
        nodes = MakeQueue(*sibs*)
    else
        if GoalTest(*node*) **then return** *w*
        next = Result(*node*, Actions(*node*))
        nodes = *enqueue*(*problem, nodes, next, w*)

Parameter Updates

We need to specify *w* = update(*w, x, sibs, nodes*)

A simple perceptron-style update rule: w = w + Δ

$$\Delta = \sum_{n \in sibs} \frac{\Phi(x,n)}{|sibs|} - \sum_{n \in nodes} \frac{\Phi(x,n)}{|nodes|}$$

It comes with the usual perceptron-style mistake bound and generalization bound. (See references)

2nd Framework:

SEARN: Search and Learning

Hal Daumé III, John Langford, Daniel Marcu (2007)

Policy

- A policy is a mapping from a state to an action
- For a given node, the policy tells what action should be taken

Policy

- A policy is a mapping from a state to an action
- For a given node, the policy tells what action should be taken

- A policy gives a search path in the search space.
  - Different policy means different search path
  - Can be thought as the “driver” in the search space

Policy

- A policy is a mapping from a state to an action
- For a given node, the policy tells what action should be taken

- A policy gives a search path in the search space.
  - Different policy means different search path
  - Can be thought as the “driver” in the search space

- A policy may be deterministic, or may contain some randomness. (More on this later)

Reference Policy and Learned Policy

Reference Policy and Learned Policy

- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

Reference Policy and Learned Policy

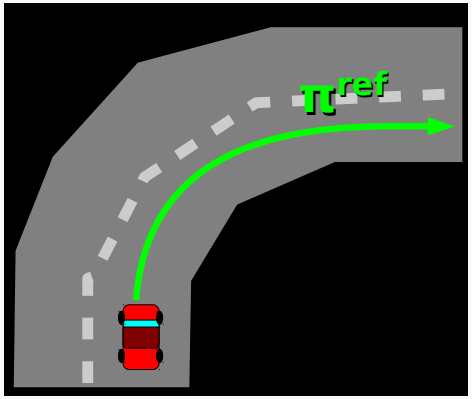
- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

- Goal: Learn a good policy for test data when we do not have access to cost vector c. (Imitation Learning)

Reference Policy and Learned Policy

- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

- Goal: Learn a good policy for test data when we do not have access to cost vector c. (Imitation Learning)

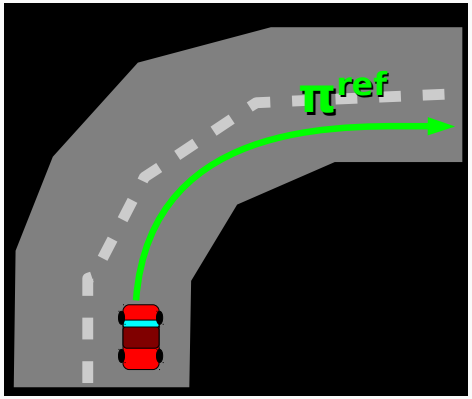


Reference Policy and Learned Policy

- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

- Goal: Learn a good policy for test data when we do not have access to cost vector c. (Imitation Learning)

For example if we are using Hamming distance for cost vector $\mathbf{c}$, then the reference policy is trivial to compute, why?



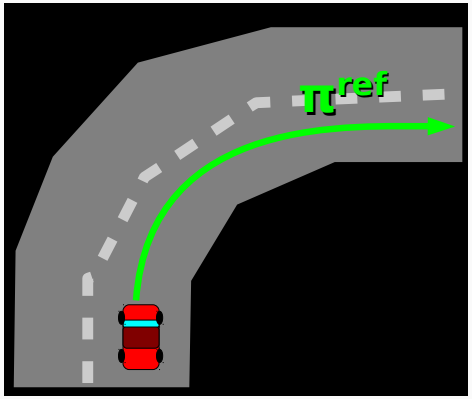
Reference Policy and Learned Policy

- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

- Goal: Learn a good policy for test data when we do not have access to cost vector c. (Imitation Learning)

For example if we are using Hamming distance for cost vector $\mathbf{c}$, then the reference policy is trivial to compute, why?

Just make the right decision at every step



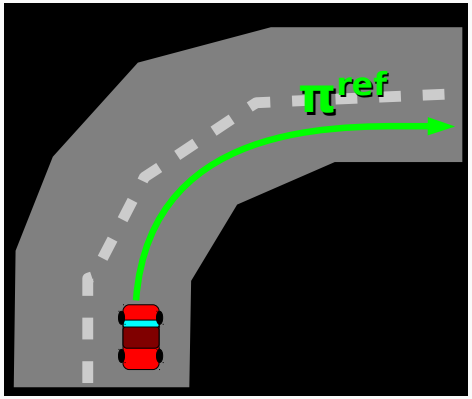
Reference Policy and Learned Policy

- We assume we already have a good *reference policy* π for training data $(\mathbf{x}, \mathbf{c})$
  - i.e. examples associated with costs for outputs

- Goal: Learn a good policy for test data when we do not have access to cost vector c. (Imitation Learning)

For example if we are using Hamming distance for cost vector $\mathbf{c}$, then the reference policy is trivial to compute, why?

Just make the right decision at every step



Suppose gold state is (A, B, C, A)
and we are at the state (A, C, -, -)
The reference policy tells us the next action is assigned C to the third slot.

Cost-Sensitive Classification

Suppose we want to learn a classifier h that maps examples to one of K labels

Standard multiclass classification

- Training data: Pairs of examples associated with labels
  - $(x, y) \in X \times [K]$
- Learning goal: To find a classifier that has low error
  - $\min_h \Pr[h(x) \neq y]$

Cost-Sensitive Classification

Suppose we want to learn a classifier h that maps examples to one of K labels

Standard multiclass classification

- Training data: Pairs of examples associated with labels
  - $(x, y) \in X \times [K]$
- Learning goal: To find a classifier that has low error
  - $\min_h \Pr[h(x) \neq y]$

Cost-sensitive classification

- Training data: An example paired with a cost vector that lists out the cost of predicting each label
  - $(x, \mathbf{c}) \in X \times [0, \infty)^K$

Cost-Sensitive Classification

Suppose we want to learn a classifier h that maps examples to one of K labels

Standard multiclass classification

- Training data: Pairs of examples associated with labels
  - $(x, y) \in X \times [K]$
- Learning goal: To find a classifier that has low error
  - $\min_h \Pr[h(x) \neq y]$

Cost-sensitive classification

- Training data: An example paired with a cost vector that lists out the cost of predicting each label
  - $(x, \mathbf{c}) \in X \times [0, \infty)^K$
- Learning goal: To find a classifier that has low cost
  - $\min_h E_{x,c}\left[c_{h(x)}\right]$

Cost-Sensitive Classification

Suppose we want to learn a classifier h that maps examples to one of K labels

Standard multiclass classification

- Training data: Pairs of examples associated with labels
  - $(x, y) \in X \times [K]$
- Learning goal: To find a classifier that has low error
  - $\min_h \Pr[h(x) \neq y]$

Cost-sensitive classification

- Training data: An example paired with a cost vector that lists out the cost of predicting each label
  - $(x, \mathbf{c}) \in X \times [0, \infty)^K$
- Learning goal: To find a classifier that has low cost
  - $\min_h E_{x,c}\left[c_{h(x)}\right]$

Exercise: How would you design a cost-sensitive learner?

Cost-Sensitive Classification

Suppose we want to learn a classifier h that maps examples to one of K labels

Standard multiclass classification

- Training data: Pairs of examples associated with labels
  - $(x, y) \in X \times [K]$
- Learning goal: To find a classifier that has low error
  - $\min_h \Pr[h(x) \neq y]$

Cost-sensitive classification

- Training data: An example paired with a cost vector that lists out the cost of predicting each label
  - $(x, \mathbf{c}) \in X \times [0, \infty)^K$
- Learning goal: To find a classifier that has low cost
  - $\min_h E_{x,c}\left[c_{h(x)}\right]$

SEARN uses a cost-sensitive learner to learn a policy

SEARN at test time

We already have learned a policy. We can use this policy to construct a sequence of decisions y and get the final structured output.

SEARN at test time

We already have learned a policy. We can use this policy to construct a sequence of decisions y and get the final structured output.

1. Use the learned policy on initial state (-,…, -) to compute y_1

SEARN at test time

We already have learned a policy. We can use this policy to construct a sequence of decisions y and get the final structured output.

1. Use the learned policy on initial state (-,…, -) to compute y_1
2. Use the learned policy on state (y_1, -,…,-) to compute y_2

SEARN at test time

We already have learned a policy. We can use this policy to construct a sequence of decisions y and get the final structured output.

1. Use the learned policy on initial state (-,…, -) to compute y_1
2. Use the learned policy on state (y_1, -,…,-) to compute y_2
3. Keep going until we get y = (y_1,…,y_n)

SEARN at training time

SEARN at training time

- The core idea in training is to notice that at each decision step, we are actually doing a cost-sensitive classification

SEARN at training time

- The core idea in training is to notice that at each decision step, we are actually doing a cost-sensitive classification

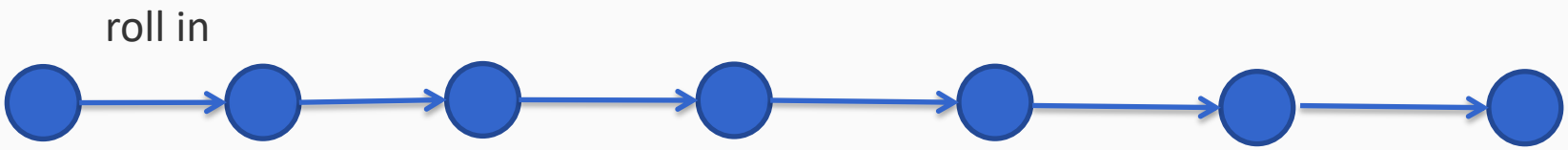
- Construct cost-sensitive classification examples (s, c) with state s and cost vector c.

SEARN at training time

- The core idea in training is to notice that at each decision step, we are actually doing a cost-sensitive classification
- Construct cost-sensitive classification examples (s, c) with state s and cost vector c.
- Learn a cost-sensitive classifier. (This is nothing but a policy)

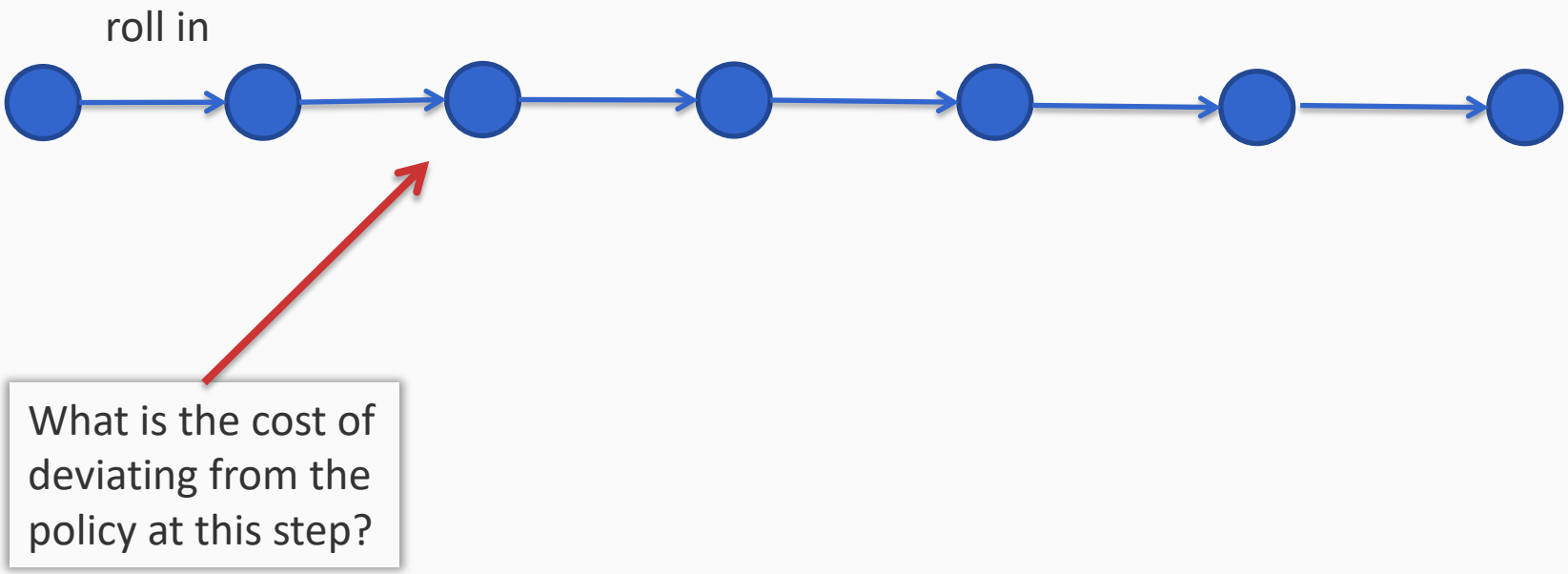
Roll-in, Roll-out

Roll-in, Roll-out

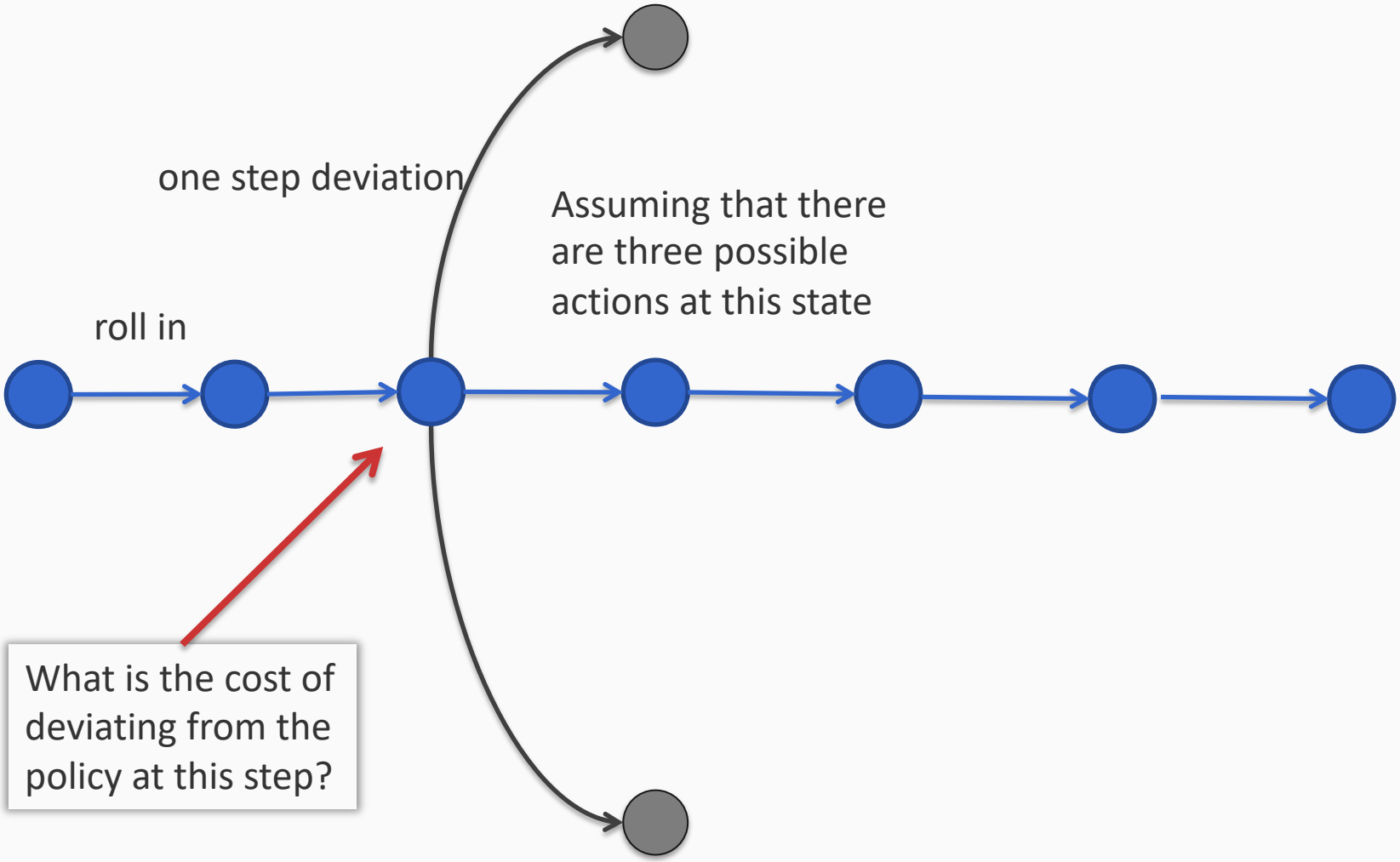


At each state, use some policy to move to a new state.

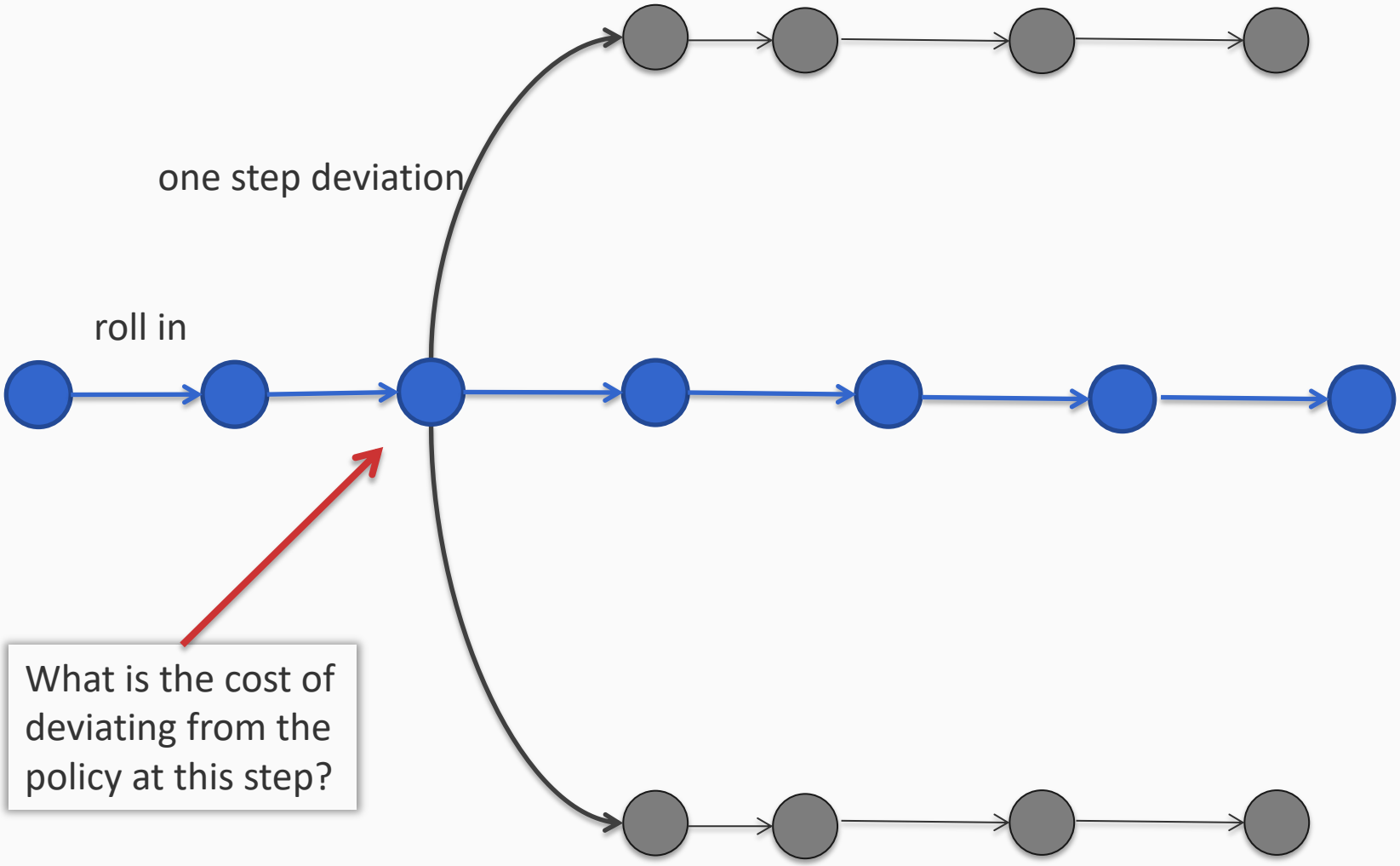
Roll-in, Roll-out



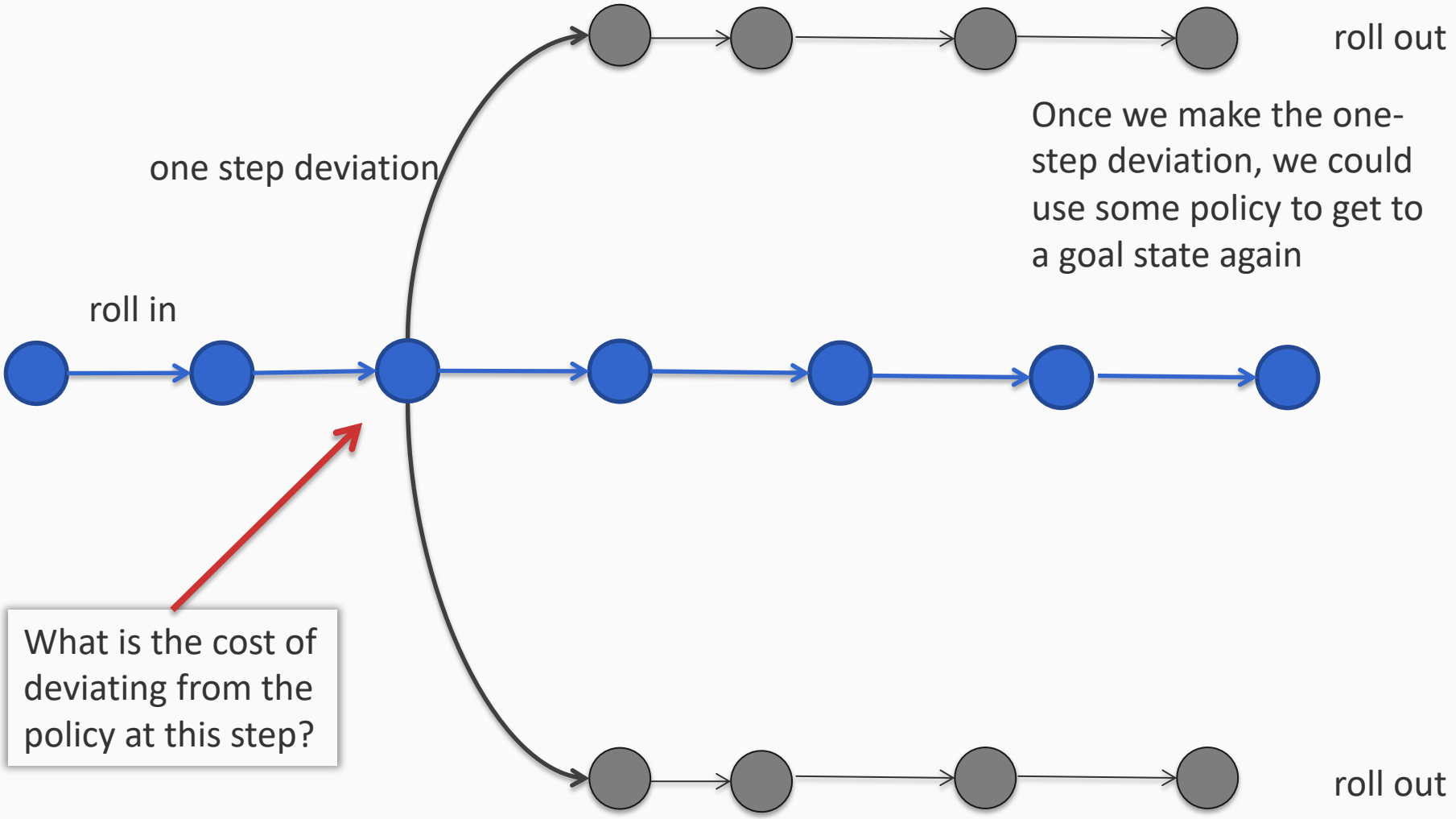
Roll-in, Roll-out



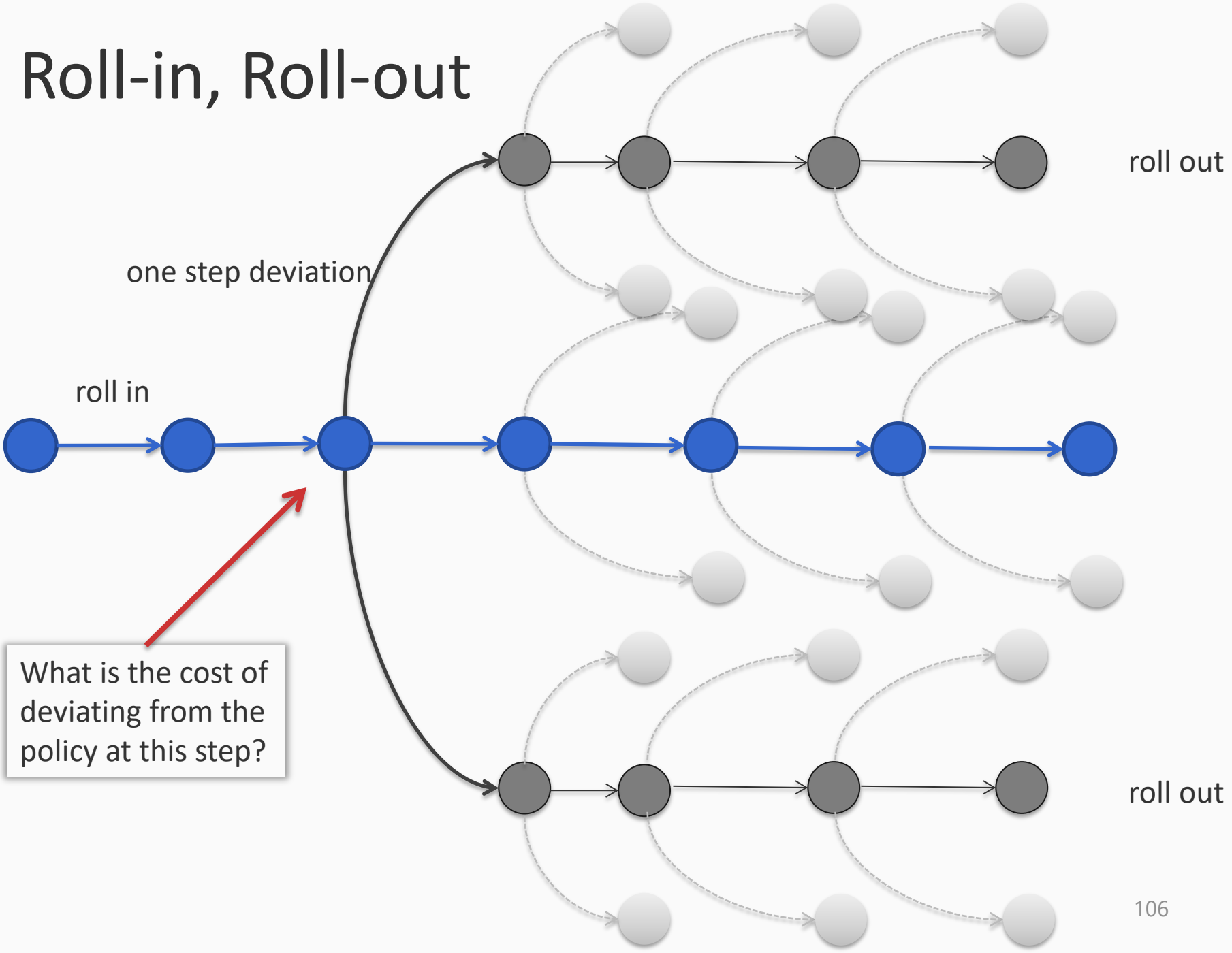
Roll-in, Roll-out



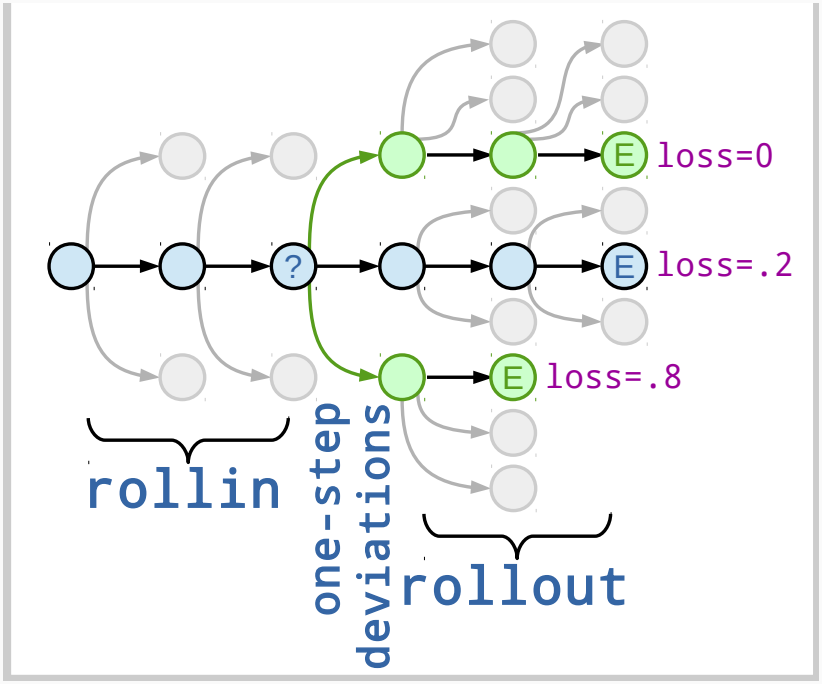
Roll-in, Roll-out



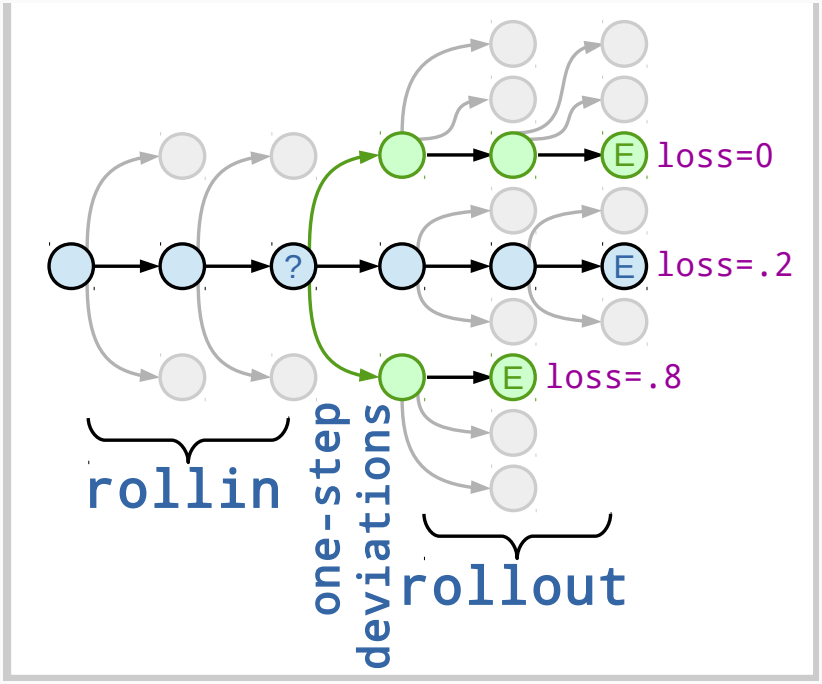
Roll-in, Roll-out



SEARN at training time (continued)

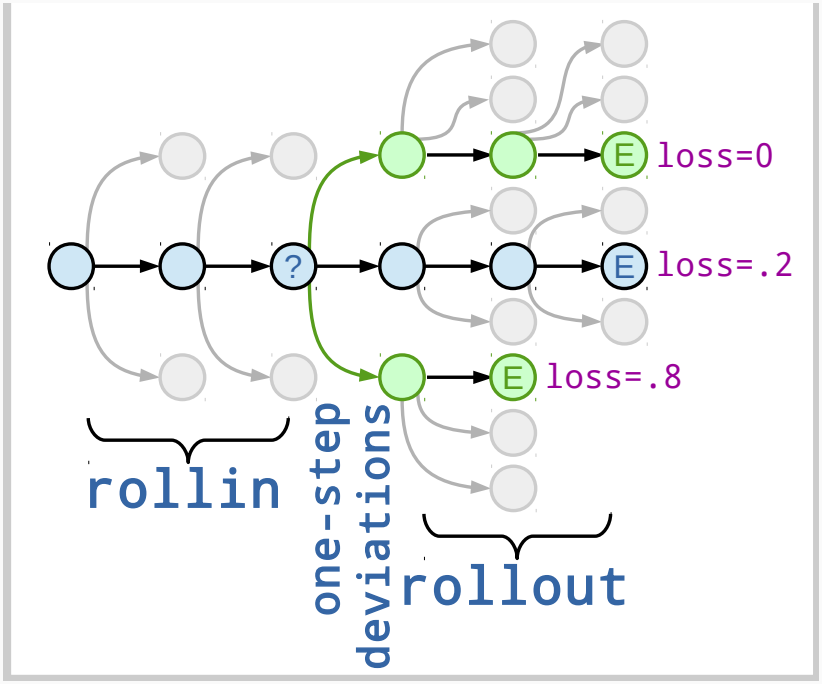


SEARN at training time (continued)



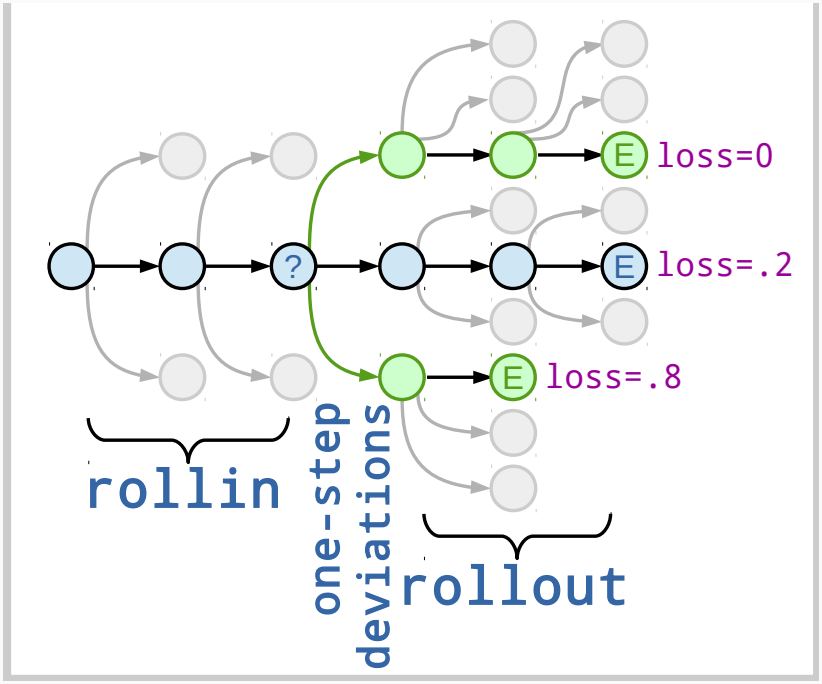
- Generate a search path

SEARN at training time (continued)



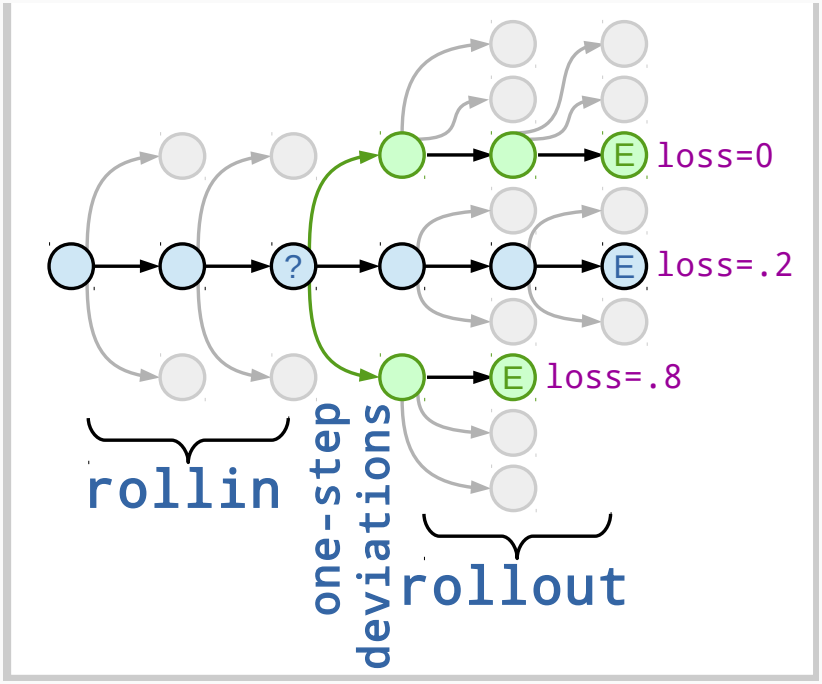
- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))

SEARN at training time (continued)



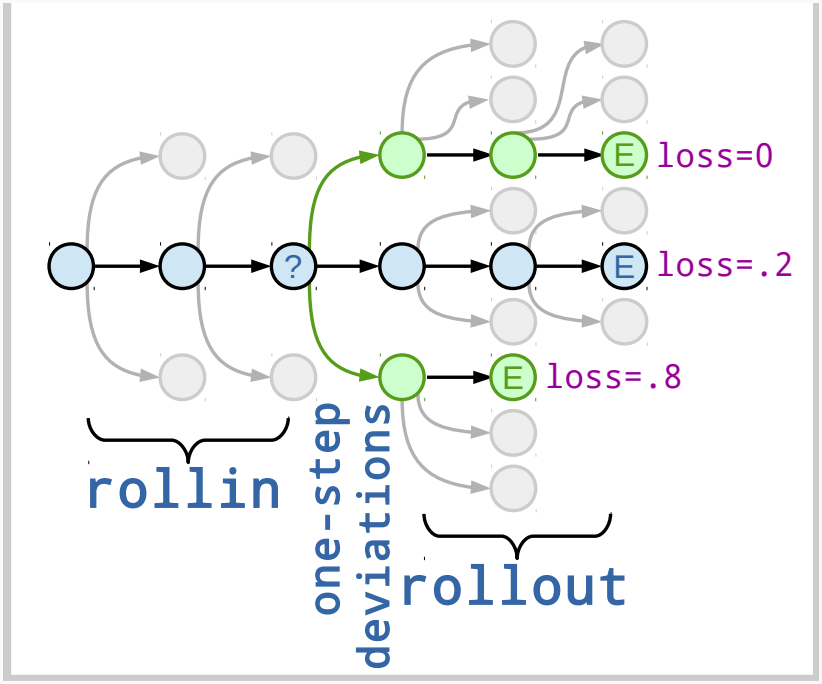
- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path

SEARN at training time (continued)



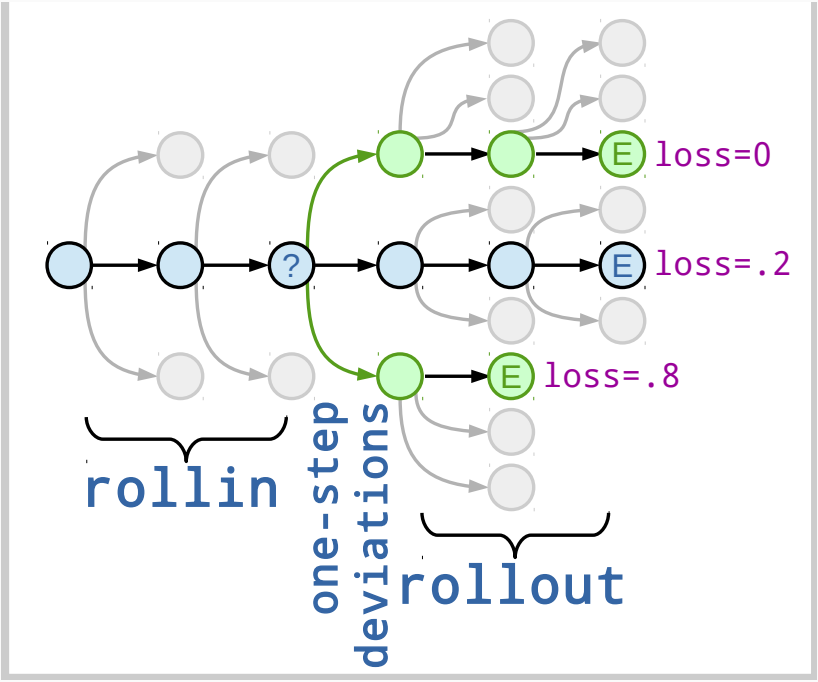
- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example

SEARN at training time (continued)



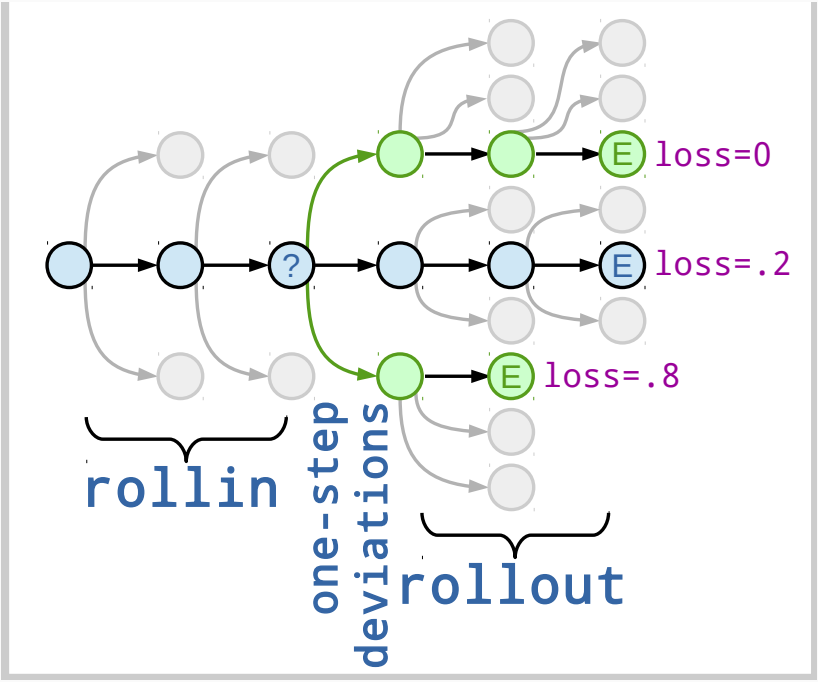
- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example
- Collect all these cost-sensitive examples to train a improved policy h'

SEARN at training time (continued)



- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example
- Collect all these cost-sensitive examples to train a improved policy h'
- Interpolate: $h \leftarrow \beta h' + (1-\beta)h$

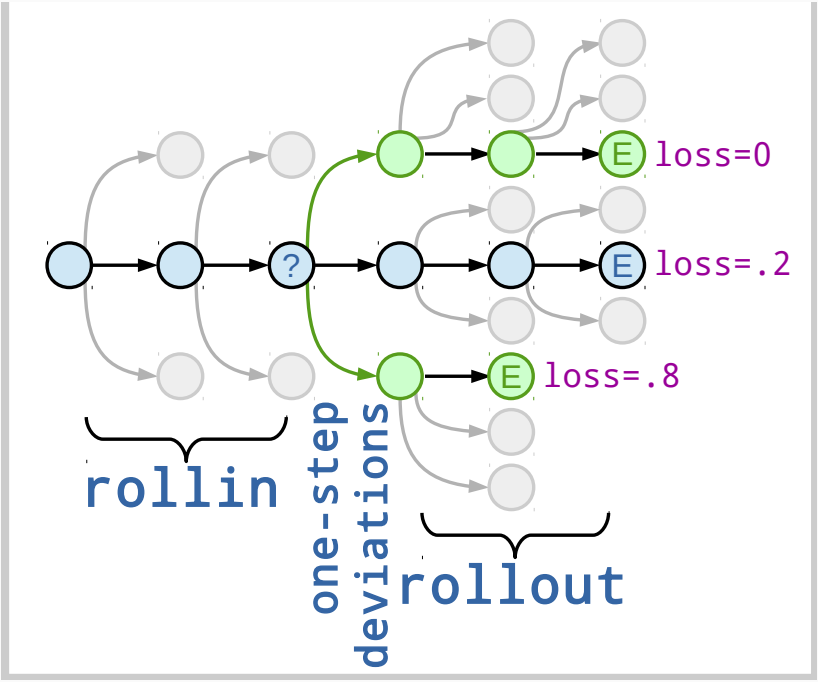
SEARN at training time (continued)



- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example
- Collect all these cost-sensitive examples to train a improved policy h'
- Interpolate: $h \leftarrow \beta h' + (1-\beta)h$
- Repeat

SEARN at training time (continued)

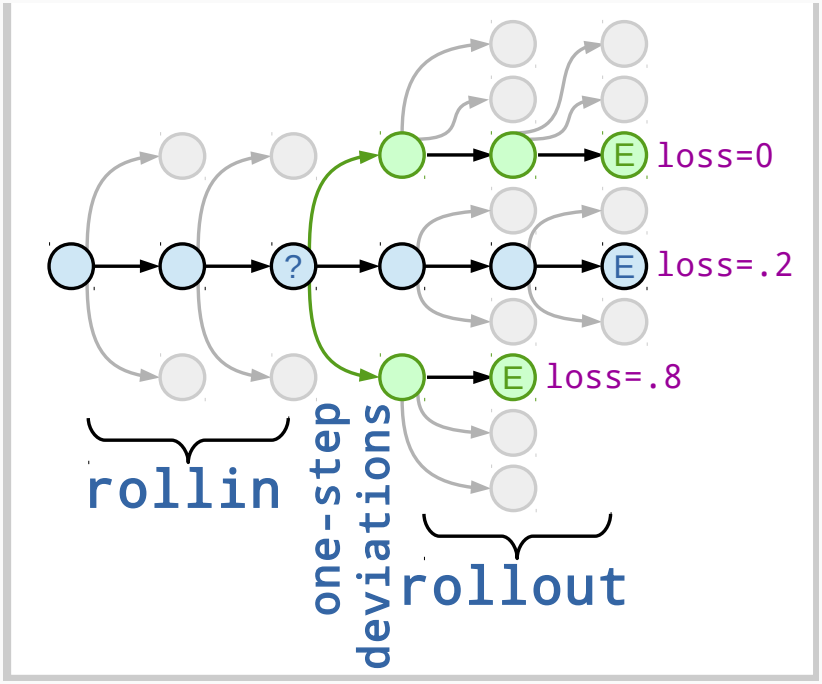
Roll-in with
current policy h



- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example
- Collect all these cost-sensitive examples to train a improved policy h'
- Interpolate: $h \leftarrow \beta h' + (1-\beta)h$
- Repeat

SEARN at training time (continued)

Roll-in with ***current policy*** h

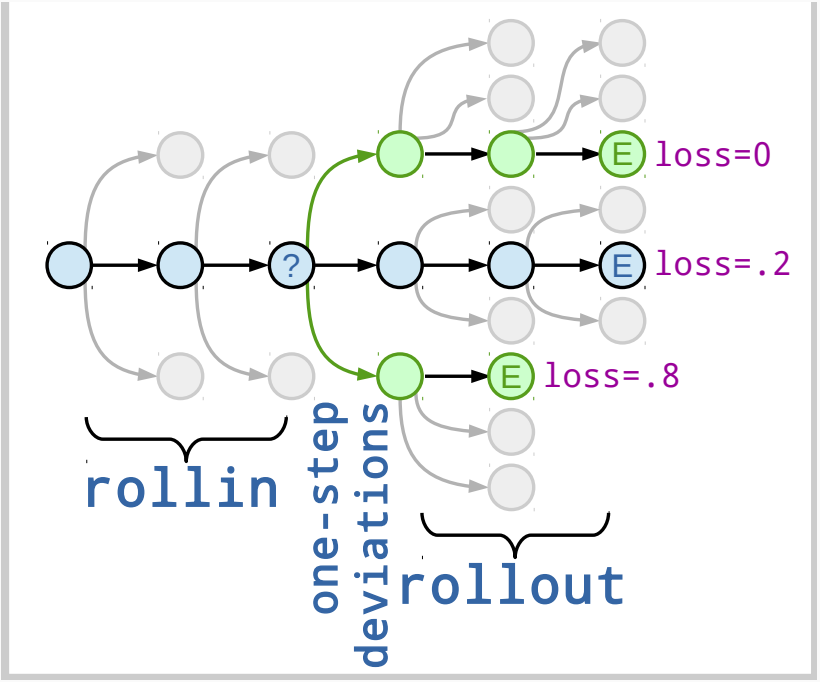


Roll-out with ***current policy*** h

- Generate a search path
- Construct a cost-sensitive example: (?-state, c=(0, 0.2, 0.8))
- Do this for every step along the path
- And for every structured training example
- Collect all these cost-sensitive examples to train a improved policy h'
- Interpolate: $h \leftarrow \beta h' + (1-\beta)h$
- Repeat

SEARN at training time (continued)

Roll-in with ***current policy*** h



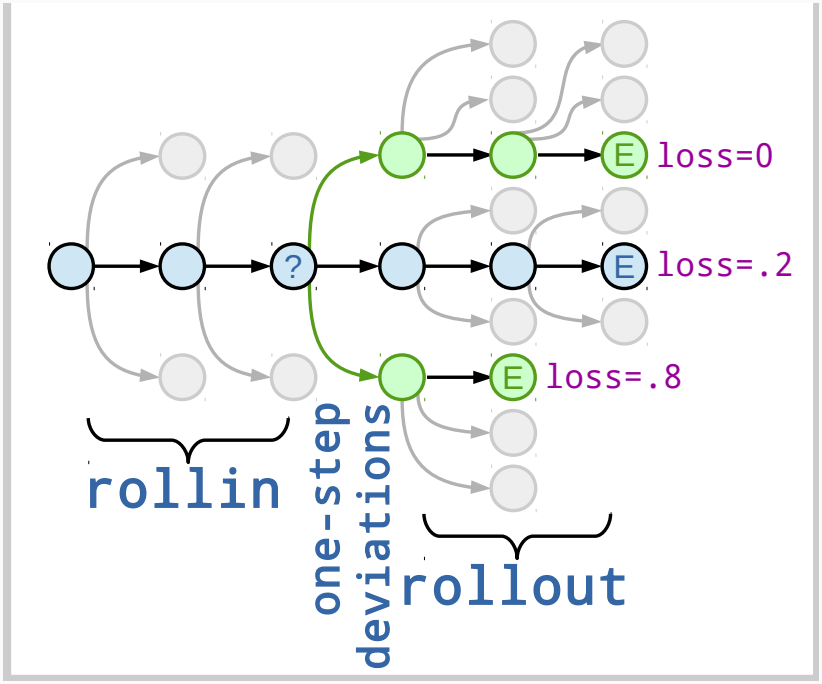
Roll-out with ***current policy*** h

- If h is deterministic:

$$l_h(c, s, a) = c_{y(s,a,h)} - \min_{a'} c_{y(s,a',h)}$$

SEARN at training time (continued)

Roll-in with ***current policy*** h



Roll-out with ***current policy*** h

- If h is deterministic:

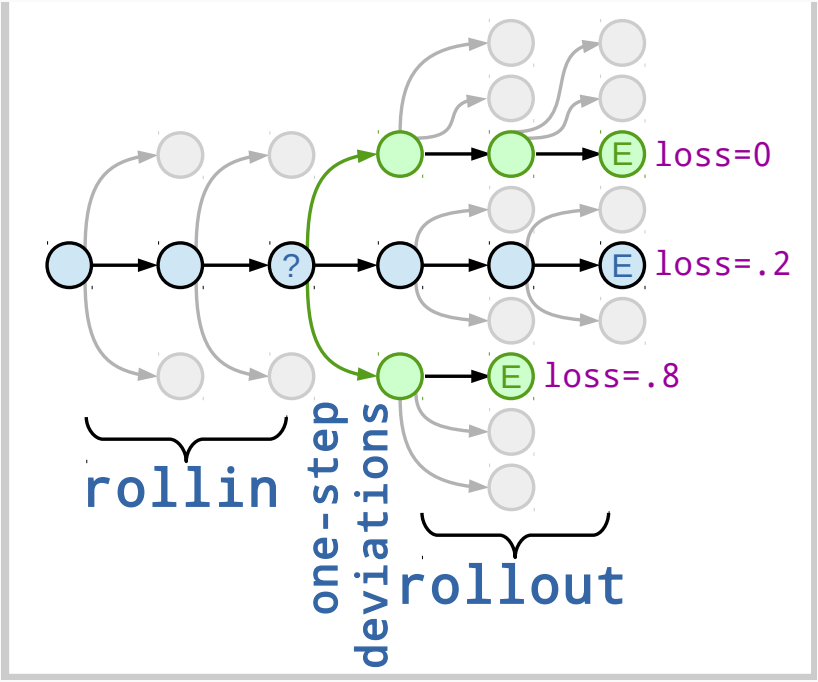
$$l_h(c, s, a) = c_{y(s,a,h)} - \min_{a'} c_{y(s,a',h)}$$

- If h contains randomness:

$$l_h(c, s, a) = E_{y\sim(s,a,h)} c_y - \min_{a'} E_{y\sim(s,a',h)} c_y$$

SEARN at training time (continued)

Roll-in with ***current policy*** h



Roll-out with ***current policy*** h

- If h is deterministic:

$$l_h(c, s, a) = c_{y(s,a,h)} - \min_{a'} c_{y(s,a',h)}$$

The loss defined this way is called regret

- If h contains randomness:

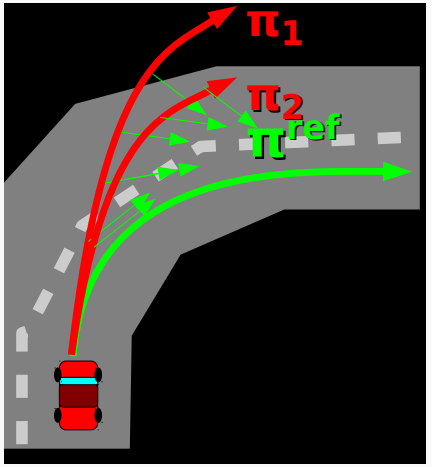
$$l_h(c, s, a) = E_{y\sim(s,a,h)} c_y - \min_{a'} E_{y\sim(s,a',h)} c_y$$

3rd Framework:

DAgger: Dataset Aggregation

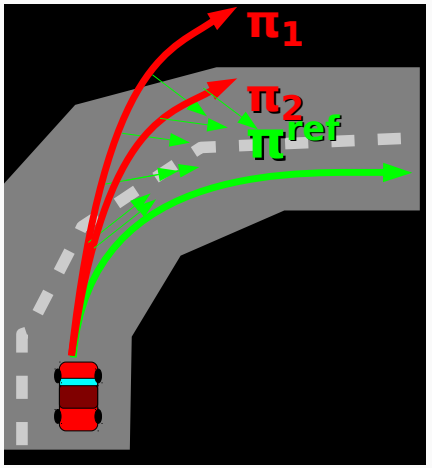
[Stéphane Ross, Geoffrey J. Gordon, J. Andrew Bagnell, 2011]

Dagger Algorithm (Simplified Version)



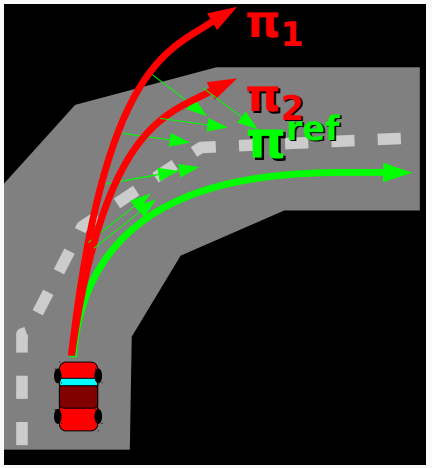
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$



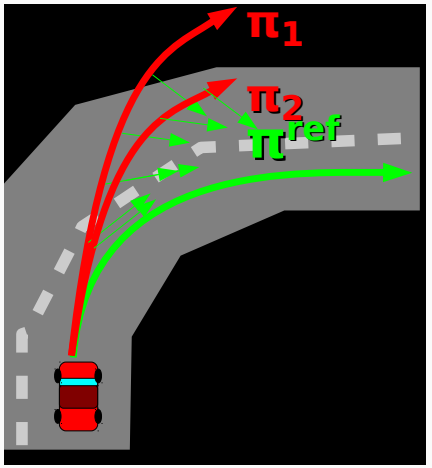
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)



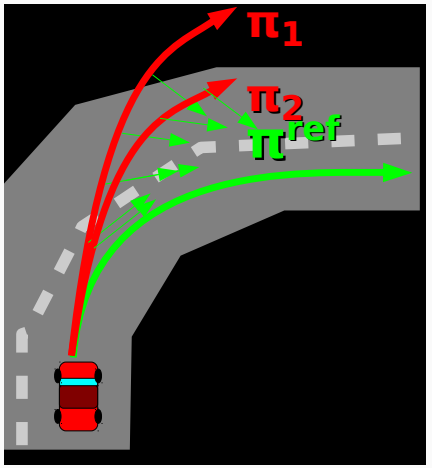
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$



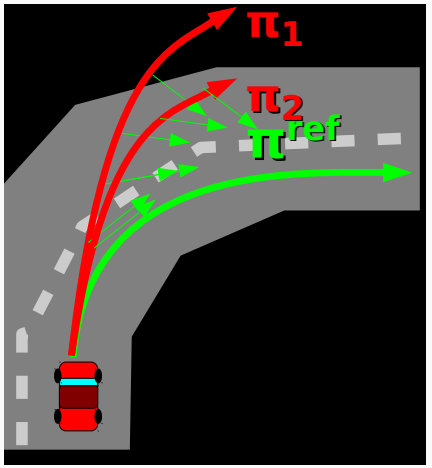
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$



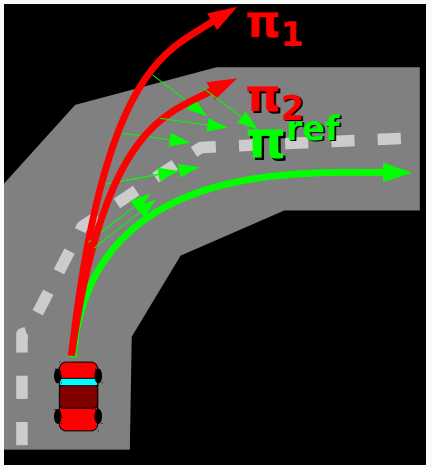
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$
- Train π_1 on D



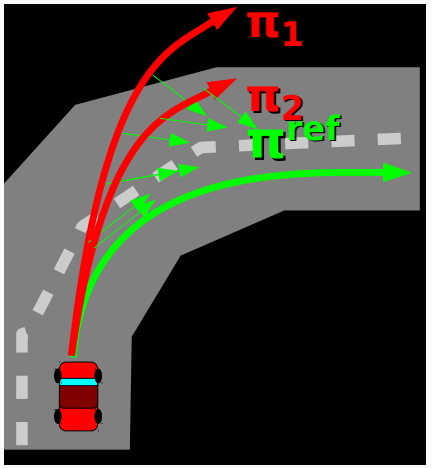
Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$
- Train π_1 on D
- Collect new trajectories with π_1



Dagger Algorithm (Simplified Version)

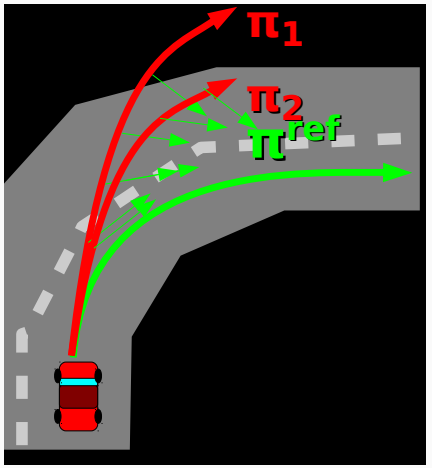
- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$
- Train π_1 on D
- Collect new trajectories with π_1
- New Dataset $D_2 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$



Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$
- Train π_1 on D

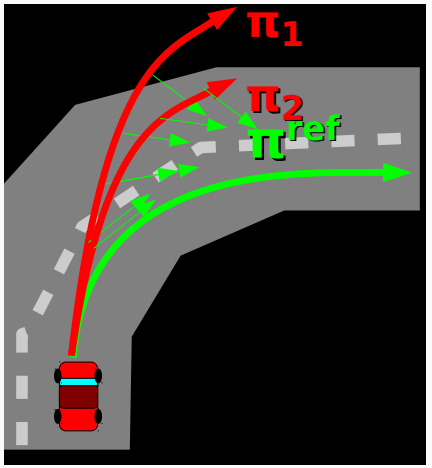
- Collect new trajectories with π_1
- New Dataset $D_2 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_2$



Dagger Algorithm (Simplified Version)

- Initialize Dataset $D = \emptyset$
- Collect trajectories with reference policy π^{ref} (the expert)
- Dataset $D_1 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_1$
- Train π_1 on D

- Collect new trajectories with π_1
- New Dataset $D_2 = \left\{\left(s, \pi^{ref}(s)\right)\right\}$
- Aggregate Datasets $D = D \cup D_2$
- Train π_2 on D



DAgger V.S. SEARN

Similarities:

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy
- Iteratively improving the current policy by learning better multiclass classifiers.

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy
- Iteratively improving the current policy by learning better multiclass classifiers.

Differences:

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy
- Iteratively improving the current policy by learning better multiclass classifiers.

Differences:

- There is no roll-out stage

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy
- Iteratively improving the current policy by learning better multiclass classifiers.

Differences:

- There is no roll-out stage
- At each step we just have a regular multiclass example (not cost-sensitive example), given by the expert.

DAgger V.S. SEARN

Similarities:

- Dagger also treats a structured prediction problem as a sequence of multiclass classification problem.
- Roll-in with current policy
- Iteratively improving the current policy by learning better multiclass classifiers.

Differences:

- There is no roll-out stage
- At each step we just have a regular multiclass example (not cost-sensitive example), given by the expert.
- Aggregate dataset

Other related algorithms

- Incremental Perceptron (2002)
  - Based on structured Perceptron
  - Instead of finishing inference during training, when inference makes its first mistake, stop and update parameters
- AggreVaTe: Aggregate Values to Imitate (2014)
  - Combines ideas from DAgger and SEARN
  - Cost-sensitive learning + dataset aggregation
- LOLS: Locally Optimal Learning to Search (2015)
  - What if the reference policy is not good?
  - Changes roll-outs to account for this

Learning to search: Summary

- Inference in structured prediction can be framed as search
  - Can we learn a model that explicitly helps inference navigate the search space?

- Several algorithms:
  - LaSO, SEARN, DAgger, etc
  - Often easy to implement with simpler building blocks
    - Can be the basis of a general purpose structured prediction framework