CATX

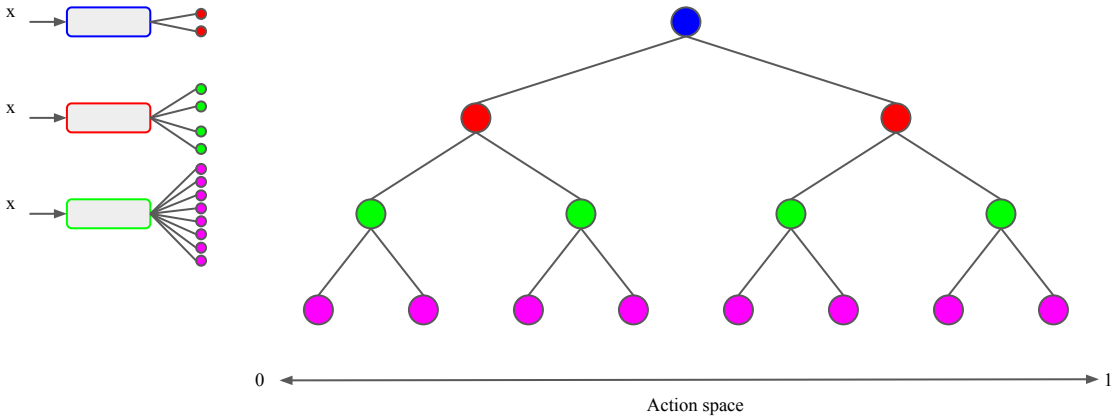
A JAX implementation of the *"Efficient Contextual Bandits with Continuous Actions"* paper

Tree

Tree

This example uses a tree of depth 3
At each depth there is neural network (depth 0: blue, depth 1: red, and depth 2: green)
Each neural network output layer dimension is 2^(depth+1)

```python
class Tree(hk.Module):
    def __init__(
        self,
        network_builder: NetworkBuilder,
        tree_params: TreeParameters,
        name: Optional[str] = None,
    ):
```

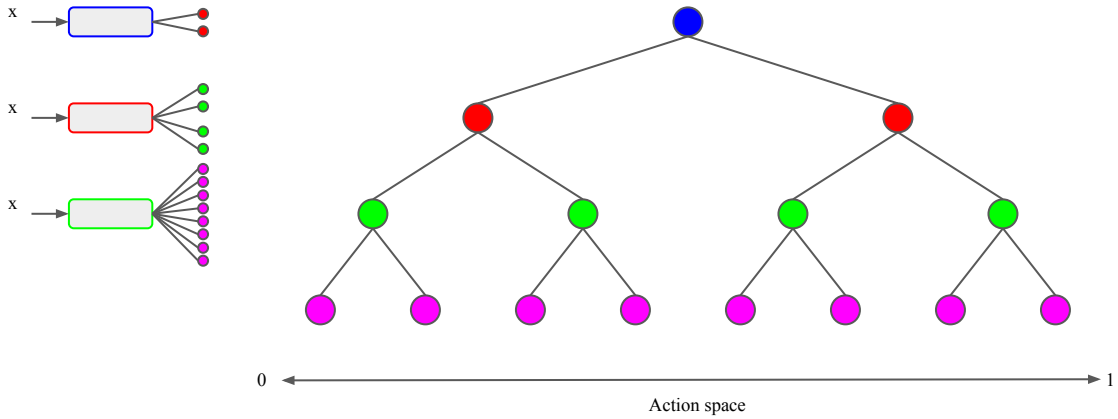


Tree parameters

Tree parameters

action space

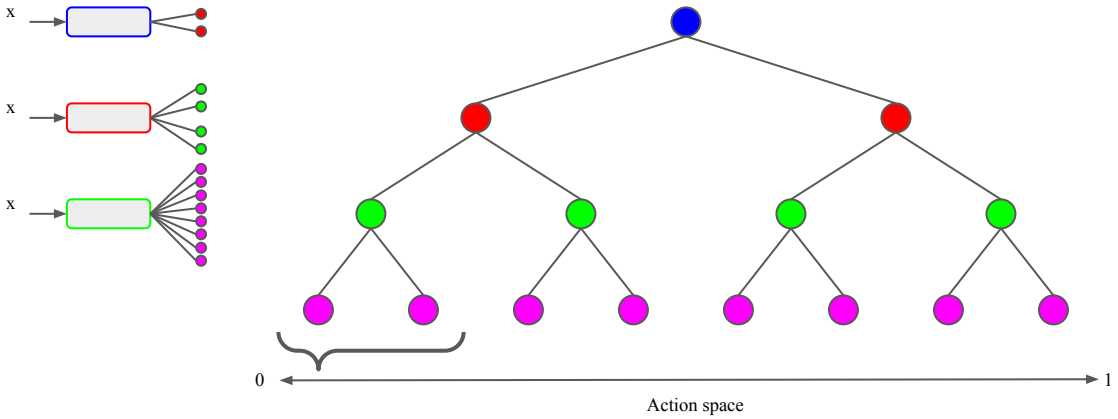
```
@dataclass
class TreeParameters:
```



Tree parameters

```
@dataclass
class TreeParameters:
```

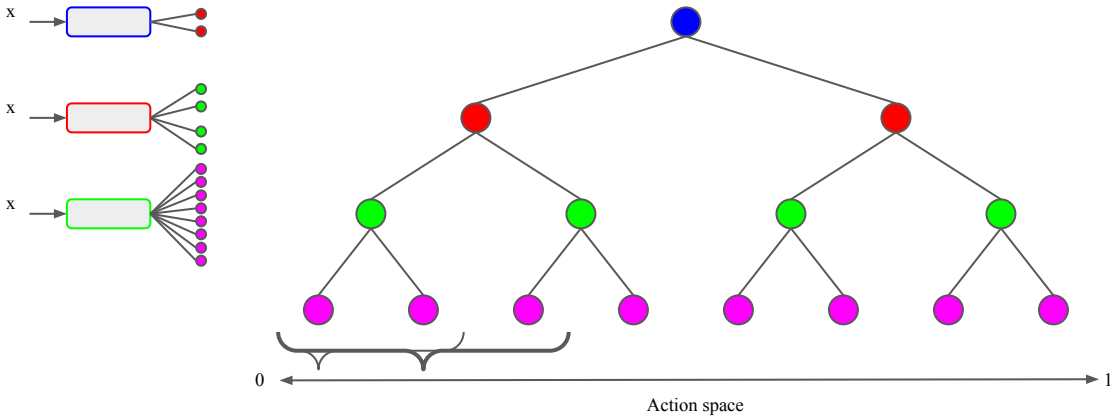
action spaces: each discretized action centroid covers 2*bandwidth of the action space



Tree parameters

```
@dataclass
class TreeParameters:
```

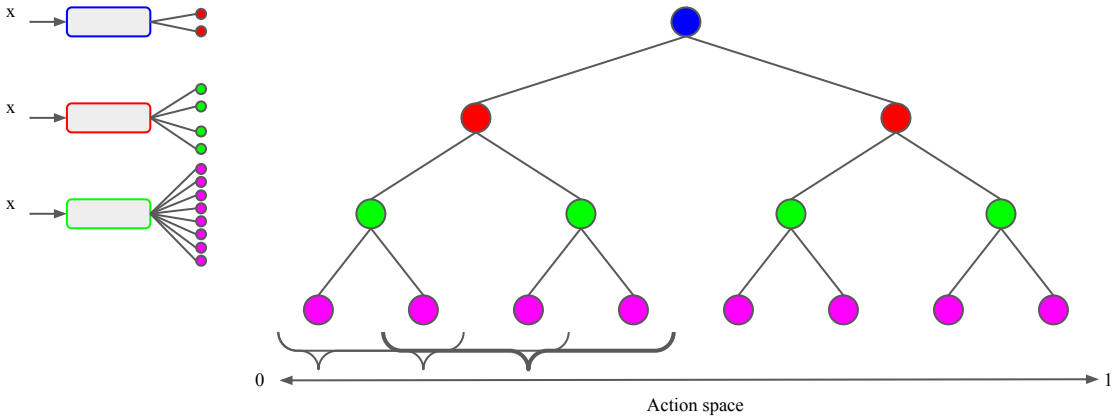
action spaces: each discretized action centroid covers 2*bandwidth of the action space



Tree parameters

```
@dataclass
class TreeParameters:
```

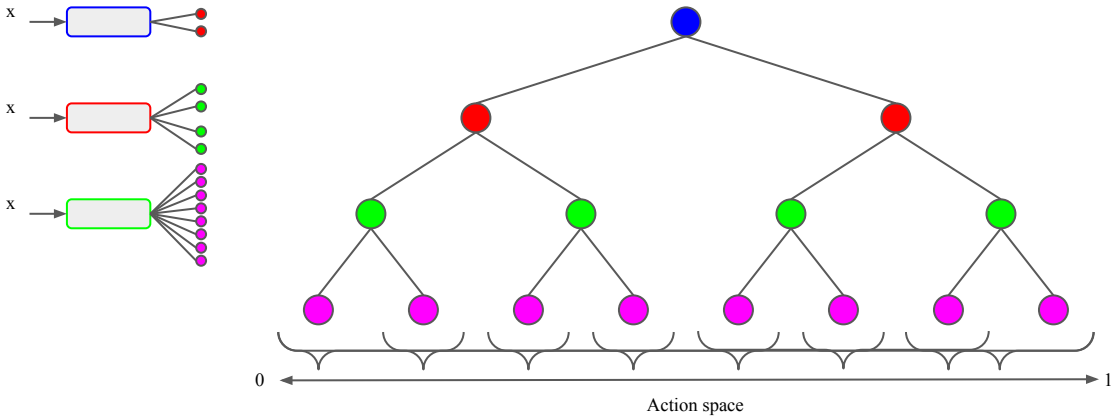
action spaces: each discretized action centroid covers 2*bandwidth of the action space



Tree parameters

```
@dataclass
class TreeParameters:
```

action spaces: each discretized action centroid covers 2*bandwidth of the action space

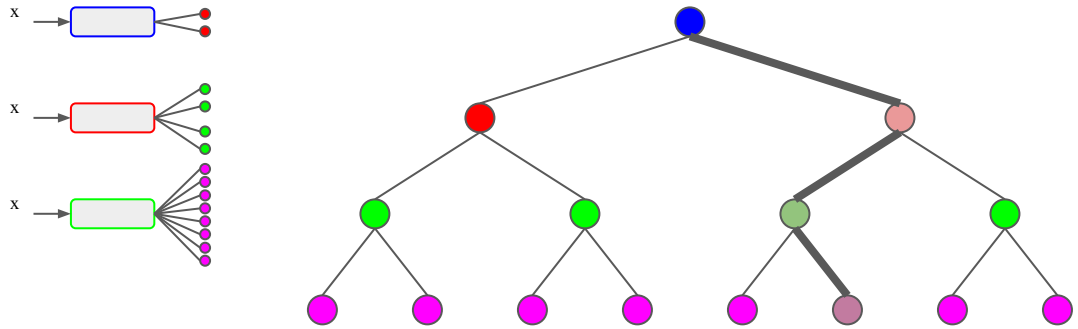


Action query

Action query

forward pass of the tree by following the max of the logits

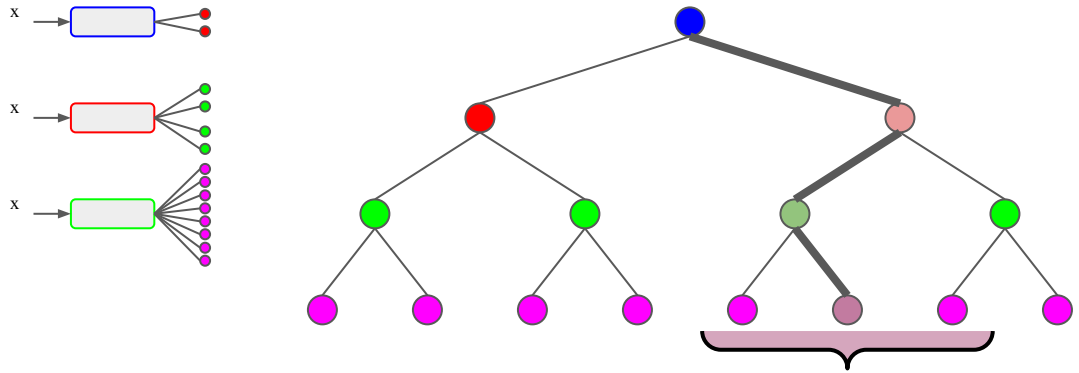
```
def sample(
    self, obs: Observations, epsilon: float
) -> Tuple[Actions, Probabilities]:
```



Action query

sample an action with eps-greedy

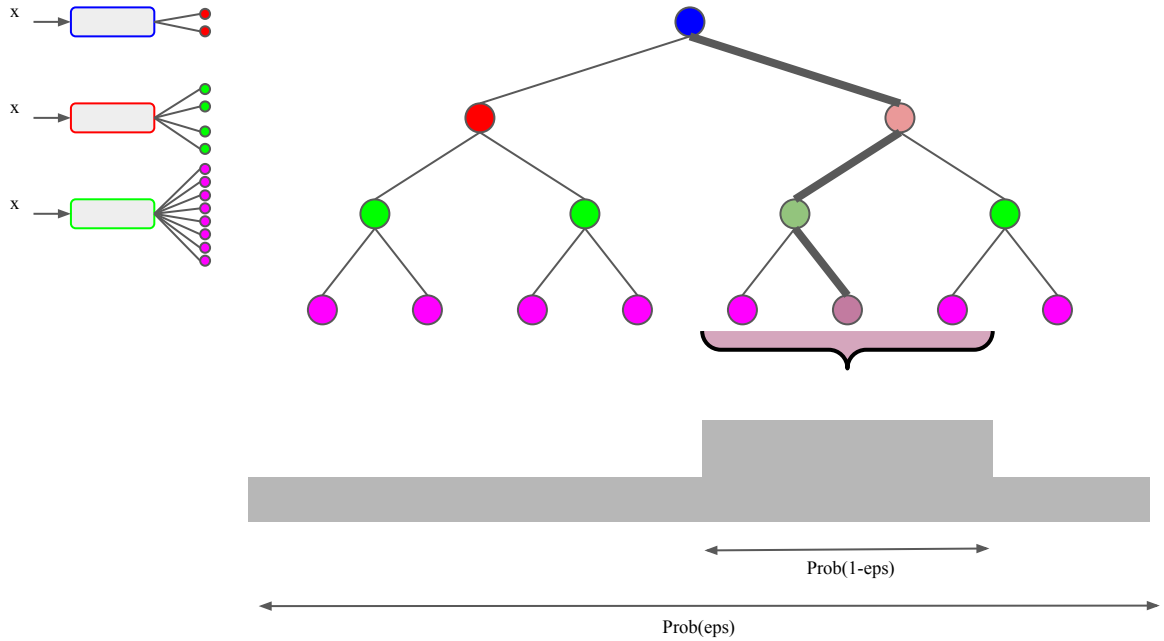
```python
def sample(
    self, obs: Observations, epsilon: float
) -> Tuple[Actions, Probabilities]:
```



Action query

sample an action with eps-greedy

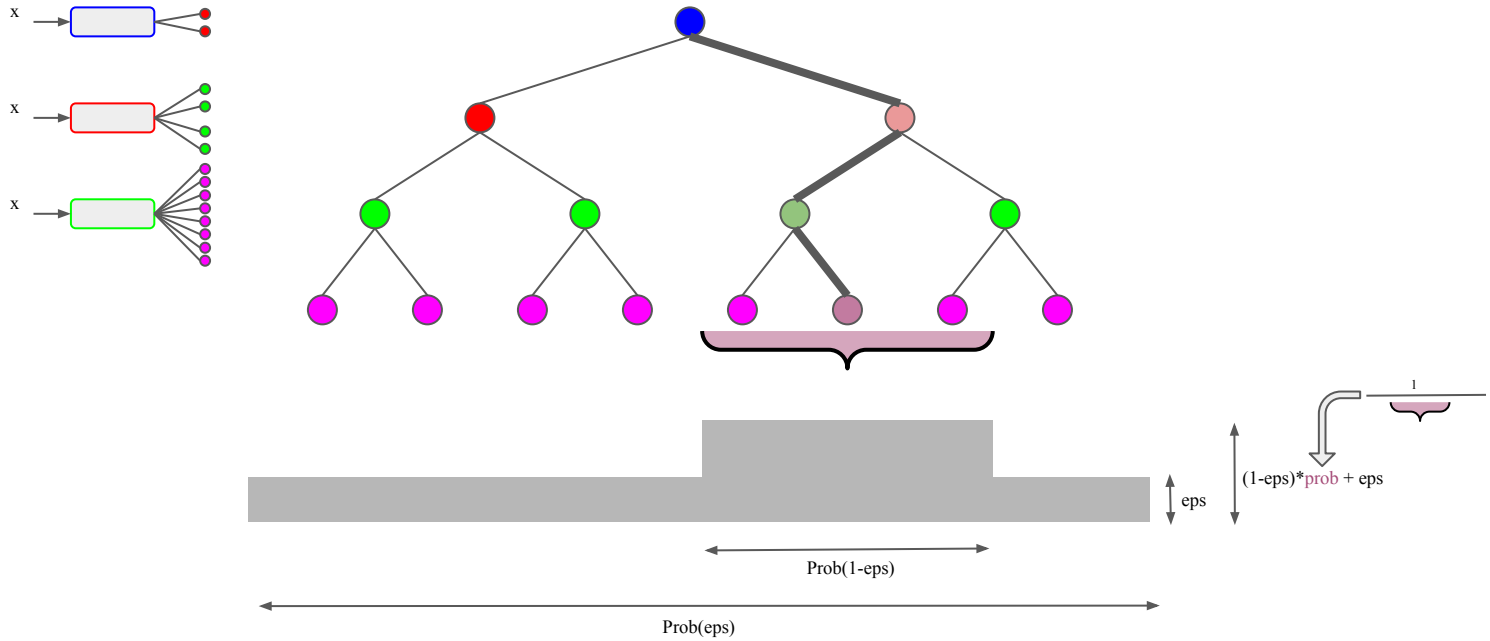
```
def sample(
    self, obs: Observations, epsilon: float
) -> Tuple[Actions, Probabilities]:
```



Action query

sample an action with eps-greedy

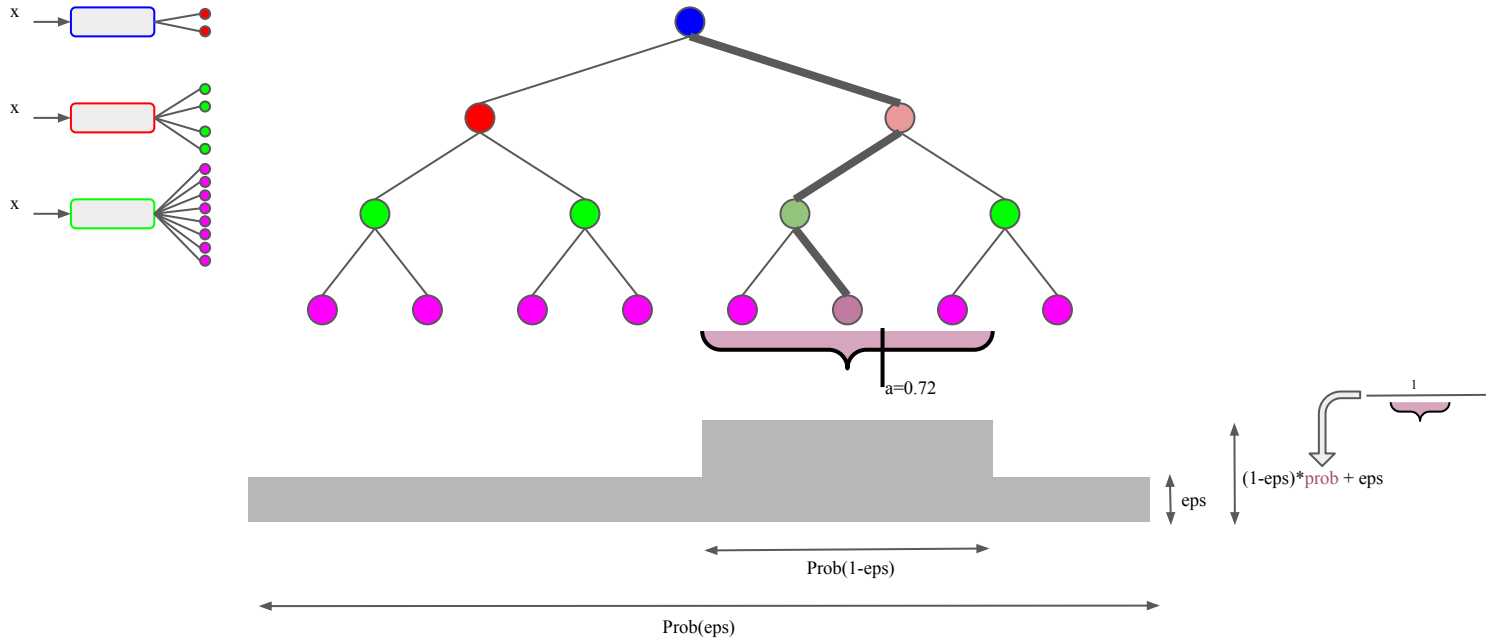
```
def sample(
    self, obs: Observations, epsilon: float
) -> Tuple[Actions, Probabilities]:
```



Action query

sample an action with eps-greedy (example: a=0.72)

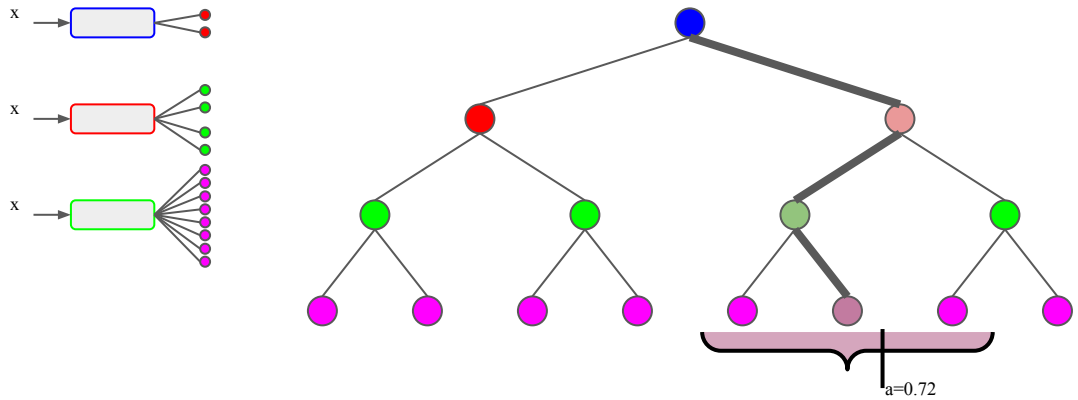
```
def sample(
    self, obs: Observations, epsilon: float
) -> Tuple[Actions, Probabilities]:
```



Action cost

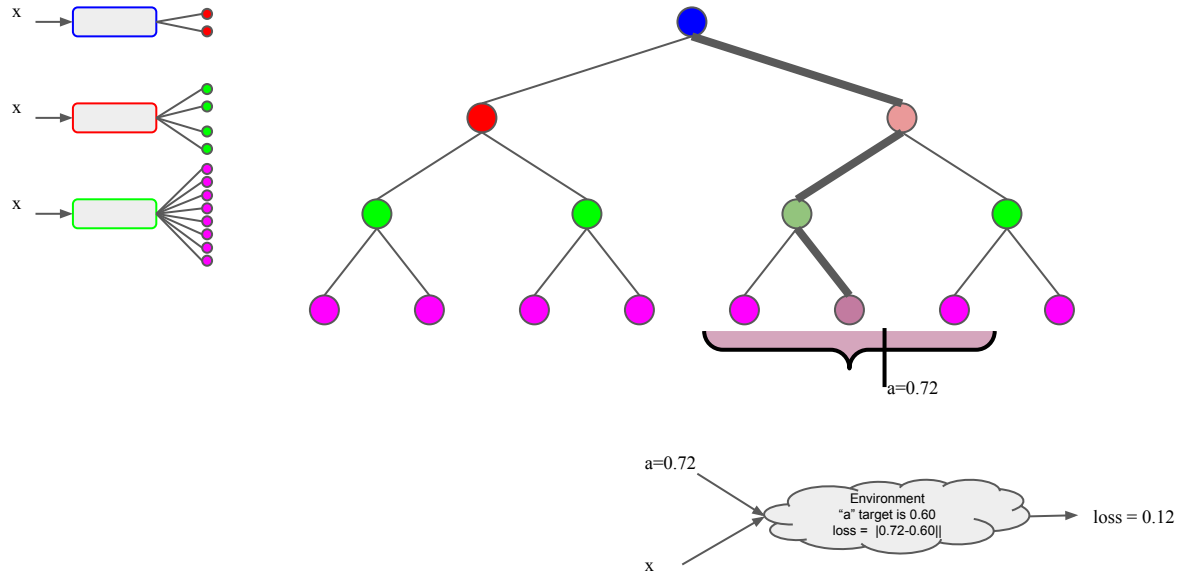
Action cost

apply action in the environment and receive cost feedback



Action cost

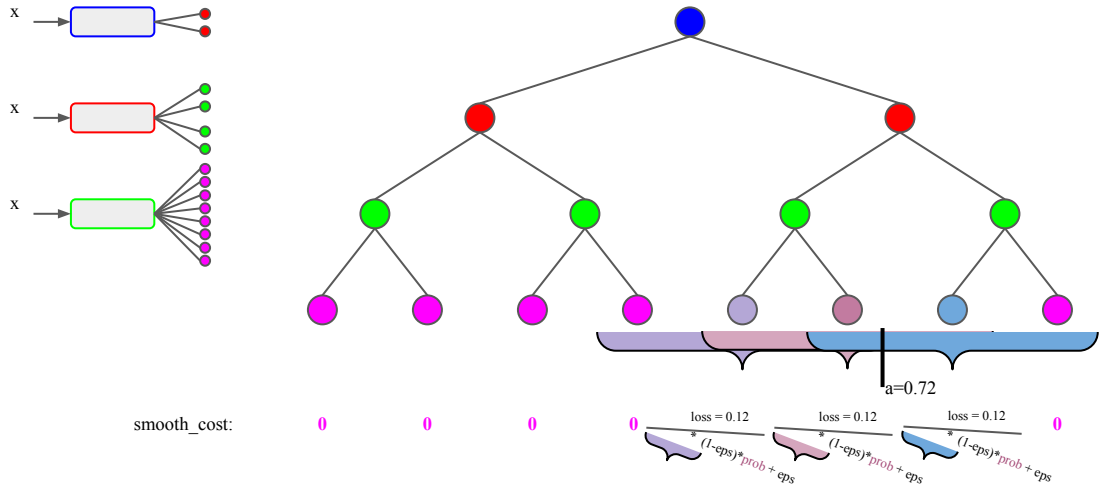
apply action in the environment and receive cost feedback



Action cost

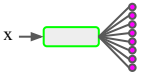
smooth the cost across the discretized actions that could have generated the applied action

```python
@functools.partial(jax.jit, static_argnames=("self",))
def _compute_smooth_costs(
    self, costs: JaxCosts, actions: JaxActions, probabilities: JaxProbabilities
) -> JaxCosts:
```

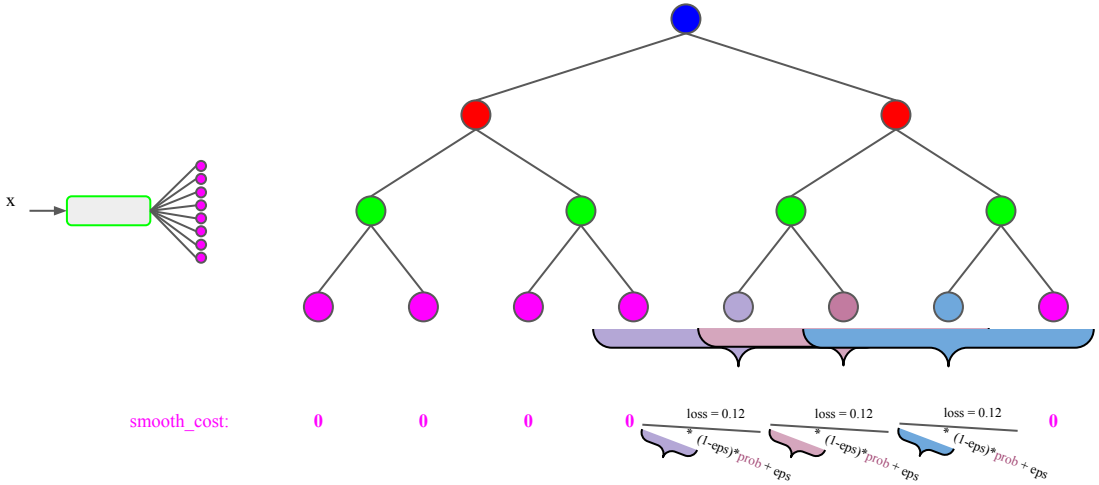


Update neural network weights

Update:



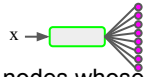

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



loss ft = sum(softmax(●●●●●●●●) * smooth cost)

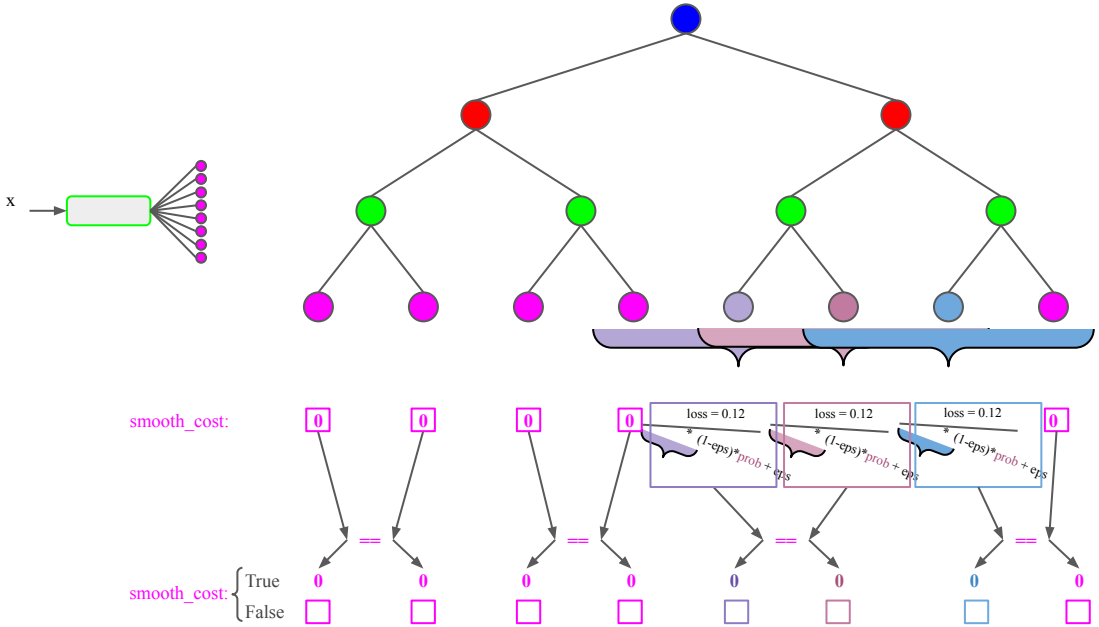
min(loss ft)

Update:



Only update nodes whose pair childs have different cost.

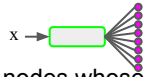
```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



loss ft = sum(softmax(●●●●●●●●) * smooth cost)

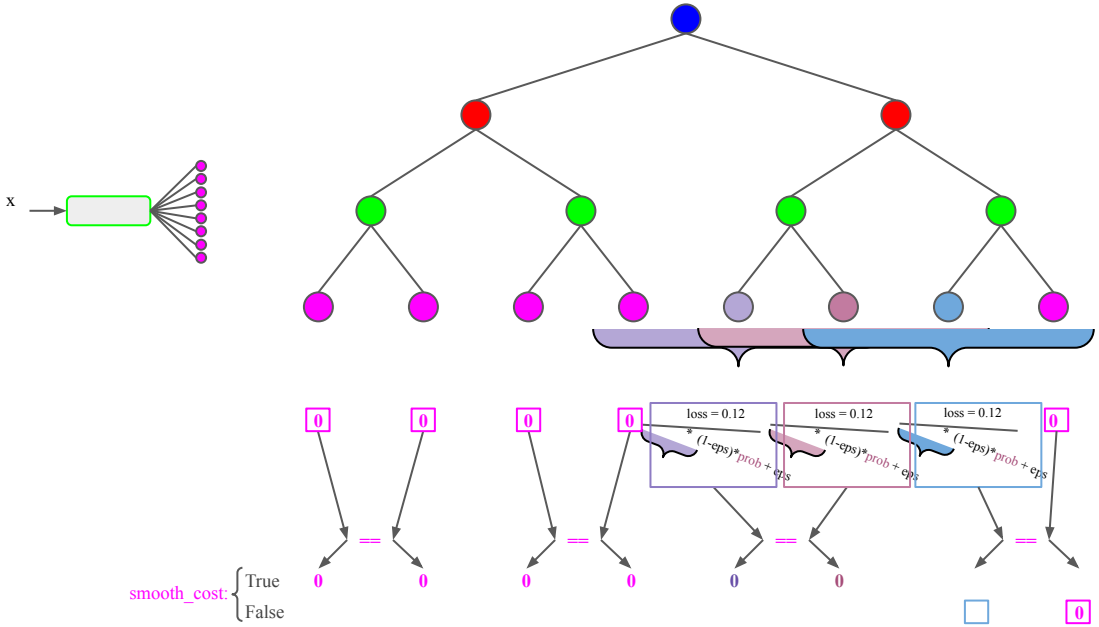
min(loss ft)

Update:



Only update nodes whose pair childs have different cost.
In this example:

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

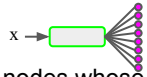


loss ft = sum(softmax( ) * smooth cost)

0 0 0 0 0 0

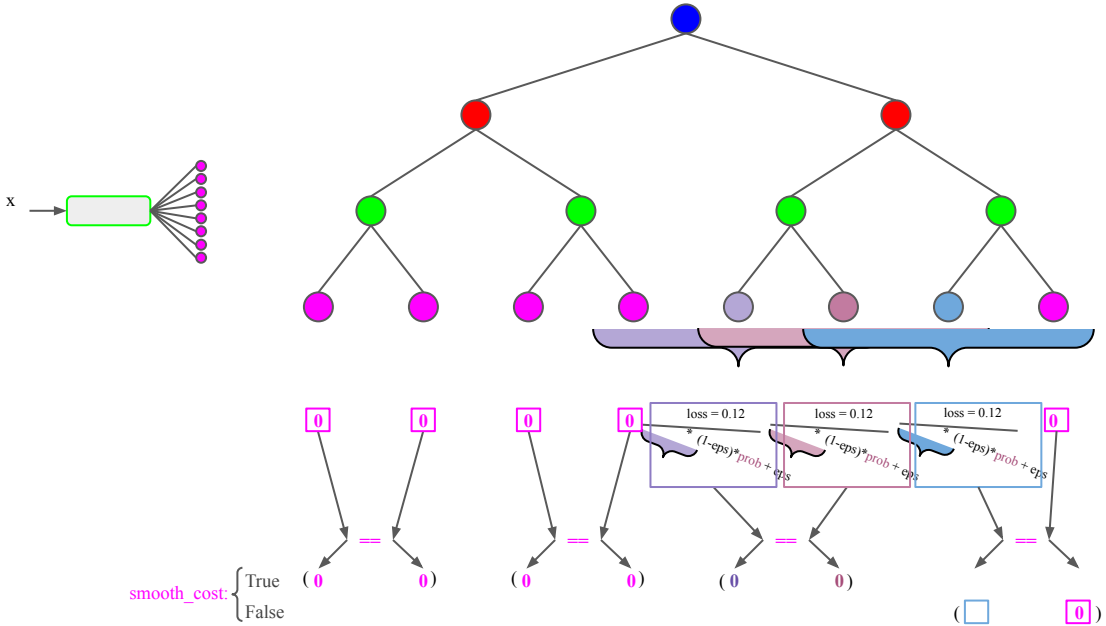
min(loss ft)

Update:



Only update nodes whose pair childs have different cost.
In this example:
Note: the softmax is performed pairwise

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

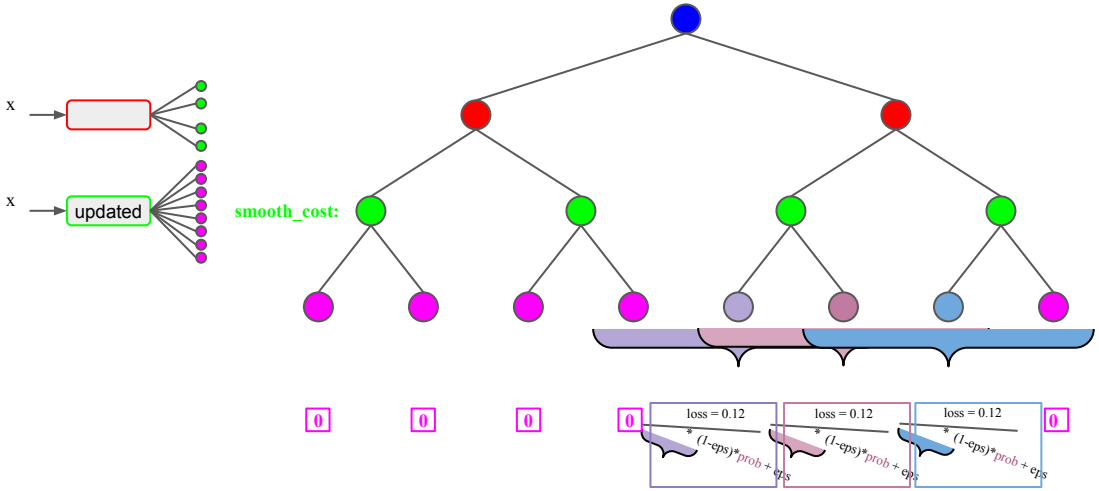


loss ft = sum(softmax( ) * smooth cost)

min(loss ft)

Update:


```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

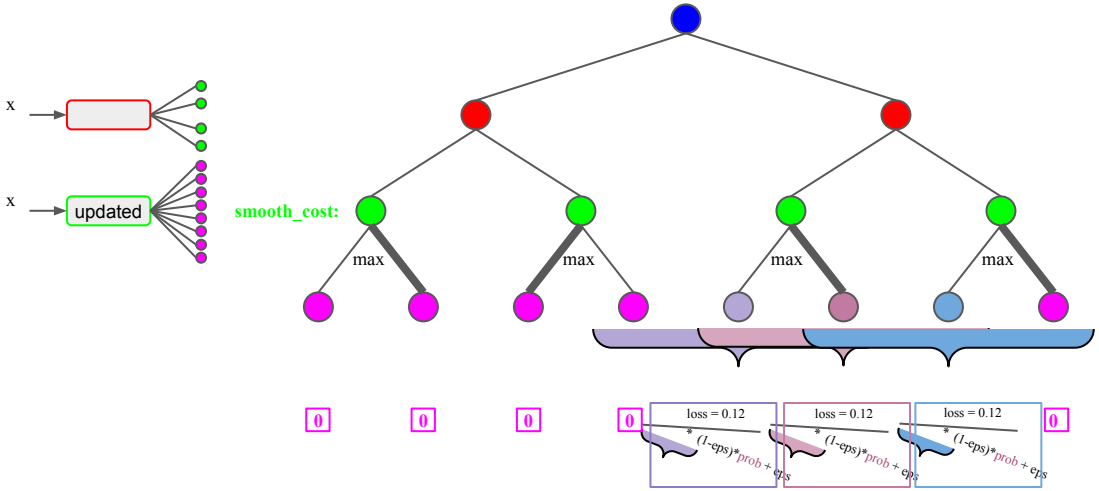


loss ft = sum(softmax((● ●) (● ●)) * **smooth cost**)

min(loss ft)

Update:


```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

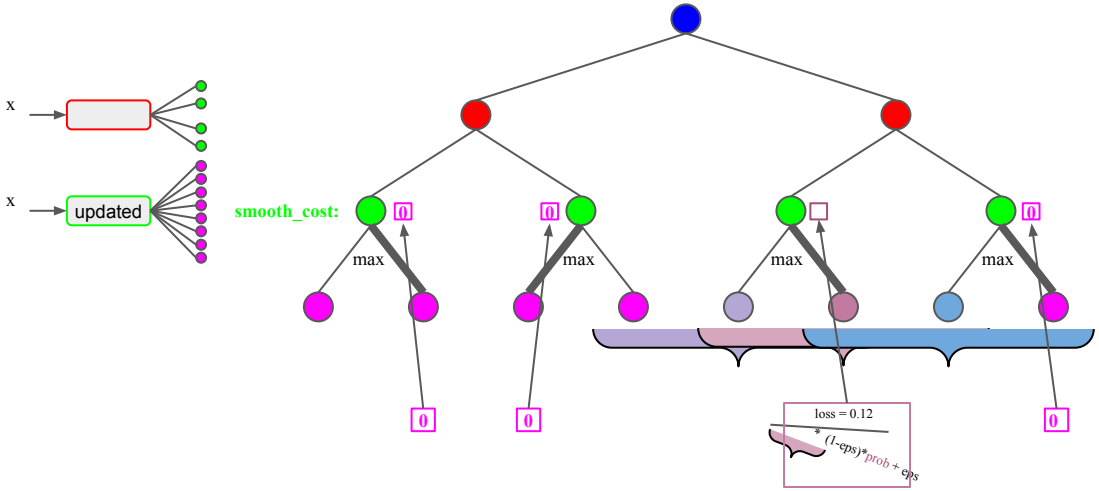


loss ft = sum(softmax((● ●) (● ●)) * **smooth cost**)

min(loss ft)

Update:


```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

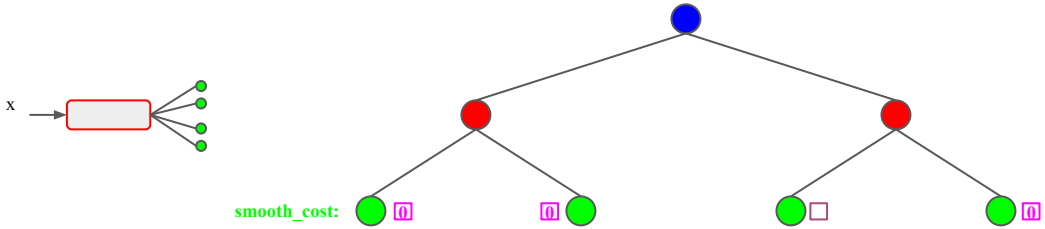


loss ft = sum(softmax(( ) ( )) ( )) * **smooth cost**)

min(loss ft)

Update:


```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

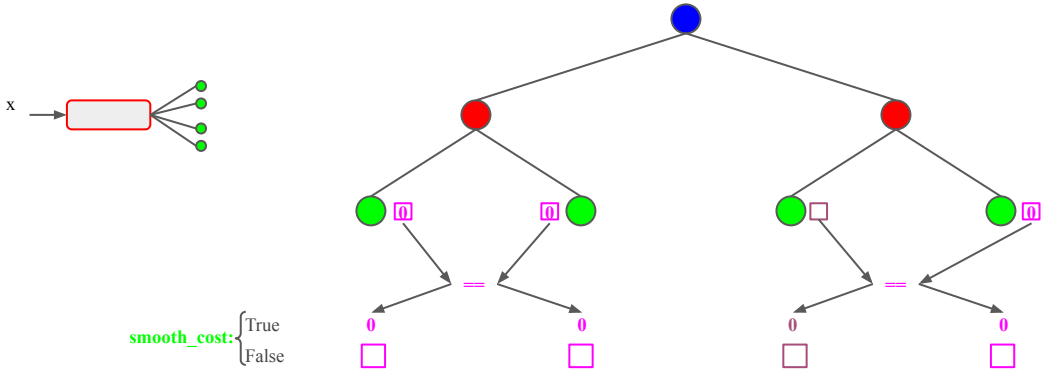


loss ft = sum(softmax(( ) ( )) * **smooth cost**)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

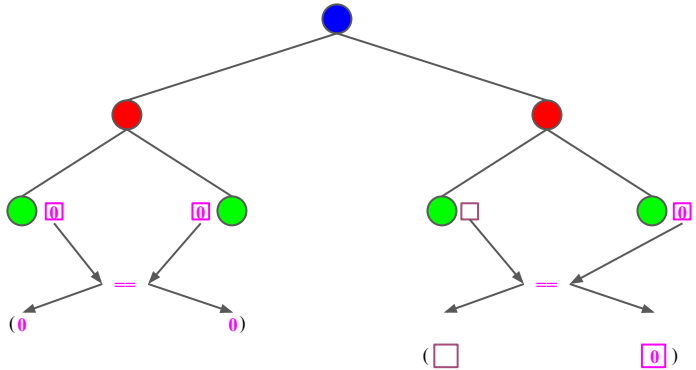


loss ft = sum(softmax((● ●) (● ●)) * **smooth cost**)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



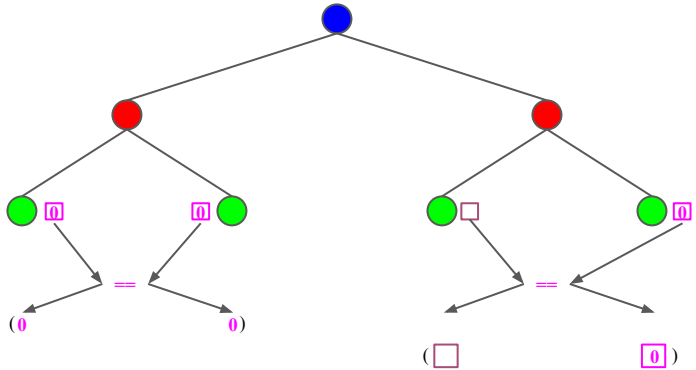
smooth_cost: { True / False

loss ft = sum(softmax((● ●) (● ●)) * **smooth cost**)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



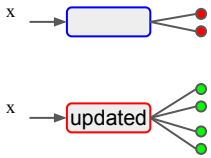
smooth_cost: { True / False

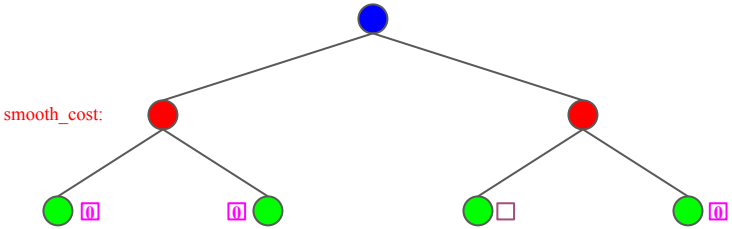
loss ft = sum(softmax( ) ( )) * **smooth cost**)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



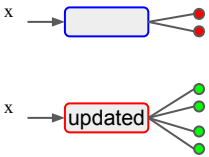


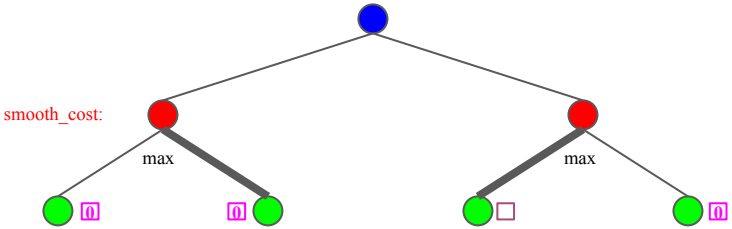
loss ft = sum(softmax(● ●) * smooth cost)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

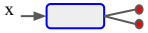




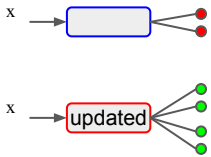
loss ft = sum(softmax(● ●) * smooth cost)

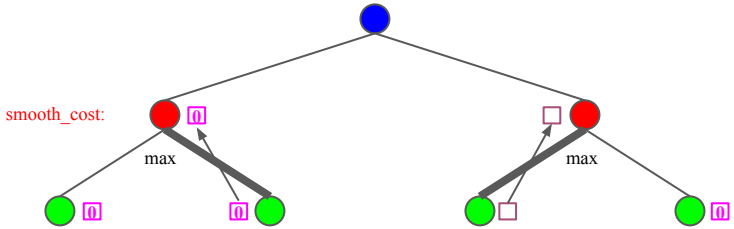
min(loss ft)

Update:




```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```

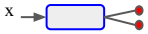




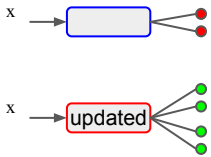
loss ft = sum(softmax(● ●) * smooth cost)

min(loss ft)

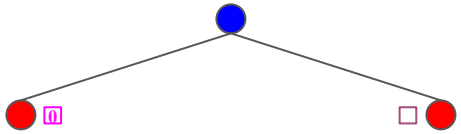
Update:




```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



smooth_cost:

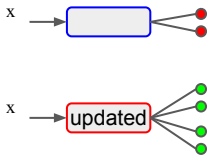


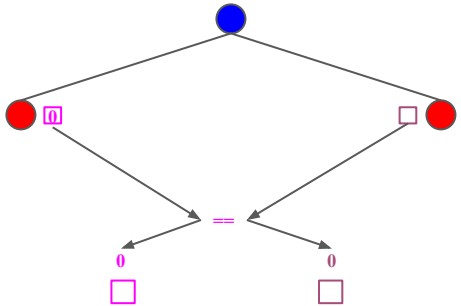
loss ft = sum(softmax(● ●) * smooth cost)

min(loss ft)

Update: x →

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```



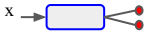


smooth_cost: { True / False

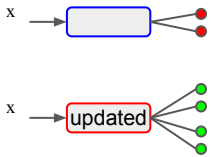
loss ft = sum(softmax(● ●) * smooth cost)

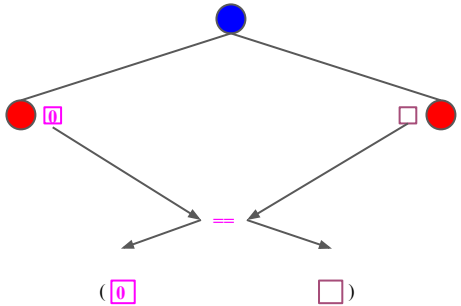
min(loss ft)

Update:




```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```





smooth_cost: { True / False

loss ft = sum(softmax(● ●)) * smooth cost)

min(loss ft)

Update:

```
def learn(
    self,
    obs: Observations,
    actions: Actions,
    probabilities: Probabilities,
    costs: Costs,
) -> None:
```