



CATX: contextual bandits with continuous action using trees with smoothing in JAX

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

InstaDeep © 2022 Copyright, all rights reserved.

Information contained within this presentation is the exclusive property of InstaDeep Limited and is protected by copyright. Any individual or organisation that wishes to duplicate, quote, amend or otherwise share such information shall only be permitted to do so upon obtaining explicit consent by InstaDeep in writing.

Software Licence Conditions.

Each user of CATX software (the “Software”), is granted, free of charge, a licence to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

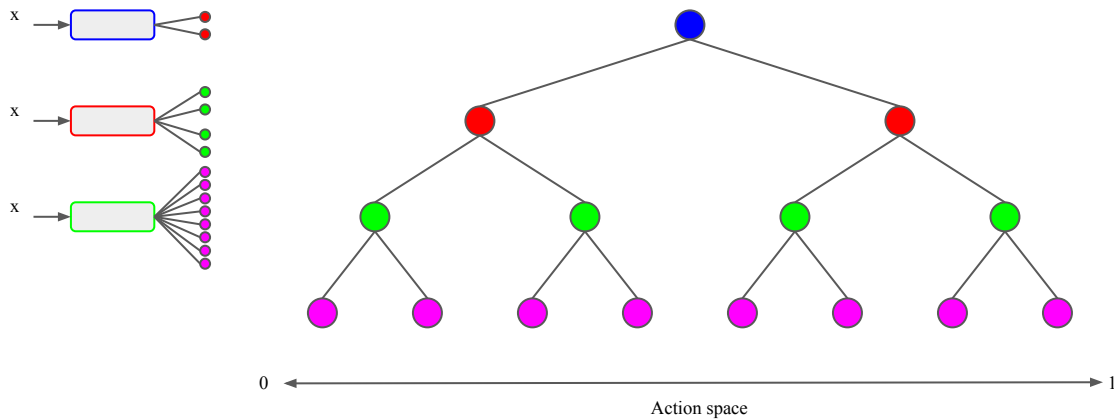
Tree

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 $2^{(\text{depth}+1)}$



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

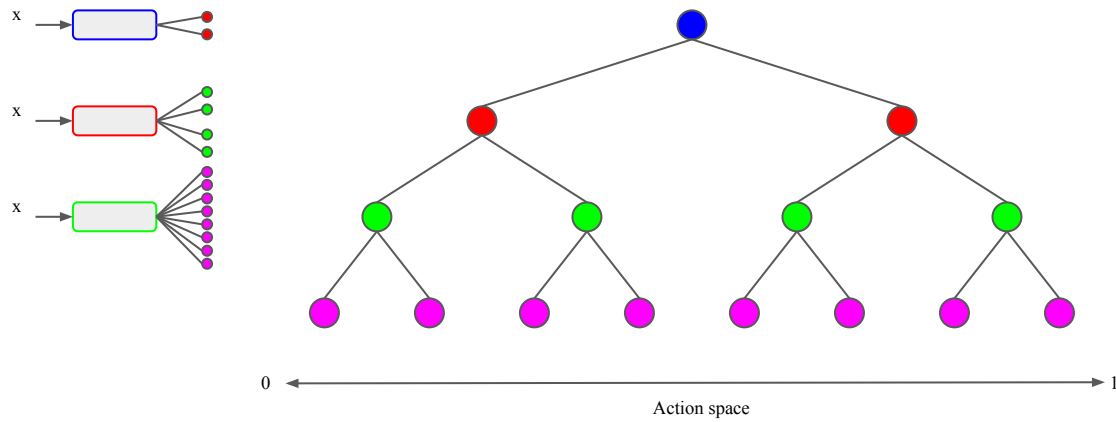
```

Tree parameters

Tree parameters

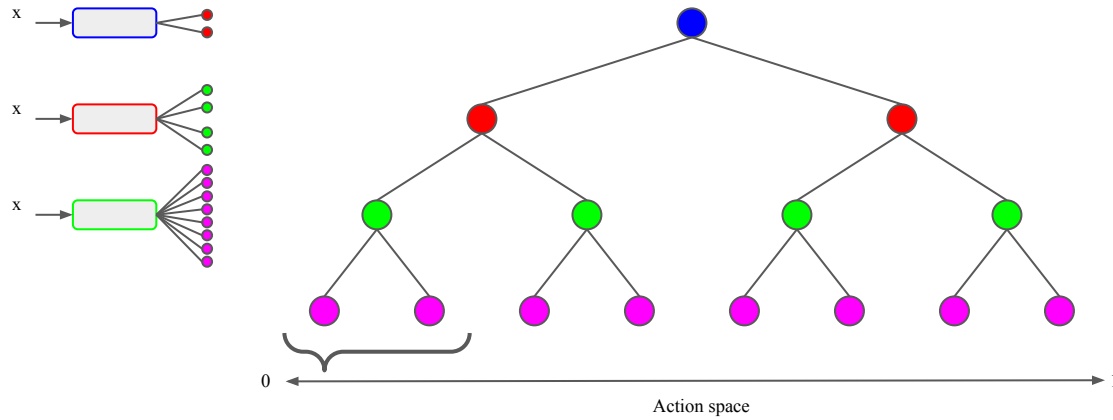
action spaces

```
@dataclass  
class TreeParameters:
```



Tree parameters

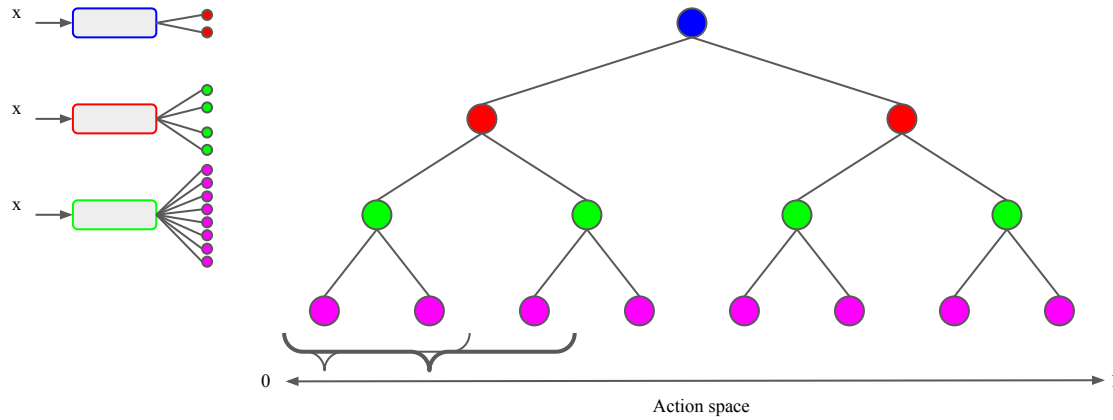
action spaces: each discretized action centroid covers $2 \times \text{bandwidth}$ of the action space



Tree parameters

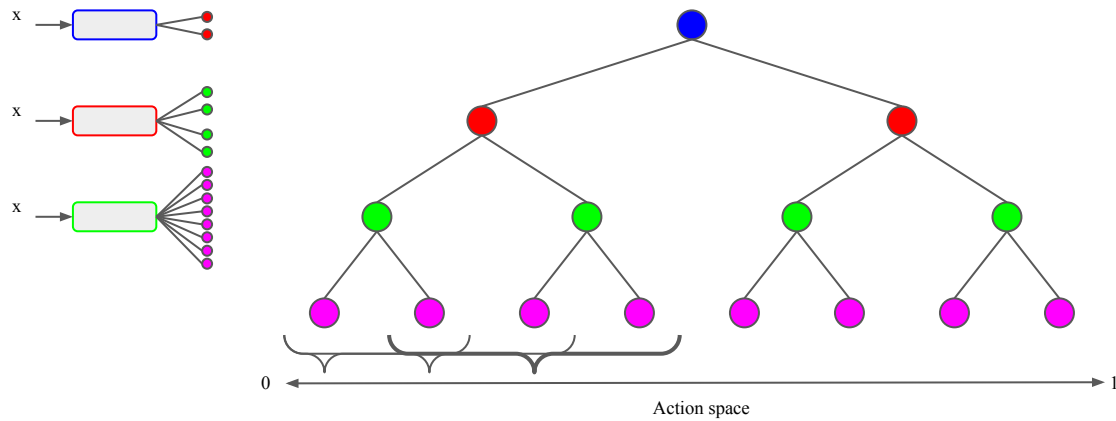
```
@dataclass  
class TreeParameters:
```

action spaces: each discretized action centroid covers $2 \times \text{bandwidth}$ of the action space



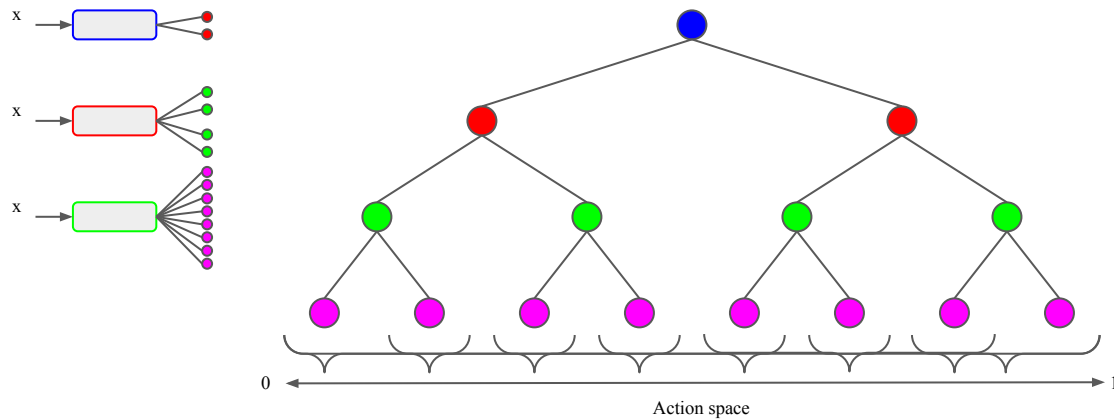
Tree parameters

action spaces: each discretized action centroid covers $2 \times \text{bandwidth}$ of the action space



Tree parameters

action spaces: each discretized action centroid covers $2 \times \text{bandwidth}$ of the action space

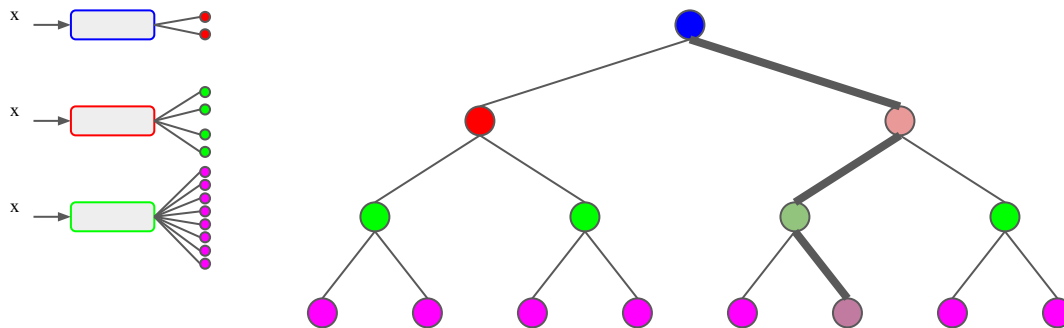


Action query

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

Action query

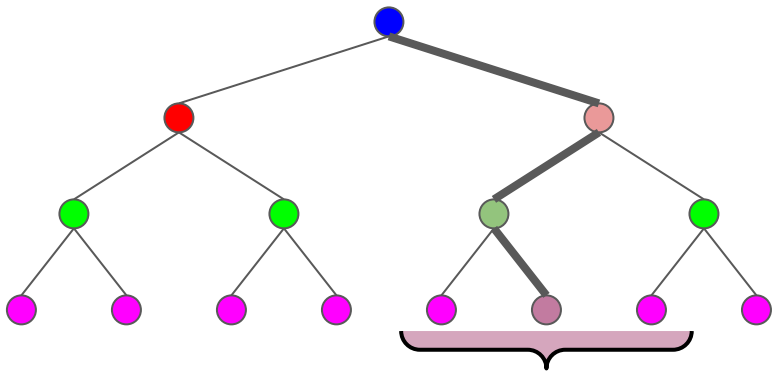
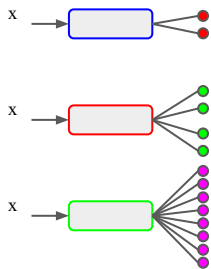
forward pass of the tree by following the max of the logits



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

Action query

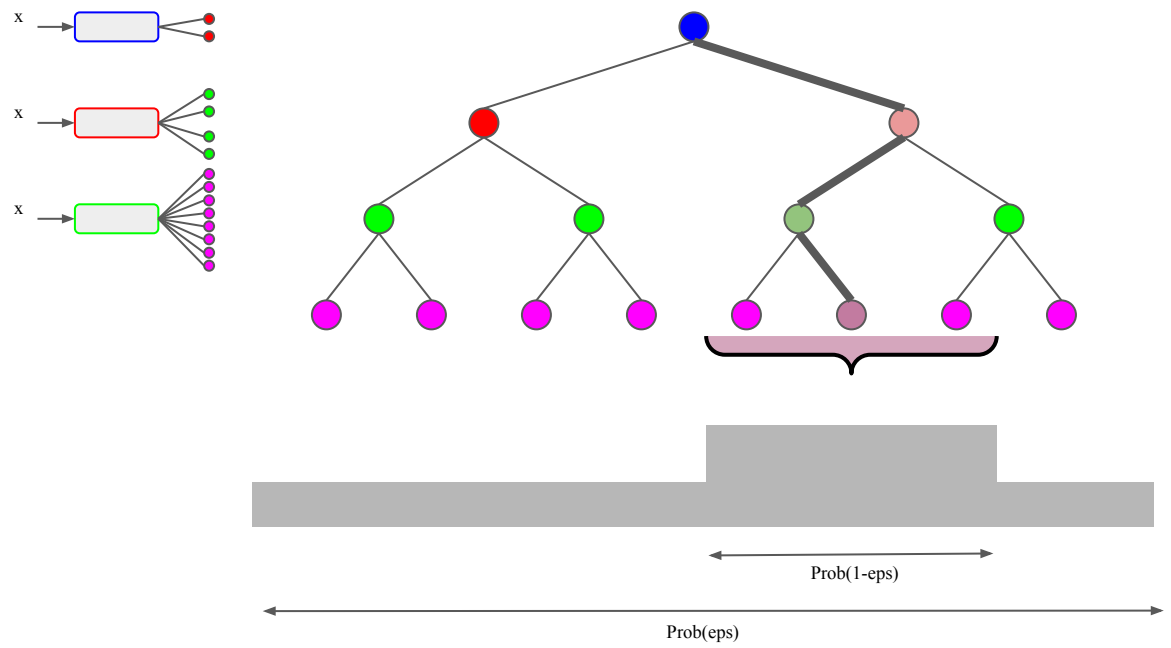
sample an action with eps-greedy



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

Action query

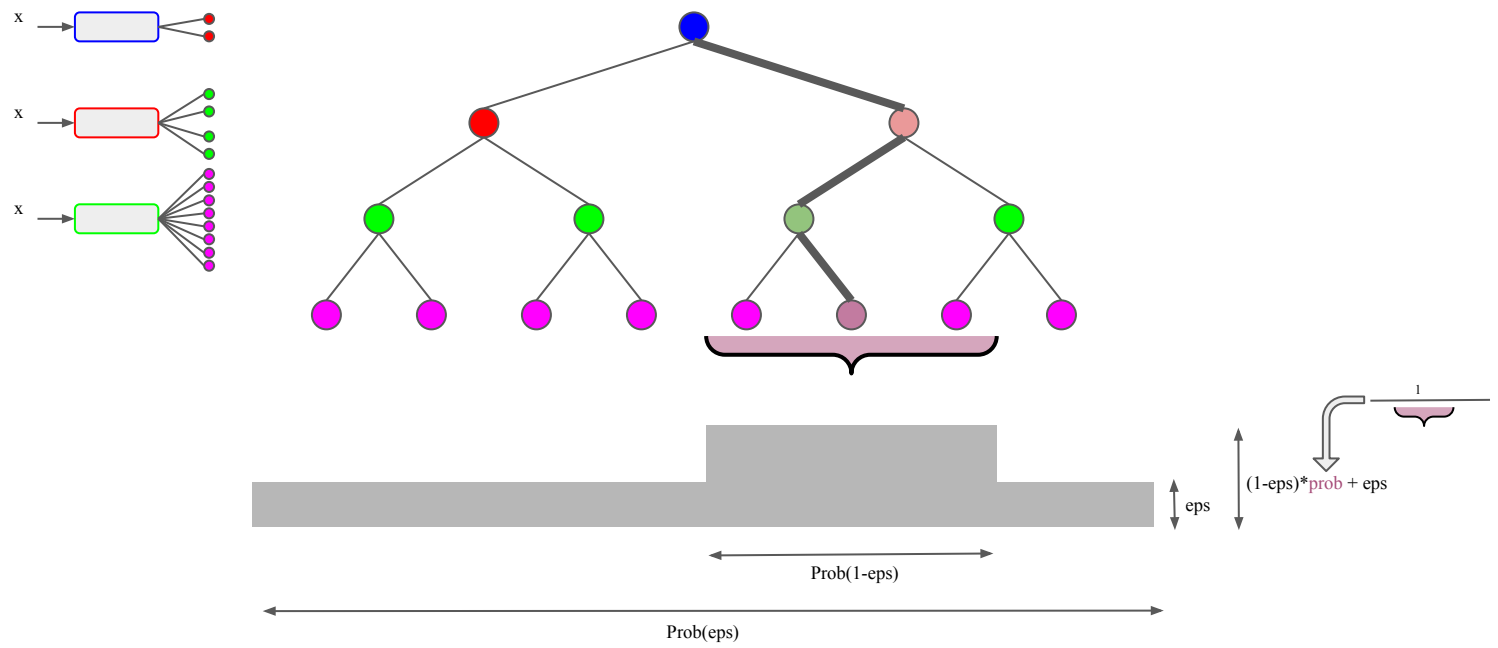
sample an action with eps-greedy



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

Action query

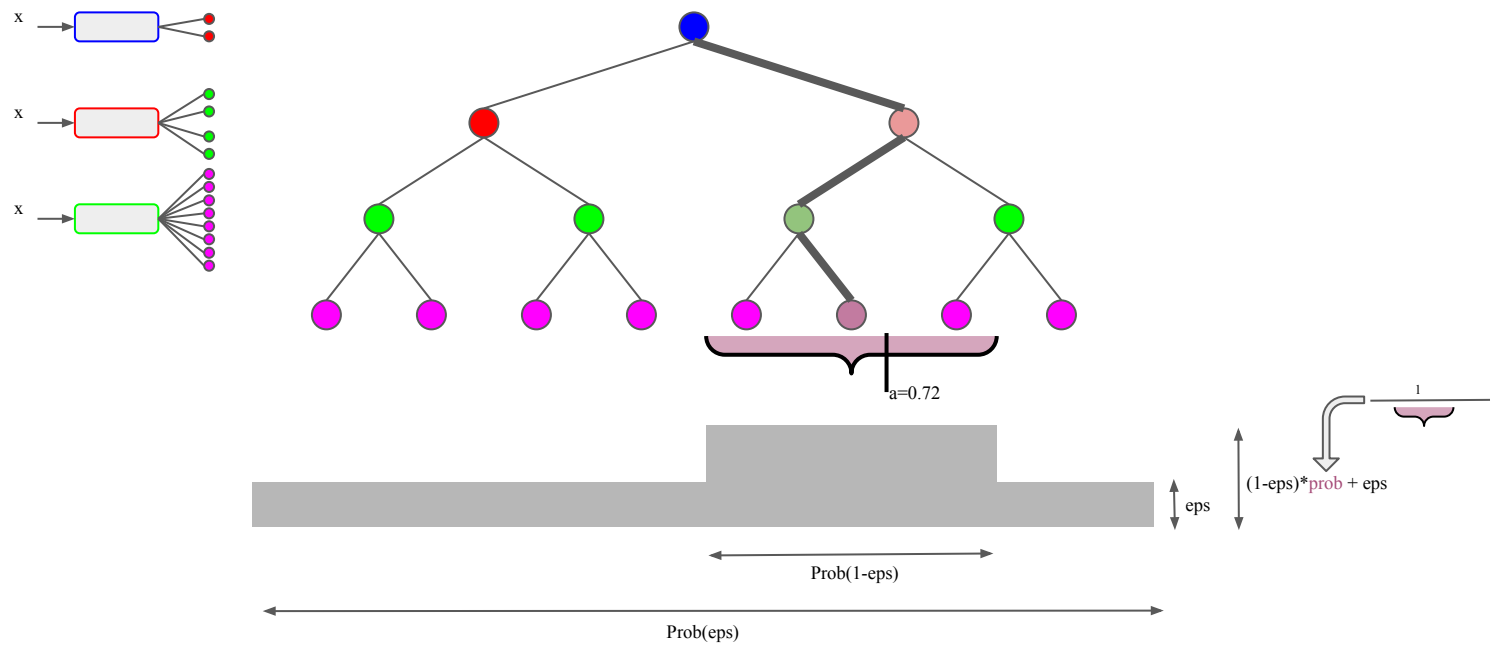
sample an action with eps-greedy



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

Action query

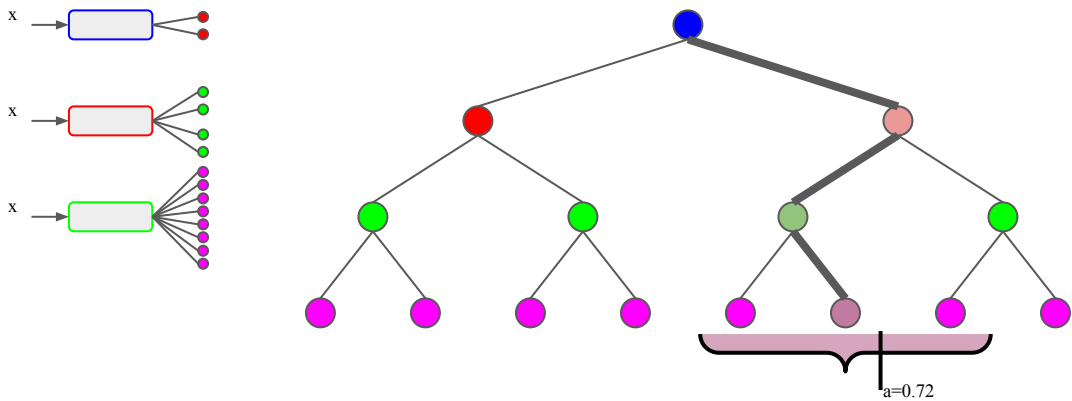
sample an action with eps-greedy (example: a=0.72)



Action cost

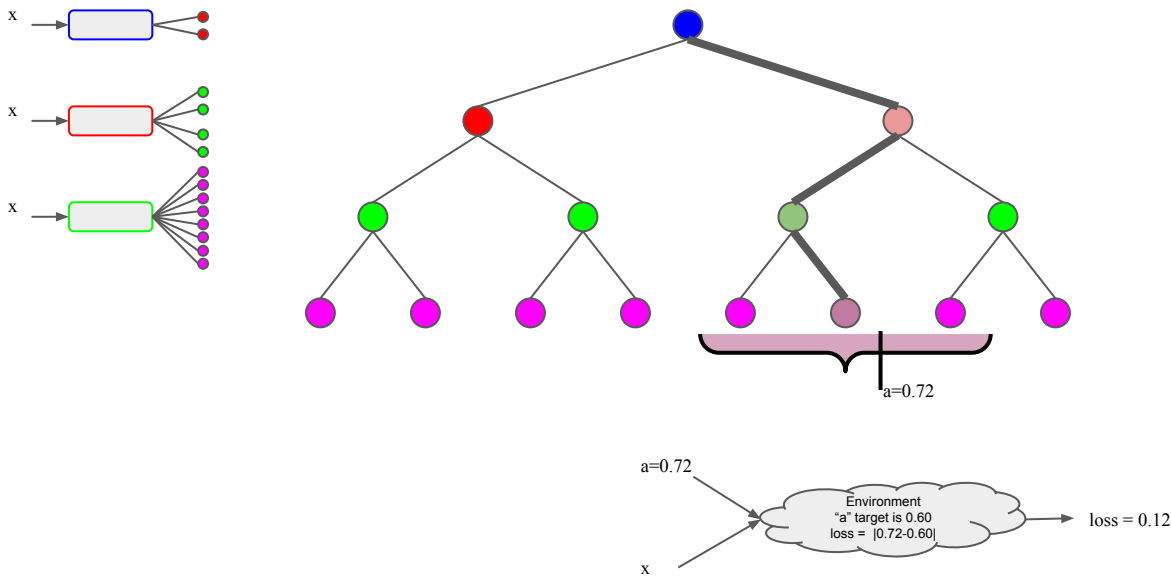
Action cost

apply action in the environment and receive cost feedback



Action cost

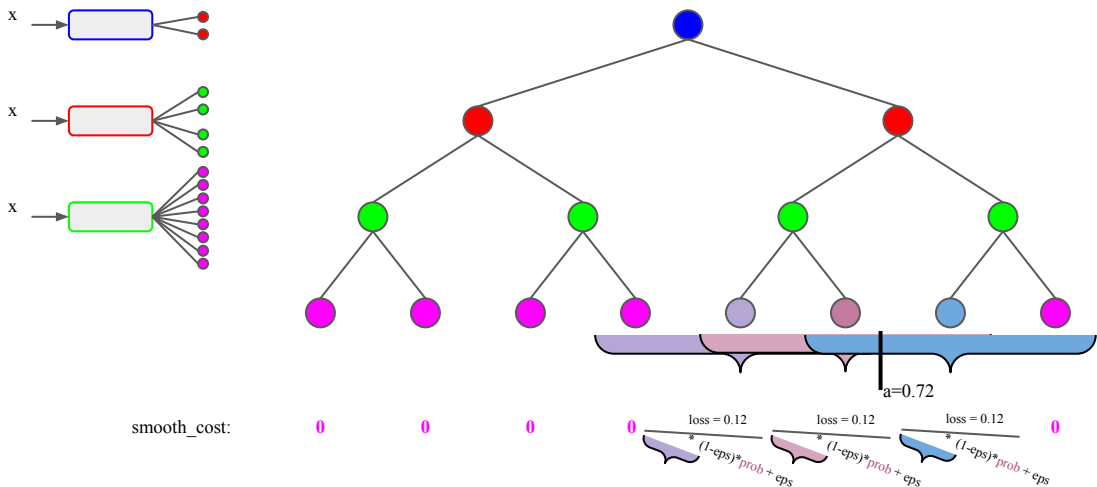
apply action in the environment and receive cost feedback




Action cost

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

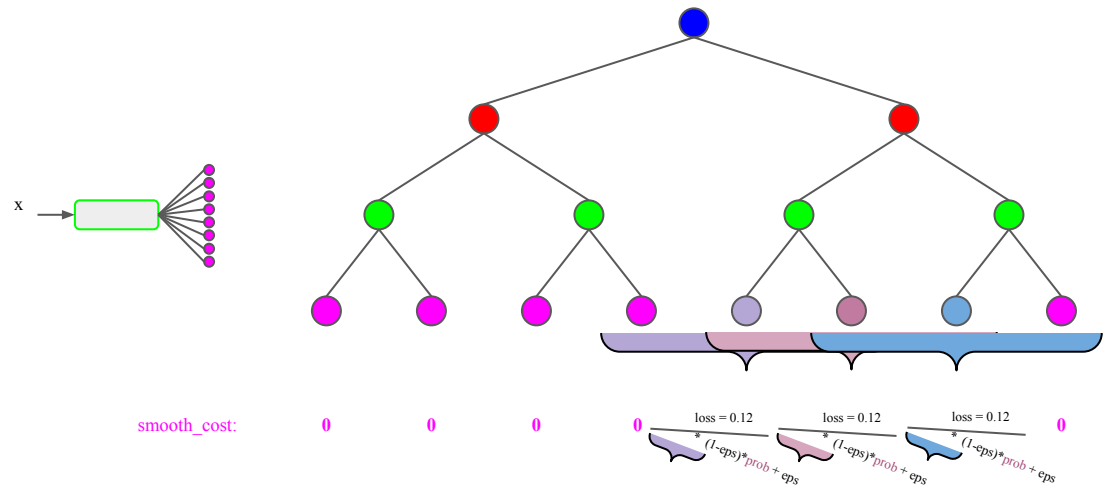
```
@functools.partial(jax.jit, static_argnames=("self",))
def _compute_smooth_costs(
    self, costs: JaxCosts, actions: JaxActions, probabilities: JaxProbabilities
) -> JaxCosts:
```



Update neural network weights

Update: $x \rightarrow$ 

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

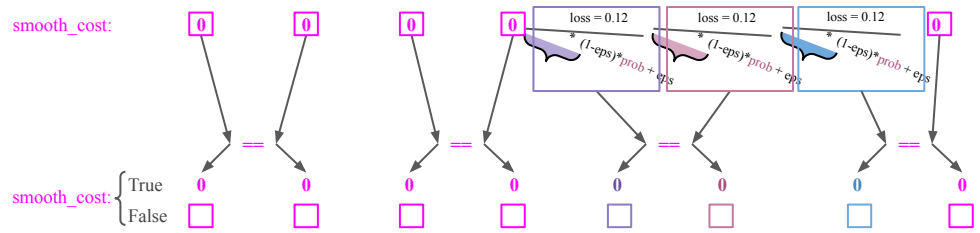
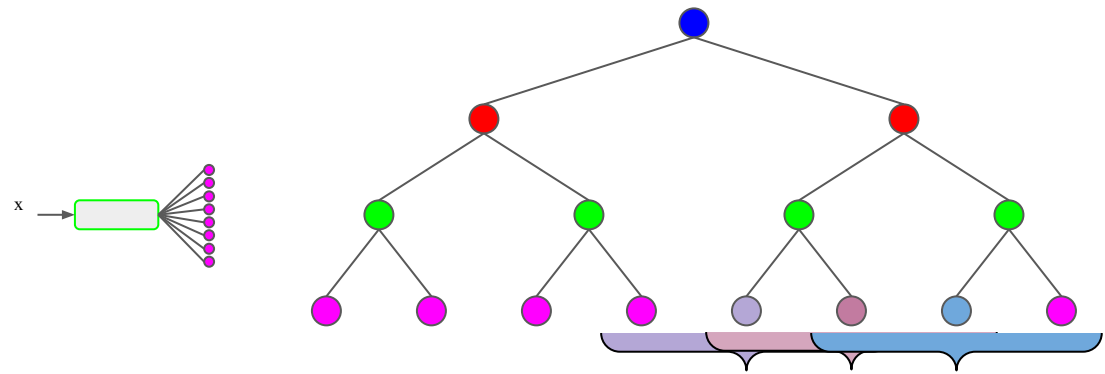


$\text{loss ft} = \text{sum}(\text{softmax}(\text{purple circles}) * \text{smooth cost})$

$\text{min}(\text{loss ft})$


Update: $x \rightarrow$  Only update nodes whose pair childs have different cost.

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

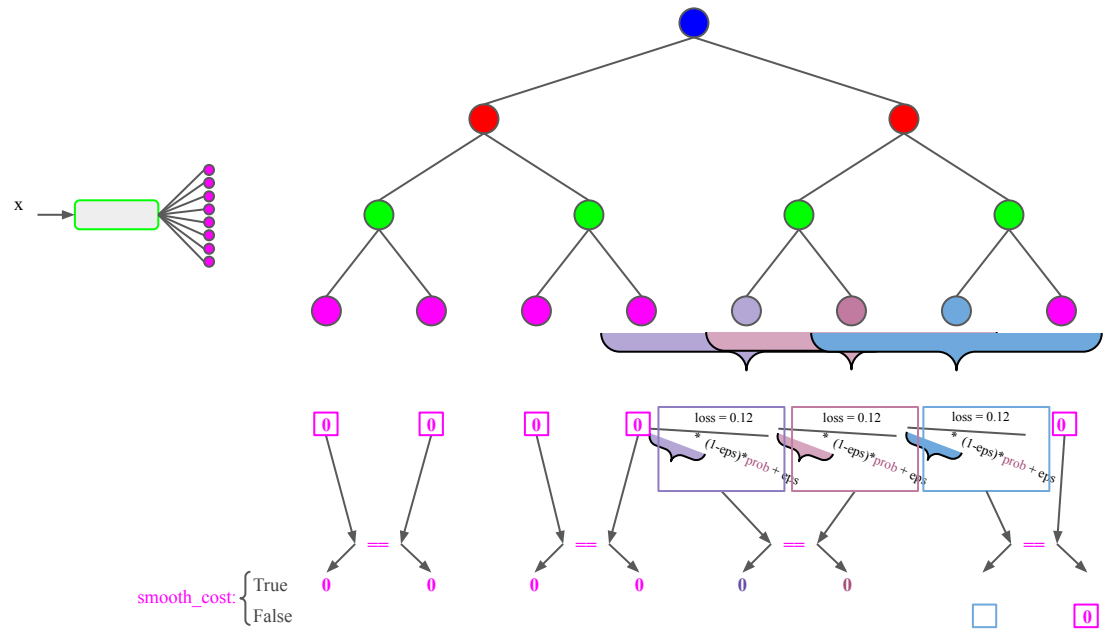


loss ft = sum(softmax() * smooth cost)

min(loss ft)

Update: $x \rightarrow$ 

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

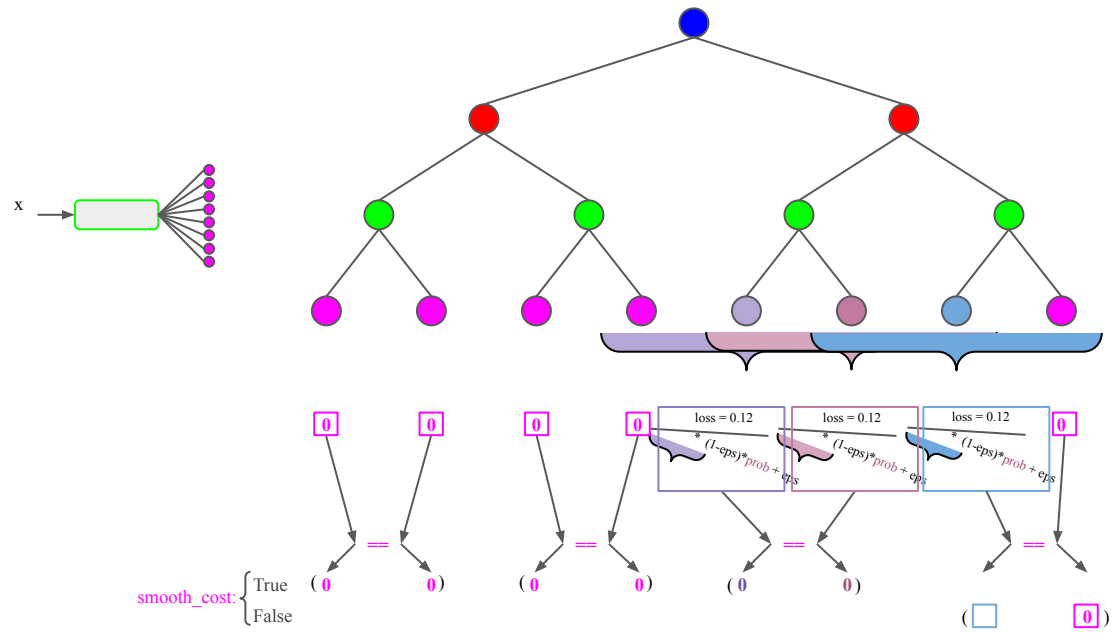


$loss_{ft} = \sum(softmax(\begin{matrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}) * smooth\ cost)$
 $min(loss_{ft})$

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


Update: $x \rightarrow$


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



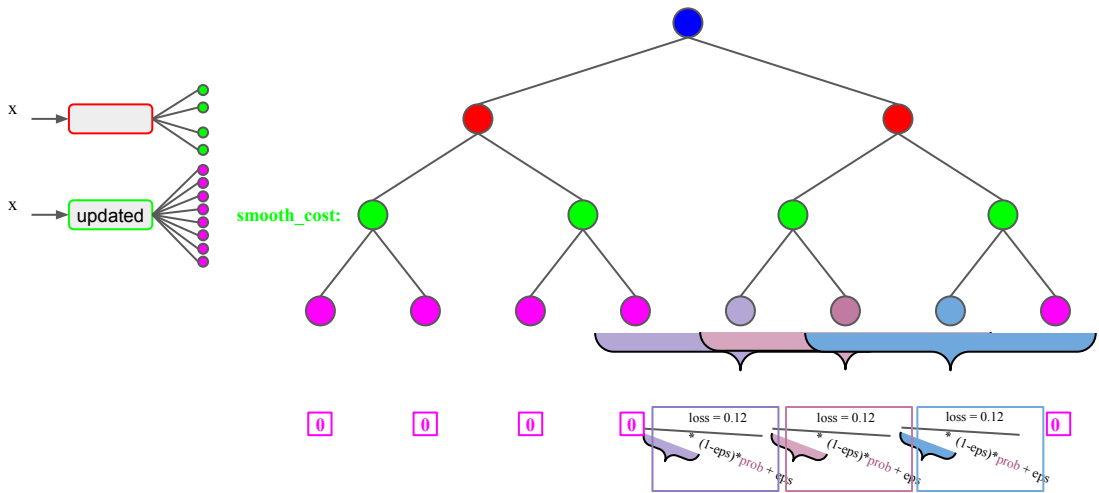
smooth_cost: {
True
False


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


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

Update: $x \rightarrow$ 

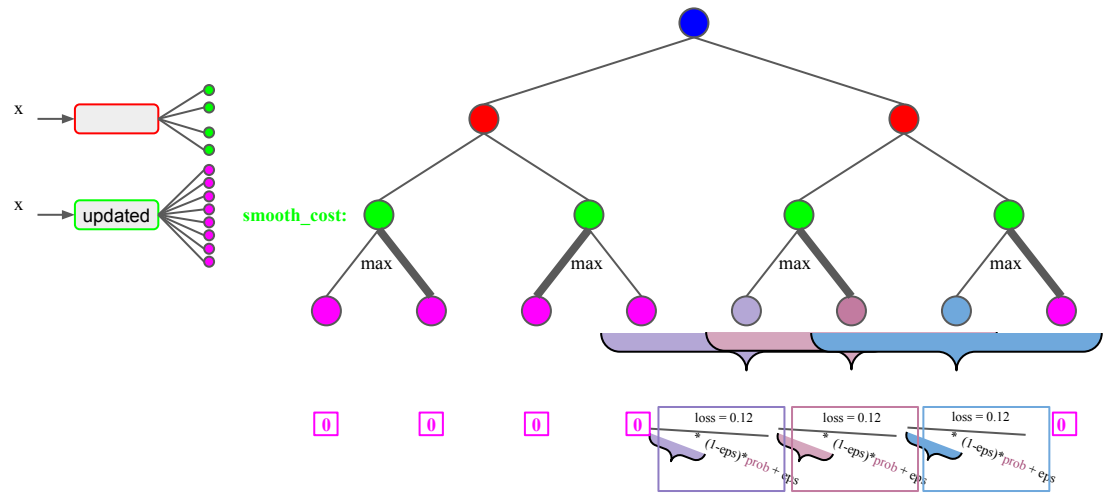
```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
) -> None:
```





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


Update: $x \rightarrow$ 

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

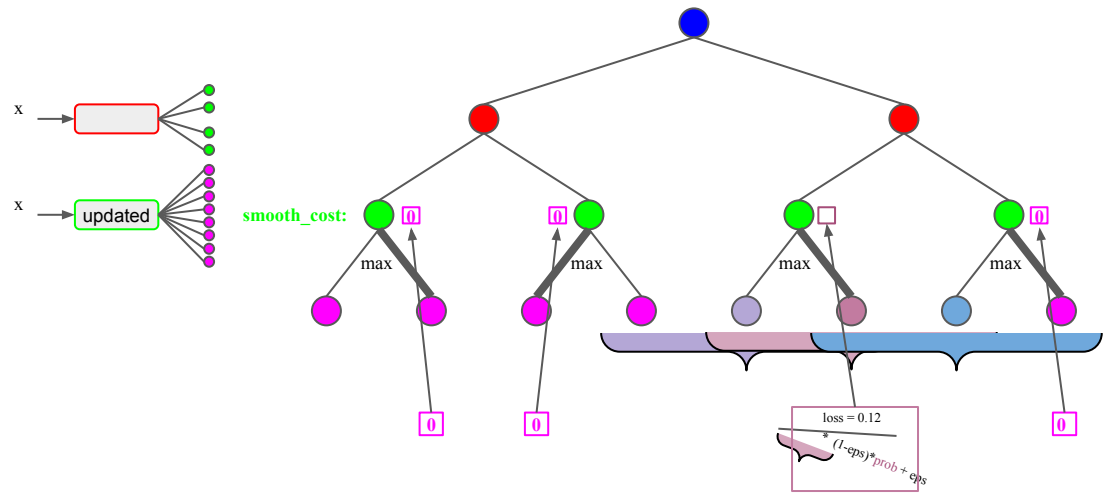


loss ft = sum(softmax() ()) * **smooth cost**)

min(loss ft)


Update: $x \rightarrow$ 

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

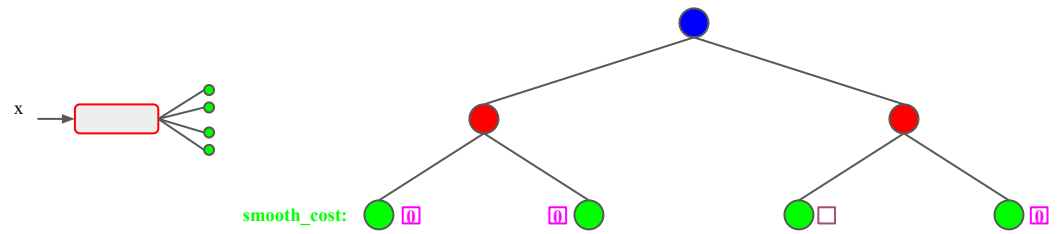





$$\text{loss ft} = \text{sum}(\text{softmax}(\text{green nodes}) * \text{smooth cost})$$


$$\text{min}(\text{loss ft})$$

Update: $x \rightarrow$ 

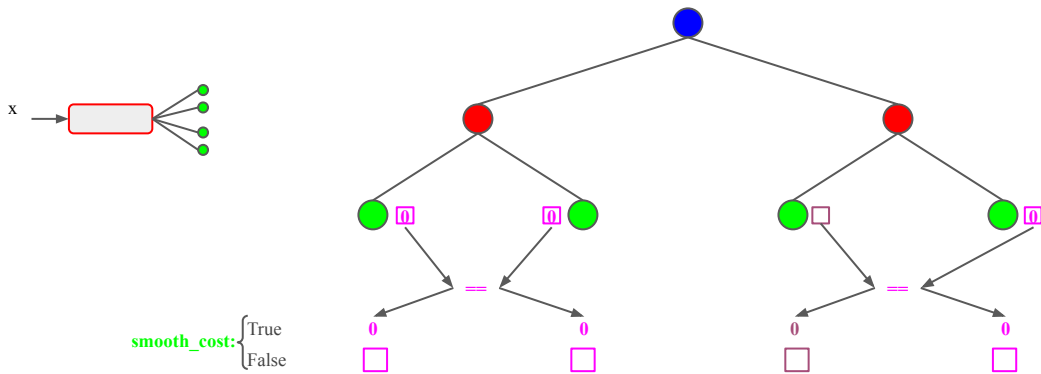
```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
) -> None:
```



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


Update: $x \rightarrow$ 

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

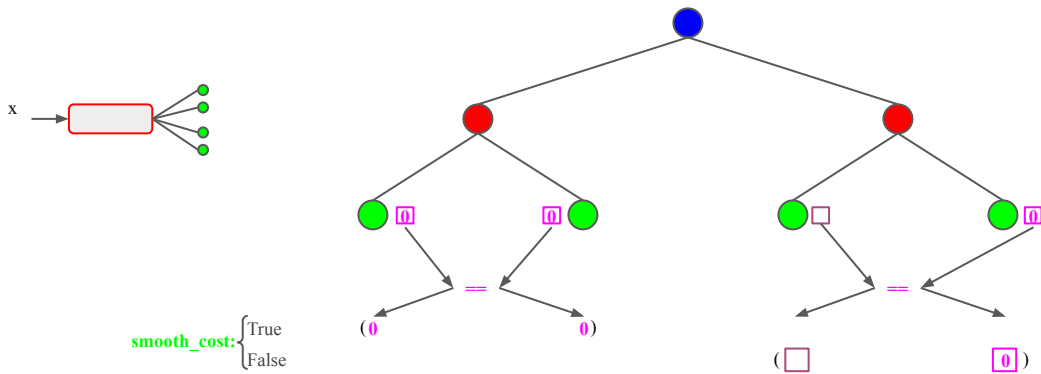


$loss\ ft = \sum (\text{softmax}(\text{green nodes}) * smooth\ cost)$

$\min(loss\ ft)$


Update: $x \rightarrow$ 

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

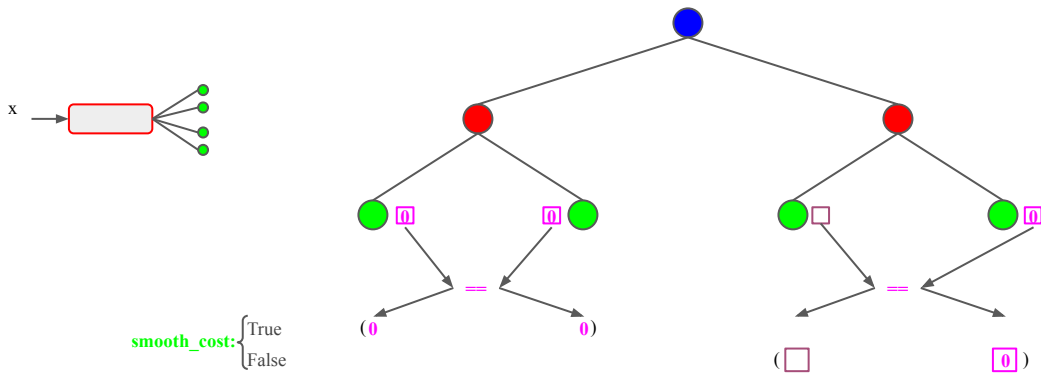


$loss\ ft = \sum(\text{softmax}(\text{green circles}) * smooth\ cost)$

$\min(loss\ ft)$


Update: $x \rightarrow$ 

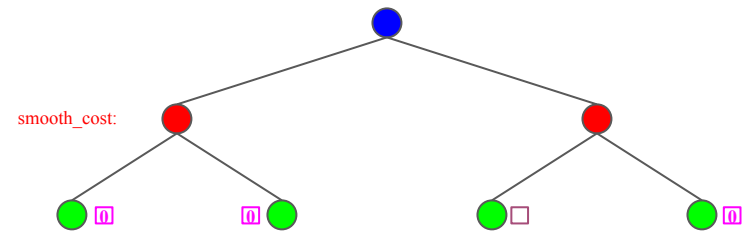
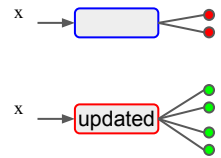
```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
    ) -> None:
```



$\text{smooth_cost} = \begin{cases} \text{True} \\ \text{False} \end{cases}$


$\text{loss ft} = \text{sum}(\text{softmax}(\begin{pmatrix} 0 & 0 \end{pmatrix}) * \text{smooth cost})$
 $\min(\text{loss ft})$

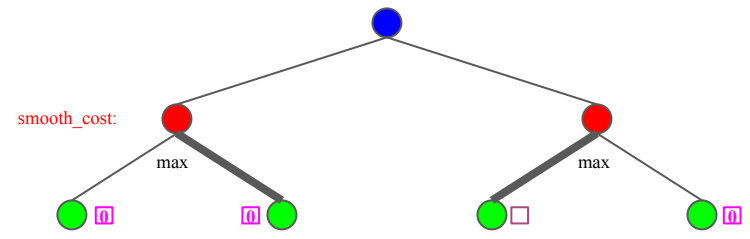
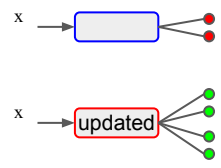
Update: $x \rightarrow$ 



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

$$\text{loss ft} = \text{sum}(\text{softmax}(\bullet \bullet) * \text{smooth cost})$$
$$\text{min}(\text{loss ft})$$


Update: $x \rightarrow$ 

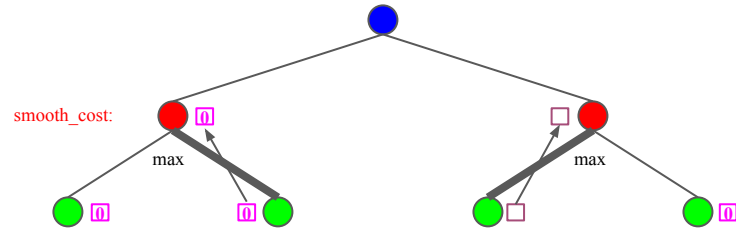
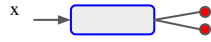


```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
) -> None:
```

$$\text{loss ft} = \text{sum}(\text{softmax}(\bullet \bullet) * \text{smooth cost})$$

$$\text{min}(\text{loss ft})$$


Update: $x \rightarrow$ 

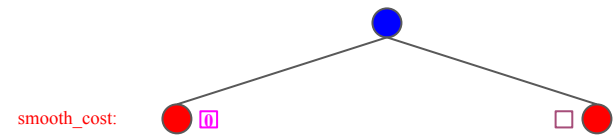
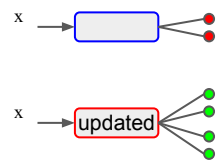


```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
) -> None:
```

$\text{loss ft} = \text{sum}(\text{softmax}(\bullet, \bullet) * \text{smooth cost})$

$\text{min}(\text{loss ft})$


Update: $x \rightarrow$ 

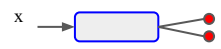


```
def learn(  
    self,  
    obs: Observations,  
    actions: Actions,  
    probabilities: Probabilities,  
    costs: Costs,  
) -> None:
```

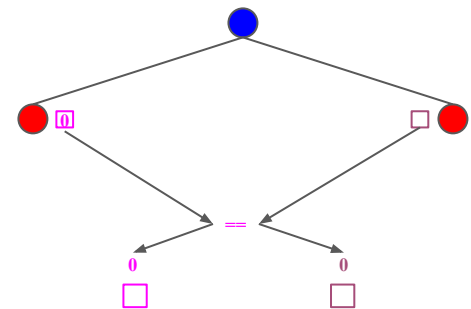
$$\text{loss ft} = \text{sum}(\text{softmax}(\text{red dots}) * \text{smooth cost})$$

$$\text{min}(\text{loss ft})$$

Update: $x \rightarrow$ 




$\text{smooth_cost} : \begin{cases} \text{True} \\ \text{False} \end{cases}$

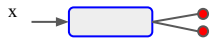


$$\text{loss ft} = \text{sum}(\text{softmax}(\bullet \bullet) * \text{smooth cost})$$

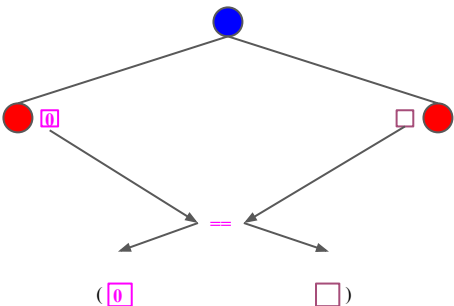
$$\text{min}(\text{loss ft})$$

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

Update: $x \rightarrow$ 



$\text{smooth_cost} = \begin{cases} \text{True} \\ \text{False} \end{cases}$

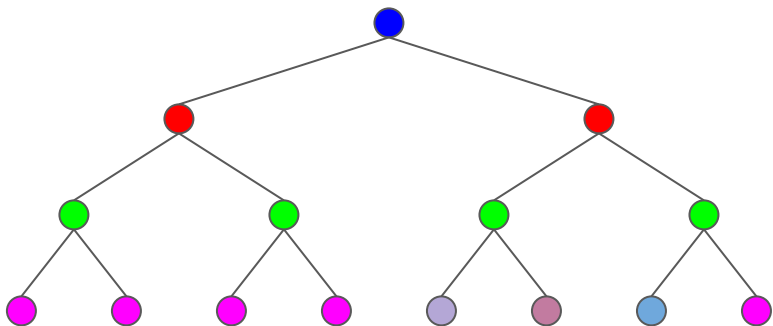
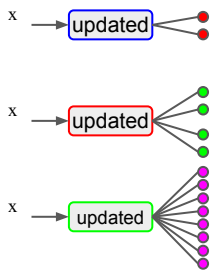


$$\text{loss ft} = \text{sum}(\text{softmax}(\text{red_dot_red_dot}) * \text{smooth cost})$$

$\min(\text{loss ft})$

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

Update:



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