

```
In [ ]: import marimo as mo
```

CartPole, decision trees, and bagging

Lecture map: random experience → one Q-tree at several depths → inspect its nested questions and four-dimensional bins → bag several Q-trees → play CartPole.

This is a **marimo** notebook. Python 3.10+ is suitable. Install once with:

```
python -m pip install marimo "gymnasium[classic-control]" scikit-learn numpy matplotlib pillow
```

The notebook uses Gymnasium's `CartPole-v1` (four state coordinates, two actions). It collects random transitions once, then uses the **same transitions** for every model. A `DecisionTreeRegressor` approximates $Q(s, a)$. The bagged model averages twelve such regressors; it is trained through the same sequence of Q-target updates.

Start locally with `marimo edit CartPole_decision_trees_and_bagging.py`, or use `marimo run` for an app view. Training takes roughly a minute and runs once; changing a plot control reuses the fitted models. The comparison is an experiment, not a guarantee that deeper trees or bagging always win.

```
In [ ]: import time
import numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.ensemble import BaggingRegressor

FEATURES = ['cart x', 'cart velocity', 'pole angle', 'angular velocity', 'action']
STATE = FEATURES[:4]
plt.rcParams.update({'figure.figsize': (10, 4), 'font.size': 11})
print('Gymnasium', gym.__version__)
```

Gymnasium 1.3.0

1. Let a random agent play

Each row contains a state, an action, reward, next state, and whether the pole actually fell. Gymnasium gives +1 per step, including the failure step. The time-limit flag `truncated` is a cap at 500, **not** a physical failure; the Bellman target below masks only `terminated`.

```
In [ ]: def collect_random(epochs=800, seed=11):
    env = gym.make('CartPole-v1')
    rng = np.random.default_rng(seed)
```

```

rows, lengths, first_episode = [], [], []
for episode in range(episodes):
    state, _ = env.reset(seed=seed + episode)
    steps = 0
    while True:
        action = int(rng.integers(2))
        next_state, reward, terminated, truncated, _ = env.step(action)
        row = (state.copy(), action, reward, next_state.copy(),
terminated)
        rows.append(row)
        if episode == 0:
            first_episode.append(row)
        state = next_state
        steps += 1
        if terminated or truncated:
            break
    lengths.append(steps)
env.close()
columns = tuple(np.asarray([row[j] for row in rows]) for j in range(5))
return columns, np.asarray(lengths), first_episode

data, random_lengths, first_episode = collect_random()
states, actions, rewards, next_states, terminated = data
print(f'{len(random_lengths)} random games → {len(states):,} transitions; '
      f'mean {random_lengths.mean():.1f} steps, median
{np.median(random_lengths):.0f}.')
print('First random game:')
print('time    angle (deg)    angular speed    action    return under RANDOM
continuation')
for t, row in enumerate(first_episode[:8]):
    state, action, _, _, _ = row
    print(f'{t:4d} {np.rad2deg(state[2]):12.2f} {state[3]:15.2f} '
          f'{"right" if action else "left":>7} {len(first_episode)-t:17d}')
print('The last column measures the random continuation, not the best
action value.')

```

800 random games → 18,302 transitions; mean 22.9 steps, median 19.

First random game:

time	angle (deg)	angular speed	action	return under RANDOM continuation
0	0.58	-0.05	left	36
1	0.53	0.25	left	35
2	0.81	0.54	right	34
3	1.44	0.26	left	33
4	1.73	0.56	right	32
5	2.37	0.27	right	31
6	2.68	-0.01	right	30
7	2.67	-0.28	left	29

The last column measures the random continuation, not the best action value.

2. Turn the transitions into repeated regression problems

At iteration k , fit the targets

$$y_i = r_i + \gamma(1 - \text{terminated}_i) \max_{a' \in \{0,1\}} Q_{k-1}(s'_i, a'), \quad \gamma = 0.99.$$

The first target is simply r_i . Fit a **new tree** at each iteration; no tree is associated with one episode. For a particular fitted model, choose the larger of $Q(s, 0)$ and $Q(s, 1)$. A shallow tree may assign both actions the same prediction, in which case `argmax` chooses left; a flat policy near 9 steps is a real failure mode, not a plotting bug.

```
In [ ]: def state_action(states, action):
    return np.column_stack((states, np.broadcast_to(action, len(states))))

def q_pair(model, states):
    return (model.predict(state_action(states, 0)),
            model.predict(state_action(states, 1)))

def greedy_action(model, state):
    qleft, qright = q_pair(model, np.asarray(state).reshape(1, 4))
    return int(qright[0] > qleft[0])

def fit_q(data, depth=10, trees=1, iterations=30, gamma=0.99):
    states, actions, rewards, next_states, terminated = data
    X = state_action(states, actions)
    model = None
    checkpoints = {}
    for k in range(iterations):
        if model is None:
            target = rewards.copy()
        else:
            qleft, qright = q_pair(model, next_states)
            target = rewards + gamma * ~terminated * np.maximum(qleft,
qright)
            tree = DecisionTreeRegressor(max_depth=depth, min_samples_leaf=20,
random_state=41)
            model = tree if trees == 1 else BaggingRegressor(estimator=tree,
n_estimators=trees, max_samples=0.8, bootstrap=True, random_state=41,
n_jobs=1)
            model.fit(X, target)
            if k in (0, 5, 14, iterations - 1):
                checkpoints[k + 1] = model
    return (model, checkpoints)

DEPTHS = (2, 4, 6, 10)
start = time.perf_counter()
models = {}
history = {}
for _depth in DEPTHS:
    models[_depth], history[_depth] = fit_q(data, depth=_depth)
    print(f'depth {_depth}: fitted in {time.perf_counter() - start:.1f} s')
bagged, bagged_history = fit_q(data, depth=10, trees=12)
print(f'12 bagged depth=10 trees: all fitted in {time.perf_counter() -
start:.1f} s')
```

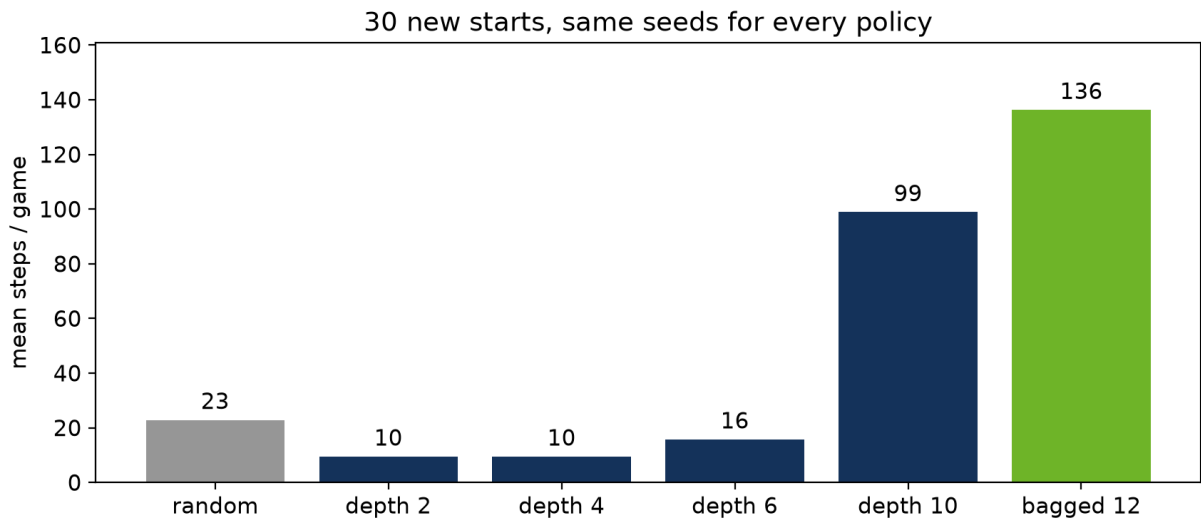
```
depth 2: fitted in 0.3 s
depth 4: fitted in 0.9 s
depth 6: fitted in 1.8 s
depth 10: fitted in 3.1 s
12 bagged depth-10 trees: all fitted in 11.7 s
```

3. Does depth help? Test on new games

Every fitted policy starts from the **same 30 seeded initial conditions**. These are online rollouts, so a long bar means the policy really kept the pole up; training loss on the random transitions would not establish that.

```
In [ ]: def rollout(model=None, seed=9000, random=False, render=False):
    env = gym.make('CartPole-v1', render_mode='rgb_array' if render else
None)
    state, _ = env.reset(seed=seed)
    rng = np.random.default_rng(seed + 76543)
    trajectory = [state.copy()]
    frames = [env.render()] if render else []
    for t in range(500):
        action = int(rng.integers(2)) if random else greedy_action(model,
state)
        state, _, term, trunc, _ = env.step(action)
        trajectory.append(state.copy())
        if render:
            frames.append(env.render())
        if term or trunc:
            break
    env.close()
    return (np.asarray(trajectory), frames)
seeds = range(9000, 9030)
scores = {'random': np.array([len(rollout(seed=s, random=True)[0]) - 1 for
s in seeds])}
for _depth, _model in models.items():
    scores[f'depth {_depth}'] = np.array([len(rollout(_model, seed=s)[0]) -
1 for s in seeds])
scores['bagged 12'] = np.array([len(rollout(bagged, seed=s)[0]) - 1 for s
in seeds])
_score_fig, ax = plt.subplots(figsize=(10, 4))
labels = list(scores)
means = [scores[key].mean() for key in labels]
ax.bar(labels, means, color=['#999999'] + ['#17365c'] * 4 + ['#71b629'])
for i, key in enumerate(labels):
    ax.text(i, means[i] + 4, f'{means[i]:.0f}', ha='center')
ax.set(ylim=(0, max(means) * 1.18), ylabel='mean steps / game', title='30
new starts, same seeds for every policy')
for key in labels:
    arr = scores[key]
    print(f'{key:>10}: mean {arr.mean():6.1f}, median
{np.median(arr):5.0f}, range {arr.min():3d}-{arr.max():3d}')
_score_fig
```

random:	mean	22.9,	median	18,	range	10– 73
depth 2:	mean	9.5,	median	9,	range	8– 11
depth 4:	mean	9.5,	median	9,	range	8– 11
depth 6:	mean	15.8,	median	14,	range	12– 30
depth 10:	mean	98.9,	median	97,	range	94–107
bagged 12:	mean	136.3,	median	136,	range	129–148



4. Open one tree

Each fitted tree asks one feature-threshold question per internal node. The root is a single question over the **five inputs** (four state coordinates and the proposed action). As the tree gets deeper, different leaves represent smaller regions of this input space.

These depth settings retrain separate trees. Their root questions can change; they are not successive snapshots of one fixed tree. The plot cuts off after a chosen level for readability. Use the controls below to reveal more levels, choose an individual bagged member, or change which fitted depth to inspect.

```
In [ ]: def tree_overview(tree):
    tr = tree.tree_
    depth = np.zeros(tr.node_count, dtype=int)
    for node in range(tr.node_count):
        if tr.children_left[node] != -1:
            depth[tr.children_left[node]] = depth[node] + 1
            depth[tr.children_right[node]] = depth[node] + 1
    counts = np.zeros((int(depth.max())+1, len(FEATURES)), dtype=int)
    for node in range(tr.node_count):
        if tr.feature[node] >= 0:
            counts[depth[node], tr.feature[node]] += 1
    return counts

def inspect_tree(depth=10, levels=2, member=0):
    tree = models[depth] if member == 0 else bagged.estimators_[member-1]
    counts = tree_overview(tree)
    print(f'{"single tree" if member == 0 else f"bagged member {member}"}:
    ,
        f'{tree.tree_.node_count} nodes, {tree.tree_.n_leaves} leaves, '
```

```

f'depth {tree.tree_.max_depth}')
fig, (ax, bx) = plt.subplots(1, 2, figsize=(16, 5),
                             gridspec_kw={'width_ratios': [3, 2]})
plot_tree(tree, max_depth=levels, feature_names=FEATURES, filled=True,
          rounded=True, impurity=False, precision=2, fontsize=8,
          node_ids=True, ax=ax)
ax.set_title(f'Top {levels+1} levels (ellipsis = hidden subtree)')
bottom = np.zeros(len(counts))
for j, name in enumerate(FEATURES):
    bx.bar(range(len(counts)), counts[:, j], bottom=bottom, label=name)
    bottom += counts[:, j]
bx.set(xlabel='node depth', ylabel='number of split questions',
       title='All split variables, including hidden levels')
bx.legend(fontsize=8)
plt.tight_layout()
return fig

```

```

In [ ]: tree_depth = mo.ui.dropdown(options=list(DEPTHS), value=10, label='Fitted
depth')
tree_levels = mo.ui.slider(start=1, stop=4, value=2, label='Visible
levels')
tree_member = mo.ui.slider(start=0, stop=12, value=0, label='Bagged member
(0 = single tree)')
mo.hstack([tree_depth, tree_levels, tree_member], justify='start', gap=2)

```

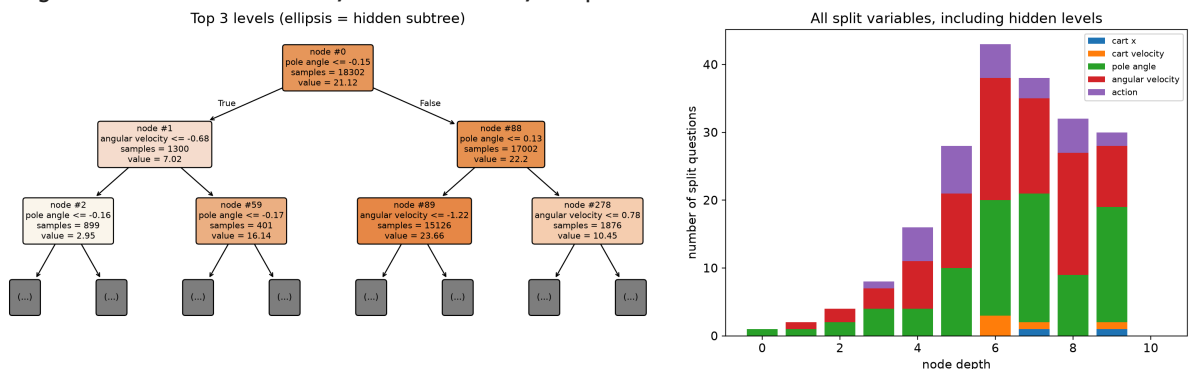


```

In [ ]: inspect_tree(tree_depth.value, tree_levels.value, tree_member.value)

```

single tree: 405 nodes, 203 leaves, depth 10



5. Follow one state down the nested questions

Take an actual state from a learned run. The two candidate actions may traverse different branches of the same Q-tree. The printed node IDs match the tree plot when `node_ids=True`; the route reports each feature, threshold, branch, and final predicted return.

```
In [ ]: def trace_path(tree, state, action):
        z = np.r_[state, action]
        tr = tree.tree_
        node = 0
        print('Input:', ', '.join(f'{n}={v:.3f}' for n, v in zip(FEATURES, z)))
        while tr.children_left[node] != -1:
            feature = tr.feature[node]
            threshold = tr.threshold[node]
            go_left = z[feature] <= threshold
            print(f'node {node:>3}: {FEATURES[feature]:>16} '
                  f'{z[feature]:>8.3f} {"≤" if go_left else ">"} '
                  f'{threshold:>8.3f} '
                  f'→ {"left" if go_left else "right"}')
            node = tr.children_left[node] if go_left else
tr.children_right[node]
        print(f'leaf {node}: Q ≈ {tr.value[node,0,0]:.2f} discounted steps')
        return node

example_state = rollout(models[10], seed=9000)[0][5]
for a in (0, 1):
    print('\nPROPOSED ACTION:', 'LEFT' if a == 0 else 'RIGHT')
    trace_path(models[10], example_state, a)
```

PROPOSED ACTION: LEFT

Input: cart x=-0.068, cart velocity=-1.014, pole angle=0.075, angular velocity=1.515, action=0.000

```
node 0:      pole angle    0.075 >  -0.147 → right
node 88:     pole angle    0.075 ≤   0.130 → left
node 89: angular velocity  1.515 >  -1.220 → right
node 135: angular velocity  1.515 >   1.217 → right
node 241:    pole angle    0.075 ≤   0.090 → left
node 242:         action    0.000 ≤   0.500 → left
node 243:    pole angle    0.075 >   0.029 → right
node 245: angular velocity  1.515 >   1.254 → right
node 247:    pole angle    0.075 ≤   0.081 → left
node 248:         cart x   -0.068 ≤  -0.034 → left
leaf 249: Q ≈ 8.19 discounted steps
```

PROPOSED ACTION: RIGHT

Input: cart x=-0.068, cart velocity=-1.014, pole angle=0.075, angular velocity=1.515, action=1.000

```
node 0:      pole angle    0.075 >  -0.147 → right
node 88:     pole angle    0.075 ≤   0.130 → left
node 89: angular velocity  1.515 >  -1.220 → right
node 135: angular velocity  1.515 >   1.217 → right
node 241:    pole angle    0.075 ≤   0.090 → left
node 242:         action    1.000 >   0.500 → right
node 252: angular velocity  1.515 ≤   1.685 → left
node 253: angular velocity  1.515 >   1.493 → right
leaf 261: Q ≈ 19.91 discounted steps
```

6. See the multidimensional bins

We cannot draw four state dimensions at once. Fix two coordinates and vary the other two. **Color = preferred action**; the contour is where $Q(s, \text{right}) - Q(s, \text{left}) = 0$. In the second panel, color is $Q(s, \text{right})$, showing the steps/plateaus from tree leaves. Try moving the fixed cart position or velocity and watch the rectangles change.

These are slices of the **learned policy**, not a claim that CartPole's dynamics depend on only the two plotted variables.

```
In [ ]: def policy_slice(model, plane='angle × angular speed', cart_x=0.0,
                        cart_v=0.0, angle=0.0, angular_v=0.0, grid=120):
    if plane == 'angle × angular speed':
        xx, yy = 2, 3
        extent = ((-0.22, 0.22), (-2.5, 2.5))
    else:
        xx, yy = 0, 1
        extent = ((-2.4, 2.4), (-2.5, 2.5))
    gx = np.linspace(*extent[0], grid)
    gy = np.linspace(*extent[1], grid)
    X, Y = np.meshgrid(gx, gy)
    s = np.tile([cart_x, cart_v, angle, angular_v], (X.size, 1))
    s[:, xx] = X.ravel(); s[:, yy] = Y.ravel()
    q0, q1 = q_pair(model, s)
    d = (q1-q0).reshape(X.shape)
    value = q1.reshape(X.shape)
    fig, axes = plt.subplots(1, 2, figsize=(12, 4.2),
                             constrained_layout=True)
    axes[0].pcolormesh(X, Y, d > 0, cmap='coolwarm', vmin=0, vmax=1,
                       shading='auto')
    if d.min() < 0 < d.max():
        axes[0].contour(X, Y, d, levels=[0], colors='k', linewidths=.6)
    axes[0].set_title('blue: left | red: right')
    mesh = axes[1].pcolormesh(X, Y, value, cmap='viridis', shading='auto')
    fig.colorbar(mesh, ax=axes[1], label='Q(state, right)')
    axes[1].set_title('Piecewise constant values in tree leaves')
    for ax in axes:
        ax.set(xlabel=STATE[xx], ylabel=STATE[yy])
    return fig
```

```
In [ ]: slice_model = mo.ui.dropdown(options=['depth 2', 'depth 4', 'depth 6',
        'depth 10', 'bagged 12'], value='depth 10', label='Model')
slice_plane = mo.ui.dropdown(options=['angle × angular speed', 'cart x ×
        cart velocity'], value='angle × angular speed', label='Plane')
slice_x = mo.ui.slider(start=-1.0, stop=1.0, step=0.25, value=0.0,
        label='Fixed cart x')
slice_v = mo.ui.slider(start=-1.5, stop=1.5, step=0.25, value=0.0,
        label='Fixed cart velocity')
slice_angle = mo.ui.slider(start=-0.15, stop=0.15, step=0.03, value=0.0,
        label='Fixed pole angle')
slice_angular_v = mo.ui.slider(start=-1.5, stop=1.5, step=0.3, value=0.0,
        label='Fixed angular velocity')
mo.vstack([mo.hstack([slice_model, slice_plane], justify='start', gap=2),
        mo.hstack([slice_x, slice_v, slice_angle, slice_angular_v],
        justify='start', gap=2)])
```

Model

depth 10

Plane

angle × angular speed

Fixed

cart

x

Fixed

cart

velocity

Fixed

pole

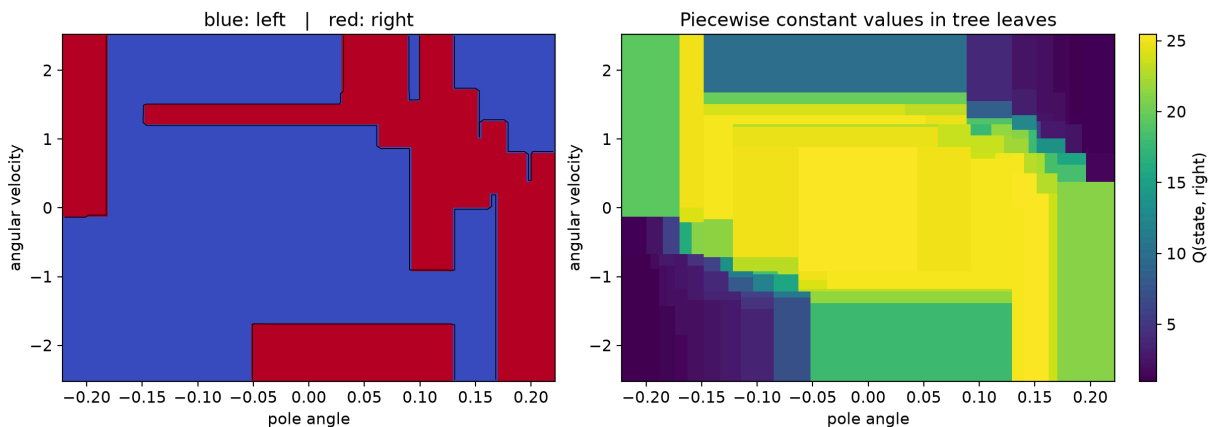
angle

F

a

v

```
In [ ]: _model = bagged if slice_model.value == 'bagged 12' else
models[int(slice_model.value.split()[-1])]
policy_slice(_model, plane=slice_plane.value, cart_x=slice_x.value,
             cart_v=slice_v.value, angle=slice_angle.value,
             angular_v=slice_angular_v.value)
```



7. Bagging: individual partitions and their average

Each member sees a bootstrap sample (80% as many draws as rows, with replacement). Here we compare three fitted members' right-minus-left value maps with their ensemble's mean. One can inspect their tree diagrams above by changing the `member` slider. This is **bagging at every fitted-Q iteration**, not one tree per trajectory.

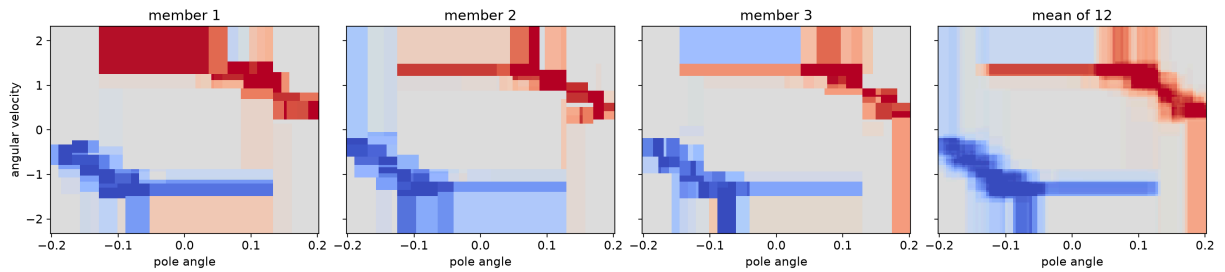
```
In [ ]: def bagging_maps(cart_x=0.0, cart_v=0.0, n=100):
    angles = np.linspace(-.20, .20, n)
    speeds = np.linspace(-2.3, 2.3, n)
    A, V = np.meshgrid(angles, speeds)
    s = np.column_stack([np.full(A.size, cart_x), np.full(A.size, cart_v),
                        A.ravel(), V.ravel()])
    selected = [bagged.estimated[i] for i in (0, 1, 2)] + [bagged]
    fig, axs = plt.subplots(1, 4, figsize=(16, 3.5), sharex=True,
                           sharey=True,
                           constrained_layout=True)
    for i, (ax, model) in enumerate(zip(axs, selected)):
        q0, q1 = q_pair(model, s)
        d = (q1-q0).reshape(A.shape)
        bound = np.percentile(np.abs(d), 95)
        ax.pcolormesh(A, V, d, shading='auto', cmap='coolwarm',
                     vmin=-bound if bound else -1, vmax=bound if bound
    else 1)
    ax.set(title=f'member {i+1}' if i < 3 else 'mean of 12',
```

```

        xlabel='pole angle')
    if i == 0: ax.set_ylabel('angular velocity')
    return fig

bagging_maps()

```



8. Watch one random and two learned episodes

The top row gives representative frames from an independently seeded run. The bottom row shows pole angle through time. A failure at $\pm 12^\circ$ ends an episode; the time limit is 500 steps. Change the seed to find successes and failures; one run is illustrative, while the 30-game comparison above measures performance.

```

In [ ]: episode_seed = mo.ui.slider(start=9000, stop=9050, value=9001,
    label='Episode seed')
    episode_seed

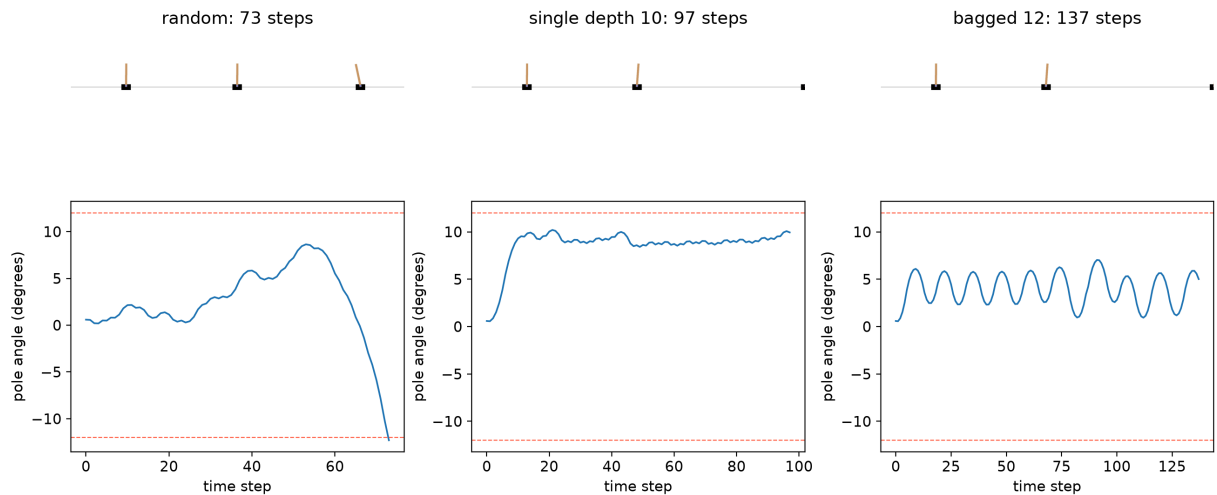
```

Episode seed

```

In [ ]: agents = [('random', None), ('single depth 10', models[10]), ('bagged 12',
    bagged)]
    _rollout_fig, axes = plt.subplots(2, 3, figsize=(13, 6),
    constrained_layout=True)
    for col, (label, _model) in enumerate(agents):
        trajectory, frames = rollout(_model, seed=episode_seed.value,
        random=_model is None, render=True)
        indices = [0, min(5, len(frames) - 1), len(frames) - 1]
        strip = np.concatenate([frames[i] for i in indices], axis=1)
        axes[0, col].imshow(strip)
        axes[0, col].set(title=f'{label}: {len(trajectory) - 1} steps')
        axes[0, col].axis('off')
        axes[1, col].plot(np.rad2deg(trajectory[:, 2]), lw=1.3)
        axes[1, col].axhline(12, c='tomato', ls='--', lw=0.8)
        axes[1, col].axhline(-12, c='tomato', ls='--', lw=0.8)
        axes[1, col].set(xlabel='time step', ylabel='pole angle (degrees)')
    _rollout_fig

```



Optional: animate any one game

This renders Gymnasium's frames to an inline GIF on demand. It does not open a separate display window.

```
In [ ]: animation_model = mo.ui.dropdown(options=['random', 'depth 10', 'bagged 12'], value='bagged 12', label='Animate')
animation_button = mo.ui.run_button(label='Render GIF')
mo.hstack([animation_model, animation_button], justify='start', gap=2)
```

Animate

```
In [ ]: mo.stop(not animation_button.value, mo.md('Press **Render GIF** to animate the chosen episode.'))
import base64
from io import BytesIO
from PIL import Image as PILImage

def animate_episode(model=None, seed=9001, random=False, max_frames=150):
    trajectory, frames = rollout(model, seed=seed, random=random, render=True)
    stride = max(1, int(np.ceil(len(frames)/max_frames)))
    pics = [PILImage.fromarray(frame).resize((300, 200)) for frame in frames[::stride]]
    output = BytesIO()
    pics[0].save(output, format='GIF', save_all=True, append_images=pics[1:],
                 duration=20*stride, loop=0)
    print(f'{len(trajectory)-1} steps; showing every {stride} frame(s)')
    encoded = base64.b64encode(output.getvalue()).decode('ascii')
    return mo.Html(f'')

_choice = animation_model.value
_selected = None if _choice == 'random' else (models[10] if _choice == 'depth 10' else bagged)
```

```
animate_episode(_selected, seed=episode_seed.value, random=_choice ==  
'random')
```

Press **Render GIF** to animate the chosen episode.

What to say in class

1. Random play supplies **transitions**, including the delayed effect of an action. The first few targets are nearly uninformative; repeated backups let terminal failure affect earlier states.
2. One fitted regression tree asks nested questions about the four state variables **and the candidate action**. Its leaves are constant-value bins in five dimensions. The policy compares two leaves, one per action.
3. Increasing tree depth adds expressiveness but can make the estimates fragile. Shallow trees can collapse to the same action everywhere; more depth is not guaranteed to improve held-out rollouts.
4. Bagging trains trees on different bootstrap samples and averages their Q predictions **before** choosing an action. It reduces some sensitivity to individual splits; it does not fix missing exploration.

Limits: this is a small, seeded teaching experiment. Offline fitted Q iteration can overestimate unsupported actions or fail with sparse exploration. The fixed random dataset makes the depth and bagging comparison controlled, but stronger CartPole policies may require collecting new data under improved, exploratory policies. The `min_samples_leaf=20`, 30 backups, 800 episodes, and 12 bagged members balance clarity and lecture time.

Sources: [Gymnasium CartPole](#), [Gymnasium classic-control installation](#), [scikit-learn tree plotting](#), [scikit-learn bagging](#), and Ernst, Geurts & Wehenkel, *Tree-Based Batch Mode Reinforcement Learning* (2005).