In [1]:

```
from enum import IntEnum
from typing import Tuple, Optional, List
from gym import Env, spaces
from gym.utils import seeding
from gym.envs.registration import register

import numpy as np

def register_env_1() -> None:
    """Register custom gym environment so that we can use `gym.make()`

    In your main file, call this function before using `gym.make()` to use the Four Ro
        register_env_1()
        env = gym.make('WindyGridWorld-v0')

    There are a couple of ways to create Gym environments of the different variants of
    1. Create separate classes for each env and register each env separately.
    2. Create one class that has flags for each variant and register each env separate

        Example:
        (Original)     register(id="WindyGridWorld-v0", entry_point="env:WindyGridWorl
        (King's moves) register(id="WindyGridWorldKings-v0", entry_point="env:WindyGri

        The kwargs will be passed to the entry_point class.

    3. Create one class that has flags for each variant and register env once. You can

        Example:
        (Original)     gym.make("WindyGridWorld-v0")
        (King's moves) gym.make("WindyGridWorld-v0", **kwargs)

        The kwargs will be passed to the __init__() function.

    Choose whichever method you like.
    """
    register(id="WindyGridWorld-v0", entry_point=__name__ + ":WindyGridWorldEnv")


class Action(IntEnum):
    """Action"""

    LEFT = 0
    DOWN = 1
    RIGHT = 2
    UP = 3


def actions_to_dxdy(action: Action) -> Tuple[int, int]:
    """
    Helper function to map action to changes in x and y coordinates
    Args:
        action (Action): taken action
    Returns:
        dxdy (Tuple[int, int]): Change in x and y coordinates
    """
    mapping = {
        Action.LEFT: (-1, 0),
        Action.DOWN: (0, -1),
```

```
        Action.RIGHT: (1, 0),
        Action.UP: (0, 1),
    }
    return mapping[action]


class WindyGridWorldEnv(Env):
    def __init__(self):
        """Windy grid world gym environment
        This is the template for Q4a. You can use this class or modify it to create th
        """

        # Grid dimensions (x, y)
        self.rows = 10
        self.cols = 7

        # Wind
        # TODO define self.wind as either a dict (keys would be states) or multidimens
        self.wind = np.array((
            np.zeros(7),
            np.zeros(7),
            np.zeros(7),
            np.ones(7),
            np.ones(7),
            np.ones(7),
            np.ones(7)*2,
            np.ones(7)*2,
            np.ones(7),
            np.zeros(7)
        ))

        self.action_space = spaces.Discrete(len(Action))
        self.observation_space = spaces.Tuple(
            (spaces.Discrete(self.rows), spaces.Discrete(self.cols))
        )

        # Set start_pos and goal_pos
        self.start_pos = (0, 3)
        self.goal_pos = (7, 3)
        self.agent_pos = None

    def seed(self, seed: Optional[int] = None) -> List[int]:
        """Fix seed of environment

        In order to make the environment completely reproducible, call this function a
            env = gym.make(...)
            env.seed(seed)
            env.action_space.seed(seed)
'
        This function does not need to be used for this assignment, it is given only f
        """

        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def reset(self):
        self.agent_pos = self.start_pos
        return self.agent_pos

    def step(self, action: Action) -> Tuple[Tuple[int, int], float, bool, dict]:
```

```
        """Take one step in the environment.

        Takes in an action and returns the (next state, reward, done, info).
        See https://github.com/openai/gym/blob/master/gym/core.py#L42-L58 foand r more

        Args:
            action (Action): an action provided by the agent

        Returns:
            observation (object): agent's observation after taking one step in environ
            reward (float) : reward for this transition
            done (bool): whether the episode has ended, in which case further step() c
            info (dict): contains auxiliary diagnostic information (helpful for debugg
        """

        # TODO
        if self.agent_pos == self.goal_pos:
            done = True
            reward = 0.0
            self.reset()
        else:
            done = False
            reward = -1.0

            move = actions_to_dxdy(action)
            wind = self.wind[int(self.agent_pos[0]), int(self.agent_pos[1])]

            next_pos = (int(self.agent_pos[0] + move[0]), int(self.agent_pos[1] + move
            if next_pos[0] < 0:
                next_pos = (0, next_pos[1])
            if next_pos[0] >= self.rows:  # If x coordinate is out of bounds, keep cur
                next_pos = (self.rows - 1, next_pos[1])
            if next_pos[1] < 0:
                next_pos = (next_pos[0], 0)
            if next_pos[1] >= self.cols:  # If y coordinate is out of bounds, keep cur
                next_pos = (next_pos[0], self.cols - 1)

            self.agent_pos = next_pos

        return self.agent_pos, reward, done, {}
```

In [58]:

```
from collections import deque
import gym
from typing import Optional, Tuple, Callable
from collections import defaultdict
import numpy as np

def generate_episode(env, policy, es=False):
    episode = []
    state = env.reset()
    complete = False
    n = 0

    while n <= 10000:  # Consider using a more practical loop condition.
        if es and len(episode) == 0:
            action = env.action_space.sample()
        else:
            action = policy(state)
```

```
        next_state, reward, done, _ = env.step(action)  # Assuming `_` captures 'info'
        # print(f"First step: {state}, {action}, {reward}")
        episode.append((state, action, reward))  # Ensure only state, action, reward a
        state = next_state
        if done:
            complete = True
            break
        n += 1

    return episode, complete



def td_zero(env, policy, num_episodes, gamma=1.0, alpha=0.1):
    V_td0 = defaultdict(float)
    for _ in range(num_episodes):
        state = env.reset()
        done = False
        while not done:
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            V_td0[state] += alpha * (reward + gamma * V_td0[next_state] - V_td0[state]
            state = next_state
            # print(f"V_td0:{V_td0}")
    return V_td0

def monte_carlo_prediction(env, policy, num_episodes, gamma=1.0):
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)
    V_mcp = defaultdict(float)

    for _ in range(num_episodes):
        episode = []
        state = env.reset()
        done = False
        while not done:
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, reward))
            state = next_state

        G = 0
        for state, reward in reversed(episode):
            G = gamma * G + reward
            returns_sum[state] += G
            returns_count[state] += 1.0
            V_mcp[state] = returns_sum[state] / returns_count[state]
    return V_mcp

    from collections import deque

def n_step_td_for_v(env, policy, num_episodes, n=4, alpha=0.1, gamma=1.0):
    """
    N-step TD for estimating V.

    Args:
        env (gym.Env): The environment.
        policy (callable): A function that maps states to actions.
        num_episodes (int): Number of episodes to run for training.
        n (int): The number of steps to look ahead to update the value function.
```

```
        alpha (float): The learning rate.
        gamma (float): The discount factor.
    Returns:
        V (defaultdict): The estimated state values.
    """
    V_nstep = defaultdict(float)

    for episode in range(num_episodes):
        states = deque(maxlen=n+1)
        rewards = deque(maxlen=n+1)

        state = env.reset()
        states.append(state)
        done = False
        t = 0

        while not done or len(states) > 1:
            if not done:
                action = policy(state)
                next_state, reward, done, _ = env.step(action)
                states.append(next_state)
                rewards.append(reward)
                state = next_state

            tau = t - n + 1  # State to be updated
            if tau >= 0:
                G = sum([gamma ** i * rewards[i] for i in range(len(rewards))])
                if len(states) > n:
                    G += gamma ** n * V_nstep[states[-1]]

                V_nstep[states[0]] += alpha * (G - V_nstep[states[0]])
                states.popleft()
                rewards.popleft()

            t += 1

            # print(f"V_nstep:{V_nstep}")
    return V_nstep
```

In [53]:

```
def calculate_learning_targets(episodes, V, gamma, n=None, method='TD(0)'):
    """
    Calculate the learning targets for the given evaluation episodes.

    Args:
        episodes (list): List of evaluation episodes, each episode is a list of (state
        V (defaultdict): Estimated state values.
        gamma (float): Discount factor.
        n (int or None): The number of steps for n-step TD. Use None for Monte Carlo.
        method (str): The method to use for calculating targets ('TD(0)', 'n-step TD',

    Returns:
        targets (list): The list of learning targets for the specified state across al
    """
    targets = []
    for episode in episodes:
        episode_length = len(episode)
        for i, (state, _, reward) in enumerate(episode):
            if method == 'TD(0)':
                if i < episode_length - 1:
```

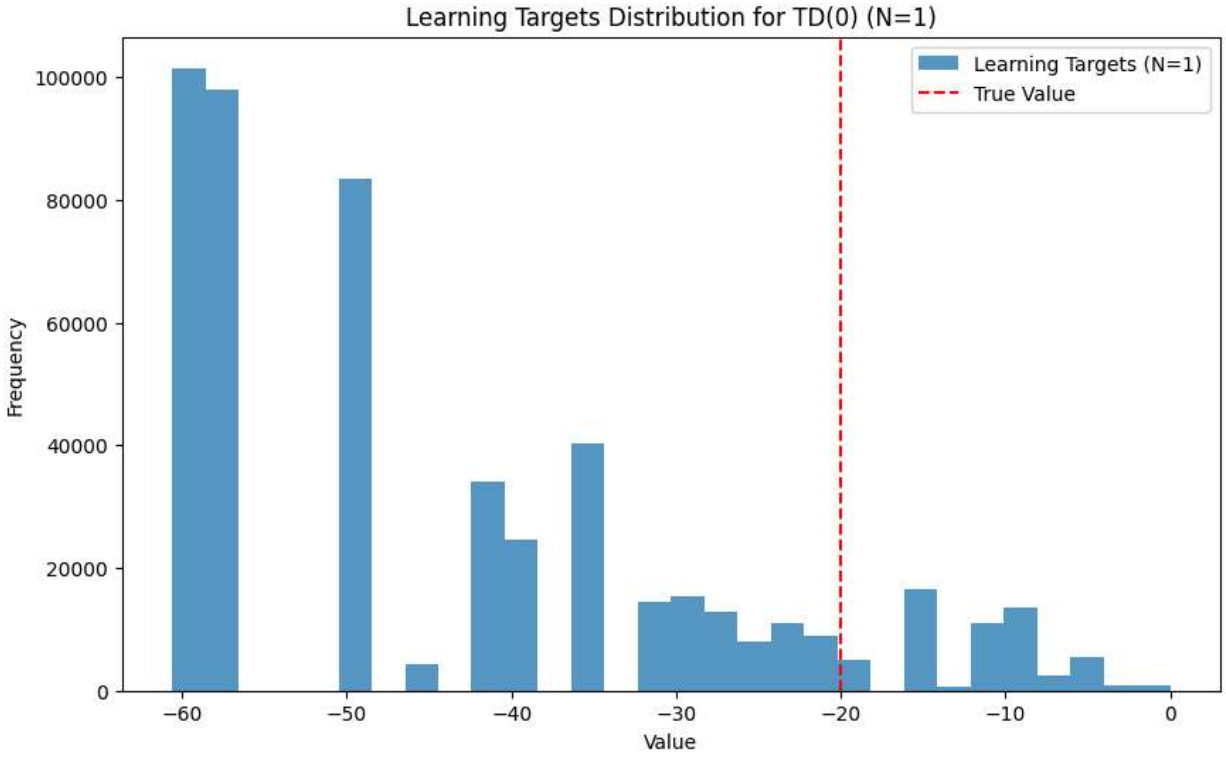
```
                    next_state = episode[i + 1][0]
                    target = reward + gamma * V.get(next_state, 0)
                else:
                    target = reward
                targets.append(target)

            elif method == 'n-step TD':
                target = 0
                for j in range(min(n, episode_length - i)):
                    reward_j = episode[i + j][2]
                    target += (gamma ** j) * reward_j
                if i + n < episode_length:
                    next_state = episode[i + n][0]
                    target += (gamma ** n) * V.get(next_state, 0)
                targets.append(target)

            elif method == 'Monte Carlo':
                G = 0
                for j in range(i, episode_length):
                    reward_j = episode[j][2]
                    G += (gamma ** (j - i)) * reward_j
                targets.append(G)

    return targets
```

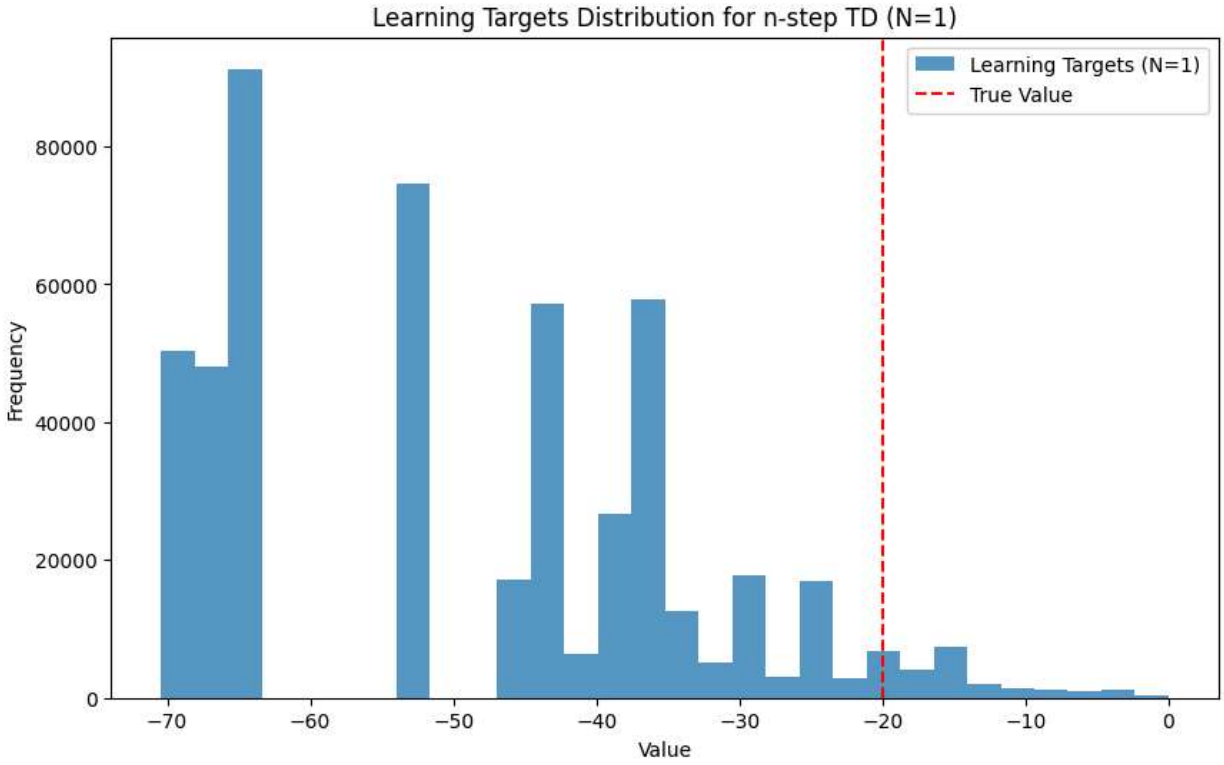
In [56]:
```
def stochastic_policy(state):
    # Example: Uniformly random policy
    action_probabilities = [0.25, 0.25, 0.25, 0.25]  # Adjust based on your environmer
    return np.random.choice([0, 1, 2, 3], p=action_probabilities)
```

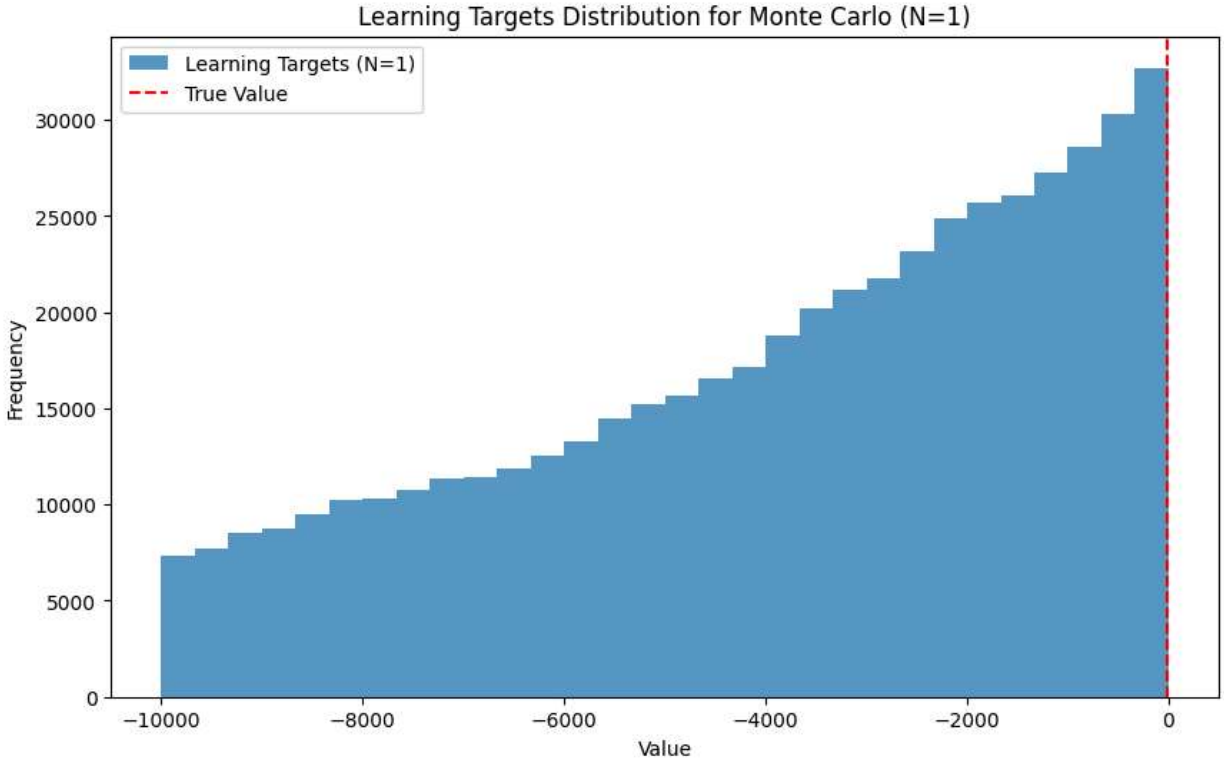
In [60]:
```
import matplotlib.pyplot as plt

def run_experiment_and_plot(env, policy, N_values=[1, 10, 50], n=4):
    gamma = 1.0
    true_value = -20  # Optional: Replace with the true value computed via DP or other

    for N in N_values:
        training_episodes = [generate_episode(env, policy)[0] for _ in range(N)]

        V_td0 = td_zero(env, policy, N, gamma)
        V_nstep = n_step_td_for_v(env, policy, N, n=n, alpha=0.1, gamma=gamma)
        print(f"value of nste sarsa:{V_nstep}")
        V_mc = monte_carlo_prediction(env, policy, N, gamma)

        evaluation_episodes = [generate_episode(env, policy)[0] for _ in range(100)]

        methods = {"TD(0)": V_td0, "n-step TD": V_nstep, "Monte Carlo": V_mc}
        for method_name, V in methods.items():
            targets = calculate_learning_targets(evaluation_episodes, V, gamma, n=n if

            # Plotting
            plt.figure(figsize=(10, 6))
            plt.hist(targets, bins=30, alpha=0.75, label=f"Learning Targets (N={N})")
            plt.axvline(x=true_value, color='r', linestyle='--', label="True Value")
            plt.title(f"Learning Targets Distribution for {method_name} (N={N})")
            plt.xlabel("Value")
            plt.ylabel("Frequency")
            plt.legend()
            plt.show()
```

```
# Assuming `env` and `policy` are defined
env = WindyGridWorldEnv()
run_experiment_and_plot(env, stochastic_policy)  # Pass the policy function correctly
```

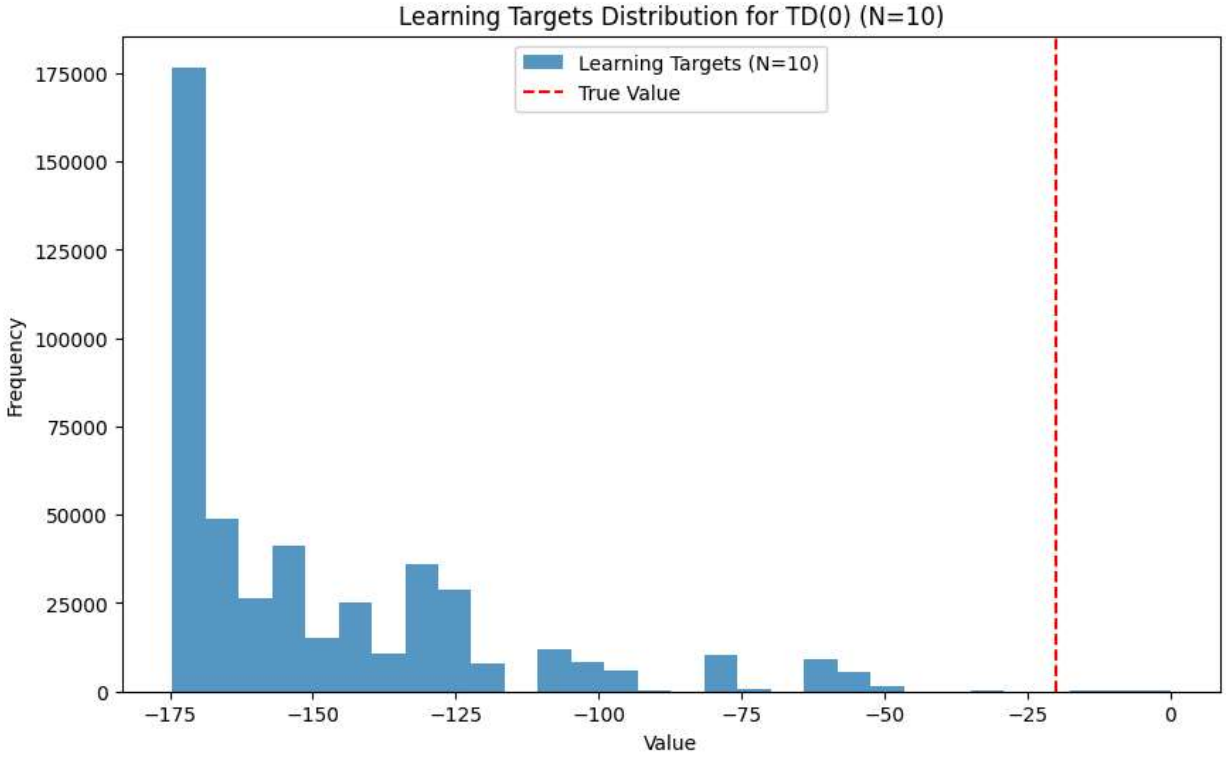
```
value of nste sarsa:defaultdict(<class 'float'>, {(0, 1): -14.0070693498635, (0, 3):
-19.889915817353142, (1, 1): -17.203401996147416, (0, 2): -19.495903681550374, (2,
1): -16.734783495176565, (2, 0): -12.399907770470325, (1, 0): -11.502304660112795,
(0, 0): -10.802658745060212, (1, 2): -20.13524780495477, (1, 3): -26.46377349686205,
(0, 4): -25.911046629286318, (1, 4): -29.128378315433636, (2, 4): -37.27315282964447
4, (2, 5): -38.71662558768603, (2, 3): -27.521263268597732, (2, 2): -22.8885093816972
5, (3, 3): -21.057874537998583, (4, 4): -13.801746644108176, (5, 5): -6.2500900664021
11, (5, 6): -66.41462527086992, (4, 6): -61.975171132710045, (3, 6): -60.581487007591
974, (2, 6): -49.47496596621456, (6, 6): -60.73561876053163, (1, 6): -41.930768571326
51, (0, 6): -34.02727706722861, (0, 5): -32.447699108991614, (1, 5): -33.902965979822
48, (3, 4): -38.36942734693196, (7, 6): -48.173750363357044, (8, 6): -40.145637851741
206, (9, 6): -32.33040074093334, (9, 5): -32.92564308449763, (9, 4): -17.080587558059
317, (9, 3): -8.100373544737494, (8, 4): -24.499240670635164, (8, 5): -29.41517225474
0305, (3, 5): -32.83670929075249, (7, 5): -6.2230951900904525, (4, 5): -25.1673426064
0014, (3, 2): -28.31943071195482, (4, 3): -22.91702601409034, (5, 4): -11.54159832357
564, (8, 3): -6.0126650050904065, (7, 4): -2.0098303806319957, (9, 2): -3.24740287126
3855, (9, 1): -1.7323776030118077, (3, 1): -11.972695450476586, (4, 2): -4.6836898736
46177, (3, 0): -2.460771945677915, (4, 1): -3.434916294569172, (8, 2): -0.46, (7, 3):
0.0})
```

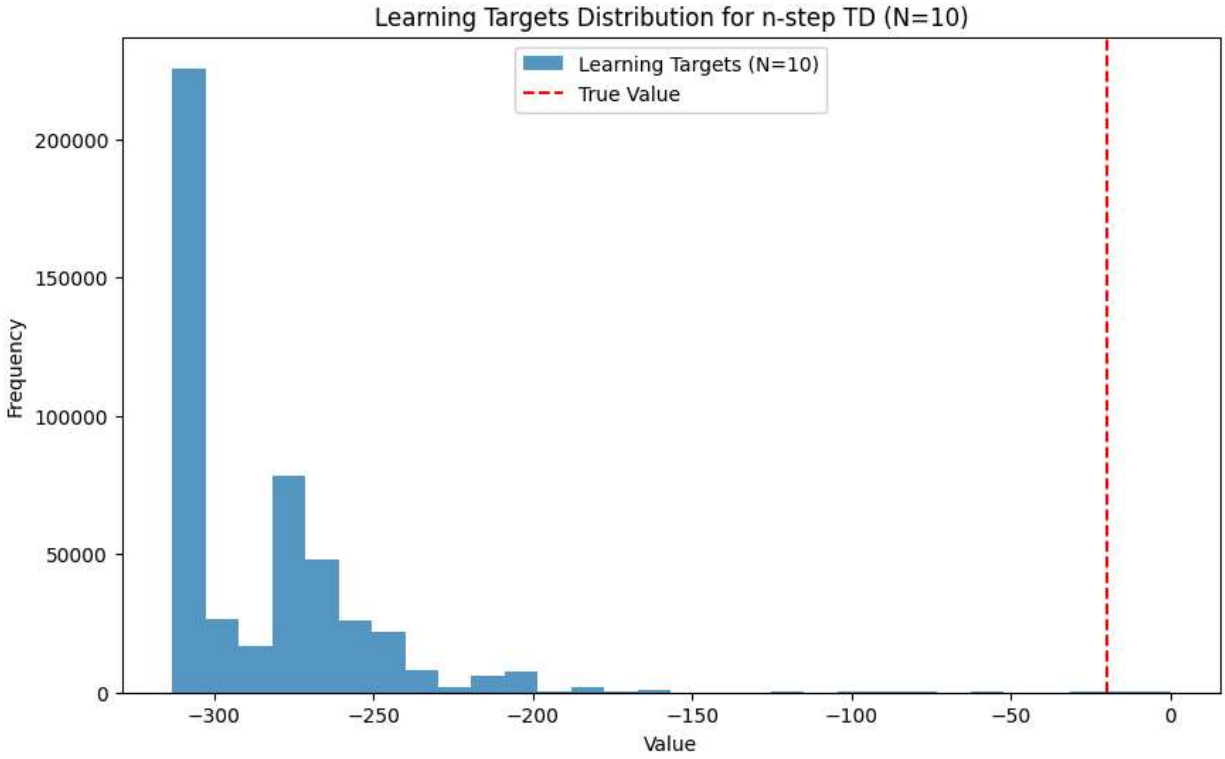


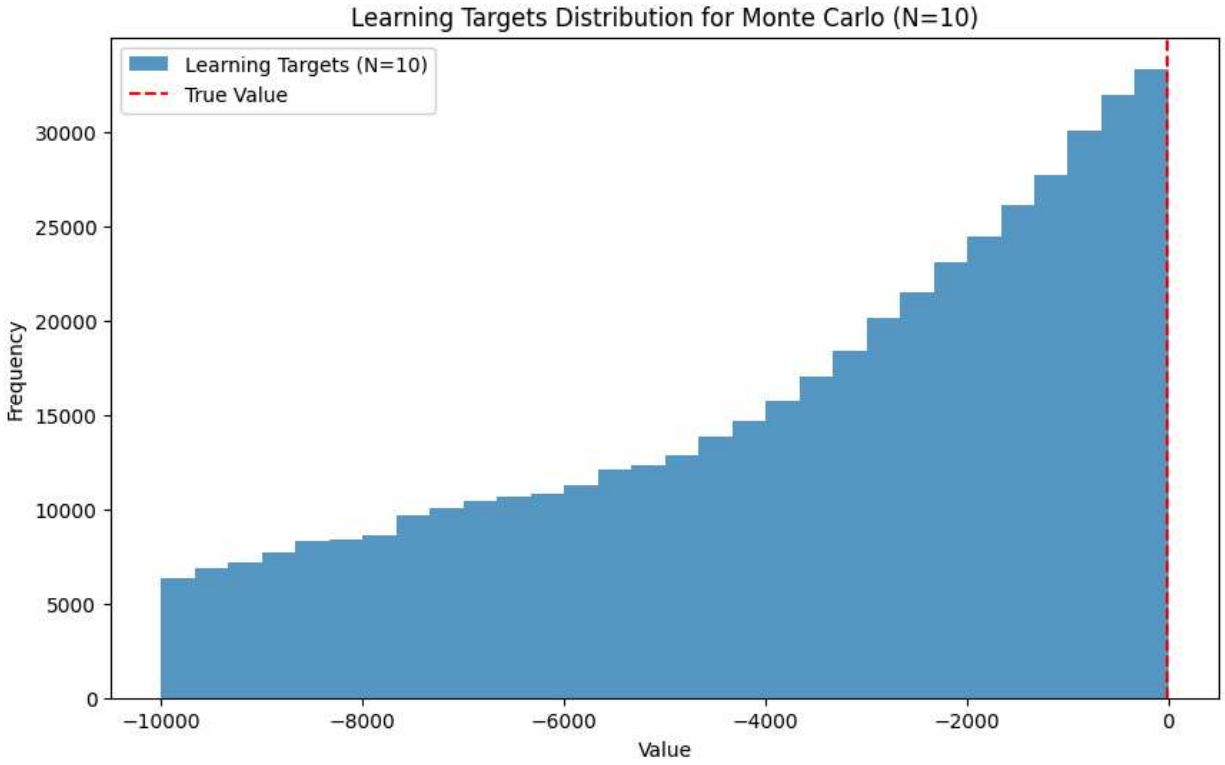

Learning Targets Distribution for n-step TD (N=1)
Learning Targets (N=1)
True Value
Frequency
Value
0
20000
40000
60000
80000
−70
−60
−50
−40
−30
−20
−10
0


Learning Targets Distribution for Monte Carlo (N=1)
Learning Targets (N=1)
True Value
Frequency
Value
0
5000
10000
15000
20000
25000
30000
−10000
−8000
−6000
−4000
−2000
0

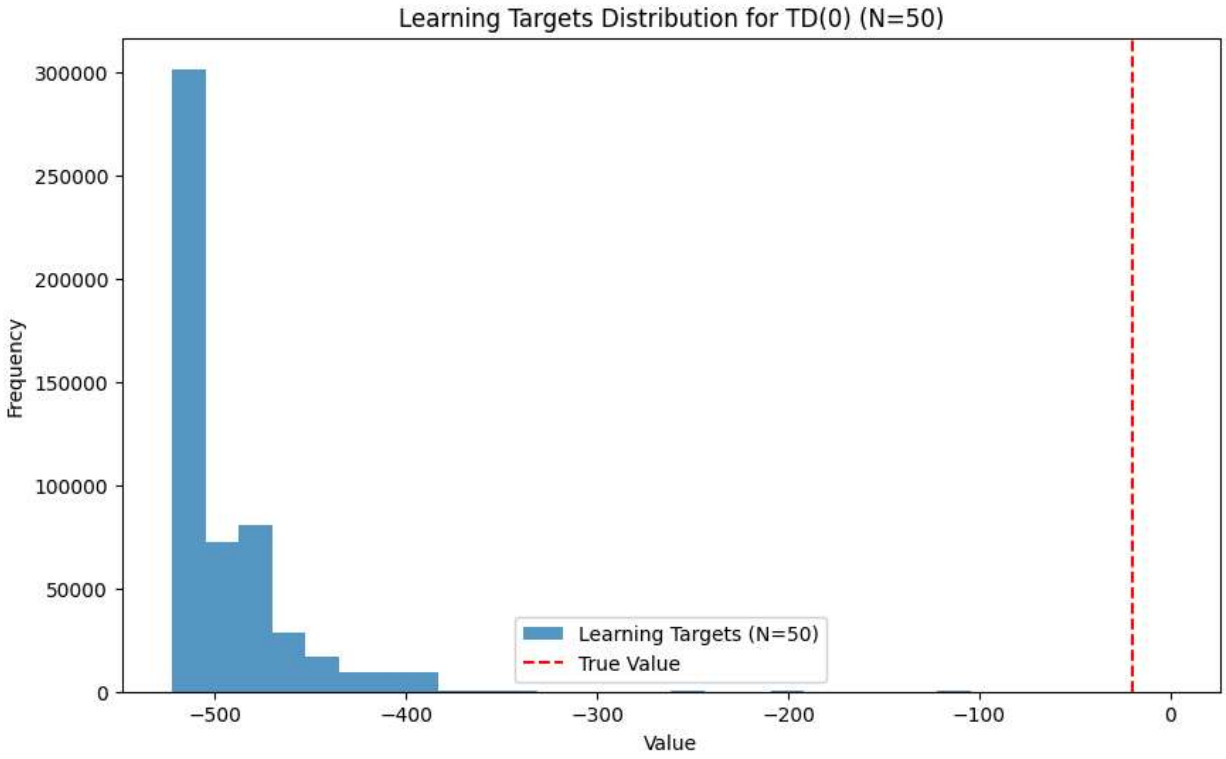
value of nste sarsa:defaultdict(<class 'float'>, {(0, 3): -242.1331613008782, (0, 2): -233.75540475028092, (0, 1): -200.5043265730485, (0, 0): -205.21137319036094, (0, 4): -243.02564337817768, (1, 0): -200.51156799967876, (2, 0): -183.3286459550415, (3, 0): -164.00741622577152, (2, 1): -203.6866106299323, (1, 1): -208.5844077454305, (2, 2): -246.91227155552667, (3, 2): -246.1623980323027, (3, 4): -265.876694645284, (3, 6): -276.65531060494413, (4, 6): -301.213806709269, (2, 6): -270.28525999690794, (5, 6): -309.0729868234708, (6, 6): -306.4460339241624, (7, 6): -306.19977766591825, (8, 6): -301.3853789804265, (9, 6): -290.2206197781793, (1, 6): -267.77891413002567, (1, 5): -262.82052402301673, (2, 5): -266.3417440813688, (3, 5): -283.42516694020674, (0, 5): -251.98835868305588, (0, 6): -261.2145129011355, (2, 4): -257.572561769609, (2, 3): -253.48575715604608, (1, 2): -233.72355242121415, (1, 3): -237.89571528222834, (1, 4): -252.29717220189679, (3, 3): -254.3694870770313, (3, 1): -179.7215745241554, (9, 5): -278.2482212257686, (8, 5): -299.35311833051304, (9, 4): -258.6473563746988, (8, 4): -280.2117696415912, (7, 5): -246.17290504540912, (4, 5): -272.2923743130926, (9, 3): -220.15452262558972, (9, 2): -154.98264549275885, (8, 2): -58.422912337263725, (7, 3): 0.0, (4, 2): -95.65227089360899, (4, 3): -207.13678026415124, (9, 1): -72.634578193867 36, (4, 4): -243.7192902600448, (5, 5): -185.39447502285498, (8, 1): -17.832565886159628, (8, 3): -239.3260568092511, (9, 0): -15.949583316873234, (4, 1): -89.20154521177436, (7, 4): -201.67329865227657, (8, 0): -34.62918492354179, (5, 4): -116.5746422913897, (7, 1): -14.730312316004344, (5, 2): -97.41973031368967, (6, 5): -22.056676089271566, (5, 3): -24.450662755585785})

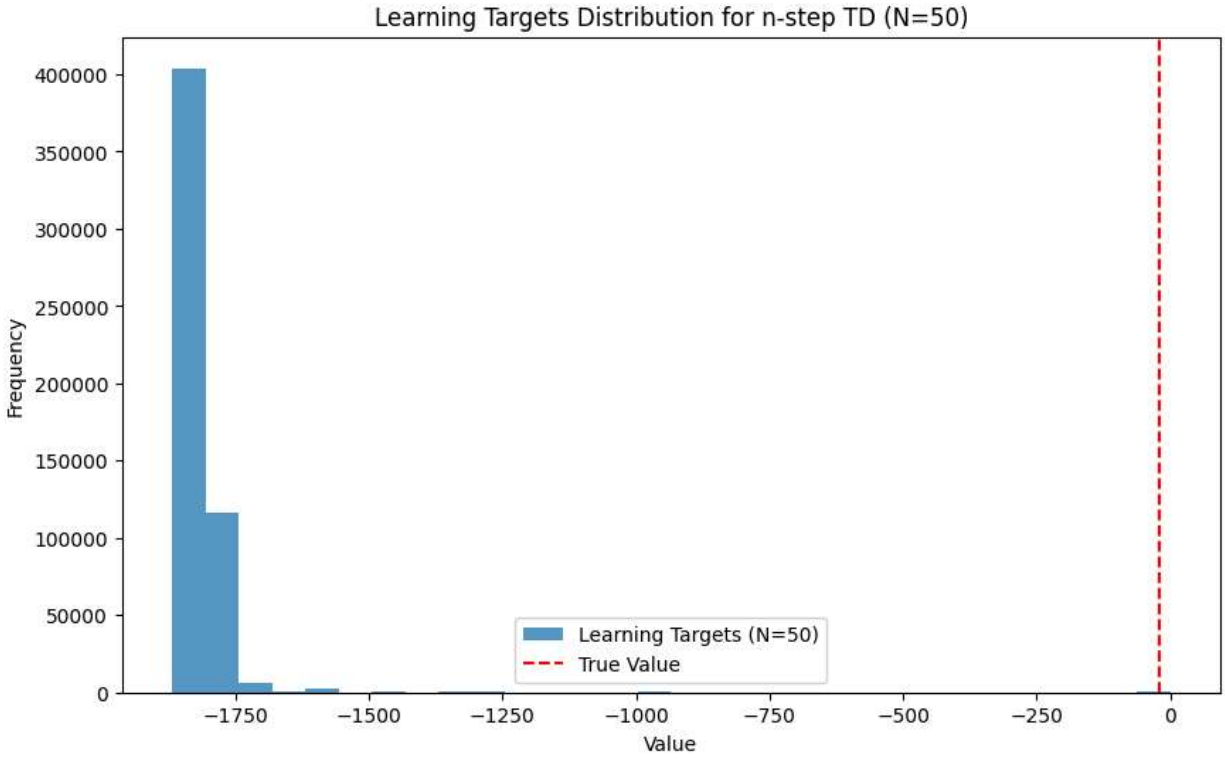


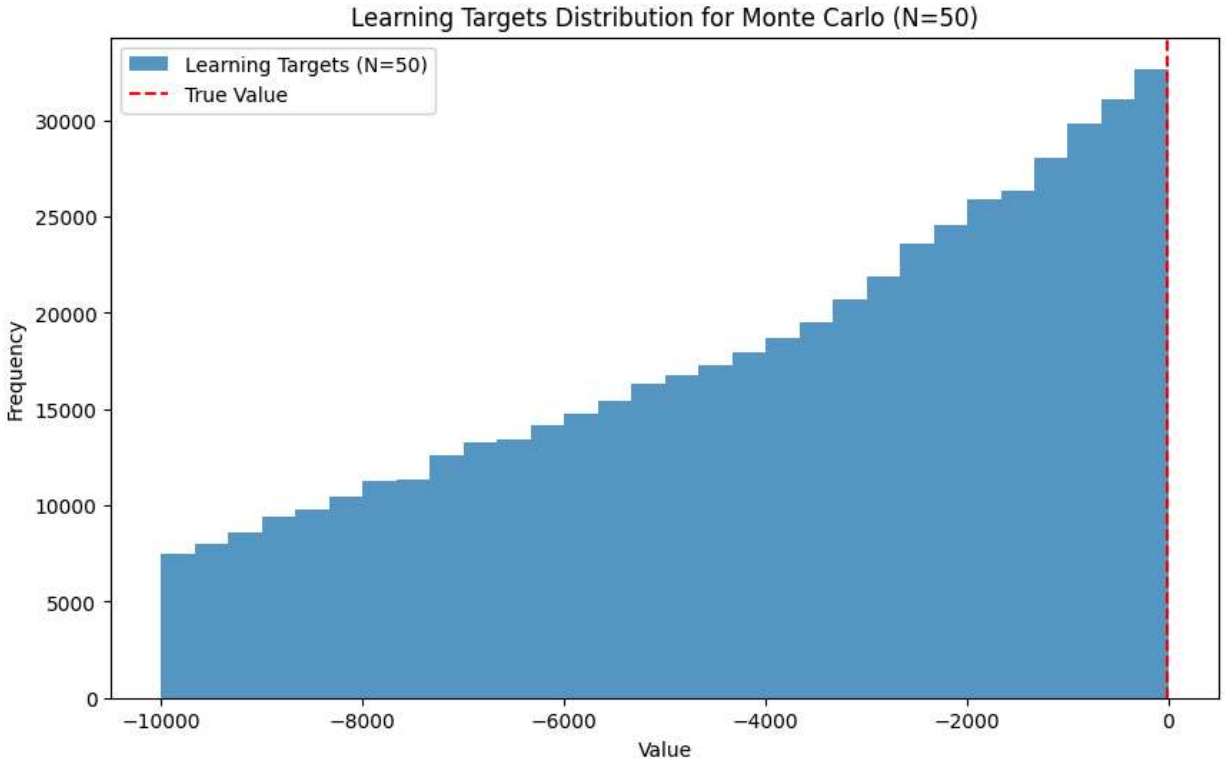

Learning Targets Distribution for n-step TD (N=10)
Learning Targets (N=10)
True Value
Frequency
200000
150000
100000
50000
0
−300
−250
−200
−150
−100
−50
0
Value


Learning Targets Distribution for Monte Carlo (N=10)
Learning Targets (N=10)
True Value
Frequency
30000
25000
20000
15000
10000
5000
0
−10000
−8000
−6000
−4000
−2000
0
Value

value of nste sarsa:defaultdict(<class 'float'>, {(0, 4): -1824.8648421208215, (0, 3): -1811.286380363686, (0, 5): -1834.1332913115973, (1, 5): -1845.1657224381788, (2, 5): -1840.5513567192254, (1, 6): -1848.4519066762894, (0, 6): -1844.3960739293404, (2, 6): -1849.6513565667478, (3, 5): -1856.1860015848542, (4, 6): -1864.1674375246535, (3, 6): -1860.3543085326942, (5, 6): -1863.1784155250136, (6, 6): -1858.478080450727, (7, 6): -1816.1539726193282, (8, 6): -1780.9846020586683, (9, 6): -1758.9066192755474, (9, 5): -1739.788672034832, (8, 5): -1815.9849408413802, (9, 4): -1686.6162995143557, (8, 4): -1822.9477866611219, (7, 5): -1821.6765415459388, (1, 4): -1828.053697584958, (1, 3): -1804.0133604236792, (1, 2): -1769.3207720390863, (0, 2): -1798.0884583770471, (0, 1): -1788.8394022303742, (2, 2): -1789.2958072249446, (3, 2): -1824.3296136909914, (3, 4): -1851.3804292202353, (2, 4): -1834.5792459155862, (2, 3): -1826.7937621215763, (2, 1): -1749.2327892188844, (2, 0): -1764.7462240148911, (3, 0): -1725.5489236973729, (4, 1): -1713.175638351069, (5, 2): -1302.6785768728375, (6, 3): -562.3963320893195, (1, 1): -1751.1984618835556, (3, 3): -1831.6092734157555, (4, 4): -1779.5894940313335, (5, 5): -1713.3748573711098, (4, 5): -1848.336569195181, (9, 3): -1581.9792068207012, (9, 2): -1437.709116745274, (9, 1): -1359.7606297999255, (9, 0): -1251.3197695467775, (0, 0): -1777.5842471576345, (1, 0): -1760.1483952345998, (8, 3): -1797.2622873680407, (8, 2): -970.1356397009483, (7, 3): 0.0, (3, 1): -1675.6136768025376, (4, 2): -1580.8547555432729, (5, 3): -1407.7438518862166, (4, 3): -1785.928522153431, (7, 4): -1749.8385221412582, (5, 4): -1776.2449177989047, (8, 0): -1349.1951037356291, (8, 1): -1499.4100751964672, (6, 5): -1387.6971509248115, (6, 4): -823.387208355862, (7, 2): -444.40786350426936, (7, 1): -467.11541877541794})







Bias refers to the error introduced by approximating the true value function, which can be due to simplifications in the learning algorithm. In other words, bias is the difference between the expected prediction and the true value function. Methods that rely on bootstrapping, like TD(0), typically have higher bias, especially with less training data, because they rely on current estimates to update future estimates.

Variance refers to the variability of the value function estimates between different training episodes. Methods that rely on full episodes for updates, like Monte Carlo, tend to have higher

variance, particularly when the number of episodes is low.

Here's how to interpret the histograms for each method with respect to bias and variance:

1. **TD(0) Histograms**:
    - With low N (e.g., N=1), the histogram might show a peak around a wrong value (if the initial estimates are poor), indicating high bias.
    - As N increases, the peak should move closer to the true value line (assuming your policy is improving), which would indicate a reduction in bias.
    - The spread of the histogram reflects variance, which might not change significantly with more data because TD(0) is less sensitive to individual episode variability.
2. **n-step TD Histograms**:
    - With low N, the spread of the histogram might be wider than for TD(0) because n-step TD incorporates more steps per update, reducing bias at the cost of higher variance.
    - As N increases, you would expect the spread to decrease and the peak to move closer to the true value, showing a reduction in both bias and variance due to more data smoothing out the updates.
3. **Monte Carlo Histograms**:
    - With low N, the histogram is likely to be very spread out, reflecting high variance since Monte Carlo relies on complete episodes.
    - As N increases, the histogram should become narrower and taller around the true value line, indicating a convergence towards the true value function and a significant reduction in variance.

Commenting on the Bias-Variance Trade-off:

- TD(0) tends to be more biased, especially with fewer episodes, because it updates estimates based on the current policy's limited knowledge.
- n-step TD balances bias and variance by considering a few steps into the future.
- Monte Carlo has low bias but high variance, especially noticeable with fewer episodes. However, it benefits the most from additional data, showing the greatest reduction in variance as N increases.

Interpreting the True Value Line:

- The dashed line representing the true value is a reference point. The closer the peaks of the histograms are to this line, the lower the bias of the method.
- If the histograms are narrow and tall near this true value line, it means the method has low variance and bias.

Conclusion:

- Ideally, you want the learning targets to be unbiased and consistent across different training datasets, with histograms centered at the true value line and with minimal spread.

- With more training data (larger N), the methods should converge, reducing both bias and variance. The histograms should become narrower and more centered around the true value.

For a specific and detailed analysis, I would need to review the numeric data or have a detailed description of the histograms' shape, center, spread, and how they align with the true value line. If you can provide this information in text form, I can give you a more precise interpretation.