KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(An Autonomous Institute)

(Accredited by NBA & NAAC, Approved by A.I.C.T.E., Reg by Govt of Telangana State & Affiliated to JNTU, Hyderabad)

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING(AI&ML)

LABORATORY RECORD

OF

REINFORCEMENT LEARNING LAB

Submitted by

Name:

Roll No:

B.Tech IV YEAR I SEM (KR21)

ACADEMIC YEAR 2024-25

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(Approved by AICTE & Govt of T.S. and Affiliated to JNTUH)
NARAYANAGUDA, HYDERABAD - 500 029.

Certificate

This is to certify that the following is a Bonafide Record of the practical work done by _____

bearing Roll No. _____ of _____ Branch of _____ year B.Tech Course in the _____ Laboratory during the Academic year _____ & _____ under our supervision.

Number of experiments conducted : _____

Signature of Staff Member Incharge

Signature of Head of the Dept.

Signature of Internal Examiner

Signature of External Examiner

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

CONTENTS

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

Daily Laboratory Assessment Sheet

Name of the Lab: Name of the Student:

Class: HT.No:

S.No.	Name of the Experiment	Date	Observation Marks (3M)	Record Marks (4M)	Viva Voice Marks (3M)	Total Marks (10M)	Signature of Faculty
	TOTAL						

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Narayanguda, Hyderabad – 500029

Department of Computer Science & Engineering (AI & ML)

Vision of the College:

- To be the fountain head in producing highly skilled, globally competent engineers.
- Producing quality graduates trained in the latest software technologies and related tools and striving to make India a world leader in software products and services.

Mission of the College:

- To provide a learning environment that inculcates problem solving skills, professional, ethical responsibilities, lifelong learning through multi modal platforms and prepare students to become successful professionals.
- To establish industry institute Interaction to make students ready for the industry.
- To provide exposure to students on latest hardware and software tools.
- To promote research-based projects/activities in the emerging areas of technology convergence.
- To encourage and enable students to not merely seek jobs from the industry but also to create new enterprises.
- To induce a spirit of nationalism which will enable the student to develop, understand lndia's challenges and to encourage them to develop effective solutions.
- To support the faculty to accelerate their learning curve to deliver excellent service to students.

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Narayanguda, Hyderabad – 500029

Department of Computer Science & Engineering (AI & ML)

Vision of the Department

To be among the region's premier teaching and research Computer Science and Engineering departments producing globally competent and socially responsible graduates in the most conducive academic environment.

Mission of the Department

- To provide faculty with state-of-the-art facilities for continuous professional development and research, both in foundational aspects and of relevance to emerging computing trends.
- To impart skills that transform students to develop technical solutions for societal needs and inculcate entrepreneurial talents.
- To inculcate an ability in students to pursue the advancement of knowledge in various specializations of Computer Science and Engineering and make them industry-ready.
- To engage in collaborative research with academia and industry and generate adequate resources for research activities for seamless transfer of knowledge resulting in sponsored projects and consultancy.
- To cultivate responsibility through sharing of knowledge and innovative computing solutions that benefits the society-at-large.
- To collaborate with academia, industry and community to set high standards in academic excellence and in fulfilling societal responsibilities.

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTE)
Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Narayanguda, Hyderabad – 500029

Department of Computer Science & Engineering (AI & ML)

PROGRAM OUTCOMES (POs)

PO1: Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTE)
Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH,
Narayanguda, Hyderabad – 500029

Department of Computer Science & Engineering (AI & ML)

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: An ability to analyze the common business functions to design and develop appropriate Computer Science **solutions** for social upliftments.

PSO2: Shall have expertise on the evolving technologies like Mobile Apps, CRM, ERP, Big Data, etc.

PROGRAM EDUCATIONAL OBJECTIVES (PEOS)

PEO1: Graduates will have successful careers in computer related engineering fields or will be able to successfully pursue advanced higher education degrees.

PEO2: Graduates will try and provide solutions to challenging problems in their profession by applying computer engineering principles.

PEO3: Graduates will engage in life-long learning and professional development by rapidly adapting changing work environment.

PEO4: Graduates will communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

B. Tech. in COMPUTER SCIENCE AND ENGINEERING (AI & ML)

IV Year I Semester Syllabus (KR21)

REINFORCEMENT LEARNING LAB (21CM702PC)

L	T	P	C
0	0	2	1

Pre-requisites/ Co-requisites:

1. 21CM501PC –Deep Learning for Vision System Course

Course Objectives: The course will help to

1. Discover How to Use the Markov Decision Process.
2. Learn how to create and use Deep Q Networks and policy gradient techniques.
3. Learn to use evolutionary algorithms and action-critic approaches.
4. Study Distributional DQN Design and Implementation.
5. Learn and implement Intrinsic curiosity module.

Course Outcomes: The student will be able to

1. Develop programs on Markov Decision Process (MDP).
2. To fix the issues with Deep Q Networks.
3. Implement advanced policy gradient algorithms.
4. Implement Distributional DQN algorithms.
5. Implement Intrinsic curiosity module.

List of Experiments: Write Python code to

1. Implement a multi-arm bandit algorithm.
2. Implement a Deep Q Network(DQN) model, train and test your model.
3. Implement and test the policy gradient method and test.
4. Implement an Actor-Critic model.
5. Implement genetic algorithms for evolving a set of random strings toward a target string.
6. Implement MNIST genetic algorithm.
7. Implement distributional DQN.
8. Implement Intrinsic curiosity module for Super Mario Bros game.

TEXT BOOKS:

1. Deep Reinforcement Learning in Action, Brandon Brown and Alexander Zai, Manning Publications, 2020.
2. Deep Reinforcement Learning Hands-On, Maxim Lapan, Packt Publishing, 2018.

REFERENCE BOOKS:

1. Reinforcement Learning: An introduction, R.S. Sutton and A.G. Barto, Second Edition, MIT Press, 2018.
2. Deep Reinforcement Learning with Python, Sudharsan Ravichandiran, Second Edition, Packt Publishing, 2020.

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTE)

Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH, Hyderabad

Department of Computer Science & Engineering (AI&ML)

Course Outcomes and CO-PO/PSO Mapping

REINFORCEMENT LEARNING LAB

COURSE OUTCOMES (COs):

Upon completion of the Course, the students will be able to:

	Course Outcome(CO)
CO1	Develop programs on Markov Decision Process (MDP).
CO2	To fix the issues with Deep Q Networks.
CO3	Implement advanced policy gradient algorithms
CO4	Implement Distributional DQN algorithms.
CO5	Implement Intrinsic curiosity module.

CO-PO-PSO Matrix:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	2				1							1		2
CO2	2	2	2	2	1							1	1	
CO3	2	2	2		1							2	1	
CO4	2	2	2	2	1							1	1	
CO5	2	2	1		1							1		1

Levels: 3: High 2: Medium 1: Low

EXPERIMENT 1

Aim: To implement N-armed Bandits

Theory:

Theory: N-Armed Bandit Problem

The N-armed bandit problem is a classic framework in reinforcement learning and decision theory that explores the trade-off between exploration and exploitation. This problem models a scenario where an agent faces a set of N slot machines (or "bandits"), each with an unknown probability distribution of rewards. The goal is to maximize the cumulative reward over a series of trials by strategically selecting which machine to play.

Context and Importance

The N-armed bandit problem serves as a simplified representation of real-world decision-making challenges, such as:

Clinical Trials: Determining the most effective treatment among multiple options.
Online Advertising: Allocating ad placements to maximize click-through rates.
Resource Allocation: Optimizing investment across different projects.

Exploration vs. Exploitation

The key challenge in the N-armed bandit problem is balancing two competing objectives:

Exploration: Trying out different actions to learn their reward distributions.
Exploitation: Choosing the action believed to yield the highest reward based on current knowledge.

Various algorithms, such as the greedy method, ε-greedy strategy, and upper confidence bound (UCB), are used to address this trade-off, enabling the agent to improve decision-making over time.

This experiment aims to simulate and analyze the N-armed bandit scenario, offering insights into the effectiveness of action-selection strategies and their impact on maximizing rewards.

Experiment-1

Aim : Implement a multi-arm bandit algorithm.

```
def get_best_action(actions):
  best_action = 0
  max_action_value = 0
  for i in range(len(actions)): #A
    cur_action_value = get_action_value(actions[i]) #B
    if cur_action_value > max_action_value:
      best_action = i
      max_action_value = cur_action_value
  return best_action
```

```
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt

n = 10
probs = np.random.rand(n) #A
eps = 0.1
```

```
def get_reward(prob, n=10):
    reward = 0;
    for i in range(n):
        if random.random() < prob:
            reward += 1
    return reward
```

```
reward_test = [get_reward(0.7) for _ in range(2000)]
```

```
np.mean(reward_test)
```

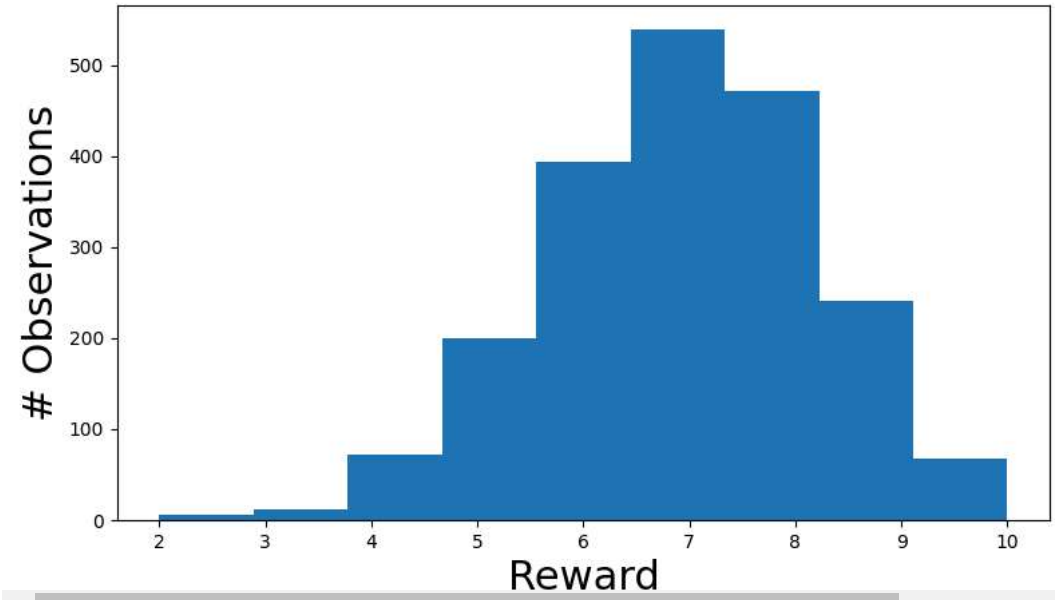
```
7.036
```

```
sum = 0
x = [4,5,6,7]
for j in range(len(x)):
    sum = sum + x[j]
sum
```

```
22
```

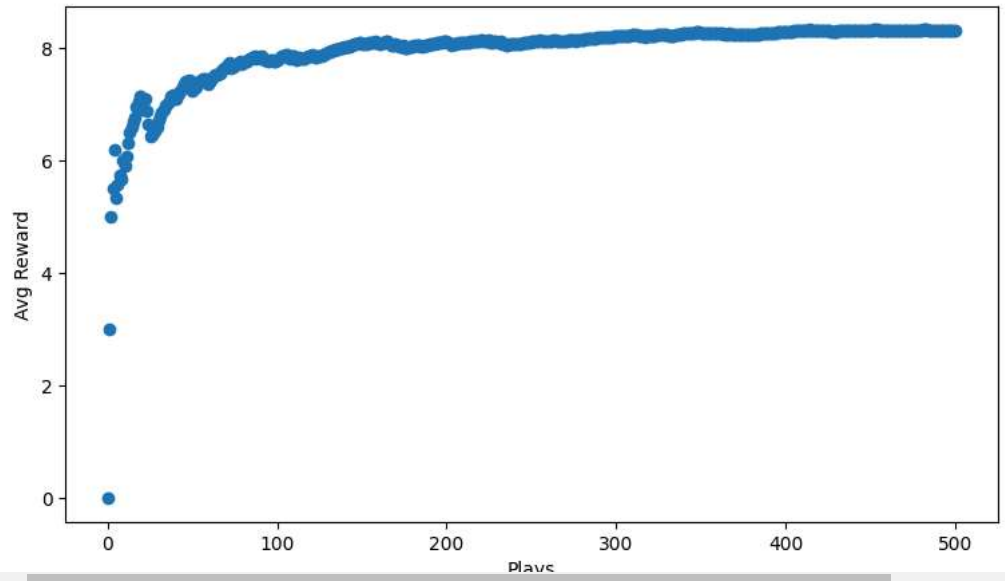
```
plt.figure(figsize=(9,5))
plt.xlabel("Reward",fontsize=22)
plt.ylabel("# Observations",fontsize=22)
plt.hist(reward_test,bins=9)
```

```
(array([  6.,  11.,  71., 200., 394., 539., 472., 240.,  67.]),
 array([ 2.        ,  2.88888889,  3.77777778,  4.66666667,  5.55555556,
         6.44444444,  7.33333333,  8.22222222,  9.11111111, 10.        ]),
 <BarContainer object of 9 artists>)
```



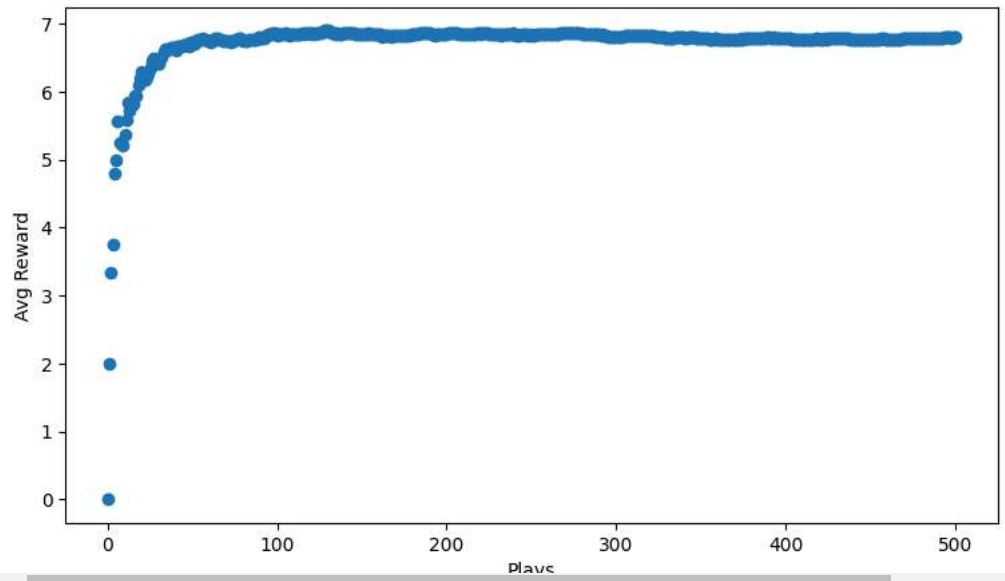

```
# 10 actions x 2 columns
# Columns: Count #, Avg Reward
record = np.zeros((n,2))


def get_best_arm(record):
    arm_index = np.argmax(record[:,1],axis=0)
    return arm_index


def update_record(record,action,r):
    new_r = (record[action,0] * record[action,1] + r) / (record[action,0] + 1)
    record[action,0] += 1
    record[action,1] = new_r
    return record


fig,ax = plt.subplots(1,1)
ax.set_xlabel("Plays")
ax.set_ylabel("Avg Reward")
fig.set_size_inches(9,5)
rewards = [0]
for i in range(500):
    if random.random() > 0.2:
        choice = get_best_arm(record)
    else:
        choice = np.random.randint(10)
    r = get_reward(probs[choice])
    record = update_record(record,choice,r)
    mean_reward = ((i+1) * rewards[-1] + r)/(i+2)
    rewards.append(mean_reward)
ax.scatter(np.arange(len(rewards)),rewards)
```

```
<matplotlib.collections.PathCollection at 0x7b6df02557b0>
```




```
def softmax(av, tau=1.12):
    softm = ( np.exp(av / tau) / np.sum( np.exp(av / tau) ) )
    return softm


probs = np.random.rand(n)
record = np.zeros((n,2))


fig,ax = plt.subplots(1,1)
ax.set_xlabel("Plays")
ax.set_ylabel("Avg Reward")
fig.set_size_inches(9,5)
rewards = [0]
for i in range(500):
    p = softmax(record[:,1],tau=0.7)
    choice = np.random.choice(np.arange(n),p=p)
    r = get_reward(probs[choice])
    record = update_record(record,choice,r)
    mean_reward = ((i+1) * rewards[-1] + r)/(i+2)
    rewards.append(mean_reward)
ax.scatter(np.arange(len(rewards)),rewards)
```

```
<matplotlib.collections.PathCollection at 0x7b6de9f53ee0>
```




```
class ContextBandit:
    def __init__(self, arms=10):
```

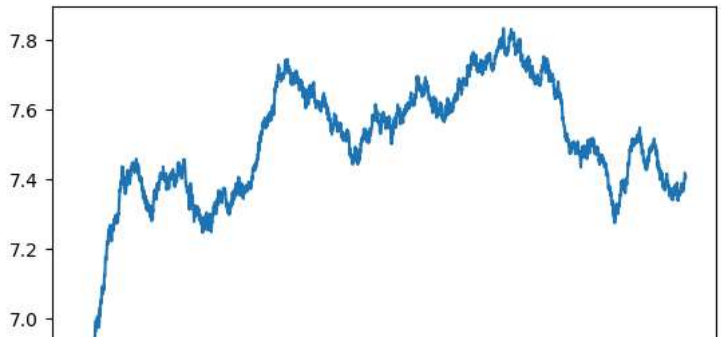
```
            self.arms = arms
            self.init_distribution(arms)
            self.update_state()

        def init_distribution(self, arms):
            # Num states = Num Arms to keep things simple
            self.bandit_matrix = np.random.rand(arms,arms)
            #each row represents a state, each column an arm

        def reward(self, prob):
            reward = 0
            for i in range(self.arms):
                if random.random() < prob:
                    reward += 1
            return reward

        def get_state(self):
            return self.state

        def update_state(self):
            self.state = np.random.randint(0,self.arms)

        def get_reward(self,arm):
            return self.reward(self.bandit_matrix[self.get_state()][arm])

        def choose_arm(self, arm):
            reward = self.get_reward(arm)
            self.update_state()
            return reward


    import numpy as np
    import torch

    arms = 10
    N, D_in, H, D_out = 1, arms, 100, arms


    env = ContextBandit(arms=10)
    state = env.get_state()
    reward = env.choose_arm(1)
    print(state)
```

```
8
```

```
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
        torch.nn.ReLU(),
    )


    loss_fn = torch.nn.MSELoss()


    env = ContextBandit(arms)


    def one_hot(N, pos, val=1):
        one_hot_vec = np.zeros(N)
        one_hot_vec[pos] = val
        return one_hot_vec


    def running_mean(x,N=50):
        c = x.shape[0] - N
        y = np.zeros(c)
        conv = np.ones(N)
        for i in range(c):
            y[i] = (x[i:i+N] @ conv)/N
        return y


    def train(env, epochs=5000, learning_rate=1e-2):
        cur_state = torch.Tensor(one_hot(arms,env.get_state())) #A
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
        rewards = []
        for i in range(epochs):
```

```
            y_pred = model(cur_state) #B
            av_softmax = softmax(y_pred.data.numpy(), tau=2.0) #C
            av_softmax /= av_softmax.sum() #D
            choice = np.random.choice(arms, p=av_softmax) #E
            cur_reward = env.choose_arm(choice) #F
            one_hot_reward = y_pred.data.numpy().copy() #G
            one_hot_reward[choice] = cur_reward #H
            reward = torch.Tensor(one_hot_reward)
            rewards.append(cur_reward)
            loss = loss_fn(y_pred, reward)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            cur_state = torch.Tensor(one_hot(arms,env.get_state())) #I
        return np.array(rewards)


    rewards = train(env)


    plt.plot(running_mean(rewards,N=500))
```

```
[<matplotlib.lines.Line2D at 0x7b6d50abf7c0>]
```



EXPERIMENT 2

Aim: To implement a Deep Q network(DQN) model, train and test the model.

Theory:
Introduction to Gridworld in Reinforcement Learning

The Gridworld environment is a foundational simulation framework in reinforcement learning. It represents a grid-based environment where an agent navigates to achieve specific goals while avoiding obstacles and unfavorable states. This framework is designed to model decision-making problems in a simplified, controlled manner, making it ideal for understanding and testing RL concepts.

In a typical Gridworld:

States correspond to grid cells, each representing a distinct position.
Actions include moving up, down, left, or right.
Rewards guide the agent's behavior by assigning values for reaching specific states, such as a positive reward for reaching the goal and a penalty for stepping into a pit or wall.
Transitions are deterministic or stochastic, defining how actions change the state.
The Gridworld experiment introduces key RL concepts such as state-value functions, policy optimization, and exploration vs. exploitation. By working in a simplified environment, agents can learn to maximize rewards through techniques like Q-learning, SARSA, or Deep Q-Networks (DQN). The Gridworld's modularity also allows for flexible configurations, like static layouts or randomized scenarios, to test different learning algorithms under various conditions.

This experiment focuses on creating and manipulating such a Gridworld to study the agent's behavior under predefined rules and reward structures, paving the way for understanding more complex RL problems.

Experiment-2

Aim :Implement a Deep Q Network(DQN) model,train and test your model.

```
from Gridworld import Gridworld
game = Gridworld(size=4, mode='static')
```

```
game.display()
```

```
array([['+', '-', ' ', 'P'],
       [' ', 'W', ' ', ' '],
       [' ', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ']], dtype='<U2')
```

```
game.makeMove('d')
game.makeMove('d')
game.makeMove('l')
game.display()
```

```
array([['+', '-', ' ', ' '],
       [' ', 'W', ' ', ' '],
       [' ', ' ', 'P', ' '],
       [' ', ' ', ' ', ' ']], dtype='<U2')
```

```
game.reward()
```

```
-1
```

```
game.board.render_np()
```

```
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 0]],

       [[1, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=uint8)
```

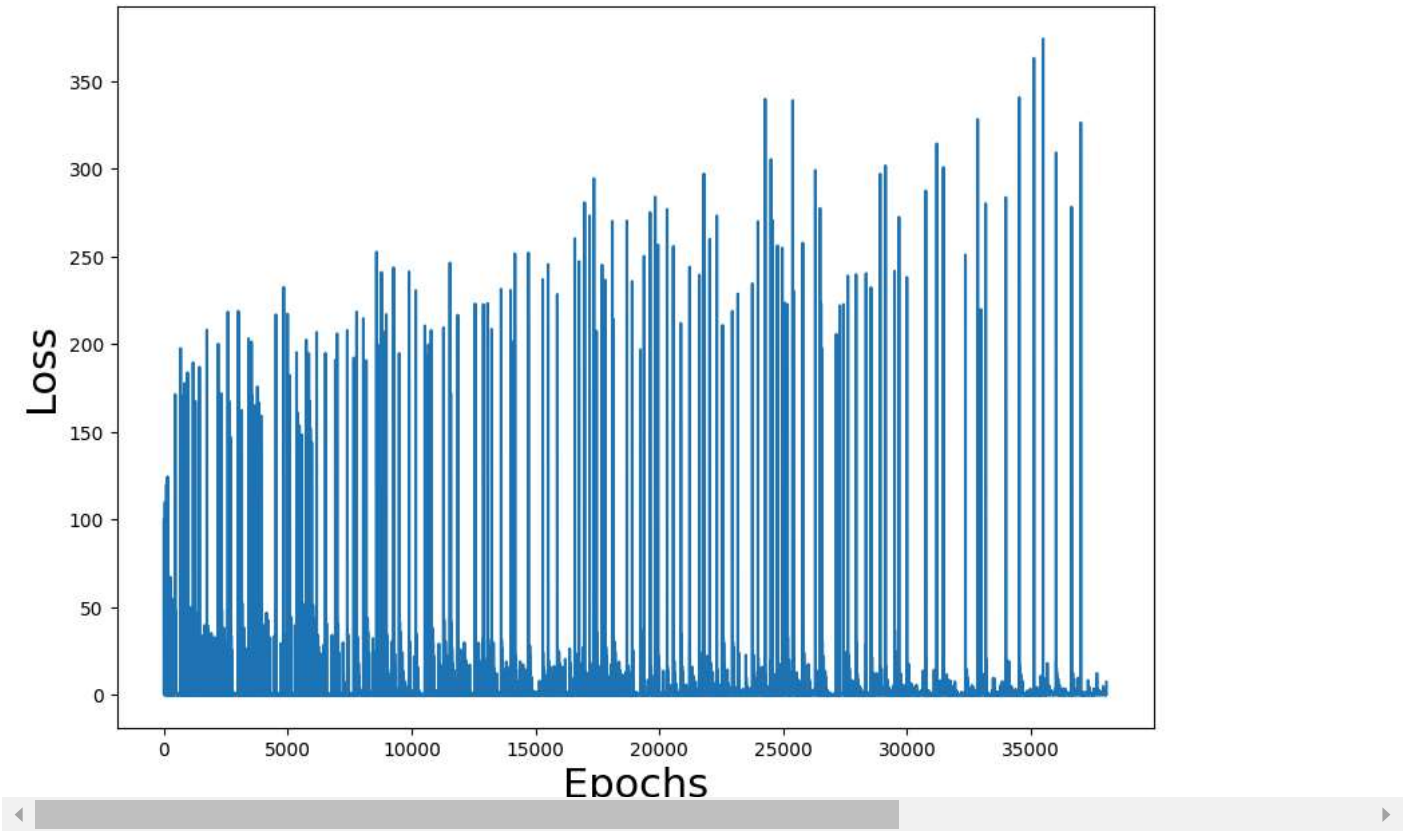
```
import numpy as np
import torch
from Gridworld import Gridworld
from IPython.display import clear_output
import random
from matplotlib import pylab as plt

l1 = 64
l2 = 150
l3 = 100
l4 = 4

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)
loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 1.0
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 1.0


action_set = {
    0: 'u',
    1: 'd',
    2: 'l',
    3: 'r',
}


epochs = 1000
losses = [] #A
for i in range(epochs): #B
    game = Gridworld(size=4, mode='static') #C
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0 #D
    state = torch.from_numpy(state_).float() #E
    status = 1 #F
    while(status == 1): #G
        qval = model(state) #H
        qval_ = qval.data.numpy()
        if (random.random() < epsilon): #I
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_] #J
        game.makeMove(action) #K
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state2 = torch.from_numpy(state2_).float() #L
        reward = game.reward()
        with torch.no_grad():
            newQ = model(state2.reshape(1,64))
        maxQ = torch.max(newQ) #M
        if reward == -1: #N
            Y = reward + (gamma * maxQ)
        else:
            Y = reward
        Y = torch.Tensor([Y]).detach()
        X = qval.squeeze()[action_] #O
        loss = loss_fn(X, Y) #P
        print(i, loss.item())
        clear_output(wait=True)
        optimizer.zero_grad()
        loss.backward()
        losses.append(loss.item())
        optimizer.step()
        state1 = state2
        if reward != -1: #Q
            status = 0
    if epsilon > 0.1: #R
        epsilon -= (1/epochs)
```

```
999 7.286319732666016
```

```
plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)
```

```
Text(0, 0.5, 'Loss')
```



PyTorch Automatic Differentiation Review

```
m = torch.Tensor([2.0])
m.requires_grad=True
b = torch.Tensor([1.0])
b.requires_grad=True
def linear_model(x,m,b):
    y = m + b#@ x + b
    return y


#with torch.no_grad():
y = linear_model(torch.Tensor([4]),m,b)


y.grad_fn
```

```
<AddBackward0 at 0x1b3285c31c0>
```

```
y.backward()


m.grad
```

```
tensor([1.])
```

```
def test_model(model, mode='static', display=True):
    i = 0
    test_game = Gridworld(mode=mode)
    state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    state = torch.from_numpy(state_).float()
    if display:
        print("Initial State:")
        print(test_game.display())
    status = 1
    while(status == 1): #A
        qval = model(state)
        qval_ = qval.data.numpy()
        action_ = np.argmax(qval_) #B
        action = action_set[action_]
```

```
        if display:
            print('Move #: %s; Taking action: %s' % (i, action))
        test_game.makeMove(action)
        state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state = torch.from_numpy(state_).float()
        if display:
            print(test_game.display())
        reward = test_game.reward()
        if reward != -1:
            if reward > 0:
                status = 2
                if display:
                    print("Game won! Reward: %s" % (reward,))
            else:
                status = 0
                if display:
                    print("Game LOST. Reward: %s" % (reward,))
        i += 1
        if (i > 15):
            if display:
                print("Game lost; too many moves.")
            break

    win = True if status == 2 else False
    return win


test_model(model, 'random')
```

```
Initial State:
[[' ' ' ' ' ' ' ' ']
 ['-' ' ' ' ' ' ' ']
 [' ' 'W' ' ' ' ' ']
 [' ' ' ' '+' 'P']]
Move #: 0; Taking action: l
[[' ' ' ' ' ' ' ' ']
 ['-' ' ' ' ' ' ' ']
 [' ' 'W' ' ' ' ' ']
 [' ' ' ' '+' ' ' ']]
Game won! Reward: 10
True
```

```
l1 = 64
l2 = 150
l3 = 100
l4 = 4

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)
loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 0.3


from collections import deque
epochs = 5000
losses = []
mem_size = 1000 #A
batch_size = 200 #B
replay = deque(maxlen=mem_size) #C
max_moves = 50 #D
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
```

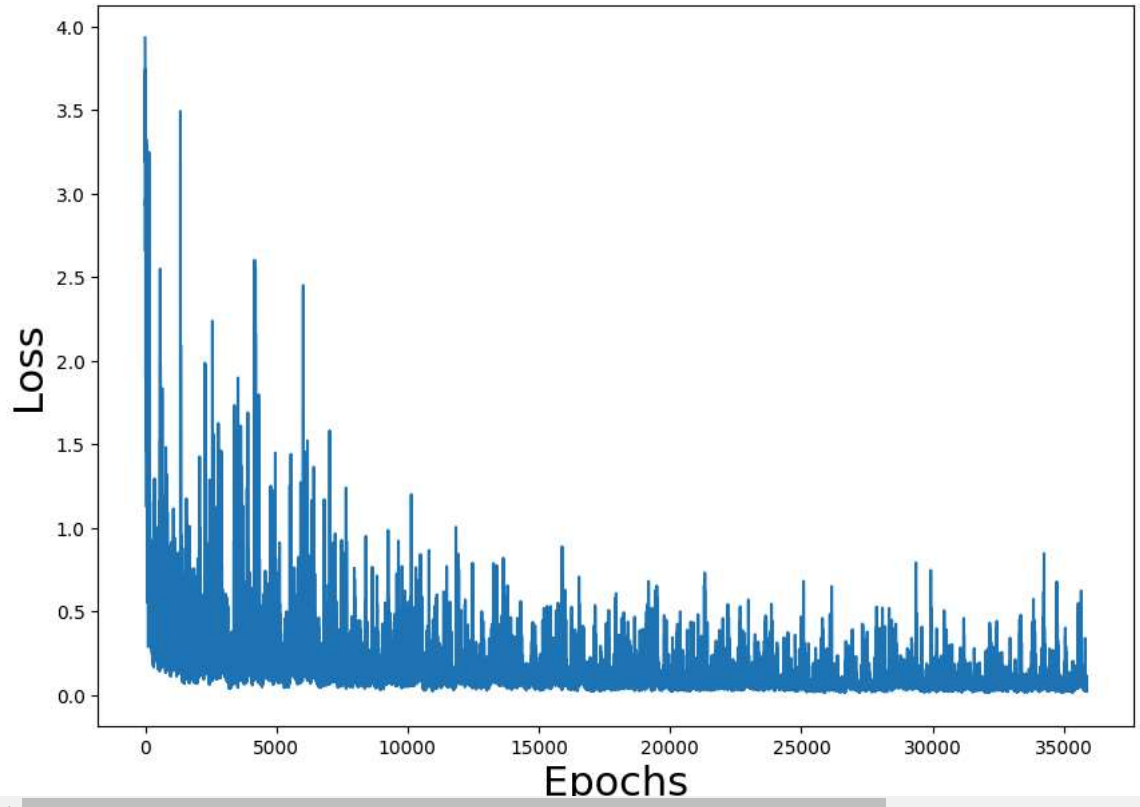
```
        mov = 0
        while(status == 1):
            mov += 1
            qval = model(state1) #E
            qval_ = qval.data.numpy()
            if (random.random() < epsilon): #F
                action_ = np.random.randint(0,4)
            else:
                action_ = np.argmax(qval_)

            action = action_set[action_]
            game.makeMove(action)
            state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
            state2 = torch.from_numpy(state2_).float()
            reward = game.reward()
            done = True if reward > 0 else False
            exp =  (state1, action_, reward, state2, done) #G
            replay.append(exp) #H
            state1 = state2

            if len(replay) > batch_size: #I
                minibatch = random.sample(replay, batch_size) #J
                state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch]) #K
                action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
                reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
                state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
                done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])

                Q1 = model(state1_batch) #L
                with torch.no_grad():
                    Q2 = model(state2_batch) #M

                Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2,dim=1)[0]) #N
                X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
                loss = loss_fn(X, Y.detach())
                print(i, loss.item())
                clear_output(wait=True)
                optimizer.zero_grad()
                loss.backward()
                losses.append(loss.item())
                optimizer.step()

            if reward != -1 or mov > max_moves: #O
                status = 0
                mov = 0
    losses = np.array(losses)

    #A Set the total size of the experience replay memory
    #B Set the minibatch size
    #C Create the memory replay as a deque list
    #D Maximum number of moves before game is over
    #E Compute Q-values from input state in order to select action
    #F Select action using epsilon-greedy strategy
    #G Create experience of state, reward, action and next state as a tuple
    #H Add experience to experience replay list
    #I If replay list is at least as long as minibatch size, begin minibatch training
    #J Randomly sample a subset of the replay list
    #K Separate out the components of each experience into separate minibatch tensors
    #L Re-compute Q-values for minibatch of states to get gradients
    #M Compute Q-values for minibatch of next states but don't compute gradients
    #N Compute the target Q-values we want the DQN to learn
    #O If game is over, reset status and mov number
```

```
4999 0.03387792035937309
```

```
    def running_mean(x,N=50):
        c = x.shape[0] - N
        y = np.zeros(c)
        conv = np.ones(N)
        for i in range(c):
            y[i] = (x[i:i+N] @ conv)/N
        return y


    plt.figure(figsize=(10,7))
    plt.plot(losses)
```

```
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)
```

```
Text(0, 0.5, 'Loss')
```




```
max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}%".format(100.0*win_perc))
```

```
Games played: 1000, # of wins: 928
Win percentage: 92.80000000000001%
```

```
test_model(model, mode='random')
```

```
Initial State:
[[' ' ' ' ' ' ' ' ']
 ['P' ' ' '-' ' ']
 [' ' ' ' 'W' '+']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: r
[[' ' ' ' ' ' ' ']
 [' ' 'P' '-' ' ']
 [' ' ' ' 'W' '+']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[[' ' ' ' ' ' ' ']
 [' ' ' ' '-' ' ']
 [' ' 'P' 'W' '+']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: r
[[' ' ' ' ' ' ' ']
 [' ' ' ' '-' ' ']
 [' ' 'P' 'W' '+']
 [' ' ' ' ' ' ' ']]
Move #: 3; Taking action: r
[[' ' ' ' ' ' ' ']
 [' ' ' ' '-' ' ']
 [' ' 'P' 'W' '+']
 [' ' ' ' ' ' ' ']]
Move #: 4; Taking action: r
```

```
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 5; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 6; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 7; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 8; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 9; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
  [' ' 'P' 'W' '+']
  [' ' ' ' ' ' ' ']]
 Move #: 10; Taking action: r
 [[' ' ' ' ' ' ' ']
  [' ' ' ' '-' ' ']
```

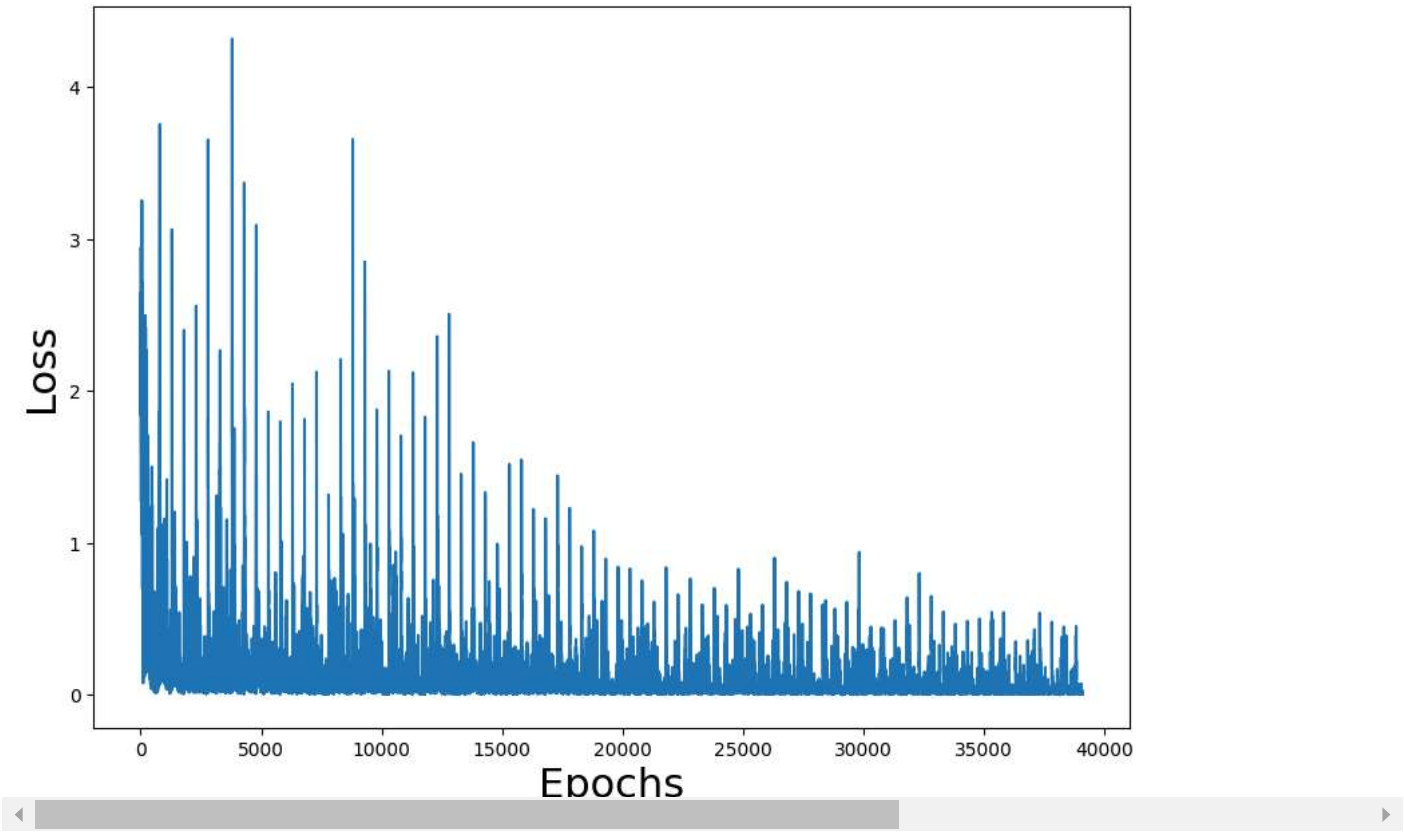
```
import copy

l1 = 64
l2 = 150
l3 = 100
l4 = 4


model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)

model2 = copy.deepcopy(model) #A
model2.load_state_dict(model.state_dict()) #B

loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 0.3


from collections import deque
epochs = 5000
losses = []
mem_size = 1000
batch_size = 200
replay = deque(maxlen=mem_size)
max_moves = 50
h = 0
sync_freq = 500 #A
j=0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        j+=1
        mov += 1
        qval = model(state1)
```

```
            qval_ = qval.data.numpy()
            if (random.random() < epsilon):
                action_ = np.random.randint(0,4)
            else:
                action_ = np.argmax(qval_)

            action = action_set[action_]
            game.makeMove(action)
            state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
            state2 = torch.from_numpy(state2_).float()
            reward = game.reward()
            done = True if reward > 0 else False
            exp =  (state1, action_, reward, state2, done)
            replay.append(exp) #H
            state1 = state2

            if len(replay) > batch_size:
                minibatch = random.sample(replay, batch_size)
                state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
                action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
                reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
                state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
                done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])
                Q1 = model(state1_batch)
                with torch.no_grad():
                    Q2 = model2(state2_batch) #B

                Y = reward_batch + gamma * ((1-done_batch) * torch.max(Q2,dim=1)[0])
                X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
                loss = loss_fn(X, Y.detach())
                print(i, loss.item())
                clear_output(wait=True)
                optimizer.zero_grad()
                loss.backward()
                losses.append(loss.item())
                optimizer.step()

                if j % sync_freq == 0: #C
                    model2.load_state_dict(model.state_dict())
            if reward != -1 or mov > max_moves:
                status = 0
                mov = 0

losses = np.array(losses)

#A Set the update frequency for synchronizing the target model parameters to the main DQN
#B Use the target network to get the maiximum Q-value for the next state
#C Copy the main model parameters to the target network
```

```
4999 0.005407183896750212
```

```
plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)
```

```
Text(0, 0.5, 'Loss')
```




```
max_games = 1000
wins = 0
for i in range(max_games):
```

EXPERIMENT 3

Aim: To implement and test Policy Gradient Methods in Reinforcement Learning.

Theory:

Policy gradient methods form a critical class of reinforcement learning (RL) algorithms that optimize the agent's policy directly by computing gradients with respect to its parameters. Unlike value-based methods, which focus on estimating state or action values, policy gradient approaches are concerned with improving the policy that determines the probabilities of selecting actions in a given state. This makes them particularly effective for environments with high-dimensional or continuous action spaces.

Context and Relevance:

A policy in reinforcement learning dictates how an agent behaves in various states of an environment. By parameterizing the policy—often using neural networks—policy gradient methods seek to identify the optimal parameters that maximize the agent's long-term cumulative reward. This direct approach avoids some of the challenges associated with value-based methods, such as instability in value function approximation.

Objective Function: The goal is to maximize the expected return $J(\theta)=E[R]$, where R is the cumulative reward collected by the agent.

Gradient Estimation: Gradients of $J(\theta)$ with respect to the policy parameters θ are estimated using algorithms like REINFORCE, which leverages the log-derivative trick for efficient computation.

This experiment implements and analyzes policy gradient methods, exploring their potential for learning optimal policies in challenging environments. By focusing on direct policy optimization, the experiment contributes to understanding how reinforcement learning can be applied effectively to real-world tasks requiring continuous decision-making.

Experiment 3

Aim : Implement and test the Policy Gradient Method and test.

```
!pip install gymnasium
```

```
Collecting gymnasium
  Downloading gymnasium-1.0.0-py3-none-any.whl.metadata (9.5 kB)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (3.1.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (4.12.2)
Collecting farama-notifications>=0.0.1 (from gymnasium)
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Downloading gymnasium-1.0.0-py3-none-any.whl (958 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 958.1/958.1 kB 11.7 MB/s eta 0:00:00
Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Installing collected packages: farama-notifications, gymnasium
Successfully installed farama-notifications-0.0.4 gymnasium-1.0.0
```

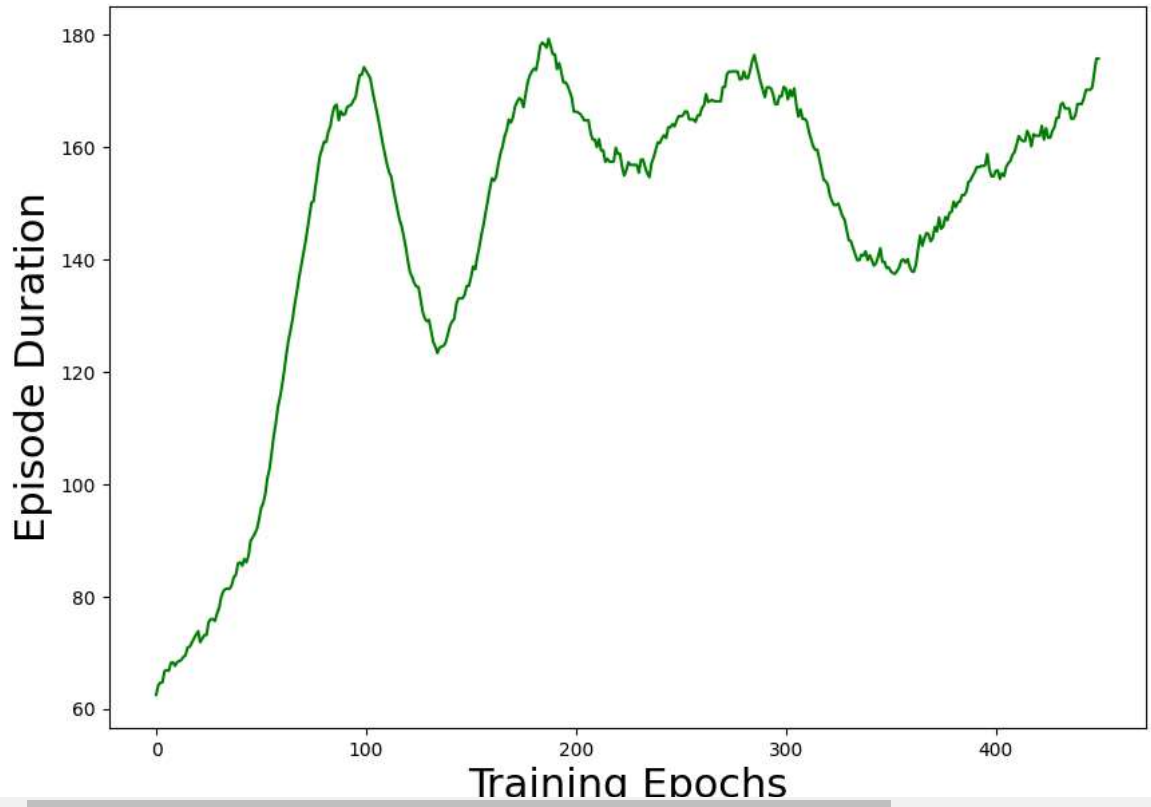
```
import numpy as np
import torch
import gymnasium as gym
from matplotlib import pyplot as plt


def running_mean(x, N=50):
    kernel = np.ones(N)
    conv_len = x.shape[0]-N
    y = np.zeros(conv_len)
    for i in range(conv_len):
        y[i] = kernel @ x[i:i+N]
        y[i] /= N
    return y


env = gym.make("CartPole-v1")


import numpy as np
import torch

l1 = 4 #A
l2 = 150
l3 = 2 #B

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.Softmax(dim=0) #C
)

learning_rate = 0.009
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

#A Input data is length 4
#B Output is a 2-length vector for the Left and the Right actions
#C Output is a softmax probability distribution over actions


state1 = env.reset()
pred = model(torch.from_numpy(state1[0])) #G
action = np.random.choice(np.array([0,1]), p=pred.data.numpy()) #H
state2, reward, done, info, _ = env.step(action) #I

#G Call policy network model to produce predicted action probabilities
#H Sample an action from the probability distribution produced by the policy network
#I Take the action, receive new state and reward. The info variable is produced by the environment but is irrelevant


def discount_rewards(rewards, gamma=0.99):
    lenr = len(rewards)
    disc_return = torch.pow(gamma,torch.arange(lenr).float()) * rewards #A
    disc_return /= disc_return.max() #B
```

```
    return disc_return

#A Compute exponentially decaying rewards
#B Normalize the rewards to be within the [0,1] interval to improve numerical stability


def loss_fn(preds, r): #A
    return -1 * torch.sum(r * torch.log(preds)) #B

#A The loss function expects an array of action probabilities for the actions that were taken and the discounted rewards.
#B It computes the log of the probabilities, multiplies by the discounted rewards, sums them all and flips the sign.


MAX_DUR = 200
MAX_EPISODES = 500
gamma = 0.99
score = [] #A
expectation = 0.0
for episode in range(MAX_EPISODES):
    curr_state = env.reset()[0]
    done = False
    transitions = [] #B

    for t in range(MAX_DUR): #C
        act_prob = model(torch.from_numpy(curr_state).float()) #D
        action = np.random.choice(np.array([0,1]), p=act_prob.data.numpy()) #E
        prev_state = curr_state
        curr_state, _, done, _, info = env.step(action) #F
        transitions.append((prev_state, action, t+1)) #G
        if done: #H
            break

    ep_len = len(transitions) #I
    score.append(ep_len)
    reward_batch = torch.Tensor([r for (s,a,r) in transitions]).flip(dims=(0,)) #J
    disc_returns = discount_rewards(reward_batch) #K
    state_batch = torch.Tensor([s for (s,a,r) in transitions]) #L
    action_batch = torch.Tensor([a for (s,a,r) in transitions]) #M
    pred_batch = model(state_batch) #N
    prob_batch = pred_batch.gather(dim=1,index=action_batch.long().view(-1,1)).squeeze() #O
    loss = loss_fn(prob_batch, disc_returns)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

#A List to keep track of the episode length over training time
#B List of state, action, rewards (but we ignore the reward)
#C While in episode
#D Get the action probabilities
#E Select an action stochastically
#F Take the action in the environment
#G Store this transition
#H If game is lost, break out of the loop
#I Store the episode length
#J Collect all the rewards in the episode in a single tensor
#K Compute the discounted version of the rewards
#L Collect the states in the episode in a single tensor
#M Collect the actions in the episode in a single tensor
#N Re-compute the action probabilities for all the states in the episode
#O Subset the action-probabilities associated with the actions that were actually taken
```

```
<ipython-input-10-90527e783f05>:24: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please consider conv
  state_batch = torch.Tensor([s for (s,a,r) in transitions]) #L
```

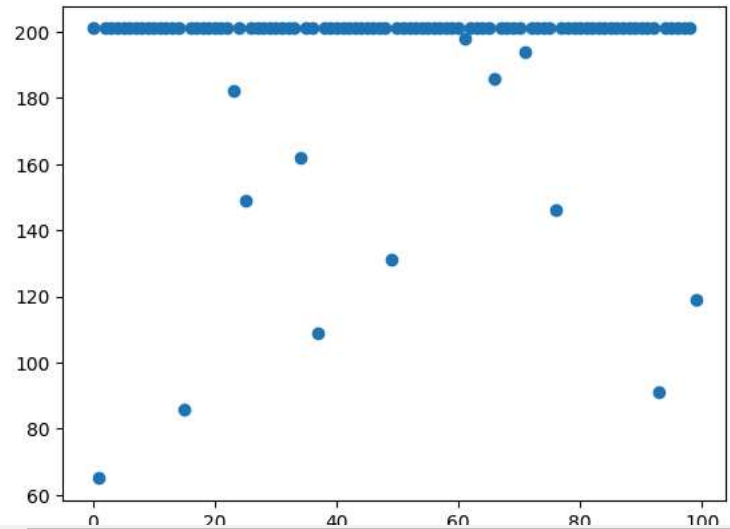
```
score = np.array(score)
avg_score = running_mean(score, 50)


plt.figure(figsize=(10,7))
plt.ylabel("Episode Duration",fontsize=22)
plt.xlabel("Training Epochs",fontsize=22)
plt.plot(avg_score, color='green')
```

```
[<matplotlib.lines.Line2D at 0x7922dd0f68f0>]
```




```
score = []
games = 100
done = False
state1 = env.reset()[0]
for i in range(games):
    t=0
    while not done: #F
        if type(state1) is tuple:
            state1 = state1[0]
        pred = model(torch.from_numpy(state1).float()) #G
        action = np.random.choice(np.array([0,1]), p=pred.data.numpy()) #H
        state2, reward, done, _, info = env.step(action) #I
        state1 = state2
        if(type(state1) == 'tuple'):
            state1 = state2[0]

        t += 1
        if t > MAX_DUR: #L
            break;
    state1 = env.reset()
    done = False
    score.append(t)
score = np.array(score)


plt.scatter(np.arange(score.shape[0]),score)
```

```
<matplotlib.collections.PathCollection at 0x7922dcedec80>
```



Start coding or generate with AI.

Start coding or generate with AI.

EXPERIMENT 4

Aim: To implement an Actor-Critic Model for reinforcement learning and demonstrate its effectiveness in solving the "CartPole-v1" environment**.**

Theory:

The Actor-Critic model is a hybrid reinforcement learning approach that combines policy-based and value-based methods. It consists of two primary components: the actor, which determines the actions to be taken based on a policy, and the critic, which evaluates those actions by estimating the value function. This dual architecture allows the model to effectively balance exploration and exploitation during learning.The actor maps states to a probability distribution over possible actions using a policy network. It learns by maximizing the expected cumulative reward, with guidance from the critic. The critic evaluates the value of states or state-action pairs, using techniques like temporal difference learning, and provides feedback to stabilize the actor's updates. This reduces the variance in policy gradient methods and accelerates convergence.

In this experiment, a neural network is implemented with shared layers followed by separate layers for the actor and critic. The actor outputs action probabilities, while the critic estimates the value of the current state. The training process leverages multiprocessing to create multiple worker processes that interact with the environment in parallel. These processes share a common model and update it based on their experiences, enabling efficient learning.

The loss function combines actor and critic components. The actor's loss is designed to encourage actions that maximize rewards, while the critic's loss minimizes the error in value estimation. A discount factor is used to prioritize immediate rewards over distant ones, ensuring better decision-making over time.

The implemented model is tested in the "CartPole-v1" environment, where it learns to balance the pole effectively through repeated interactions. This experiment demonstrates how the Actor-Critic model leverages the strengths of both policy and value-based approaches, making it a robust solution for complex reinforcement learning problems.

Experiment-4

Aim : Implement an Actor-Critic Model.

```
import multiprocessing as mp
import numpy as np
def square(x): #A
    return np.square(x)
x = np.arange(64) #B
print(x)
print(mp.cpu_count())
pool = mp.Pool(8) #C
squared = pool.map(square, [x[8*i:8*i+8] for i in range(8)])
print(squared)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63]
2
[array([ 0,  1,  4,  9, 16, 25, 36, 49]), array([ 64,  81, 100, 121, 144, 169, 196, 225]), array([256, 289, 324, 361, 400, 441, 484, 529
```

```
def square(i, x, queue):
    print("In process {}".format(i,))
    queue.put(np.square(x))
processes = [] #A
queue = mp.Queue() #B
x = np.arange(64) #C
for i in range(8): #D
    start_index = 8*i
    proc = mp.Process(target=square,args=(i,x[start_index:start_index+8], queue))
    proc.start()
    processes.append(proc)

for proc in processes: #E
    proc.join()

for proc in processes: #F
    proc.terminate()
results = []
while not queue.empty(): #G
    results.append(queue.get())

print(results)
```

```
In process 0
In process 1
In process 2

In process 3In process 5In process 4

In process 6
In process 7
[array([ 0,  1,  4,  9, 16, 25, 36, 49]), array([ 64,  81, 100, 121, 144, 169, 196, 225]), array([256, 289, 324, 361, 400, 441, 484, 529
```

```
import torch
from torch import nn
from torch import optim
import numpy as np
from torch.nn import functional as F
import gym
import torch.multiprocessing as mp #A

class ActorCritic(nn.Module): #B
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.l1 = nn.Linear(4,25)
        self.l2 = nn.Linear(25,50)
        self.actor_lin1 = nn.Linear(50,2)
        self.l3 = nn.Linear(50,25)
```

```
        self.critic_lin1 = nn.Linear(25,1)
    def forward(self,x):
        x = F.normalize(x,dim=0)
        y = F.relu(self.l1(x))
        y = F.relu(self.l2(y))
        actor = F.log_softmax(self.actor_lin1(y),dim=0) #C
        c = F.relu(self.l3(y.detach()))
        critic = torch.tanh(self.critic_lin1(c)) #D
        return actor, critic #E


def worker(t, worker_model, counter, params):
    worker_env = gym.make("CartPole-v1")
    worker_env.reset()
    worker_opt = optim.Adam(lr=1e-4,params=worker_model.parameters()) #A
    worker_opt.zero_grad()
    for i in range(params['epochs']):
        worker_opt.zero_grad()
        values, logprobs, rewards = run_episode(worker_env,worker_model) #B
        actor_loss,critic_loss,eplen = update_params(worker_opt,values,logprobs,rewards) #C
        counter.value = counter.value + 1 #D


def run_episode(worker_env, worker_model):
    state = torch.from_numpy(worker_env.env.state).float() #A
    values, logprobs, rewards = [],[],[] #B
    done = False
    j=0
    while (done == False): #C
        j+=1
        policy, value = worker_model(state) #D
        values.append(value)
        logits = policy.view(-1)
        action_dist = torch.distributions.Categorical(logits=logits)
        action = action_dist.sample() #E
        logprob_ = policy.view(-1)[action]
        logprobs.append(logprob_)
        state_, _, done, _, info = worker_env.step(action.detach().numpy())
        state = torch.from_numpy(state_).float()
        if done: #F
            reward = -10
            worker_env.reset()
        else:
            reward = 1.0
        rewards.append(reward)
    return values, logprobs, rewards


def update_params(worker_opt,values,logprobs,rewards,clc=0.1,gamma=0.95):
        rewards = torch.Tensor(rewards).flip(dims=(0,)).view(-1) #A
        logprobs = torch.stack(logprobs).flip(dims=(0,)).view(-1)
        values = torch.stack(values).flip(dims=(0,)).view(-1)
        Returns = []
        ret_ = torch.Tensor([0])
        for r in range(rewards.shape[0]): #B
            ret_ = rewards[r] + gamma * ret_
            Returns.append(ret_)
        Returns = torch.stack(Returns).view(-1)
        Returns = F.normalize(Returns,dim=0)
        actor_loss = -1*logprobs * (Returns - values.detach()) #C
        critic_loss = torch.pow(values - Returns,2) #D
        loss = actor_loss.sum() + clc*critic_loss.sum() #E
        loss.backward()
        worker_opt.step()
        return actor_loss, critic_loss, len(rewards)


MasterNode = ActorCritic() #A
MasterNode.share_memory() #B
processes = [] #C
params = {
    'epochs':1000,
    'n_workers':7,
}
counter = mp.Value('i',0) #D
for i in range(params['n_workers']):
    p = mp.Process(target=worker, args=(i,MasterNode,counter,params)) #E
    p.start()
```

```
    processes.append(p)
for p in processes: #F
    p.join()
for p in processes: #G
    p.terminate()

print(counter.value,processes[1].exitcode) #H
```

```
      values, logprobs, rewards = run_episode(worker_env,worker_model) #B
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
Traceback (most recent call last):
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
Process Process-19:
Traceback (most recent call last):
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
  File "/usr/lib/python3.10/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
Traceback (most recent call last):
  File "/usr/lib/python3.10/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
ValueError: not enough values to unpack (expected 5, got 4)
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
ValueError: not enough values to unpack (expected 5, got 4)
  File "/usr/lib/python3.10/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
ValueError: not enough values to unpack (expected 5, got 4)
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
  File "/usr/lib/python3.10/multiprocessing/process.py", line 108, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
ValueError: not enough values to unpack (expected 5, got 4)
ValueError: not enough values to unpack (expected 5, got 4)
  File "<ipython-input-12-cacb6a87066b>", line 8, in worker
    values, logprobs, rewards = run_episode(worker_env,worker_model) #B
ValueError: not enough values to unpack (expected 5, got 4)
  File "<ipython-input-13-4c109c72850c>", line 15, in run_episode
    state_, _, done, _, info = worker_env.step(action.detach().numpy())
ValueError: not enough values to unpack (expected 5, got 4)
0 1
```

Test the trained agent

```
!pip install gym
!pip install torch
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.10/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym) (3.1.0)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym) (0.0.8)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.5.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
```

```
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2024.10.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (3.0.2)
```

```
import gym
import numpy
import torch
env = gym.make("CartPole-v1")
env.reset()

for i in range(100):
    state_ = np.array(env.env.state)
    state = torch.from_numpy(state_).float()
    logits,value = MasterNode(state)
    action_dist = torch.distributions.Categorical(logits=logits)
    action = action_dist.sample()
    state2, reward, done, info = env.step(action.detach().numpy())
    if done:
        print("Lost")
        env.reset()
    state_ = np.array(env.env.state)
    state = torch.from_numpy(state_).float()
    env.render()
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:49: DeprecationWarning: WARN: You are calling render method, but you didn't specifie
If you want to render in human mode, initialize the environment in this way: gym.make('EnvName', render_mode='human') and don't call the
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
/usr/local/lib/python3.10/dist-packages/pygame/pkgdata.py:25: DeprecationWarning: pkg_resources is deprecated as an API. See https://set
  from pkg_resources import resource_stream, resource_exists
/usr/local/lib/python3.10/dist-packages/pkg_resources/__init__.py:3154: DeprecationWarning: Deprecated call to `pkg_resources.declare_na
Implementing implicit namespace packages (as specified in PEP 420) is preferred to `pkg_resources.declare_namespace`. See https://setupt
  declare_namespace(pkg)
/usr/local/lib/python3.10/dist-packages/pkg_resources/__init__.py:3154: DeprecationWarning: Deprecated call to `pkg_resources.declare_na
Implementing implicit namespace packages (as specified in PEP 420) is preferred to `pkg_resources.declare_namespace`. See https://setupt
  declare_namespace(pkg)
/usr/local/lib/python3.10/dist-packages/pkg_resources/__init__.py:3154: DeprecationWarning: Deprecated call to `pkg_resources.declare_na
Implementing implicit namespace packages (as specified in PEP 420) is preferred to `pkg_resources.declare_namespace`. See https://setupt
  declare_namespace(pkg)
/usr/local/lib/python3.10/dist-packages/gym/core.py:49: DeprecationWarning: WARN: You are calling render method, but you didn't specifie
If you want to render in human mode, initialize the environment in this way: gym.make('EnvName', render_mode='human') and don't call the
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
Lost
Lost
```

```
def run_episode(worker_env, worker_model, N_steps=10):
    raw_state = np.array(worker_env.env.state)
    state = torch.from_numpy(raw_state).float()
    values, logprobs, rewards = [],[],[]
    done = False
    j=0
    G=torch.Tensor([0]) #A
    while (j < N_steps and done == False): #B
        j+=1
        policy, value = worker_model(state)
        values.append(value)
        logits = policy.view(-1)
        action_dist = torch.distributions.Categorical(logits=logits)
        action = action_dist.sample()
        logprob_ = policy.view(-1)[action]
        logprobs.append(logprob_)
        state_, _, done, info = worker_env.step(action.detach().numpy())
        state = torch.from_numpy(state_).float()
        if done:
            reward = -10
            worker_env.reset()
        else: #C
            reward = 1.0
            G = value.detach()
        rewards.append(reward)
    return values, logprobs, rewards, G
```

```
#Simulated rewards for 3 steps
r1 = [1,1,-1]
```

```
r2 = [1,1,1]
R1,R2 = 0.0,0.0
#No bootstrapping
for i in range(len(r1)-1,0,-1):
    R1 = r1[i] + 0.99*R1
for i in range(len(r2)-1,0,-1):
    R2 = r2[i] + 0.99*R2
print("No bootstrapping")
print(R1,R2)
#With bootstrapping
R1,R2 = 1.0,1.0
for i in range(len(r1)-1,0,-1):
    R1 = r1[i] + 0.99*R1
for i in range(len(r2)-1,0,-1):
    R2 = r2[i] + 0.99*R2
print("With bootstrapping")
print(R1,R2)
```

```
No bootstrapping
0.010000000000000009 1.99
With bootstrapping
0.9901 2.9701
```

Start coding or generate with AI.

EXPERIMENT 5

Aim: To implement a genetic algorithm that evolves a population of random strings towards a given target string through selection, crossover, and mutation.

Theory:

Genetic algorithms (GAs) are optimization techniques inspired by natural selection. In this experiment, GAs are used to evolve a population of random strings to match a target string, "Hello World!". Each string, represented as an *Individual*, is evaluated for fitness based on its similarity to the target string using a similarity metric.

Initially, a population of random strings is generated. Parents are selected based on their fitness scores, with higher fitness individuals more likely to reproduce. Selected parents undergo *crossover*, exchanging string segments to produce offspring. A small mutation rate introduces random changes to maintain diversity. Over successive generations, the population evolves, with the average fitness improving as strings become more similar to the target.

The experiment highlights the balance between exploration and exploitation in genetic algorithms. While selection and crossover ensure progress toward the solution, mutation prevents the algorithm from getting stuck in local optima. This balance allows GAs to navigate large search spaces effectively.

Through visualizations of fitness progression, such as a graph of average fitness over generations, the experiment showcases the gradual convergence toward the target string. This demonstrates the power and flexibility of genetic algorithms in solving optimization problems.

Experiment-5

Aim : Implement genetic algorithms for evolving a set of random strings toward a target string.

```
import random
from matplotlib import pyplot as plt


alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.! "
target = "Hello World!"


class Individual:
    def __init__(self, string, fitness=0):
        self.string = string
        self.fitness = fitness


from difflib import SequenceMatcher

def similar(a, b):
    return SequenceMatcher(None, a, b).ratio()


def spawn_population(length=26,size=100):
    pop = []
    for i in range(size):
        string = ''.join(random.choices(alphabet,k=length))
        individual = Individual(string)
        pop.append(individual)
    return pop


def mutate(x, mut_rate=0.01):
    new_x_ = []
    for char in x.string:
        if random.random() < mut_rate:
            new_x_.extend(random.choices(alphabet,k=1))
        else:
            new_x_.append(char)
    new_x = Individual(''.join(new_x_))
    return new_x


def recombine(p1_, p2_): #produces two children from two parents
    p1 = p1_.string
    p2 = p2_.string
    child1 = []
    child2 = []
    cross_pt = random.randint(0,len(p1))
    child1.extend(p1[0:cross_pt])
    child1.extend(p2[cross_pt:])
    child2.extend(p2[0:cross_pt])
    child2.extend(p1[cross_pt:])
    c1 = Individual(''.join(child1))
    c2 = Individual(''.join(child2))
    return c1, c2


def evaluate_population(pop, target):
    avg_fit = 0
    for i in range(len(pop)):
        fit = similar(pop[i].string, target)
        pop[i].fitness = fit
        avg_fit += fit
    avg_fit /= len(pop)
    return pop, avg_fit


def next_generation(pop, size=100, length=26, mut_rate=0.01):
    new_pop = []
    while len(new_pop) < size:
        parents = random.choices(pop,k=2, weights=[x.fitness for x in pop])
        offspring_ = recombine(parents[0],parents[1])
        offspring = [mutate(offspring_[0], mut_rate=mut_rate), mutate(offspring_[1], mut_rate=mut_rate)]
```

```
        new_pop.extend(offspring) #add offspring to next generation
    return new_pop


pop = spawn_population(length=len(target))


pop, avg_fit = evaluate_population(pop, target)
print(avg_fit)
```

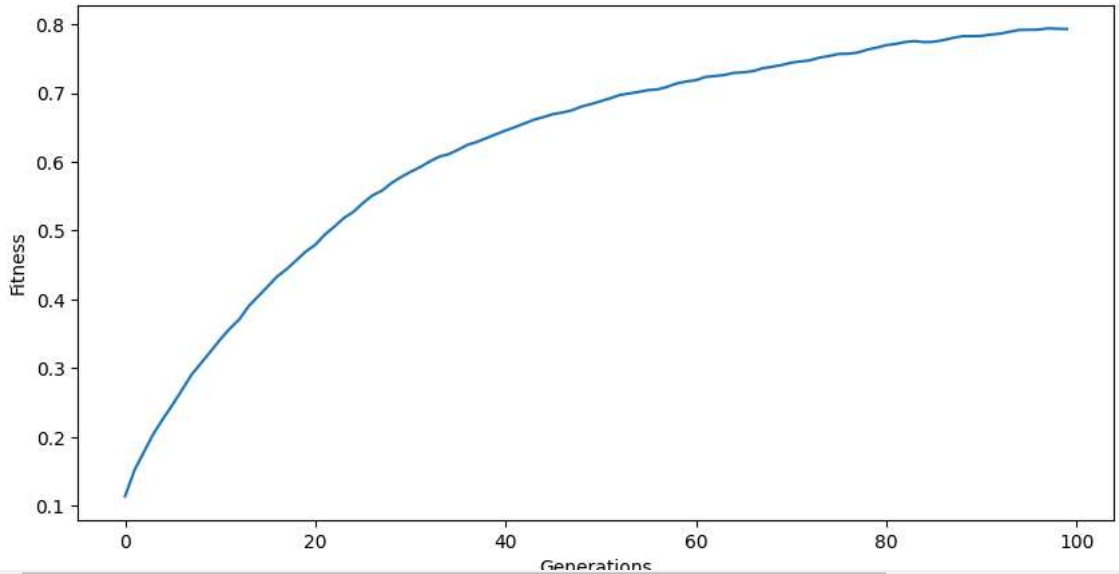
```
0.13333333333333336
```

```
new_pop = next_generation(pop, length=len(target))


new_pop[10].fitness
```

```
0
```

```
num_generations = 100
population_size = 3000
str_len = len(target)
mutation_rate = 0.001 # 0.1% mutation rate per character


pop_fit = []
pop = spawn_population(size=population_size, length=str_len) #initial population
for gen in range(num_generations):
    # trainning
    pop, avg_fit = evaluate_population(pop, target)
    pop_fit.append(avg_fit) #record population average fitness
    new_pop = next_generation(pop, size=population_size, length=str_len, mut_rate=mutation_rate)
    pop = new_pop


plt.figure(figsize=(10,5))
plt.xlabel("Generations")
plt.ylabel("Fitness")
plt.plot(pop_fit)
```

```
[<matplotlib.lines.Line2D at 0x7b4acedaef80>]
```




```
pop.sort(key=lambda x: x.fitness, reverse=True) #sort in place, highest fitness first


pop[0].string
```

Start coding or generate with AI.

EXPERIMENT 6

Aim: To implement a genetic algorithm for optimizing the weights of a linear classifier on the MNIST dataset, thereby evolving the parameters for improved model performance.

Theory:

Genetic algorithms (GAs) simulate natural selection to optimize model parameters without relying on gradient-based methods. In this experiment, GAs are applied to train a linear classifier on the MNIST dataset, a collection of grayscale images of handwritten digits. Each *Individual* in the population represents a weight matrix, and its fitness is determined by the model's classification loss on a batch of training data.The process begins with generating an initial population of random weight matrices. Parents are selected based on their fitness, and offspring are created through recombination, inheriting traits from both parents. Mutation introduces random changes in the offspring's weights to maintain genetic diversity and prevent premature convergence. Over successive generations, the population evolves to minimize the loss, improving the classifier's performance.

The experiment highlights the flexibility of GAs, especially in scenarios where traditional gradient-based methods might face challenges. Fitness progression over generations demonstrates the algorithm's effectiveness in converging toward optimal weight configurations for the classifier.

A comparison with gradient-descent methods, such as the Adam optimizer, underscores the strengths and trade-offs of evolutionary and gradient-based techniques. While gradient descent typically converges faster, GAs provide a robust alternative for non-differentiable problems or for exploring unconventional solution spaces.

This experiment underscores the potential of evolutionary computation in solving practical machine-learning problems and lays a foundation for exploring advanced optimization and learning algorithms.

Experiment-6

Aim : Implement MNIST genetic algorithm

```
import os
import torch
import torchvision.datasets as dset
from torch.distributions import Bernoulli
import torchvision.transforms as transforms
import numpy as np
import random
from matplotlib import pyplot as plt
from scipy.stats import halfnorm
```

Setup a directory to store the MNIST dataset/

```
root = './data'
if not os.path.exists(root):
    os.mkdir(root)
```

Setup a transformer to normalize the data.

```
trans = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (1.0,))])
```

```
train_set = dset.MNIST(root=root, train=True, transform=trans, download=True)
test_set = dset.MNIST(root=root, train=False, transform=trans, download=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|| 9.91M/9.91M [00:00<00:00, 15.0MB/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|| 28.9k/28.9k [00:00<00:00, 452kB/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|| 1.65M/1.65M [00:00<00:00, 4.20MB/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|| 4.54k/4.54k [00:00<00:00, 5.53MB/s]Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

```
batch_size = 100

train_loader = torch.utils.data.DataLoader(
                dataset=train_set,
                batch_size=batch_size,
                shuffle=True)
test_loader = torch.utils.data.DataLoader(
```

```
                dataset=test_set,
                batch_size=batch_size,
                shuffle=False)
```

We define a simple linear classifier (or you can think of it as a single layer neural network). It simply multiplies a weight/parameter matrix by the input vector and applies a softmax.

```
x = next(iter(train_loader))[0]
```

```
x = x.reshape(100,784)
```

```
class Individual:
    def __init__(self,param, fitness=0):
        self.param = param
        self.fitness = fitness
```

```
def model(x,W):
    return torch.nn.Softmax()(x @ W)
```

```
model(x,torch.rand(784,10))
```

```
def spawn_population(param_size=(784,10),pop_size=1000):
    return [Individual(torch.randn(*param_size)) for i in range(pop_size)]


loss_fn = torch.nn.CrossEntropyLoss()


random.randint(0,10)
```

```
0
```

```
def evaluate_population(pop):
    avg_fit = 0 #avg population fitness
    for individual in pop:
        x,y = next(iter(train_loader))
        pred = model(x.reshape(batch_size,784),individual.param)
        loss = loss_fn(pred,y)
        fit = loss
        individual.fitness = 1.0 / fit
        avg_fit += fit
    avg_fit = avg_fit / len(pop)
    return pop, avg_fit


def recombine(x1,x2): #x1,x2 : Individual
    w1 = x1.param.view(-1) #flatten
    w2 = x2.param.view(-1)
    cross_pt = random.randint(0,w1.shape[0])
    child1 = torch.zeros(w1.shape)
    child2 = torch.zeros(w1.shape)
    child1[0:cross_pt] = w1[0:cross_pt]
    child1[cross_pt:] = w2[cross_pt:]
    child2[0:cross_pt] = w2[0:cross_pt]
    child2[cross_pt:] = w1[cross_pt:]
    child1 = child1.reshape(784,10)
    child2 = child2.reshape(784,10)
    c1 = Individual(child1)
    c2 = Individual(child2)
    return [c1,c2]


def mutate(pop, mut_rate=0.01):
    param_shape = pop[0].param.shape
    l = torch.zeros(*param_shape)
    l[:] = mut_rate
    m = Bernoulli(l)
    for individual in pop:
        mut_vector = m.sample() * torch.randn(*param_shape)
        individual.param = mut_vector + individual.param
    return pop
```

```
def seed_next_population(pop,pop_size=1000, mut_rate=0.01):
    new_pop = []
    while len(new_pop) < pop_size: #until new pop is full
        parents = random.choices(pop,k=2, weights=[x.fitness for x in pop])
        offspring = recombine(parents[0],parents[1])
        new_pop.extend(offspring)
    new_pop = mutate(new_pop,mut_rate)
    return new_pop


pop = spawn_population()


%%time
pop, avg_fit = evaluate_population(pop)
```

```
CPU times: user 23.8 s, sys: 59.5 ms, total: 23.9 s
Wall time: 24.2 s
```

```
new_pop = seed_next_population(pop)


len(new_pop)
```

```
1000
```

Now we need to spawn a population of weight matrices, run the model using the different individuals, calculate the loss for each one, and then breed the ones with the highest fitness score (lowest loss).

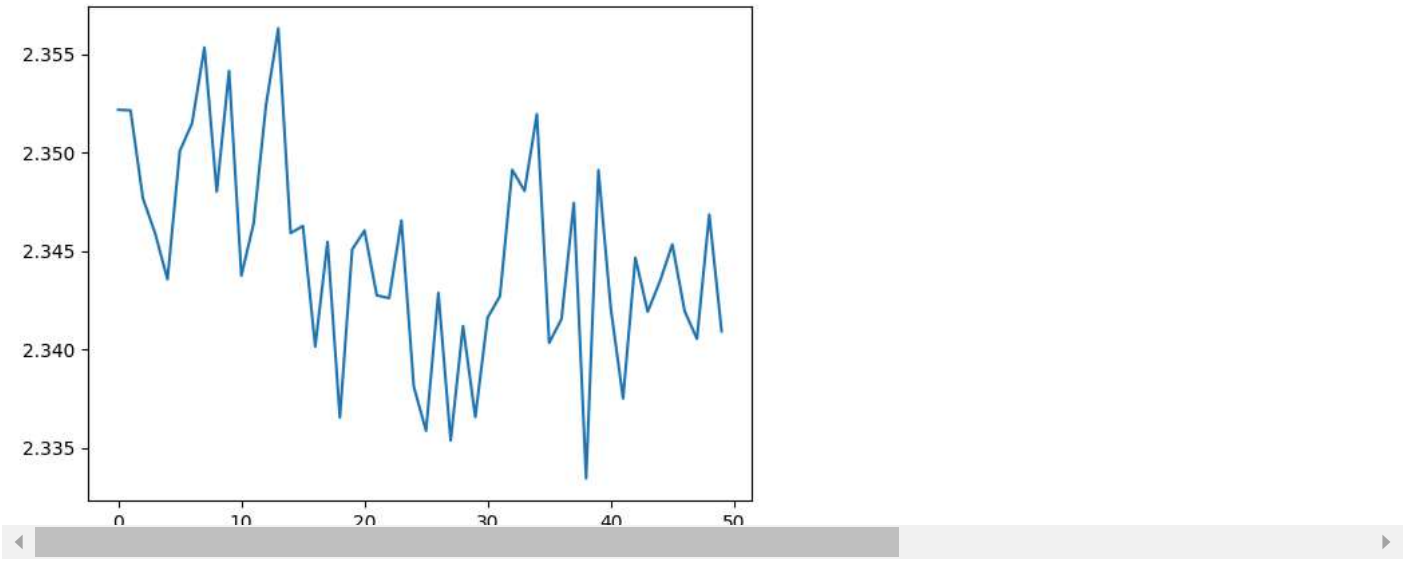
```
num_generations = 50
population_size = 100
mutation_rate = 0.001 # 1% mutation rate per generation
```

Main Evolution (Training) Loop

```
pop_fit = []
pop = spawn_population(pop_size=population_size) #initial population
for gen in range(num_generations):
    # trainning
    pop, avg_fit = evaluate_population(pop)
    pop_fit.append(avg_fit) #record population average fitness
    new_pop = seed_next_population(pop, pop_size=population_size, mut_rate=mutation_rate)
    pop = new_pop
```

```
plt.plot(pop_fit)
```

```
[<matplotlib.lines.Line2D at 0x7e931491bf10>]
```




```
avg_loss = 0
for i in range(len(pop)):
    x,y = next(iter(train_loader))
    pred = model(x.reshape(batch_size,784),pop[i].param)
    loss = loss_fn(pred,y)
    avg_loss += loss
avg_loss /= len(pop)
print(avg_loss)
```

```
tensor(2.3425)
```

Avg Loss new pop: 2.3336 Avg Loss after 10 gens: 2.3435

Train with gradient-descent (comparison)

```
p = torch.randn(784,10, requires_grad=True)
optim = torch.optim.Adam(lr=0.1, params=[p])
```

```
loss_list = []
for i in range(50):
    for x, y in train_loader:
        optim.zero_grad()
```

```
            pred = model(x.reshape(batch_size, 784), p)
            loss = loss_fn(pred, y)
            loss_list.append(loss.detach().numpy())  # Detach and convert to NumPy
            loss.backward()
            optim.step()
        print(loss)

    plt.plot(loss_list)
    plt.show()
```

```
tensor(2.0408, grad_fn=<NllLossBackward0>)
tensor(2.2201, grad_fn=<NllLossBackward0>)
```

Start coding or generate with AI.

```
tensor(2.1704, grad_fn=<NllLossBackward0>)
tensor(2.1811, grad_fn=<NllLossBackward0>)
tensor(2.1509, grad_fn=<NllLossBackward0>)
tensor(2.1310, grad_fn=<NllLossBackward0>)
tensor(2.2012, grad_fn=<NllLossBackward0>)
tensor(2.0410, grad_fn=<NllLossBackward0>)
tensor(2.1712, grad_fn=<NllLossBackward0>)
tensor(2.1010, grad_fn=<NllLossBackward0>)
tensor(2.1209, grad_fn=<NllLossBackward0>)
tensor(2.1910, grad_fn=<NllLossBackward0>)
tensor(2.1007, grad_fn=<NllLossBackward0>)
tensor(2.1811, grad_fn=<NllLossBackward0>)
tensor(2.1212, grad_fn=<NllLossBackward0>)
tensor(2.1908, grad_fn=<NllLossBackward0>)
tensor(2.1711, grad_fn=<NllLossBackward0>)
tensor(2.2411, grad_fn=<NllLossBackward0>)
tensor(2.2412, grad_fn=<NllLossBackward0>)
tensor(2.2012, grad_fn=<NllLossBackward0>)
tensor(2.1407, grad_fn=<NllLossBackward0>)
tensor(2.1705, grad_fn=<NllLossBackward0>)
tensor(2.1911, grad_fn=<NllLossBackward0>)
tensor(2.1711, grad_fn=<NllLossBackward0>)
tensor(2.2208, grad_fn=<NllLossBackward0>)
```

EXPERIMENT 7

Aim: To implement and evaluate a Distributional Deep Q-Network (DQN), a reinforcement learning algorithm that estimates the distribution of returns rather than their expected value, enhancing decision-making under uncertainty.

Theory:

Distributional Deep Q-Network (DQN), a reinforcement learning approach that estimates the probability distribution of returns instead of just their expected value. The experiment began by setting up the environment using the gymnasium library and initializing support distributions over the reward space. These support distributions were divided into a grid ranging from vmin to vmax with 51 support points. The update_dist function was designed to update these distributions based on observed rewards by applying the Bellman update with a discount factor, ensuring that the agent accurately captures the distribution of future rewards.

The core of the experiment involved creating the dist_dqn function, a neural network that maps input states to probability distributions over possible actions. The network architecture included multiple layers with torch.selu activations and a softmax operation to maintain valid probability distributions. The agent interacted with the environment (ALE/Freeway) using an experience replay buffer to store state-action-reward transitions. The training process involved sampling batches from the buffer, calculating target distributions using the get_target_dist function, and optimizing the neural network using the cross-entropy loss function.

Throughout the experiment, an **epsilon-greedy** strategy was implemented to balance exploration and exploitation. The agent initially explored random actions but gradually decreased the exploration rate (epsilon) over time to focus more on exploitation as learning progressed. The loss was tracked over iterations to monitor the network's convergence. This experiment highlighted how Distributional DQN effectively models uncertainty and variability in returns, making the agent's decision-making process more robust and adaptive to complex environments.

Experiment-7

Aim : Implement distributional DQN

```
!pip install gymnasium
```

```
Collecting gymnasium
  Downloading gymnasium-1.0.0-py3-none-any.whl.metadata (9.5 kB)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (3.1.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (4.12.2)
Collecting farama-notifications>=0.0.1 (from gymnasium)
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Downloading gymnasium-1.0.0-py3-none-any.whl (958 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 958.1/958.1 kB 13.3 MB/s eta 0:00:00
Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Installing collected packages: farama-notifications, gymnasium
Successfully installed farama-notifications-0.0.4 gymnasium-1.0.0
```

```
!pip install ale-py
!pip install autorom[accept-rom-license]
```

```
Collecting ale-py
  Downloading ale_py-0.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (7.6 kB)
Requirement already satisfied: numpy>1.20 in /usr/local/lib/python3.10/dist-packages (from ale-py) (1.26.4)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from ale-py) (4.12.2)
Downloading ale_py-0.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 21.6 MB/s eta 0:00:00
Installing collected packages: ale-py
Successfully installed ale-py-0.10.1
Collecting autorom[accept-rom-license]
  Downloading AutoROM-0.6.1-py3-none-any.whl.metadata (2.4 kB)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (8.1.7)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (2.32.3)
Collecting AutoROM.accept-rom-license (from autorom[accept-rom-license])
  Downloading AutoROM.accept-rom-license-0.6.1.tar.gz (434 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 434.7/434.7 kB 7.1 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-li
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (3.1
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]
Downloading AutoROM-0.6.1-py3-none-any.whl (9.4 kB)
Building wheels for collected packages: AutoROM.accept-rom-license
  Building wheel for AutoROM.accept-rom-license (pyproject.toml) ... done
  Created wheel for AutoROM.accept-rom-license: filename=AutoROM.accept_rom_license-0.6.1-py3-none-any.whl size=446667 sha256=92d9f2e686
  Stored in directory: /root/.cache/pip/wheels/6b/1b/ef/a43ff1a2f1736d5711faa1ba4c1f61be1131b8899e6a057811
Successfully built AutoROM.accept-rom-license
Installing collected packages: AutoROM.accept-rom-license, autorom
Successfully installed AutoROM.accept-rom-license-0.6.1 autorom-0.6.1
```

```
import numpy as np
probs = np.array([0.6, 0.1, 0.1, 0.1, 0.1])
outcomes = np.array([18, 21, 17, 17, 21])
expected_value = 0.0
for i in range(probs.shape[0]):
    expected_value += probs[i] * outcomes[i]

print(expected_value)
```

```
18.4
```

```
expected_value = probs @ outcomes
print(expected_value)
```

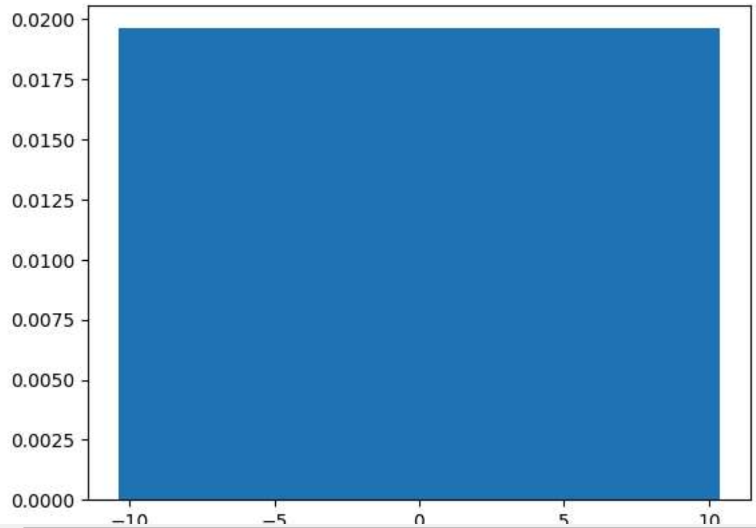
```
18.4
```

```
t0 = 18.4
T = lambda: t0 + np.random.randn(1)
T()
```

```
array([17.95607589])
```

```
import torch
import numpy as np
from matplotlib import pyplot as plt

vmin,vmax = -10.,10.
nsup=51
support = np.linspace(vmin,vmax,nsup)
probs = np.ones(nsup)
probs /= probs.sum()
z3 = torch.from_numpy(probs).float()
plt.bar(support,probs)
```

```
<BarContainer object of 51 artists>
```



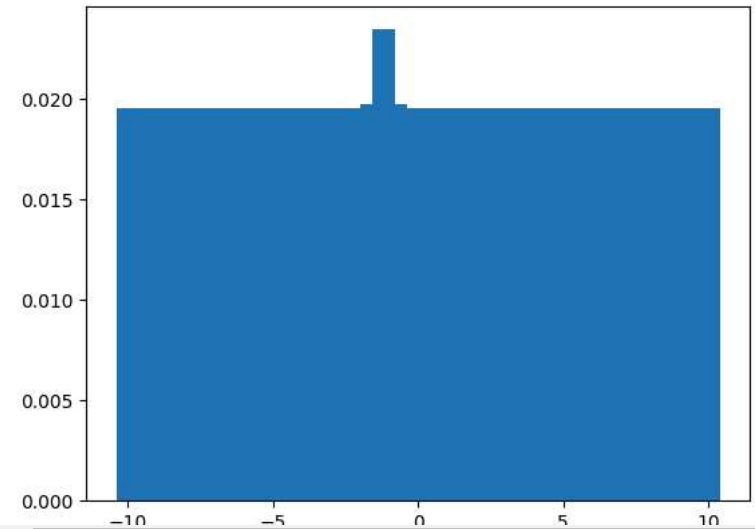

```
def update_dist(r,support,probs,lim=(-10.,10.),gamma=0.8):
    nsup = probs.shape[0]
    vmin,vmax = lim[0],lim[1]
    dz = (vmax-vmin)/(nsup-1.)
    bj = np.round((r-vmin)/dz)
    bj = int(np.clip(bj,0,nsup-1))
    m = probs.clone()
    j = 1
    for i in range(bj,1,-1):
        m[i] += np.power(gamma,j) * m[i-1]
        j += 1
    j = 1
    for i in range(bj,nsup-1,1):
        m[i] += np.power(gamma,j) * m[i+1]
        j += 1
    m /= m.sum()
    return m
```

```
probs
```

```
array([0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784, 0.01960784, 0.01960784, 0.01960784, 0.01960784,
       0.01960784])
```

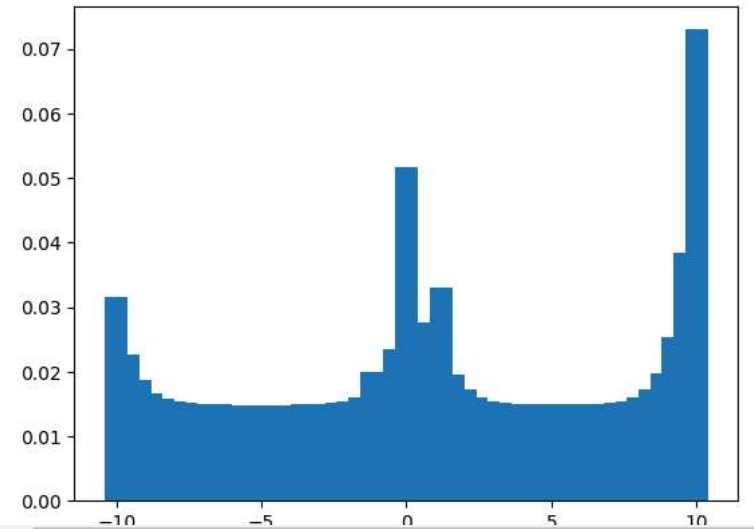
```
ob_reward = -1
Z = torch.from_numpy(probs).float()
Z = update_dist(ob_reward,torch.from_numpy(support).float(),Z,lim=(vmin,vmax),gamma=0.1)
plt.bar(support,Z)
```

```
<BarContainer object of 51 artists>
```



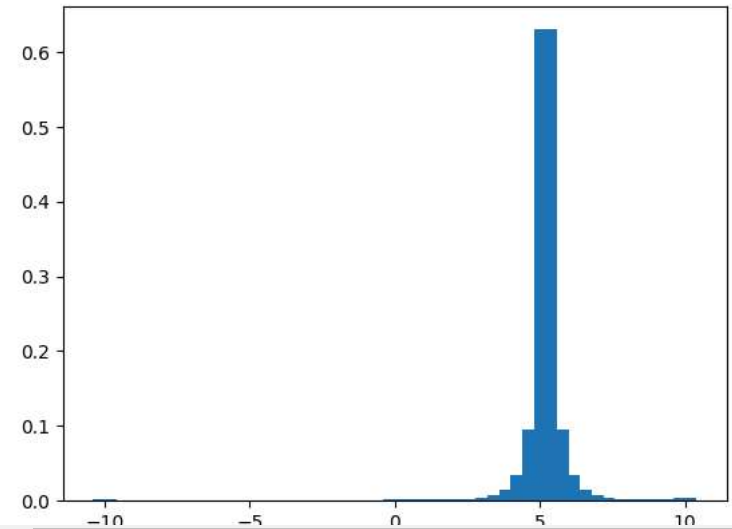

```
ob_rewards = [10,10,10,0,1,0,-10,-10,10,10]
for i in range(len(ob_rewards)):
    Z = update_dist(ob_rewards[i], torch.from_numpy(support).float(), Z, lim=(vmin,vmax), gamma=0.5)
plt.bar(support, Z)
```

```
<BarContainer object of 51 artists>
```




```
ob_rewards = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
for i in range(len(ob_rewards)):
    Z = update_dist(ob_rewards[i], torch.from_numpy(support).float(), \
    Z, lim=(vmin,vmax), gamma=0.7)
plt.bar(support, Z)
```

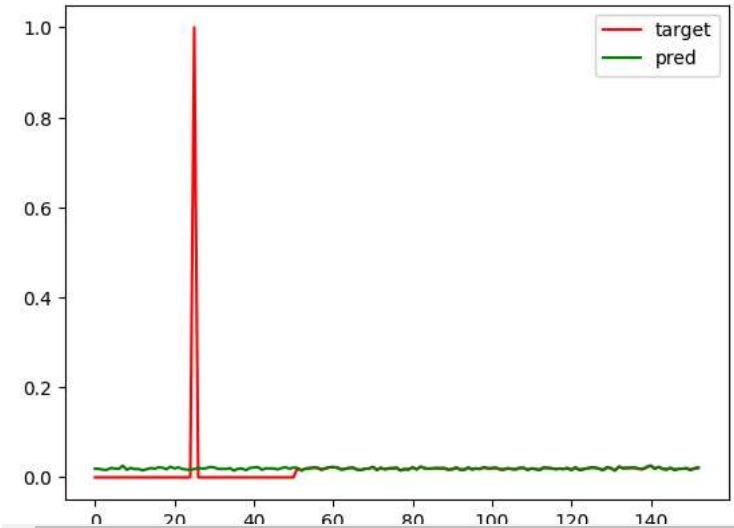
```
<BarContainer object of 51 artists>
```




```
def dist_dqn(x,theta,aspace=3):
    dim0,dim1,dim2,dim3 = 128,100,25,51
    t1 = dim0*dim1
    t2 = dim2*dim1
    theta1 = theta[0:t1].reshape(dim0,dim1)
    theta2 = theta[t1:t1 + t2].reshape(dim1,dim2)
    l1 = x @ theta1 #D
    l1 = torch.selu(l1)
    l2 = l1 @ theta2 #E
    l2 = torch.selu(l2)
    l3 = []
    for i in range(aspace):
        step = dim2*dim3
        theta5_dim = t1 + t2 + i * step
        theta5 = theta[theta5_dim:theta5_dim+step].reshape(dim2,dim3)
        l3_ = l2 @ theta5
        l3.append(l3_)
    l3 = torch.stack(l3,dim=1)
    l3 = torch.nn.functional.softmax(l3,dim=2)
    return l3.squeeze()


def get_target_dist(dist_batch,action_batch,reward_batch,support,lim=(-10,10),gamma=0.8):
    nsup = support.shape[0]
    vmin,vmax = lim[0],lim[1]
    dz = (vmax-vmin)/(nsup-1.)
    target_dist_batch = dist_batch.clone()
    for i in range(dist_batch.shape[0]):
        dist_full = dist_batch[i]
        action = int(action_batch[i].item())
        dist = dist_full[action]
        r = reward_batch[i]
        if r != -1:
            target_dist = torch.zeros(nsup)
            bj = np.round((r-vmin)/dz)
            bj = int(np.clip(bj,0,nsup-1))
            target_dist[bj] = 1.
        else:
            target_dist = update_dist(r,support,dist,lim=lim,gamma=gamma)
        target_dist_batch[i,action,:] = target_dist

    return target_dist_batch


def lossfn(x,y):#A
    loss = torch.Tensor([0.])
    loss.requires_grad=True
    for i in range(x.shape[0]):
        loss_ = -1 *  torch.log(x[i].flatten(start_dim=0)) @ y[i].flatten(start_dim=0)
        loss = loss + loss_
    return loss
```

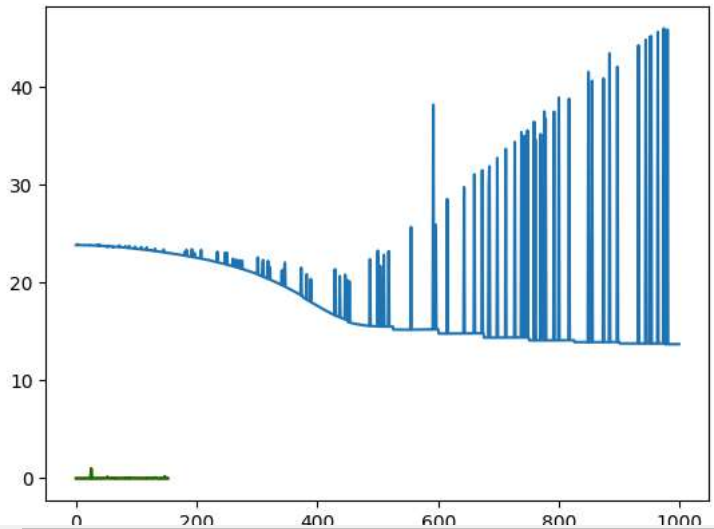
```
aspace = 3 #A
tot_params = 128*100 + 25*100 + aspace*25*51
theta = torch.randn(tot_params)/10.
theta.requires_grad=True
theta_2 = theta.detach().clone()

vmin,vmax= -10,10
gamma=0.9
lr = 0.00001
update_rate = 75
support = torch.linspace(-10,10,51)
state = torch.randn(2,128)/10.
action_batch = torch.Tensor([0,2])
reward_batch = torch.Tensor([0,10])
losses = []
pred_batch = dist_dqn(state,theta,aspace=aspace)
target_dist = get_target_dist(pred_batch,action_batch,reward_batch, \
                              support, lim=(vmin,vmax),gamma=gamma)

plt.plot((target_dist.flatten(start_dim=1)[0].data.numpy()),color='red',label='target')
plt.plot((pred_batch.flatten(start_dim=1)[0].data.numpy()),color='green',label='pred')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7b0d26ae81c0>
```

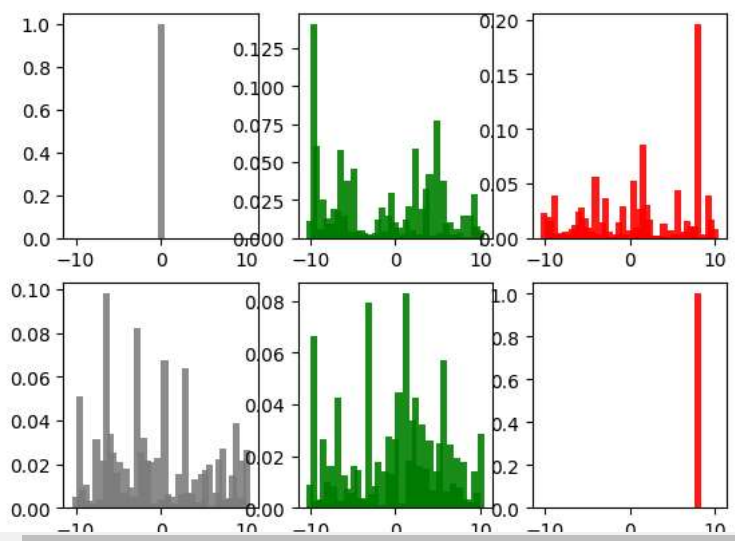



```
for i in range(1000):
    reward_batch = torch.Tensor([0,8]) + torch.randn(2)/10.0
    pred_batch = dist_dqn(state,theta,aspace=aspace)
    pred_batch2 = dist_dqn(state,theta_2,aspace=aspace)
    target_dist = get_target_dist(pred_batch2,action_batch,reward_batch, \
                                  support, lim=(vmin,vmax),gamma=gamma)
    loss = lossfn(pred_batch,target_dist.detach())
    losses.append(loss.item())
    loss.backward()

    with torch.no_grad():
        theta -= lr * theta.grad
    theta.requires_grad = True

    if i % update_rate == 0:
        theta_2 = theta.detach().clone()

plt.plot((target_dist.flatten(start_dim=1)[0].data.numpy()),color='red',label='target')
plt.plot((pred_batch.flatten(start_dim=1)[0].data.numpy()),color='green',label='pred')
plt.plot(losses)
```

```
[<matplotlib.lines.Line2D at 0x7b0d25b067d0>]
```




```
tpred = pred_batch
cs = ['gray','green','red']
num_batch = 2
labels = ['Action {}'.format(i,) for i in range(aspace)]
fig,ax = plt.subplots(nrows=num_batch,ncols=aspace)

for j in range(num_batch):
    for i in range(tpred.shape[1]):
        ax[j,i].bar(support.data.numpy(),tpred[j,i,:].data.numpy(),\
                label='Action {}'.format(i),alpha=0.9,color=cs[i])
```

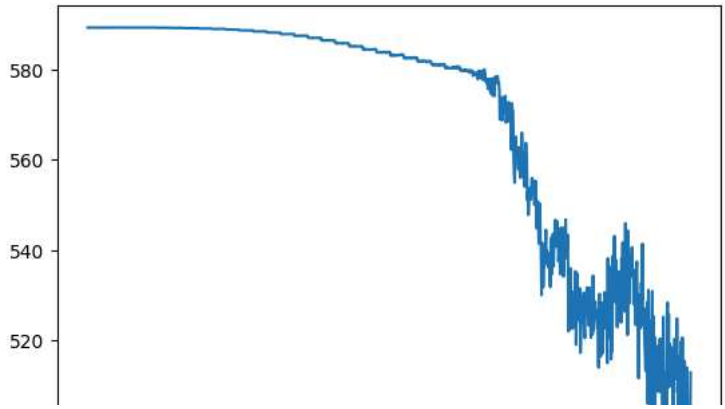



```
def preproc_state(state):
    p_state = torch.from_numpy(state).unsqueeze(dim=0).float()
    p_state = torch.nn.functional.normalize(p_state,dim=1)
    return p_state

def get_action(dist,support):
    actions = []
    for b in range(dist.shape[0]):
        expectations = [support @ dist[b,a,:] for a in range(dist.shape[1])]
        action = int(np.argmax(expectations))
        actions.append(action)
    actions = torch.Tensor(actions).int()
    return actions


import gymnasium as gym
import ale_py
from collections import deque
```

```
env = gym.make('ALE/Freeway-ram-v5')
aspace = 3
env.unwrapped.get_action_meanings()


vmin,vmax = -10,10
replay_size = 200
batch_size = 50
nsup = 51
dz = (vmax - vmin) / (nsup-1)
support = torch.linspace(vmin,vmax,nsup)

replay = deque(maxlen=replay_size)
lr = 0.0001
gamma = 0.1
epochs = 1300
eps = 0.20
eps_min = 0.05
priority_level = 5
update_freq = 25


tot_params = 128*100 + 25*100 + aspace*25*51
theta = torch.randn(tot_params)/10.
theta.requires_grad=True
theta_2 = theta.detach().clone()

losses = []
cum_rewards = []
renders = []
state = preproc_state(env.reset()[0])


from random import shuffle
for i in range(epochs):
    pred = dist_dqn(state,theta,aspace=aspace)
    if i < replay_size or np.random.rand(1) < eps:
        action = np.random.randint(aspace)
    else:
        action = get_action(pred.unsqueeze(dim=0).detach(),support).item()
    state2, reward, done, info, _ = env.step(action)
    state2 = preproc_state(state2)
    if reward == 1: cum_rewards.append(1)
    reward = 10 if reward == 1 else reward
    reward = -10 if done else reward
    reward = -1 if reward == 0 else reward
    exp = (state,action,reward,state2)
    replay.append(exp)

    if reward == 10:
        for e in range(priority_level):
            replay.append(exp)

    shuffle(replay)
    state = state2

    if len(replay) == replay_size:
        indx = np.random.randint(low=0,high=len(replay),size=batch_size)
        exps = [replay[j] for j in indx]
        state_batch = torch.stack([ex[0] for ex in exps],dim=1).squeeze()
        action_batch = torch.Tensor([ex[1] for ex in exps])
        reward_batch = torch.Tensor([ex[2] for ex in exps])
        state2_batch = torch.stack([ex[3] for ex in exps],dim=1).squeeze()
        pred_batch = dist_dqn(state_batch.detach(),theta,aspace=aspace)
        pred2_batch = dist_dqn(state2_batch.detach(),theta_2,aspace=aspace)
        target_dist = get_target_dist(pred2_batch,action_batch,reward_batch, \
                                      support, lim=(vmin,vmax),gamma=gamma)
        loss = lossfn(pred_batch,target_dist.detach())
        losses.append(loss.item())
        loss.backward()
        with torch.no_grad():
            theta -= lr * theta.grad
        theta.requires_grad = True

    if i % update_freq == 0:
        theta_2 = theta.detach().clone()
```

```
        if i > 100 and eps > eps_min:
            dec = 1./np.log2(i)
            dec /= 1e3
            eps -= dec

        if done:
            state = preproc_state(env.reset())
            done = False


plt.plot(losses)
```

```
[<matplotlib.lines.Line2D at 0x7b0d245f90f0>]
```



EXPERIMENT 8

Aim: To implement and evaluate an Intrinsic Curiosity Module (ICM) in the Super Mario Bros environment, enhancing exploration by combining intrinsic curiosity-driven rewards with extrinsic game rewards.

Theory:

The Intrinsic Curiosity Module (ICM) is a reinforcement learning approach that encourages an agent to explore unfamiliar states by generating intrinsic rewards based on prediction errors of state transitions. In this experiment, the environment was set up using the gym_super_mario_bros library, and observations were processed by downscaling game frames to focus on essential visual features.

The ICM architecture includes an encoder for processing game frames, an inverse model (Gnet) to predict actions, and a forward model (Fnet) to predict next states. The intrinsic curiosity reward is calculated as the error between predicted and actual state transitions. These intrinsic rewards are then combined with extrinsic rewards from the game environment to guide the agent's exploration.

During training, an epsilon-greedy policy balanced exploration and exploitation, starting with random actions and gradually shifting to known strategies. The agent's memory was managed using an experience replay buffer, which stored state-action-reward transitions. This setup allowed the agent to effectively explore new states and transitions, ultimately demonstrating how curiosity-driven reinforcement learning can improve state-space coverage and decision-making in dynamic environments like Super Mario Bros.

Experiment 8

Aim : Intrinsic Curiosity Module for Super Mario Bros game

```
!pip install gym==0.26.2 gym_super_mario_bros==7.3.0 nes-py==8.2.1 matplotlib torch scikit-image
```

```
Collecting gym==0.26.2
  Downloading gym-0.26.2.tar.gz (721 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 721.7/721.7 kB 8.8 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting gym_super_mario_bros==7.3.0
  Downloading gym_super_mario_bros-7.3.0-py2.py3-none-any.whl.metadata (9.4 kB)
Collecting nes-py==8.2.1
  Downloading nes_py-8.2.1.tar.gz (77 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 77.7/77.7 kB 8.5 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.5.1+cu121)
Requirement already satisfied: scikit-image in /usr/local/lib/python3.10/dist-packages (0.24.0)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym==0.26.2) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym==0.26.2) (3.1.0)
Requirement already satisfied: gym_notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym==0.26.2) (0.0.8)
Collecting pyglet<=1.5.21,>=1.4.0 (from nes-py==8.2.1)
  Downloading pyglet-1.5.21-py3-none-any.whl.metadata (7.6 kB)
Requirement already satisfied: tqdm>=4.48.2 in /usr/local/lib/python3.10/dist-packages (from nes-py==8.2.1) (4.66.6)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.55.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (11.0.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2024.10.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch) (1.
Requirement already satisfied: scipy>=1.9 in /usr/local/lib/python3.10/dist-packages (from scikit-image) (1.13.1)
Requirement already satisfied: imageio>=2.33 in /usr/local/lib/python3.10/dist-packages (from scikit-image) (2.36.1)
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.10/dist-packages (from scikit-image) (2024.9.20
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.10/dist-packages (from scikit-image) (0.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib) (
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (3.0.2)
Downloading gym_super_mario_bros-7.3.0-py2.py3-none-any.whl (198 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 198.6/198.6 kB 20.2 MB/s eta 0:00:00
Downloading pyglet-1.5.21-py3-none-any.whl (1.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 40.4 MB/s eta 0:00:00
Building wheels for collected packages: gym, nes-py
  Building wheel for gym (pyproject.toml) ... done
  Created wheel for gym: filename=gym-0.26.2-py3-none-any.whl size=827626 sha256=6050a9398ec2e88ede55ffdfcf47aa7726bd3333e69
  Stored in directory: /root/.cache/pip/wheels/b9/22/6d/3e7b32d98451b4cd9d12417052affbeeeea012955d437da1da
  Building wheel for nes-py (setup.py) ... done
  Created wheel for nes-py: filename=nes_py-8.2.1-cp310-cp310-linux_x86_64.whl size=535719 sha256=a59e7b5da156cbc85c12640c49
  Stored in directory: /root/.cache/pip/wheels/34/a7/d5/9aa14b15df740a53d41f702e4c795731b6c4da7925deb8476c
Successfully built gym nes-py
Installing collected packages: pyglet, gym, nes-py, gym_super_mario_bros
  Attempting uninstall: gym
    Found existing installation: gym 0.25.2
```

```
# Import libraries
from nes_py.wrappers import JoypadSpace
import gym_super_mario_bros
from gym_super_mario_bros.actions import COMPLEX_MOVEMENT
import matplotlib.pyplot as plt
from skimage.transform import resize
import numpy as np
import torch
```

```
from torch import nn, optim
import torch.nn.functional as F
from collections import deque
from random import shuffle

# Initialize the environment
env = gym_super_mario_bros.make('SuperMarioBros-v3', apply_api_compatibility=True, render_mode="rgb_array")
env = JoypadSpace(env, COMPLEX_MOVEMENT)
```

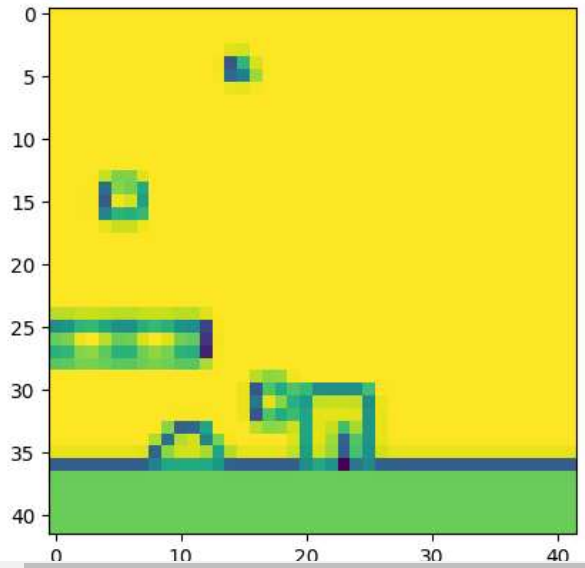
```
/usr/local/lib/python3.10/dist-packages/gym/envs/registration.py:627: UserWarning: WARN: The environment creator metadata doesn
  logger.warn(
```

```
done = True
for step in range(2500): #D
    if done:
        state = env.reset()
    state, reward, done, trunc, info = env.step(env.action_space.sample())
    env.render()
#env.close()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:233: DeprecationWarning: `np.bool8` is a deprecated al
  if not isinstance(terminated, (bool, np.bool8)):
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:272: UserWarning: WARN: No render modes was declared i
  logger.warn(
/usr/local/lib/python3.10/dist-packages/gym_super_mario_bros/smb_env.py:148: RuntimeWarning: overflow encountered in scalar sub
  return (self.ram[0x86] - self.ram[0x071c]) % 256
```

```
import matplotlib.pyplot as plt
from skimage.transform import resize #A
import numpy as np

def downscale_obs(obs, new_size=(42,42), to_gray=True):
    if to_gray:
        return resize(obs, new_size, anti_aliasing=True).max(axis=2) #B
    else:
        return resize(obs, new_size, anti_aliasing=True)


plt.imshow(env.render())
plt.imshow(downscale_obs(env.render()))
```

```
<matplotlib.image.AxesImage at 0x7c4b3cab5a50>
```




```
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
```

```
from collections import deque

def prepare_state(state): #A
    return torch.from_numpy(downscale_obs(state, to_gray=True)).float().unsqueeze(dim=0)


def prepare_multi_state(state1, state2): #B
    state1 = state1.clone()
    tmp = torch.from_numpy(downscale_obs(state2, to_gray=True)).float()
    state1[0][0] = state1[0][1]
    state1[0][1] = state1[0][2]
    state1[0][2] = tmp
    return state1


def prepare_initial_state(state,N=3): #C
    state_ = torch.from_numpy(downscale_obs(state, to_gray=True)).float()
    tmp = state_.repeat((N,1,1))
    return tmp.unsqueeze(dim=0)


def policy(qvalues, eps=None): #A
    if eps is not None:
        if torch.rand(1) < eps:
            return torch.randint(low=0,high=7,size=(1,))
        else:
            return torch.argmax(qvalues)
    else:
        return torch.multinomial(F.softmax(F.normalize(qvalues)), num_samples=1) #B


from random import shuffle
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F

class ExperienceReplay:
    def __init__(self, N=500, batch_size=100):
        self.N = N #A
        self.batch_size = batch_size #B
        self.memory = []
        self.counter = 0

    def add_memory(self, state1, action, reward, state2):
        self.counter +=1
        if self.counter % 500 == 0: #C
            self.shuffle_memory()

        if len(self.memory) < self.N: #D
            self.memory.append( (state1, action, reward, state2) )
        else:
            rand_index = np.random.randint(0,self.N-1)
            self.memory[rand_index] = (state1, action, reward, state2)

    def shuffle_memory(self): #E
        shuffle(self.memory)

    def get_batch(self): #F
        if len(self.memory) < self.batch_size:
            batch_size = len(self.memory)
        else:
            batch_size = self.batch_size
        if len(self.memory) < 1:
            print("Error: No data in memory.")
            return None
        #G
        ind = np.random.choice(np.arange(len(self.memory)),batch_size,replace=False)
        batch = [self.memory[i] for i in ind] #batch is a list of tuples
        state1_batch = torch.stack([x[0].squeeze(dim=0) for x in batch],dim=0)
        action_batch = torch.Tensor([x[1] for x in batch]).long()
        reward_batch = torch.Tensor([x[2] for x in batch])
```

```
        state2_batch = torch.stack([x[3].squeeze(dim=0) for x in batch],dim=0)
        return state1_batch, action_batch, reward_batch, state2_batch


class Phi(nn.Module): #A
    def __init__(self):
        super(Phi, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv4 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)

    def forward(self,x):
        x = F.normalize(x)
        y = F.elu(self.conv1(x))
        y = F.elu(self.conv2(y))
        y = F.elu(self.conv3(y))
        y = F.elu(self.conv4(y)) #size [1, 32, 3, 3] batch, channels, 3 x 3
        y = y.flatten(start_dim=1) #size N, 288
        return y

class Gnet(nn.Module): #B
    def __init__(self):
        super(Gnet, self).__init__()
        self.linear1 = nn.Linear(576,256)
        self.linear2 = nn.Linear(256,12)

    def forward(self, state1,state2):
        x = torch.cat( (state1, state2) ,dim=1)
        y = F.relu(self.linear1(x))
        y = self.linear2(y)
        y = F.softmax(y,dim=1)
        return y

class Fnet(nn.Module): #C
    def __init__(self):
        super(Fnet, self).__init__()
        self.linear1 = nn.Linear(300,256)
        self.linear2 = nn.Linear(256,288)

    def forward(self,state,action):
        action_ = torch.zeros(action.shape[0],12) #D
        indices = torch.stack( (torch.arange(action.shape[0]), action.squeeze()), dim=0)
        indices = indices.tolist()
        action_[indices] = 1.
        x = torch.cat( (state,action_) ,dim=1)
        y = F.relu(self.linear1(x))
        y = self.linear2(y)
        return y


class Qnetwork(nn.Module):
    def __init__(self):
        super(Qnetwork, self).__init__()
        #in_channels, out_channels, kernel_size, stride=1, padding=0
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3,3), stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv4 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.linear1 = nn.Linear(288,100)
        self.linear2 = nn.Linear(100,12)

    def forward(self,x):
        x = F.normalize(x)
        y = F.elu(self.conv1(x))
        y = F.elu(self.conv2(y))
        y = F.elu(self.conv3(y))
        y = F.elu(self.conv4(y))
        y = y.flatten(start_dim=2)
        y = y.view(y.shape[0], -1, 32)
        y = y.flatten(start_dim=1)
        y = F.elu(self.linear1(y))
        y = self.linear2(y) #size N, 12
```

```
        return y


params = {
    'batch_size':150,
    'beta':0.2,
    'lambda':0.1,
    'eta': 1.0,
    'gamma':0.2,
    'max_episode_len':100,
    'min_progress':15,
    'action_repeats':6,
    'frames_per_state':3
}

replay = ExperienceReplay(N=1000, batch_size=params['batch_size'])
Qmodel = Qnetwork()
encoder = Phi()
forward_model = Fnet()
inverse_model = Gnet()
forward_loss = nn.MSELoss(reduction='none')
inverse_loss = nn.CrossEntropyLoss(reduction='none')
qloss = nn.MSELoss()
all_model_params = list(Qmodel.parameters()) + list(encoder.parameters()) #A
all_model_params += list(forward_model.parameters()) + list(inverse_model.parameters())
opt = optim.Adam(lr=0.001, params=all_model_params)


def loss_fn(q_loss, inverse_loss, forward_loss):
    loss_ = (1 - params['beta']) * inverse_loss
    loss_ += params['beta'] * forward_loss
    loss_ = loss_.sum() / loss_.flatten().shape[0]
    loss = loss_ + params['lambda'] * q_loss
    return loss

def reset_env():
    """
    Reset the environment and return a new initial state
    """
    env.reset()
    state1 = prepare_initial_state(env.render())
    return state1


def ICM(state1, action, state2, forward_scale=1., inverse_scale=1e4):
    state1_hat = encoder(state1) #A
    state2_hat = encoder(state2)
    state2_hat_pred = forward_model(state1_hat.detach(), action.detach()) #B
    forward_pred_err = forward_scale * forward_loss(state2_hat_pred, \
                        state2_hat.detach()).sum(dim=1).unsqueeze(dim=1)
    pred_action = inverse_model(state1_hat, state2_hat) #C
    inverse_pred_err = inverse_scale * inverse_loss(pred_action, \
                                        action.detach().flatten()).unsqueeze(dim=1)
    return forward_pred_err, inverse_pred_err


def minibatch_train(use_extrinsic=True):
    state1_batch, action_batch, reward_batch, state2_batch = replay.get_batch()
    action_batch = action_batch.view(action_batch.shape[0],1) #A
    reward_batch = reward_batch.view(reward_batch.shape[0],1)

    forward_pred_err, inverse_pred_err = ICM(state1_batch, action_batch, state2_batch) #B
    i_reward = (1. / params['eta']) * forward_pred_err #C
    reward = i_reward.detach() #D
    if use_explicit: #E
        reward += reward_batch
    qvals = Qmodel(state2_batch) #F
    reward += params['gamma'] * torch.max(qvals)
    reward_pred = Qmodel(state1_batch)
    reward_target = reward_pred.clone()
    indices = torch.stack( (torch.arange(action_batch.shape[0]), \
    action_batch.squeeze()), dim=0)
    indices = indices.tolist()
    reward_target[indices] = reward.squeeze()
```

```
    q_loss = 1e5 * qloss(F.normalize(reward_pred), F.normalize(reward_target.detach()))
    return forward_pred_err, inverse_pred_err, q_loss


epochs = 5000
env.reset()
state1 = prepare_initial_state(env.render())
eps=0.15
losses = []
episode_length = 0
switch_to_eps_greedy = 1000
state_deque = deque(maxlen=params['frames_per_state'])
e_reward = 0.
last_x_pos = 0 #A
#ep_lengths = []
use_explicit = False
for i in range(epochs):
    opt.zero_grad()
    episode_length += 1
    q_val_pred = Qmodel(state1) #B
    if i > switch_to_eps_greedy: #C
        action = int(policy(q_val_pred,eps))
    else:
        action = int(policy(q_val_pred))
    for j in range(params['action_repeats']): #D
        state2, e_reward_, done, trunc, info = env.step(action)
        last_x_pos = info['x_pos']
        if done:
            state1 = reset_env()
            break
        e_reward += e_reward_
        state_deque.append(prepare_state(state2))
    state2 = torch.stack(list(state_deque),dim=1) #E
    replay.add_memory(state1, action, e_reward, state2) #F
    e_reward = 0
    if episode_length > params['max_episode_len']: #G
        if (info['x_pos'] - last_x_pos) < params['min_progress']:
            done = True
        else:
            last_x_pos = info['x_pos']
    if done or trunc:
        #ep_lengths.append(info['x_pos'])
        state1 = reset_env()
        last_x_pos = 0
        episode_length = 0
    else:
        state1 = state2
    if len(replay.memory) < params['batch_size']:
        continue
    forward_pred_err, inverse_pred_err, q_loss = minibatch_train(use_extrinsic=False) #H
    loss = loss_fn(q_loss, forward_pred_err, inverse_pred_err) #I
    loss_list = (q_loss.mean(), forward_pred_err.flatten().mean(),\
    inverse_pred_err.flatten().mean())
    losses.append(loss_list)
    loss.backward()
    opt.step()
```

```
<ipython-input-7-8acb8fb27d16>:8: UserWarning: Implicit dimension choice for softmax has been deprecated. Change the call to in
  return torch.multinomial(F.softmax(F.normalize(qvalues)), num_samples=1) #B
/usr/local/lib/python3.10/dist-packages/gym_super_mario_bros/smb_env.py:148: RuntimeWarning: overflow encountered in scalar sub
  return (self.ram[0x86] - self.ram[0x071c]) % 256
```

```
done = True
state_deque = deque(maxlen=params['frames_per_state'])
for step in range(500):
    if done:
        env.reset()
        state1 = prepare_initial_state(env.render())
    q_val_pred = Qmodel(state1)
    action = int(policy(q_val_pred,eps))
    state2, reward, done, trunc, info = env.step(action)
```

```
        state2 = prepare_multi_state(state1,state2)
        state1=state2
        env.render()
```

Start coding or generate with AI.