SOC 2025

END-TERM REPORT

RIDDHI YEOLA

MENTOR NAME: SACHIN MEENA

PROJECT ID: 127

SyanptiRL:

Meta- reinforcement learning with adaptive spiking neural networks on neuromorphic simulators.

Week 1: foundations and setup

Spiking neural networks (SNN)

What are Spiking Neural Networks?

Spiking Neural Networks are a class of *artificial neural networks* that mimic the behaviour of biological neurons more closely than traditional neural networks. In SNNs, neurons communicate by sending discrete spikes, which represent changes in voltage across a neuron's membrane.

SNNs operate on discrete events called "spikes."

Key Concepts in Spiking Neural Networks

1. Neurons and Spikes

In SNNs, each neuron emits spikes based on its membrane potential

When the membrane potential reaches a certain threshold, the neuron "fires" and emits a spike.

2. Temporal Coding

SNNs use temporal coding, where the timing of spikes carries information.

information is represented by the frequency of neuron firing

3. Synaptic Weights and Plasticity

Connections between neurons in SNNs are governed by synaptic weights, which determine the influence of one neuron's spike on another.

Synaptic plasticity, often governed by rules such as Spike-Timing-Dependent Plasticity (STDP), allows these weights to change based on the timing of spikes, enabling learning.

Mechanisms of Spiking Neural Networks

1. Membrane Potential and Firing Threshold

Each neuron has a membrane potential that integrates incoming spikes. When the potential crosses a threshold, the neuron fires a spike and the potential resets.

2. Synaptic Integration

3. Learning Rules

- **Spike-Timing-Dependent Plasticity (STDP)**: The strength of synapses is adjusted based on the relative timing of spikes. If a presynaptic neuron fires shortly before a postsynaptic neuron, the connection is strengthened (LTP). If the order is reversed, the connection is weakened (LTD).

4. Neuron Models

- **Leaky Integrate-and-Fire (LIF)**: A simple model where the membrane potential decays over time unless it's boosted by incoming spikes.

- **Hodgkin-Huxley Model**: A more complex and biologically realistic model that describes the ionic mechanisms underlying the initiation and propagation of action potentials.

Implementation of Spiking Neural Network

Step 1: Define Neuron and Synapse Classes

- The LIFNeuron class models the behaviour of a leaky integrate-and-fire neuron.
- The Synapse class represents the connection between neurons with an associated weight.

Step 2: Define the STDP Learning Rule

- The stdp function adjusts the synaptic weights based on the timing difference between the pre- and post-synaptic spikes.

Step 3: Initialize Simulation Parameters and Network

- Set the number of time steps and the sizes of input, hidden, and output layers.
- Initialize neurons and synapses with their parameters and random weights.

Step 4: Define the Spike Train Pattern to Detect

Step 5: Simulation Loop

- Run the simulation for the defined number of time steps.
- Update neurons and synapses at each time step.
- Apply the STDP learning rule to adjust synaptic weights.
- Check if the pattern is detected.

Here's the code which I have written for snn

```
import numpy as np


# Neuron Parameters
class LIFNeuron:
    def __init__(self, threshold, reset_value, decay_factor, 
refractory_period):
        self.threshold = threshold
        self.reset_value = reset_value
        self.decay_factor = decay_factor
        self.refractory_period = refractory_period
        self.membrane_potential = 0
        self.spike_time = -1
```

```
        self.refractory_end_time = -1

    def update(self, incoming_spikes, current_time):
        if current_time < self.refractory_end_time:
            return False

        self.membrane_potential *= self.decay_factor
        self.membrane_potential += np.sum(incoming_spikes)

        if self.membrane_potential >= self.threshold:
            self.spike_time = current_time
            self.membrane_potential = self.reset_value
            self.refractory_end_time = current_time +
self.refractory_period
            return True
        return False

# Synapse Parameters
class Synapse:
    def __init__(self, weight):
        self.weight = weight

# Spike-Timing-Dependent Plasticity (STDP)
def stdp(pre_spike_time, post_spike_time, weight, learning_rate,
tau_positive, tau_negative):
    if pre_spike_time > 0 and post_spike_time > 0:
        delta_t = post_spike_time - pre_spike_time
        if delta_t > 0:
            return weight + learning_rate * np.exp(-delta_t / tau_positive)
        else:
            return weight - learning_rate * np.exp(delta_t / tau_negative)
    return weight

# Simulation Parameters
time_steps = 100
```

```
input_size = 5
hidden_size = 3
output_size = 1

# Network Initialization
input_neurons = [LIFNeuron(threshold=1.0, reset_value=0.0,
decay_factor=0.9, refractory_period=2) for _ in range(input_size)]
hidden_neurons = [LIFNeuron(threshold=1.0, reset_value=0.0,
decay_factor=0.9, refractory_period=2) for _ in range(hidden_size)]
output_neurons = [LIFNeuron(threshold=1.0, reset_value=0.0,
decay_factor=0.9, refractory_period=2) for _ in range(output_size)]

input_to_hidden_synapses = np.random.rand(input_size, hidden_size)
hidden_to_output_synapses = np.random.rand(hidden_size, output_size)

learning_rate = 0.01
tau_positive = 20
tau_negative = 20

# Spike Train Pattern to Detect
pattern = [1, 0, 1, 0, 1]

# Simulation Loop
for t in range(time_steps):
    # Generate input spike trains (random for this example)
    input_spikes = np.random.randint(0, 2, size=input_size)

    # Update input neurons
    hidden_spikes = np.zeros(hidden_size)
    for i, neuron in enumerate(input_neurons):
        if neuron.update(input_spikes[i] * input_to_hidden_synapses[i], t):
            hidden_spikes += input_to_hidden_synapses[i]

    # Update hidden neurons
    output_spikes = np.zeros(output_size)
    for j, neuron in enumerate(hidden_neurons):
```

```
        if neuron.update(hidden_spikes[j] * hidden_to_output_synapses[j], t):
            output_spikes += hidden_to_output_synapses[j]

    # Update output neurons
    for k, neuron in enumerate(output_neurons):
        neuron.update(output_spikes[k], t)

    # STDP Learning
    for i in range(input_size):
        for j in range(hidden_size):
            input_to_hidden_synapses[i, j] = stdp(input_neurons[i].spike_time, hidden_neurons[j].spike_time, input_to_hidden_synapses[i, j], learning_rate, tau_positive, tau_negative)
    for j in range(hidden_size):
        for k in range(output_size):
            hidden_to_output_synapses[j, k] = stdp(hidden_neurons[j].spike_time, output_neurons[k].spike_time, hidden_to_output_synapses[j, k], learning_rate, tau_positive, tau_negative)

    # Check if pattern is detected
    if all(neuron.spike_time == t for neuron, pat in zip(input_neurons, pattern) if pat == 1):
        print(f"Pattern detected at time step {t}")
```

Reinforcement learning (RL)

RL

RL focuses on teaching agents through trial & error.

Concepts:

1. Agent
2. Environment
3. Action, ~~Reward~~
4. Reward, Observation

Limitations & considerations

1. For simple problem RL can be overkill
2. Assumes environment is Markovian
3. Training can take long time & is not always stable

1. Import dependencies

algorithm

```
import os
import gym
from stable_baselines3 import PPO
from    ——"——   ——  .common.vec_env  import DummyVecEnv
from    ——"——   ——  .evaluation  import evaluate_policy
```

Open AI gym spaces:

1. Box – n dimentional tensor, range of values
   eg: Box (0, 1, shape = (3, 3))

2. Discrete – set of items
   eg: Discrete (3)

3. Tuple - tuple of other spaces eg. Box or Discrete
eg: Tuple ((Discrete (2), Box (0,100, shape (1,))))

4. Dict - dictionary of spaces eg: Box or discrete
eg: Dict ({'height': Discrete(2), "speed": Box (0,100, ----)})

5. MultiBinary - one hot encoded binary values
eg: MultiBinary(4)

6. MultiDiscrete - multiple discrete values
eg: MultiDiscrete ([5,2,2])

2. Load Environment

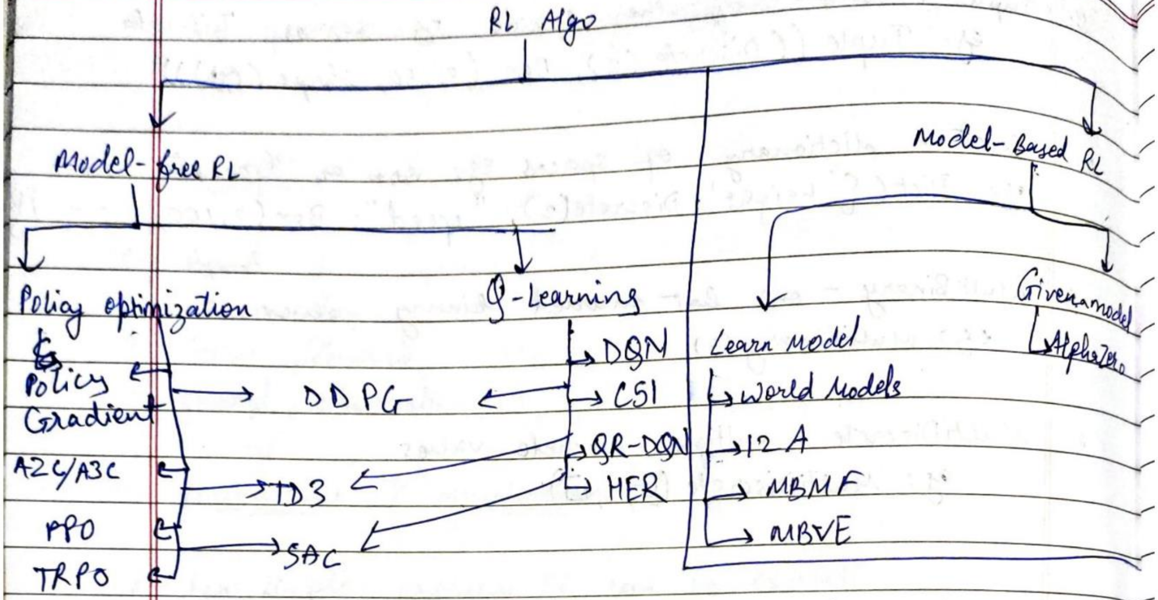
```
environment_name = 'CartPole-v0'
env = gym.make (environment_name)

environment_name
```

3. Understanding the Environment

```
env.action_space.sample()
env.observation_space.sample().
```

4. Training.



Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
A2C	✓	✓	✓	✓	✓
DDPG	✓	X	X	X	X
DQN	X	✓	X	X	X
HER	✓	✓	X	X	X
PPO	✓	✓	✓	✓	✓
SAC	✓	X	X	X	X
TD3	✓	X	X	X	X

Action Space

DQN: Discrete single process
PPO or A2C: Discrete multi processed
SAC or TD3: continuous single processed
PPO or A2C: Continuous multi processed

Understanding Training Metrics:
for A2C Algo

1. Evaluation Metrics:
   Ep_len_mean, ep_rew_mean

2. Time Metrics:
   fps, iterations, time_elapsed, total_timesteps

3. Loss Metrics:
   Entropy_loss, Policy_loss, value_loss

4. Other Metrics:
   Explained_variance; Learning_rate, n_updates

5. Save Model & Reload Model

6. Evaluation

7. Test Model.

8. Viewing Logs in TensorBoard.

Training Strategies:
1. Train for longer
2. Hyperparameter tuning
3. Try diffⁿ algorithm.

9. Adding a Callback to training stage
10. Changing policies.
11. Using Alternate Algo.

Here’s the code which I have written for RL

```
import os
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.evaluation import evaluate_policy

environment_name='CartPole-v0'
env=gym.make(environment_name)

#environment_name

episodes=5
for episode in range(1,episodes+1):
    state=env.reset()
    done = False
    score=0

    while not done:
        env.render()
        action= env.action_space.sample()
        n_state, reward, done, info=env.step(action)
        score+=reward
    print('episode:{} Score:{}'.format(episode,score))
env.close()

#env.reset()

'''episodes=5
for episode in range(1,episodes+1):
    print(episode)

#env.reset()
env.step(1)'''
```

```
env.action_space.sample()
env.observation_space.sample()

env=gym.make(environment_name)
env=DummyVecEnv([lambda: env])
model=PPO('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=20000)

PPO_path=os.path.join ('Training', 'Saved Models', 'PPO_model')
model.save(PPO_path)
del model
model=PPO.load('PPO_model', env=env)

from stable_baselines3.common.evaluation import evaluate_policy
evaluate_policy(model, env, n_eval_episodes=10, render=True)
env.close()

obs=env.reset()
while True:
    action, _states=model.predict(obs)
    obs, rewards, done, info=env.step(action)
    env.render()
    if done:
        print('info', info)
        break
env.close()

from stable_baselines3.common.callbacks import EvalCallback,
StopTrainingOnRewardThreshold
import os
save_path= os.path.join('Training', 'Saved Models')
log_path=os.path.join('Training', 'Logs')
env=gym.make(environment_name)
env=DummyVecEnv([lambda: env])
```

```
stop_callback=StopTrainingOnRewardThreshold(reward_threshold=190,
verbose=1)

eval_callback=(env, callback_on_new_best=stop_callback, eval_freq=10000,
best_model_save_path=save_path, verbose=1)

model=PPO('MlpPolicy', env, verbose=1,tensorboard_log=log_path)

model.learn(total_timesteps=20000, callback=eval_callback)

model_path=os.path.join('Training', 'Saved Models', 'best_model')

model=PPO.load(model_path, env=env)

evaluate_policy(model, env, n_eval_episodes=10, render=True)

env.close()


net_arch=[dict(pi=[128,128,128,128], vf=[128,128,128,128])]

model=PPO('MlpPolicy',env,verbose=1, policy_kwargs={'net_arch': net_arch})

model.learn(total_timesteps=20000, callback=eval_callback)


from stable_baselines3 import DQN

model= DQN('MlpPolicy', env, verbose=1, tensorboard_log=log_path)

model.learn(total_timesteps=20000, callback=eval_callback)

dqn_path=os.path.join('Training', 'Saved Models', 'DQN_model')

model.save(dqn_path)

model= DQN.load(dqn_path, env=env)

evaluate_policy(model, env, n_eval_episodes=10, render=True)

env.close()
```

Week 2: SNNs + RL Fundamentals

Implement a Basic Spiking Neural Network (SNN) for Binary Classification

Introduction to Spiking Neural Networks (SNNs)

Spiking Neural Networks (SNNs) are a class of biologically-inspired neural models that process data using discrete-time events called **spikes**. Unlike traditional neural networks, which rely on continuous activation functions, SNNs model neuron behaviour more closely by transmitting information only when membrane potentials reach a specific threshold. This leads to greater **temporal sparsity**, **energy efficiency**, and potential deployment on neuromorphic hardware.

Generating Spike Trains with snntorch

The first step in training an SNN is to convert static input data (like MNIST images) into **spike trains**. This is achieved using encoders such as **rate encoding** and **latency encoding**.

- **Rate Encoding**: The pixel intensity determines the likelihood of spiking at each time step.
- **Poisson Process**: A probabilistic method to generate spikes over time using the input intensity as the firing probability.

Building and Training an SNN on MNIST

This tutorial demonstrates how to construct and train a **multi-layer SNN** using surrogate gradients.

- **Leaky Integrate-and-Fire (LIF) neurons** are used for spiking behaviour.
- beta=0.9 is the decay factor for membrane potential.
- Uses **temporal processing** over multiple time steps.

Surrogate Gradient Training

Backpropagation is not directly applicable in SNNs due to the non-differentiable spike function. So, **surrogate gradients** are used — approximating the gradient of the spike function for training.

- Loss is computed based on the **number of output spikes** matching the target class.
- Optimizer and training loop use standard PyTorch syntax.

Mathematical Model of LIF Neuron

A Leaky Integrate-and-Fire neuron evolves as:

$$U[t+1] = \beta \cdot U[t] + WX[t] - R[t]$$

$$S[t+1] = \Theta(U[t+1] - U_{\text{thresh}})$$

- U: membrane potential
- β: decay factor
- W: input weights
- S[t]: spike output
- Θ: step function

After spiking, a **reset mechanism** ensures the neuron drops back below threshold.

Outcome

After training:

- The SNN classifies two digits (e.g., 0 and 1) from MNIST.
- Performance is comparable to traditional models in low-resource setups.
- Real advantage: sparsity, temporal coding, energy efficiency.

Develop a basic Q-learning agent to understand the fundamentals of reinforcement learning.

Introduction to Q-Learning

Q-learning is a **model-free reinforcement learning algorithm** that enables an agent to learn the best actions to take in a given environment to maximize long-term rewards. Unlike supervised learning, Q-learning does not need labelled data—it learns by interacting with the environment and observing rewards.

It is **off-policy**, meaning it learns the value of the optimal policy independently of the agent's actions.

- **Environment**: The world with which the agent interacts.
- **Agent**: Learner/decision maker.
- **State (S)**: Current situation of the agent.
- **Action (A)**: Choices available to the agent.
- **Reward (R)**: Feedback from the environment.
- **Q-value (Q[s][a])**: Expected cumulative reward from state s taking action a.

Bellman Equation

The Q-value update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- α: Learning rate (how much to update)
- γ: Discount factor (importance of future rewards)
- r: Immediate reward
- s′: Next state
- $\max_{a'} Q(s',a')$: Maximum future reward

Q-Learning Algorithm

1. Initialize the Q-table with zeros.
2. For each episode:
    - Initialize the state.

- Repeat for each step:
  - Choose action using an **ε-greedy** policy:
    - With probability ε, select random action (exploration).
    - With probability 1−ε, select action with max Q-value (exploitation).
  - Perform the action; observe reward and next state.
  - Update Q-value using Bellman Equation.
  - Set new state = next state.
- Repeat until the goal is reached.

Combine biologically inspired learning rules (STDP) with reinforcement learning techniques (Q-learning)

Introduction

In traditional machine learning, reinforcement learning (RL) and supervised learning often rely on backpropagation. However, **biological neural systems learn without backpropagation**, instead using **local learning rules** like **Spike-Timing Dependent Plasticity (STDP)**.

This objective explores how **STDP**, a biologically plausible mechanism of synaptic change, can be integrated with **Q-learning**, to create reinforcement learning agents that align more closely with real neural circuits, particularly in **spiking neural networks (SNNs)**.

What is STDP?

Spike-Timing Dependent Plasticity (STDP) is a **Hebbian learning rule** that updates synaptic weights based on the precise timing of spikes between presynaptic and postsynaptic neurons.

STDP Rule:

- **If pre-synaptic neuron spikes before post-synaptic → Weight is increased** (Long-Term Potentiation)
- **If post-synaptic neuron spikes before pre-synaptic → Weight is decreased** (Long-Term Depression)

$$\Delta w = \begin{cases} A_+ \cdot e^{-\Delta t/\tau_+} & \text{if } \Delta t > 0 \\ -A_- \cdot e^{\Delta t/\tau_-} & \text{if } \Delta t < 0 \end{cases}$$

Where:

- $\Delta t = t_{post} - t_{pre}$
- A_+, A_- are learning rates
- τ_+, τ_- are decay constants

Combining STDP with Q-Learning

In RL, learning involves associating **states and actions with rewards**. When combined with STDP:

- **Spiking activity encodes state/action representations**
- **STDP adjusts synaptic weights locally**
- **Global reward signals (like dopamine spikes) modulate the plasticity**

This is often termed **Reward-Modulated STDP (R-STDP)**.

R-STDP Update Rule

$\Delta w = R \cdot STDP$

Where R is the reinforcement (reward or punishment). This integrates local spike-based learning with a global reinforcement signal, approximating **policy gradient updates** in an SNN context.

- SNNs with R-STDP can learn to navigate environments and solve control tasks.
- Modulating STDP with **reward signals** allows **unsupervised learning to become goal-directed**.
- **Eligibility traces** act as short-term memory for credit assignment (which STDP lacks alone).

Neuromatch

Simple spiking model using the Brian2 simulator:

- Neurons are connected with STDP synapses.
- A **dopamine signal** acts as a reward modulator.
- A spike-triggered eligibility trace stores the temporal correlation between spikes.
- This shows a synapse whose weight changes only if reward is present.
- Modifies weight using Hebbian principle **only when timing AND reward align**.

Practical Integration with Q-learning

To apply STDP in Q-learning:

1. **State and action representations** are encoded using spiking neurons.
2. A **global Q-learning reward signal** (based on state transitions) modulates the STDP.
3. An **SNN replaces or augments the Q-table**, learning to represent Q-values via synaptic strengths.

This method is useful for:

- Energy-efficient neuromorphic control

- Learning with **limited or no supervision**
- Applications in **robotics**, **neuromorphic chips**, and **bio-inspired AI**

Here are the codes which I have written

Minst:

```
import snntorch as snn
import torch

# Training Parameters
batch_size=128
data_path='/tmp/data/mnist'
num_classes = 10  # MNIST has 10 output classes

# Torch Variables
dtype = torch.float

from torchvision import datasets, transforms
# Define a transform
transform = transforms.Compose([
            transforms.Resize((28,28)),
            transforms.Grayscale(),
            transforms.ToTensor(),
            transforms.Normalize((0,), (1,))])

mnist_train = datasets.MNIST(data_path, train=True, download=True,
transform=transform)

from snntorch import utils

subset = 10
mnist_train = utils.data_subset(mnist_train, subset)

print(f"The size of mnist_train is {len(mnist_train)}")
```

```
from torch.utils.data import DataLoader

train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle=True)



# Temporal Dynamics
num_steps = 10

# create vector filled with 0.5
raw_vector = torch.ones(num_steps)*0.5

# pass each sample through a Bernoulli trial
rate_coded_vector = torch.bernoulli(raw_vector)
print(f"Converted vector: {rate_coded_vector}")

print(f"The output is spiking
{rate_coded_vector.sum()*100/len(rate_coded_vector):.2f}% of the time.")

from snntorch import spikegen

# Iterate through minibatches
data = iter(train_loader)
data_it, targets_it = next(data)

# Spiking Data
spike_data = spikegen.rate(data_it, num_steps=num_steps)
print(spike_data.size())
torch.Size([100, 128, 1, 28, 28])

import matplotlib.pyplot as plt
import snntorch.spikeplot as splt
from IPython.display import HTML

spike_data_sample = spike_data[:, 0, 0]
print(spike_data_sample.size())
```

```
torch.Size([100, 28, 28])

fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)

HTML(anim.to_html5_video())

fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample, fig, ax)

HTML(anim.to_html5_video())

spike_data = spikegen.rate(data_it, num_steps=num_steps, gain=0.25)

spike_data_sample2 = spike_data[:, 0, 0]
fig, ax = plt.subplots()
anim = splt.animator(spike_data_sample2, fig, ax)
HTML(anim.to_html5_video())

plt.figure(facecolor="w")
plt.subplot(1,2,1)
plt.imshow(spike_data_sample.mean(axis=0).reshape((28,-1)).cpu(),
cmap='binary')
plt.axis('off')
plt.title('Gain = 1')

plt.subplot(1,2,2)
plt.imshow(spike_data_sample2.mean(axis=0).reshape((28,-1)).cpu(),
cmap='binary')
plt.axis('off')
plt.title('Gain = 0.25')

plt.show()

# Reshape
```

```
spike_data_sample2 = spike_data_sample2.reshape((num_steps, -1))

# raster plot
fig = plt.figure(facecolor="w", figsize=(10, 5))
ax = fig.add_subplot(111)
splt.raster(spike_data_sample2, ax, s=1.5, c="black")

plt.title("Input Layer")
plt.xlabel("Time step")
plt.ylabel("Neuron Number")
plt.show()

idx = 210  # index into 210th neuron

fig = plt.figure(facecolor="w", figsize=(8, 1))
ax = fig.add_subplot(111)

splt.raster(spike_data_sample.reshape(num_steps, -1)[:, idx].unsqueeze(1), ax, s=100, c="black", marker="|")

plt.title("Input Neuron")
plt.xlabel("Time step")
plt.yticks([])
plt.show()
```

Q-learning

```
import numpy as np
import gym

env = gym.make("FrozenLake-v1", is_slippery=False)  # simple grid world

# Q-table initialization
q_table = np.zeros([env.observation_space.n, env.action_space.n])

# Hyperparameters
```

```
alpha = 0.8         # learning rate
gamma = 0.95        # discount factor
epsilon = 0.1       # exploration rate
episodes = 1000

for episode in range(episodes):
    state = env.reset()[0]
    done = False

    while not done:
        if np.random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()  # Explore
        else:
            action = np.argmax(q_table[state])  # Exploit learned values

        next_state, reward, done, _, _ = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        # Q-Learning formula
        new_value = old_value + alpha * (reward + gamma * next_max -
old_value)
        q_table[state, action] = new_value

        state = next_state

print("Training complete!\n")
```

Week 3: Maze Navigation + STDP

Here's the code that I have written:

```
import pygame
from random import choice

class Cell:
def __init__(self, x, y, thickness):
self.x, self.y = x, y
self.thickness = thickness
self.walls = {'top': True, 'right': True, 'bottom': True, 'left': True}
self.visited = False

# cell.py
class Cell:
...
# draw grid cell walls
def draw(self, sc, tile):
x, y = self.x * tile, self.y * tile
if self.walls['top']:
pygame.draw.line(sc, pygame.Color('darkgreen'), (x, y), (x + tile, y),
self.thickness)
if self.walls['right']:
pygame.draw.line(sc, pygame.Color('darkgreen'), (x + tile, y), (x + tile, y
+ tile), self.thickness)
if self.walls['bottom']:
pygame.draw.line(sc, pygame.Color('darkgreen'), (x + tile, y + tile), (x ,
y + tile), self.thickness)
if self.walls['left']:
pygame.draw.line(sc, pygame.Color('darkgreen'), (x, y + tile), (x, y),
self.thickness)

# cell.py

class Cell:
...
# checks if cell does exist and returns it if it does
def check_cell(self, x, y, cols, rows, grid_cells):
find_index = lambda x, y: x + y * cols
if x < 0 or x > cols - 1 or y < 0 or y > rows - 1:
return False
return grid_cells[find_index(x, y)]

# checking cell neighbors of current cell if visited (carved) or not
def check_neighbors(self, cols, rows, grid_cells):
neighbors = []
top = self.check_cell(self.x, self.y - 1, cols, rows, grid_cells)
right = self.check_cell(self.x + 1, self.y, cols, rows, grid_cells)
bottom = self.check_cell(self.x, self.y + 1, cols, rows, grid_cells)
left = self.check_cell(self.x - 1, self.y, cols, rows, grid_cells)
if top and not top.visited:
neighbors.append(top)
if right and not right.visited:
neighbors.append(right)
```

```
if bottom and not bottom.visited:
neighbors.append(bottom)
if left and not left.visited:
neighbors.append(left)
return choice(neighbors) if neighbors else False

# maze.py
import pygame
from cell import Cell

class Maze:
def __init__(self, cols, rows):
self.cols = cols
self.rows = rows
self.thickness = 4
self.grid_cells = [Cell(col, row, self.thickness) for row in
range(self.rows) for col in range(self.cols)]

# maze.py

class Maze:
...
# carve grid cell walls
def remove_walls(self, current, next):
dx = current.x - next.x
if dx == 1:
current.walls['left'] = False
next.walls['right'] = False
elif dx == -1:
current.walls['right'] = False
next.walls['left'] = False
dy = current.y - next.y
if dy == 1:
current.walls['top'] = False
next.walls['bottom'] = False
elif dy == -1:
current.walls['bottom'] = False
next.walls['top'] = False

# maze.py

class Maze:
...
# generates maze
def generate_maze(self):
current_cell = self.grid_cells[0]
array = []
break_count = 1
while break_count != len(self.grid_cells):
current_cell.visited = True
next_cell = current_cell.check_neighbors(self.cols, self.rows,
self.grid_cells)
if next_cell:
next_cell.visited = True
break_count += 1
array.append(current_cell)
self.remove_walls(current_cell, next_cell)
```

```
current_cell = next_cell
elif array:
current_cell = array.pop()
return self.grid_cells

# player.py
import pygame

class Player:
def __init__(self, x, y):
self.x = int(x)
self.y = int(y)
self.player_size = 10
self.rect = pygame.Rect(self.x, self.y, self.player_size, self.player_size)
self.color = (250, 120, 60)
self.velX = 0
self.velY = 0
self.left_pressed = False
self.right_pressed = False
self.up_pressed = False
self.down_pressed = False
self.speed = 4
# player.py
class Player:
...
# get current cell position of the player
def get_current_cell(self, x, y, grid_cells):
for cell in grid_cells:
if cell.x == x and cell.y == y:
return cell

# stops player to pass through walls
def check_move(self, tile, grid_cells, thickness):
current_cell_x, current_cell_y = self.x // tile, self.y // tile
current_cell = self.get_current_cell(current_cell_x, current_cell_y,
grid_cells)
current_cell_abs_x, current_cell_abs_y = current_cell_x * tile,
current_cell_y * tile
if self.left_pressed:
if current_cell.walls['left']:
if self.x <= current_cell_abs_x + thickness:
self.left_pressed = False
if self.right_pressed:
if current_cell.walls['right']:
if self.x >= current_cell_abs_x + tile - (self.player_size + thickness):
self.right_pressed = False
if self.up_pressed:
if current_cell.walls['top']:
if self.y <= current_cell_abs_y + thickness:
self.up_pressed = False
if self.down_pressed:
if current_cell.walls['bottom']:
if self.y >= current_cell_abs_y + tile - (self.player_size + thickness):
self.down_pressed = False

class Player:
...
```

```
# drawing player to the screen
def draw(self, screen):
pygame.draw.rect(screen, self.color, self.rect)

# updates player position while moving
def update(self):
self.velX = 0
self.velY = 0
if self.left_pressed and not self.right_pressed:
self.velX = -self.speed
if self.right_pressed and not self.left_pressed:
self.velX = self.speed
if self.up_pressed and not self.down_pressed:
self.velY = -self.speed
if self.down_pressed and not self.up_pressed:
self.velY = self.speed
self.x += self.velX
self.y += self.velY
self.rect = pygame.Rect(int(self.x), int(self.y), self.player_size,
self.player_size)
# game.py
import pygame

pygame.font.init()

class Game:
def __init__(self, goal_cell, tile):
self.font = pygame.font.SysFont("impact", 35)
self.message_color = pygame.Color("darkorange")
self.goal_cell = goal_cell
self.tile = tile

# add goal point for player to reach
def add_goal_point(self, screen):
# adding gate for the goal point
img_path = 'img/gate.png'
img = pygame.image.load(img_path)
img = pygame.transform.scale(img, (self.tile, self.tile))
screen.blit(img, (self.goal_cell.x * self.tile, self.goal_cell.y *
self.tile))

# winning message
def message(self):
msg = self.font.render('You Win!!', True, self.message_color)
return msg

# checks if player reached the goal point
def is_game_over(self, player):
goal_cell_abs_x, goal_cell_abs_y = self.goal_cell.x * self.tile,
self.goal_cell.y * self.tile
if player.x >= goal_cell_abs_x and player.y >= goal_cell_abs_y:
return True
else:
return False
# clock.py
import pygame, time
```

```
pygame.font.init()

class Clock:
def __init__(self):
self.start_time = None
self.elapsed_time = 0
self.font = pygame.font.SysFont("monospace", 35)
self.message_color = pygame.Color("yellow")

# Start the timer
def start_timer(self):
self.start_time = time.time()

# Update the timer
def update_timer(self):
if self.start_time is not None:
self.elapsed_time = time.time() - self.start_time

# Display the timer
def display_timer(self):
secs = int(self.elapsed_time % 60)
mins = int(self.elapsed_time / 60)
my_time = self.font.render(f"{mins:02}:{secs:02}", True,
self.message_color)
return my_time

# Stop the timer
def stop_timer(self):
self.start_time = None
# main.py
import pygame, sys
from maze import Maze
from player import Player
from game import Game
from clock import Clock

pygame.init()
pygame.font.init()

class Main():
def __init__(self, screen):
self.screen = screen
self.font = pygame.font.SysFont("impact", 30)
self.message_color = pygame.Color("cyan")
self.running = True
self.game_over = False
self.FPS = pygame.time.Clock()
# main.py
class Main():
...
def instructions(self):
instructions1 = self.font.render('Use', True, self.message_color)
instructions2 = self.font.render('Arrow Keys', True, self.message_color)
instructions3 = self.font.render('to Move', True, self.message_color)
self.screen.blit(instructions1,(655,300))
self.screen.blit(instructions2,(610,331))
self.screen.blit(instructions3,(630,362))
```

```
# draws all configs; maze, player, instructions, and time
def _draw(self, maze, tile, player, game, clock):
# draw maze
[cell.draw(self.screen, tile) for cell in maze.grid_cells]
# add a goal point to reach
game.add_goal_point(self.screen)
# draw every player movement
player.draw(self.screen)
player.update()
# instructions, clock, winning message
self.instructions()
if self.game_over:
clock.stop_timer()
self.screen.blit(game.message(),(610,120))
else:
clock.update_timer()
self.screen.blit(clock.display_timer(), (625,200))
pygame.display.flip()

# main.py
class Main():
...
# main game loop
def main(self, frame_size, tile):
cols, rows = frame_size[0] // tile, frame_size[-1] // tile
maze = Maze(cols, rows)
game = Game(maze.grid_cells[-1], tile)
player = Player(tile // 3, tile // 3)
clock = Clock()
maze.generate_maze()
clock.start_timer()
while self.running:
self.screen.fill("gray")
self.screen.fill( pygame.Color("darkslategray"), (603, 0, 752, 752))
for event in pygame.event.get():
if event.type == pygame.QUIT:
pygame.quit()
sys.exit()
# if keys were pressed still
if event.type == pygame.KEYDOWN:
if not self.game_over:
if event.key == pygame.K_LEFT:
player.left_pressed = True
if event.key == pygame.K_RIGHT:
player.right_pressed = True
if event.key == pygame.K_UP:
player.up_pressed = True
if event.key == pygame.K_DOWN:
player.down_pressed = True
player.check_move(tile, maze.grid_cells, maze.thickness)
# if pressed key released
if event.type == pygame.KEYUP:
if not self.game_over:
if event.key == pygame.K_LEFT:
player.left_pressed = False
if event.key == pygame.K_RIGHT:
```

```
player.right_pressed = False
if event.key == pygame.K_UP:
player.up_pressed = False
if event.key == pygame.K_DOWN:
player.down_pressed = False
player.check_move(tile, maze.grid_cells, maze.thickness)
if game.is_game_over(player):
self.game_over = True
player.left_pressed = False
player.right_pressed = False
player.up_pressed = False
player.down_pressed = False
self._draw(maze, tile, player, game, clock)
self.FPS.tick(60)


if __name__ == "__main__":
window_size = (602, 602)
screen = (window_size[0] + 150, window_size[-1])
tile_size = 30
screen = pygame.display.set_mode(screen)
pygame.display.set_caption("Maze")

game = Main(screen)
game.main(window_size, tile_size)
```

The summary for the above code is as follows:

Cell class (cell.py)

- Represents each cell in the maze.
- Stores wall info (top, right, bottom, left) and visited status.
- Provides methods to draw walls and check for unvisited neighbors during maze generation.

Maze class (maze.py)

- Creates a grid of Cell objects.
- Generates a maze using **Depth-First Search with backtracking**.
- Carves paths by removing walls between current and next unvisited neighbors.

Player class (player.py)

- Controls player movement based on arrow keys.
- Checks for wall collisions to prevent illegal movement.
- Draws the player on screen and updates position.

Game class (game.py)

- Sets the goal point (bottom-right cell).
- Loads an image for the goal (gate).

- Checks if the player has reached the goal.
- Displays a win message if the goal is reached.

Clock class (clock.py)

- Implements a simple timer using time.time().
- Displays elapsed time in MM:SS format.

Main class (main.py)

- Handles the game loop:
    - Initializes the maze, player, goal, and clock.
    - Handles keyboard input.
    - Updates player position, checks for win condition.
    - Renders all elements on screen (maze, player, goal, timer).
- Displays instructions and win message when the goal is reached.

Week 4: meta learning and Q-learning

Meta learning:

Meta-learning, or "learning to learn," is a machine learning technique where models are trained to quickly adapt to new tasks by leveraging experience from previous tasks.

Two-phase process:

1. **Meta-Training**: Learn across multiple tasks to gain general knowledge.
2. **Meta-Testing**: Apply that knowledge to new, unseen tasks with minimal data.

Goal:
Enable models to adapt to new tasks quickly and with few training examples (few-shot learning).

Types of Meta-Learning Approaches

1. **Metric-based**:
    - Learn a distance/similarity metric between data points.
    - Example: Prototypical Networks.
2. **Optimization-based**:
    - Learn how to learn using fast adaptation techniques.
    - Example: MAML (Model-Agnostic Meta-Learning).
3. **Model-based**:
    - Use memory-augmented networks or meta-controllers for fast learning.

- Example: Neural Turing Machines.

Advantages

- **Data-efficient**: Performs well even with limited task-specific data.
- **Fast adaptation**: Learns new tasks quickly with few training steps.
- **Better generalization**: Applies knowledge across a wide range of tasks.
- **Supports AutoML**: Automates algorithm selection and tuning.

Challenges

- High computational cost due to nested training loops.
- Sensitive to task diversity and hyperparameters.
- Risk of overfitting if task distribution is narrow.

Applications

- Few-shot image classification
- Personalized recommendation systems
- Robotics and reinforcement learning
- Automated machine learning (AutoML)

Here is the code for the same

```
import comet_ml
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

from omniglot_dataset import OmniglotDataset
from model import MetaLearner

num_classes = 5
num_shots = 5
```

```
num_queries = 5
num_epochs = 10
learning_rate = 0.001

transform = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
])

train_dataset = OmniglotDataset(root_dir="path/to/omniglot/train",
transform=transform)
train_loader = DataLoader(train_dataset, batch_size=num_classes,
shuffle=True)

meta_learner = MetaLearner(num_classes, num_shots, num_queries)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(meta_learner.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    for batch_idx, (support_set, query_set) in enumerate(train_loader):
        optimizer.zero_grad()

        # Move data to device (e.g., GPU)
        support_set = support_set.to(device)
        query_set = query_set.to(device)

        # Forward pass and backward pass
        loss = meta_learner(support_set, query_set)
        loss.backward()
        optimizer.step()

        # Log loss to Comet ML
        experiment.log_metric("loss", loss.item(), step=batch_idx + epoch *
len(train_loader))
```

```
experiment.log_metric("final_loss", loss.item())
experiment.end()
```

Q-learning :

- Q-Learning is a **model-free** and **off-policy** reinforcement learning algorithm.
- It learns the optimal **state–action value function** (Q-table) based on experience.

The agent learns by **interacting with the environment**, receiving rewards, and updating Q-values.

Q-value `Q(s, a)` estimates the expected future reward for taking action `a` in state `s`.

```
Q(s,a) ← Q(s,a) + α [ r + γ · max_a' Q(s',a') − Q(s,a) ]
```

α: learning rate

γ: discount factor

r: reward

s': next state

Uses **ε-greedy** policy:

- With probability ε → explore random action.
- With probability 1−ε → exploit best known action.

ε decays over time to favour exploitation as learning progresses.

Commonly used with **OpenAI Gym environments** (e.g., Taxi-v3).

Steps:

- Initialize Q-table.
- Loop through episodes: select actions, update Q-values.
- Adjust ε after each episode.

Here is the code for the same:

```
import numpy as np


n_states = 16
n_actions = 4
goal_state = 15
```

```
Q_table = np.zeros((n_states, n_actions))

learning_rate = 0.8
discount_factor = 0.95
exploration_prob = 0.2
epochs = 1000

for epoch in range(epochs):
    current_state = np.random.randint(0, n_states)
    while current_state != goal_state:
        if np.random.rand() < exploration_prob:
            action = np.random.randint(0, n_actions)
        else:
            action = np.argmax(Q_table[current_state])

        next_state = (current_state + 1) % n_states

        reward = 1 if next_state == goal_state else 0

        Q_table[current_state, action] += learning_rate * \
            (reward + discount_factor *
             np.max(Q_table[next_state]) - Q_table[current_state, action])

        current_state = next_state

        q_values_grid = np.max(Q_table, axis=1).reshape((4, 4))

# Plot the grid of Q-values
plt.figure(figsize=(6, 6))
plt.imshow(q_values_grid, cmap='coolwarm', interpolation='nearest')
plt.colorbar(label='Q-value')
plt.title('Learned Q-values for each state')
plt.xticks(np.arange(4), ['0', '1', '2', '3'])
```

```
plt.yticks(np.arange(4), ['0', '1', '2', '3'])
plt.gca().invert_yaxis()  # To match grid layout
plt.grid(True)

# Annotating the Q-values on the grid
for i in range(4):
    for j in range(4):
        plt.text(j, i, f'{q_values_grid[i, j]:.2f}', ha='center',
va='center', color='black')

plt.show()

# Print learned Q-table
print("Learned Q-table:")
print(Q_table)
```

Week 5: policy gradient method

Policy Gradient methods in Reinforcement Learning (RL) to directly optimize the policy, unlike value-based methods that estimate the value of states. These methods are particularly useful in environments with continuous action spaces or complex tasks where value-based approaches struggle. Given a policy π parameterized by θ, the goal is to optimize the objective:

$$J(\theta) = \mathbb{E}\left[\sum_t R_t\right]$$

Where Rt is the reward at time *t* and the expectation is taken over states and actions under the policy πθ.

Working of Policy Gradient Methods

1. **Rollout**: The agent interacts with the environment following the current policy, collecting states, actions and rewards.
2. **Compute the Return**: The return Gt is the cumulative reward obtained from time step t onwards. This is often computed as the discounted sum of rewards.
3. **Compute the Gradient**: The gradient of the objective function with respect to the policy parameters is computed using the collected data.
4. **Update the Policy**: The policy parameters are updated using gradient ascent to improve the expected return.

Types of Policy Gradient Methods

1. **Reinforce Algorithm**
    - A Monte Carlo-based on-policy method.
    - Updates policy parameters using the **entire episode returns**.
    - Simple to implement but prone to **high variance** in gradient estimates.
2. **Actor–Critic Architectures**
    - Combines a **policy network (actor)** with a **value function estimator (critic)**.
    - The critic estimates state-value or advantage function A(s, a), which serves as a **baseline to reduce variance** in policy updates.

3. PPO

- **Proximal Policy Optimization (PPO)** stabilizes learning via clipped surrogate objectives, limiting policy update magnitude for safer optimization.

Here is the code for the assignment given:

```
import gym

import torch

import torch.nn as nn

import torch.optim as optim


# A tiny neural network to decide which action to take

class PolicyNet(nn.Module):

    def __init__(self):

        super().__init__()

        self.fc = nn.Linear(4, 2)  # CartPole has 4 state values, 2 actions


    def forward(self, x):

        return torch.softmax(self.fc(x), dim=0)


# Discounted rewards

def get_returns(rewards, gamma=0.99):

    G = 0

    returns = []

    for r in reversed(rewards):

        G = r + gamma * G

        returns.insert(0, G)
```

```
    return returns

env = gym.make("CartPole-v1")
policy = PolicyNet()
optimizer = optim.Adam(policy.parameters(), lr=0.01)

for episode in range(500):
    state = env.reset()
    log_probs = []
    rewards = []

    done = False
    while not done:
        state_tensor = torch.tensor(state, dtype=torch.float32)
        probs = policy(state_tensor)
        dist = torch.distributions.Categorical(probs)
        action = dist.sample()

        log_probs.append(dist.log_prob(action))
        state, reward, done, _ = env.step(action.item())
        rewards.append(reward)

    returns = get_returns(rewards)
    loss = 0
    for log_prob, G in zip(log_probs, returns):
        loss -= log_prob * G

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if episode % 50 == 0:
        print(f"Episode {episode}, Total reward: {sum(rewards)}")

env.close()
```