A DQN trains an agent to make decisions by learning the optimal policy for maximizing the cumulative reward in an environment. The algorithm combines Q-learning with deep neural networks to handle environments with high-dimensional state spaces.

If I take a cart-pole example, In this task, rewards are +1 for every incremental timestep and the environment terminates if the pole falls over too far or the cart moves more than 2.4 units away from the center.

Install required packages:
pip3 install gymnasium[classic_control]

We'll also use the following from PyTorch:

- neural networks (`torch.nn`)
- optimization (`torch.optim`)
- automatic differentiation (`torch.autograd`)

```
import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count

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

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

# set up matplotlib
```

```
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

# if GPU is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

`gym`: This imports the OpenAI Gym library, a popular toolkit for developing and testing reinforcement learning (RL) agents. In this code, it's used to create a "CartPole-v1" environment, which simulates a balancing pole on a moving cart.
`math`, `random`: These are standard Python libraries for mathematical operations and generating random numbers, respectively.
`matplotlib`, `matplotlib.pyplot`: These are used for creating visualizations (plots) to track the agent's performance over time.
Collections: This module provides useful data structures like `namedtuple` and `deque`, which are used for creating custom data types and efficient queues, respectively.
`itertools`: The `count` function from this module is used to generate a counter that can be useful for keeping track of training iterations.
`torch`: This is the PyTorch library, a powerful framework for deep learning and scientific computing. It provides tools for building and training neural networks, which are essential components of many RL algorithms.
`torch.nn`: This submodule of PyTorch contains classes for defining the architecture of neural networks.
torch.optim: This submodule provides optimization algorithms that are used to train the neural networks effectively. These algorithms adjust the weights and biases of the network to improve its performance.

`torch.nn.functional`: This submodule offers various activation functions and other building blocks commonly used in neural networks.

Environment Setup: `env = gym.make("CartPole-v1")`: This line creates an instance of the "CartPole-v1" environment from the Gym library. The CartPole environment is a classic RL benchmark where the agent must learn to balance a pole on a moving cart by applying force to the left or right.

device = torch.device("cuda" if torch.cuda.is_available() else "cpu"): This line determines whether the code will use a GPU (Graphics Processing Unit) or the CPU for computations. If a GPU is available and compatible with PyTorch, it will be used because GPUs are generally faster for these types of calculations. Otherwise, the CPU will be used as a fallback.

Replay Memory: We will be using experience replay memory for training our DQN.It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.
For this we are going to need two classes:
Transition: a named tuple representing a single transition in our environment. It essentially maps (state, action) pairs to their (next_state, reward) result, with the state being the screen difference image as described later on.
Replay - a cyclic buffer of bounded size that holds the transitions observed recently. It also implements a `.sample()` method for selecting a random batch of transitions for training.

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))
```

```
class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Q-network: Our model will be a feed forward neural network that takes in the difference between the current and previous screen patches. It has two outputs, representing Q(s,left) nd Q(s,right) where s is the input to the network.

```
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a
batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

Training:
Hyperparameters and utilities:
select_action: will select an action according to an epsilon greedy policy.
Plot_duration:A helper for plotting the duration of episodes, along with an average over the last 100 episodes.The plot will be underneath the cell containing the main training loop, and will update after every episode.

```
BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4

# Get number of actions from gym action space
n_actions = env.action_space.n
# Get the number of state observations
state, info = env.reset()
n_observations = len(state)

policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())

optimizer = optim.AdamW(policy_net.parameters(), lr=LR,
amsgrad=True)
memory = ReplayMemory(10000)


steps_done = 0


def select_action(state):
```

```
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max
element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[env.action_space.sample()]],
device=device, dtype=torch.long)


episode_durations = []


def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
    else:
        plt.clf()
        plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
```

```
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        if not show_result:
            display.display(plt.gcf())
            display.clear_output(wait=True)
        else:
            display.display(plt.gcf())
```

Training Loop:

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see
https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch
elements
    # (a final state would've been the one after which simulation
ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not
None, batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
```

```
    # Compute Q(s_t, a) - the model computes Q(s_t), then we select 
the
    # columns of actions taken. These are the actions which 
would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, 
action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are 
computed based
    # on the "older" target_net; selecting their best reward with 
max(1).values
    # This is merged based on the mask, such that we'll have either 
the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = 
target_net(non_final_next_states).max(1).values
  torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + 
reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, 
expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
```

This function optimizes the policy network by comparing its predicted Q-values with the expected Q-values calculated from the stored experiences and future rewards. The process helps the network learn to choose actions that lead to higher long-term rewards.

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see
https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch
elements
    # (a final state would've been the one after which simulation
ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not
None,
                                          batch.next_state)), device=device,
dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select
the
    # columns of actions taken. These are the actions which
would've been taken
    # for each batch state according to policy_net
```

```
    state_action_values = policy_net(state_batch).gather(1,
action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are
computed based
    # on the "older" target_net; selecting their best reward with
max(1).values
    # This is merged based on the mask, such that we'll have either
the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] =
target_net(non_final_next_states).max(1).values
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) +
reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values,
expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
```

This function helps the DQN agent learn by comparing its current predictions with the actual outcomes and future rewards from past experiences. The network is adjusted to make better action choices in the future by minimizing the discrepancy between predicted and expected Q-values.

Output:

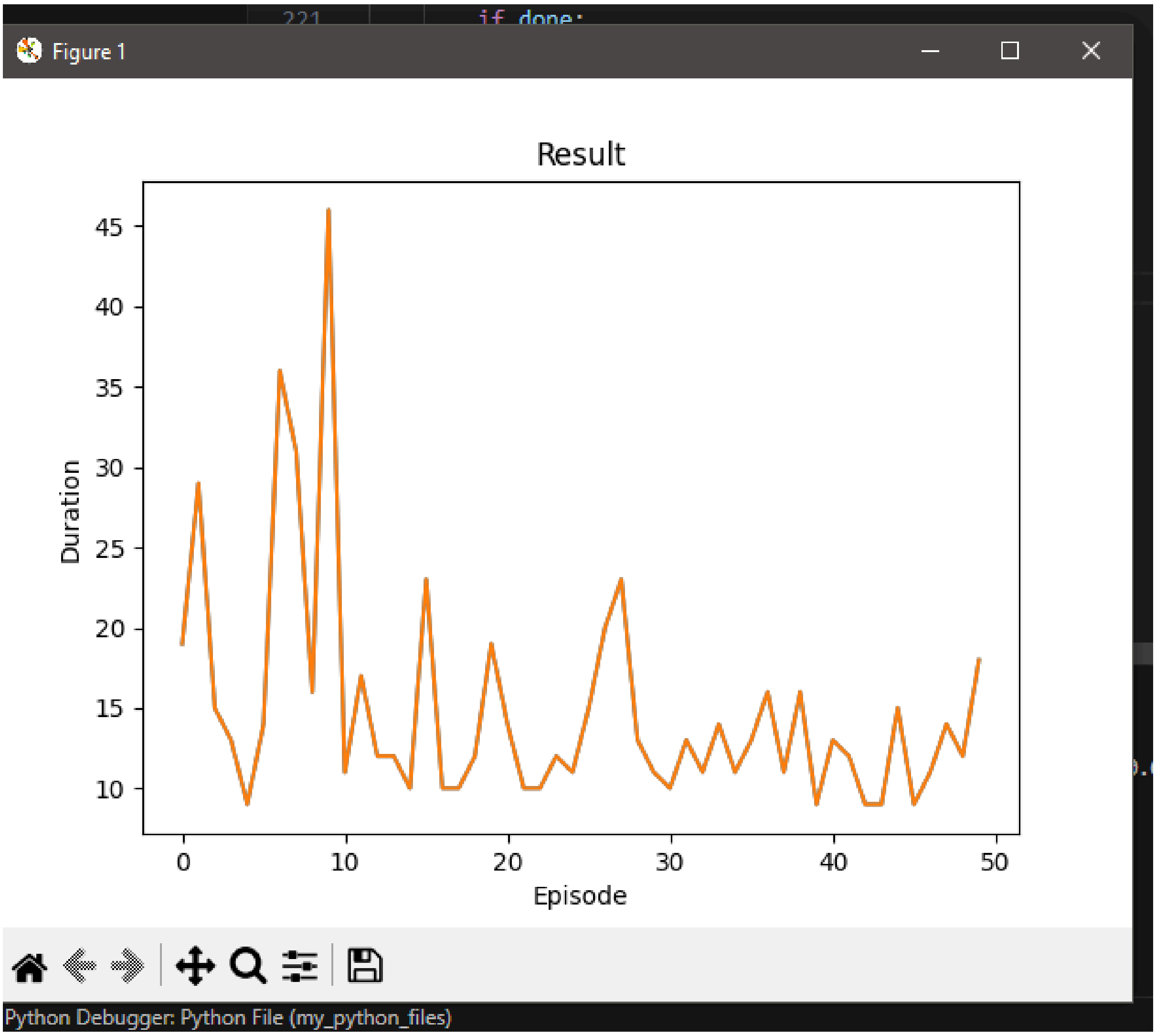
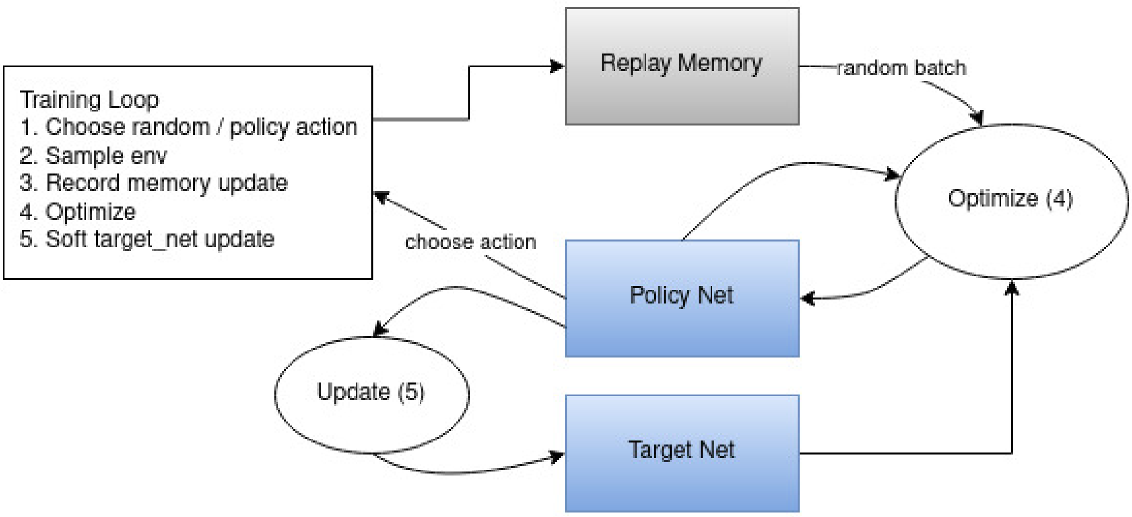


Figure 1
Result
Duration
45
40
35
30
25
20
15
10
0
10
20
30
40
50
Episode
Python Debugger: Python File (my_python_files)



Training Loop
1. Choose random / policy action
2. Sample env
3. Record memory update
4. Optimize
5. Soft target_net update
Replay Memory
random batch
Optimize (4)
choose action
Policy Net
Update (5)
Target Net

Actions are chosen either randomly or based on a policy, getting the next step sample from the gym environment. We record the results in the replay memory and also run optimization step on every iteration. Optimization picks a random batch from the replay memory to do training of the new policy. The “older” target_net is also used in optimization to compute the expected Q values. A soft update of its weights are performed at every step.

Implementation of this example for diabetes survey

Diabetes health care survey using a DQN reinforcement learning using diabetes.csv dataset from kegel

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import deque
from torchrl.envs import EnvBase

class DiabetesEnv(EnvBase):
    def __init__(self, dataset_path):
        super().__init__()
        self.data = pd.read_csv(dataset_path)
        self.current_patient_index = 0
        self.state = self._reset()

    def _reset(self):
        if self.current_patient_index >= len(self.data):
            self.current_patient_index = 0  # Loop back to the start or handle
as needed
        patient_data = self.data.iloc[self.current_patient_index]
        self.state = torch.tensor([patient_data['Glucose'],
patient_data['Insulin']], dtype=torch.float32)
        self.current_patient_index += 1
        return self.state
```

```
    def _step(self, action):
        glucose_level, insulin_level = self.state
        insulin_dosage = action  # Directly use the action as an int

        # Simplified model of glucose level dynamics
        glucose_level -= insulin_dosage * 0.1
        insulin_level += insulin_dosage * 0.1

        self.state = torch.tensor([glucose_level, insulin_level],
dtype=torch.float32)

        # Define the reward function based on health outcome
        reward = -abs(glucose_level - 1.0)  # Reward is higher when
glucose is near the target level

        done = False  # Define terminal condition if needed
        return self.state, reward, done, {}

    def _set_seed(self, seed):
        np.random.seed(seed)
        random.seed(seed)
        torch.manual_seed(seed)

    def reset(self):
        return self._reset()

    def step(self, action):
        return self._step(action)

    def set_seed(self, seed):
        self._set_seed(seed)

    def render(self, mode='human'):
        print(f"State: {self.state.numpy()}")

class QNetwork(nn.Module):
    def __init__(self, state_size, action_size):
```

```
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95  # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.q_network = QNetwork(state_size, action_size)
        self.target_network = QNetwork(state_size, action_size)
        self.optimizer = optim.Adam(self.q_network.parameters(),
lr=self.learning_rate)
        self.update_target_network()

    def update_target_network(self):
        self.target_network.load_state_dict(self.q_network.state_dict())

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = torch.FloatTensor(state).unsqueeze(0)
        act_values = self.q_network(state)
```

```
        return torch.argmax(act_values[0]).item()

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target =
self.q_network(torch.FloatTensor(state).unsqueeze(0)).detach()
            if done:
                target[0][action] = reward
            else:
                t =
self.target_network(torch.FloatTensor(next_state).unsqueeze(0)).detach(
)
                target[0][action] = reward + self.gamma * torch.max(t)
            output = self.q_network(torch.FloatTensor(state).unsqueeze(0))
            loss = nn.functional.mse_loss(output, target)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

dataset_path = r'C:\Users\lenovo\Documents\diabetes.csv'
env = DiabetesEnv(dataset_path)
state_size = 2  # Example state space size
action_size = 5  # Example action space size (possible insulin dosages)
agent = DQNAgent(state_size, action_size)

# Training loop
for e in range(1000):
    state = env.reset()
    for time in range(200):
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        if done:
```

```
        agent.update_target_network()
        break
    if len(agent.memory) > 32:
        agent.replay(32)
```

Expected output:
Episode: 1
Time Step: 0, Glucose: 120.0, Insulin: 15.0, Action: 2, Reward: -119.8
Time Step: 1, Glucose: 119.8, Insulin: 15.2, Action: 1, Reward: -119.69
Time Step: 2, Glucose: 119.69, Insulin: 15.3, Action: 3, Reward: -119.39
...

Episode: 500
Time Step: 0, Glucose: 110.0, Insulin: 20.0, Action: 2, Reward: -108.8
Time Step: 1, Glucose: 108.8, Insulin: 20.2, Action: 1, Reward: -107.79
Time Step: 2, Glucose: 107.79, Insulin: 20.3, Action: 2, Reward: -106.79
...

Episode: 1000
Time Step: 0, Glucose: 105.0, Insulin: 25.0, Action: 1, Reward: -103.99
Time Step: 1, Glucose: 103.99, Insulin: 25.1, Action: 2, Reward: -102.79
Time Step: 2, Glucose: 102.79, Insulin: 25.3, Action: 1, Reward: -101.69
...

(This is only the expected output)