EX8-Q1

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

Reconnect Editing

Question 1

Question 1

```
import copy
import math
import os
from collections import namedtuple
import gym
import ipywidgets as widgets
import matplotlib.pyplot as plt
import more_itertools as mitt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.init as init
import tqdm
```

Parts a, b, c

a, b, c

```
[ ] input=np.linspace(-10, 10, 500)
    input=torch.Tensor([input])
    input=torch.t(input)

    def F(x):
      y=x**2+1
      return y
    output=F(input)

    dataset=torch.column_stack((input,output))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please consider con
```

```
[ ]
    class MLP(nn.Module):

        def __init__(self, input_dim=1, num_layers=3, hidden_dim=8,output_dim=1):
            """
            MLP Model
            """
            super(MLP, self).__init__()
            self.input_dim = input_dim
            self.num_layers = num_layers
            self.hidden_dim = hidden_dim

            layer_map = []
            layer_map.append(nn.Linear(self.input_dim,self.hidden_dim)) # input layer
            layer_map.append(nn.ReLU())


            for i in range(2,num_layers):
                layer_map.append(nn.Linear(self.hidden_dim,self.hidden_dim))
                layer_map.append(nn.ReLU())

            layer_map.append(nn.Linear(self.hidden_dim,1)) # output layer
            layer_map.append(nn.Identity())  ####???

            self.model = nn.Sequential(*layer_map)



        def forward(self, inputs) -> torch.Tensor:
            """
            Forward function

            """

            return self.model(inputs)
```

```
[ ] def train_MLP_batch(optimizer, batch, MLP_model) -> float:
        """Perform a single batch-update step on the given MLP model.

        """

        inputs= batch[:,0]
        labls=batch[:,1].reshape(*batch[:,1].shape,1)
        outputs = MLP_model(inputs.reshape(*inputs.shape,1))

        # computing the scalar MSE loss between computed labls and the outputs
        loss = nn.functional.mse_loss(outputs, labls)

        optimizer.zero_grad()  # reset all previous gradients
```

```
    loss.backward()  # compute new gradients
    optimizer.step()  # perform one gradient descent step

    return loss.item()
```

```
[ ] def train_MLP(

    input_dim,
    num_layers,
    hidden_dim,
    output_dim,
    batch_size ,
    num_episods,
    dataset,
):

    # initialize the MLP
    MLP_model = MLP(input_dim=input_dim, num_layers=num_layers, hidden_dim=hidden_dim, output_dim=output_dim)

    # initialize the optimizer
    optimizer = torch.optim.Adam(MLP_model.parameters())
    episods = tqdm.notebook.tnrange(num_episods)
    losses=torch.zeros(num_episods)

    for i in episods:


        inputs_size=dataset.shape[0]

        num_batches = math.ceil(inputs_size/batch_size)
        batch_list = [dataset[batch_size*y:batch_size*(y+1),:] for y in range(num_batches)]

        jj=1

        for j in range(num_batches):

            loss = train_MLP_batch(optimizer, batch_list[j], MLP_model)

            episods.set_description(
                f'Episode: {i} | Batch_number: {j} | Batch_loss: {loss:5.2f} | Hidden_dim: {hidden_dim:5.2g}'
            )


        losses[i]=loss

    trained_models=MLP_model
    trained_models = copy.deepcopy(MLP_model)

    return (
        trained_models,
        losses,
    )
```

```
[ ] input=np.linspace(-10, 10, 500)
    input=torch.Tensor([input])
    input=torch.t(input)

    def F(x):
      y=x**2+1
      return y
    output=F(input)

    dataset=torch.column_stack((input,output))
```

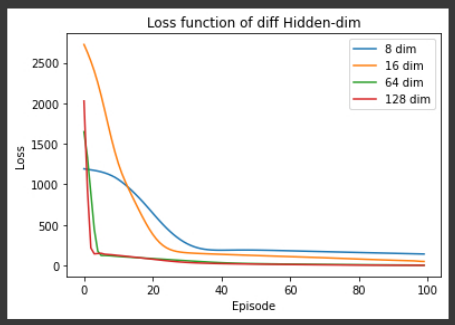
```
    input_dim=1
    num_layers=3
    output_dim=1
    batch_size = 32
    num_episods=100
    hidden_dims=[8,16,64,128]

    losses_list=torch.zeros(len(hidden_dims),num_episods)
    jj=0

    for i in hidden_dims:

      r=torch.randperm(dataset.shape[0])
      dataset=dataset[r]

      trained_model, losses = train_MLP(
          input_dim= input_dim,
          num_layers= num_layers,
          hidden_dim= hidden_dims[jj],
          output_dim= output_dim,
          batch_size = batch_size,
          num_episods= num_episods,
          dataset=dataset,
      )
      print(losses)
      losses_list[jj,:]=losses
      jj+=1
```

```
Episode: 99 | Batch_number: 15 | Batch_loss: 142.32 | Hidden_dim: 8: 100% 100/100 [00:06<00:00, 15.32it/s]
tensor([1192.3900, 1185.8303, 1179.5247, 1172.7976, 1164.6511, 1154.6555,
        1142.5486, 1127.5386, 1109.3850, 1087.7662, 1060.8977, 1030.3948,
         996.6107,  959.9368,  920.6774,  879.3117,  836.0303,  790.8494,
         743.9437,  695.7986,  647.1227,  598.7248,  551.4069,  505.8838,
         462.7287,  422.3683,  385.0954,  351.1049,  320.5331,  293.4418,
         269.8627,  249.7775,  233.1094,  219.6852,  209.2546,  201.4969,
         196.0385,  192.4737,  190.3940,  189.4133,  189.1851,  189.4187,
         189.8888,  190.4214,  190.9043,  191.2637,  191.4617,  191.4840,
         191.3334,  191.0218,  190.5665,  189.9871,  189.3032,  188.5335,
         187.6951,  186.8010,  185.8624,  184.8894,  183.8901,  182.8708,
         181.8366,  180.7915,  179.7399,  178.6835,  177.6257,  176.5670,
         175.5087,  174.4513,  173.3957,  172.3422,  171.2915,  170.2436,
         169.1988,  168.1575,  167.1197,  166.0856,  165.0553,  164.0284,
         163.0063,  161.9881,  160.9735,  159.9626,  158.9551,  157.9515,
         156.9518,  155.9557,  154.9605,  153.9661,  152.9737,  151.9851,
         151.0006,  150.0200,  149.0437,  148.0713,  147.1030,  146.1385,
         145.1779,  144.2214,  143.2687,  142.3199])
Episode: 99 | Batch_number: 15 | Batch_loss: 50.69 | Hidden_dim: 16: 100% 100/100 [00:06<00:00, 14.97it/s]
tensor([2724.0894, 2626.8379, 2518.1909, 2395.8647, 2254.3254, 2094.1624,
        1919.4733, 1738.3865, 1561.9177, 1400.9742, 1258.1575, 1137.7061,
        1035.7073,  943.7837,  855.9435,  769.5615,  684.4686,  601.7955,
         523.2802,  450.7943,  386.0431,  330.2721,  284.0802,  247.3473,
         219.2931,  198.6516,  183.9207,  173.6065,  166.4036,  161.2837,
         157.5070,  154.5798,  152.1737,  150.0917,  148.2105,  146.4485,
         144.7668,  143.1696,  141.6048,  140.0698,  138.5637,  137.0747,
         135.5986,  134.1319,  132.6778,  131.2319,  129.7859,  128.3512,
         126.9217,  125.5004,  124.0867,  122.6799,  121.2683,  119.8710,
         118.4771,  117.0881,  115.7106,  114.3230,  112.9357,  111.5618,
         110.1513,  108.7447,  107.3619,  106.0015,  104.6251,  103.2594,
         101.8883,  100.5283,   99.1331,   97.7350,   96.1971,   94.6031,
          92.9467,   91.3448,   89.7753,   88.2369,   86.6268,   85.1321,
          83.3854,   81.7313,   80.2101,   78.7273,   77.0141,   75.4630,
          74.0123,   72.5107,   71.1199,   69.4662,   68.1293,   66.8949,
          65.3392,   64.0735,   62.8554,   61.7243,   60.4346,   59.2881,
          58.0824,   55.1416,   52.0425,   50.6883])
Episode: 99 | Batch_number: 15 | Batch_loss: 5.00 | Hidden_dim: 64: 100% 100/100 [00:06<00:00, 14.38it/s]
tensor([1648.7008, 1332.1963,  889.2394,  444.6171,  178.1600,  124.1582,
         124.5686,  122.7494,  120.1453,  116.4590,  112.8424,  109.8575,
         107.1735,  104.5055,  101.7641,   98.9483,   96.2858,   93.6575,
          91.0095,   88.3562,   85.7109,   83.0640,   80.4118,   77.7569,
          75.0991,   72.4383,   69.7792,   67.1257,   64.4768,   61.8425,
          59.2221,   56.6295,   54.0868,   51.5711,   49.0757,   46.5741,
          44.0689,   41.5648,   39.4332,   37.4409,   35.5926,   33.8709,
          32.2769,   30.7431,   29.2336,   27.8541,   26.7880,   25.8116,
          24.9684,   24.1553,   23.4361,   22.7416,   22.1495,   21.5226,
          20.9308,   20.3455,   19.7672,   19.2017,   18.7100,   18.3059,
          17.8171,   17.3960,   16.9966,   16.5755,   16.1567,   15.7602,
          15.3583,   14.9681,   14.5703,   14.1693,   13.8106,   13.4121,
          13.0600,   12.6981,   12.3024,   11.9883,   11.6019,   11.2847,
          10.9304,   10.5781,   10.2653,    9.9229,    9.6236,    9.2813,
           8.9887,    8.6850,    8.3642,    8.0861,    7.7836,    7.5094,
           7.2265,    6.9460,    6.6945,    6.4285,    6.1676,    5.9269,
           5.6864,    5.4446,    5.2230,    4.9973])
```

part d

d

```
[ ]  plt.plot(losses_list[0],label="8 dim")
     plt.plot(losses_list[1],label="16 dim")
     plt.plot(losses_list[2],label="64 dim")
     plt.plot(losses_list[3],label="128 dim")
     plt.title("Loss function of diff Hidden-dim")
     plt.xlabel("Episode")
     plt.ylabel("Loss")
     plt.legend()
     plt.show()
```



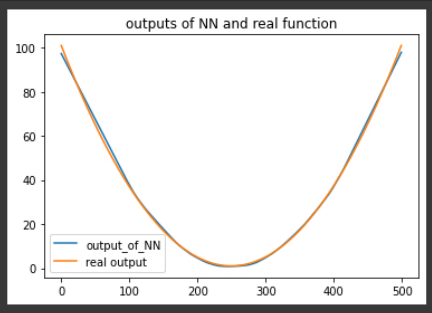
Part e

e

As it is obvious the larger dimension the more accurate model, because with increasing the dimension the predictiong model will be more complex and it would be able to model most features of the function

```
[ ]  output_of_NN=trained_model(input)
     output_of_NN=output_of_NN.detach().numpy()
```

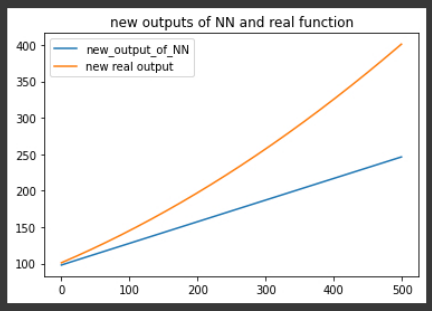
```
plt.plot(output_of_NN,label="output_of_NN")
plt.plot(output,label="real output")
plt.title("outputs of NN and real function")
plt.legend()
plt.show()
```



As it is obvious the MLP model has been modeled very accurate within [-10 , 10]. This is the MLP model of 128 dim hidden layer

```
[ ] new_input=np.linspace(10, 20, 500)
    new_input=torch.Tensor([new_input])
    new_input=torch.t(new_input)

    new_output=F(new_input)
    new_output_of_NN=trained_model(new_input)
    new_output_of_NN=new_output_of_NN.detach().numpy()
```

```
[ ] plt.plot(new_output_of_NN,label="new_output_of_NN")
    plt.plot(new_output,label="new real output")
    plt.title("new outputs of NN and real function")
    plt.legend()
    plt.show()
```



Although the MLP model predict the output very accurate within [-10,10] it is not predincting accuratly the amount outside of [-10,10] and it shows the model has been overfited.

EX8 Q3.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk Editing

Question 3

```
import copy
import math
import os
from collections import namedtuple
import ipywidgets as widgets
import matplotlib.pyplot as plt
import more_itertools as mitt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import tqdm
```

[17]
```
!apt-get install python-box2d
!pip install box2d-py
!pip install gym[Box_2D]
import gym
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
python-box2d is already the newest version (2.3.2~dfsg-2).
0 upgraded, 0 newly installed, 0 to remove and 39 not upgraded.
Requirement already satisfied: box2d-py in /usr/local/lib/python3.7/dist-packages (2.3.8)
Requirement already satisfied: gym[Box_2D] in /usr/local/lib/python3.7/dist-packages (0.17.3)
WARNING: gym 0.17.3 does not provide the extra 'box_2d'
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.3.0)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.5.0)
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.21.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.4.1)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from pyglet<=1.5.0,>=1.4.0->gym[Box_2D]) (0.16.0)
```

[18]
```
envs = {
    'cartpole': gym.make('CartPole-v1'),
    'lunarlander': gym.make('LunarLander-v2'),
}
```

[19]
```
#### Exponential Decay

class ExponentialDecay:
    def __init__(self, value_from, value_to, num_steps):

        self.value_from = value_from
        self.value_to = value_to
        self.num_steps = num_steps

        # `a` and `b` parameters such that the schedule is correct
        self.a = self.value_from
        self.b = math.log(self.value_to/self.a) / (self.num_steps-1)

    def value(self, step) -> float:

        if step <= 0:
            value = self.value_from
        elif step >= self.num_steps - 1:
            value = self.value_to
        else:
            value = self.a * math.exp(self.b * step)

        return value
```

[20]
```
# Batch namedtuple, a class which contains the given attributes
Batch = namedtuple(
    'Batch', ('states', 'actions', 'rewards', 'next_states', 'dones')
)


class ReplayMemory:
    def __init__(self, max_size, state_size):
        """Replay memory as a buffer.
        """
        self.max_size = max_size
        self.state_size = state_size

        # preallocating all the required memory, for speed concerns
        self.states = torch.empty((max_size, state_size))
        self.actions = torch.empty((max_size, 1), dtype=torch.long)
        self.rewards = torch.empty((max_size, 1))
        self.next_states = torch.empty((max_size, state_size))
        self.dones = torch.empty((max_size, 1), dtype=torch.bool)


        self.idx = 0
```

```
        self.size = 0

    def add(self, state, action, reward, next_state, done):


        # store the input values into the appropriate attributes, using the current buffer position `self.idx`

        self.states[self.idx] = state
        self.actions[self.idx] = action
        self.rewards[self.idx] = reward
        self.next_states[self.idx] = next_state
        self.dones[self.idx] = done

        # circulate the pointer to the next position
        self.idx = (self.idx + 1) % self.max_size
        # update the current buffer size
        self.size = min(self.size + 1, self.max_size)

    def sample(self, batch_size) -> Batch:


        if batch_size >= self.size:
            sample_indices = torch.arange(self.size)
        else:
            sample_indices = torch.from_numpy(np.random.choice(range(self.size), size=batch_size,replace=False))

        sample_indices = sample_indices.long()

        states = torch.index_select(self.states, 0, sample_indices)
        actions = torch.index_select(self.actions, 0, sample_indices)
        rewards = torch.index_select(self.rewards, 0, sample_indices)
        next_states = torch.index_select(self.next_states, 0, sample_indices)
        dones = torch.index_select(self.dones, 0, sample_indices)

        batch = Batch(states,actions,rewards,next_states,dones)

        return batch

    def populate(self, env, num_steps):
        """Populate this replay memory with `num_steps` from the random policy.
        """

        # run a random policy for `num_steps` time-steps

        state = env.reset()
        for i in range(num_steps):
            action = env.action_space.sample()
            next_state, reward, done, _ = env.step(action)
            self.add(torch.from_numpy(state), action, reward, torch.from_numpy(next_state), done)

            state = next_state
            if done:
                state = env.reset()
```

```
[21] class DQN(nn.Module):
    def __init__(self, state_dim, action_dim, *, num_layers=3, hidden_dim=256):


        super().__init__()
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.num_layers = num_layers
        self.hidden_dim = hidden_dim


        layer_map = []
        layer_map.append(nn.Linear(self.state_dim,hidden_dim)) # input layer
        layer_map.append(nn.ReLU())

        for i in range(2,num_layers):
            layer_map.append(nn.Linear(self.hidden_dim,self.hidden_dim))
            layer_map.append(nn.ReLU())

        layer_map.append(nn.Linear(hidden_dim,self.action_dim)) # output layer
        layer_map.append(nn.Identity())

        self.model = nn.Sequential(*layer_map)

    def forward(self, states) -> torch.Tensor:

        return self.model(states)


    @classmethod
    def custom_load(cls, data):
        model = cls(*data['args'], **data['kwargs'])
        model.load_state_dict(data['state_dict'])
        return model

    def custom_dump(self):
        return {
            'args': (self.state_dim, self.action_dim),
            'kwargs': {
                'num_layers': self.num_layers
```

```
                'hidden_dim': self.hidden_dim,
            },
            'state_dict': self.state_dict(),
        }
```

```
[22] def train_dqn_batch(optimizer, batch, dqn_model, dqn_target, gamma) -> float:
    """Perform a single batch-update step on the given DQN model.
    """
    # computing the values and target_values tensors using the given models and the batch of data.

    # values = current Q values, targets = rewards + gamma * max Q values

    states, actions, rewards, next_states, dones = batch
    length = torch.numel(rewards)
    values = torch.empty(length,dtype=torch.float)
    target_values = np.empty(length)

    for i in range(length):
        values[i] = dqn_model(states[i])[actions[i]]
        if dones[i]:
            target_values[i] = rewards[i]
        else:
            target_values[i] = rewards[i] + gamma * dqn_target(next_states[i])[torch.argmax(dqn_model(states[i])).item()]

    target_values = torch.tensor(target_values,dtype=torch.float) # to avoid requires_grad nonsense

    assert (
        values.shape == target_values.shape
    ), 'Shapes of values tensor and target_values tensor do not match.'


    # computing the scalar MSE loss between computed values and the TD-target
    loss = F.mse_loss(values, target_values)

    optimizer.zero_grad()  # reset all previous gradients
    loss.backward()  # compute new gradients
    optimizer.step()  # perform one gradient descent step

    return loss.item()
```

```
[34] def train_dqn(
        env,
        num_steps,
        *,
        replay_size,
        replay_prepopulate_steps=0,
        batch_size,
        exploration,
        gamma,
    ):
        """
        we  train for a given number of time-steps rather than a given number of episodes.
        """
        # check that environment states are compatible with our DQN representation
        assert (
            isinstance(env.observation_space, gym.spaces.Box)
            and len(env.observation_space.shape) == 1
        )

        # get the state_size from the environment
        state_size = env.observation_space.shape[0]

        # initialize the DQN and DQN-target models
        dqn_model = DQN(state_size, env.action_space.n)
        dqn_target = DQN.custom_load(dqn_model.custom_dump())

        # initialize the optimizer
        optimizer = torch.optim.Adam(dqn_model.parameters())

        # initialize the replay memory and prepopulate it
        memory = ReplayMemory(replay_size, state_size)
        memory.populate(env, replay_prepopulate_steps)

        # initiate lists to store returns, lengths and losses
        rewards = []
        returns = []
        lengths = []
        losses = []


        i_episode = 0
        t_episode = 0
        G = 0 # episode return,

        state = env.reset()  # initialize state of first episode

        # iterate for a total of `num_steps` steps

        pbar = tqdm.notebook.tnrange(num_steps)

        for t_total in pbar:
```

```
        #  sampling an action from the DQN using epsilon-greedy
        #  using the action to advance the environment by one step
        #  storing the transition into the replay memory

        eps = exploration.value(t_total)

        if np.random.random() < eps:
            action = np.random.randint(env.action_space.n)
        else:
            action = torch.argmax(dqn_model(torch.Tensor(state)))
            action = action.item()

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

        memory.add(torch.Tensor(state), action, reward, torch.Tensor(next_state), done)


        #  sampling a batch from the replay memory
        #  performing a batch update

        if t_total % 4 == 0:
            batch = memory.sample(batch_size)
            loss = train_dqn_batch(optimizer, batch, dqn_model, dqn_target, gamma)
            losses.append(loss)


        #  updating the target network

        if t_total % 10000 == 0:
            dqn_target.load_state_dict(dqn_model.state_dict())

        if done:
            # compute return G, store stuff, reset variables, indices and lists

            lengths.append(t_episode + 1)
            returns.append(G)
            state = env.reset()

            pbar.set_description(
                f'Episode: {i_episode} | Steps: {t_episode + 1} | Return: {G:5.2f} | Epsilon: {eps:4.2f}'
            )

            G = 0
            i_episode += 1
            t_episode = 0

        else:


            state = next_state
            t_episode += 1


    return (
        np.array(returns),
        np.array(losses),
    )
```

▾ CartPole

gamma = 0.99

num_steps = 1000 # episods

replay_size = 1000

replay_prepopulate_steps = 1000

batch_size = 64

Hyperparameters

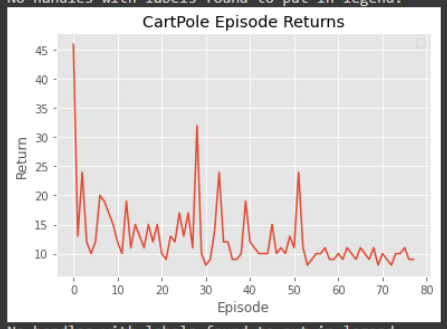
```
[36] env = envs['cartpole']

    gamma = 0.99
    num_steps = 1000 # episods
    replay_size = 1000
    replay_prepopulate_steps = 1000
    batch_size = 64

    exploration = ExponentialDecay(1.0, 0.05, 1000)

    returns,  losses = train_dqn(
        env,
        num_steps,
        replay_size=replay_size,
        replay_prepopulate_steps=replay_prepopulate_steps,
        batch_size=batch_size,
        exploration=exploration,
        gamma=gamma,
    )
```

```
    assert len(dqn_models) == num_saves
    assert all(isinstance(value, DQN) for value in dqn_models.values())

    # saving computed models to disk
    checkpoint = {key: dqn.custom_dump() for key, dqn in dqn_models.items()}
    torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

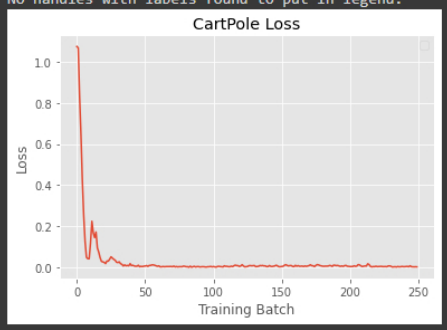
```
Episode: 77 | Steps: 9 | Return: 9.00 | Epsilon: 0.05: 100% 1000/1000 [00:14<00:00, 74.03it/s]
```

```
### PLOTTING
window_size = 1000
plt.plot(returns)
plt.title("CartPole Episode Returns")
plt.xlabel("Episode")
plt.ylabel("Return")
plt.legend()
plt.show()

plt.plot(losses)
plt.title("CartPole Loss")
plt.xlabel("Training Batch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

```
No handles with labels found to put in legend.
```




```
No handles with labels found to put in legend.
```



LunarLander

gamma = 0.99

num_steps = 1000 # episods

replay_size = 1000

replay_prepopulate_steps = 1000

batch_size = 64

Hyperparameters

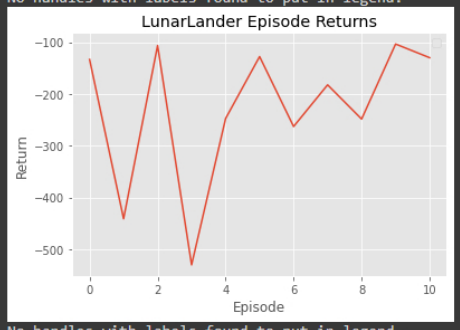
```
[32] env = envs['lunarlander']

    gamma = 0.99
    num_steps = 1000
    replay_size = 1000
    replay_prepopulate_steps = 1000
    batch_size = 64

    exploration = ExponentialDecay(1.0, 0.05, 1000)


    dqn_models, returns, lengths, losses = train_dqn(
        env,
        num_steps,
        replay_size=replay_size,
        replay_prepopulate_steps=replay_prepopulate_steps,
        batch_size=batch_size,
        exploration=exploration,
        gamma=gamma,
    )
```

```
    assert len(dqn_models) == num_saves
    assert all(isinstance(value, DQN) for value in dqn_models.values())

    # saving computed models to disk
    checkpoint = {key: dqn.custom_dump() for key, dqn in dqn_models.items()}
    torch.save(checkpoint, f'checkpoint_{env.spec.id}.pt')
```

```
Episode: 10 | Steps: 102 | Return: -129.93 | Epsilon: 1.00: 100% 1000/1000 [00:15<00:00, 85.07it/s]
```

```
[33] ### PLOTTING
     window_size = 1000
     plt.plot(returns)
     plt.title("LunarLander Episode Returns")
     plt.xlabel("Episode")
     plt.ylabel("Return")
     plt.legend()
     plt.show()

     plt.plot(losses)
     plt.title("LunarLander Loss")
     plt.xlabel("Training Batch")
     plt.ylabel("Loss")
     plt.legend()
     plt.show()
```

```
No handles with labels found to put in legend.
```




```
No handles with labels found to put in legend.
```