

ASSIGNMENT 3

Submission Deadline: 20/11/21, 9 AM

Submission Link: <https://forms.gle/fq7g53adYcZW7UMz8>

Name: Shiven Tripathi

Roll No.: 190816

IITK EMail: shiven@iitk.ac.in

Read all the instructions below carefully before you start working on the assignment.

- The purpose of this course is that you learn RL and the best way to do that is by implementation and experimentation.
- The assignment requires you to implement some algorithms and you are required to report your findings after experimenting with those algorithms.
- **You are required to submit ZIP file containing a Jupyter notebook (.ipynb), and an image folder. The notebook would include the code, graphs/plots of the experiments you run and your findings/observations. Image folder is the folder having plots, images, etc.**
- In case you use any maths in your explanations, render it using latex in the Jupyter notebook.
- You are expected to implement algorithms on your own and not copy it from other sources/class mates. Of course, you can refer to lecture slides.
- If you use any reference or material (including code), please cite the source, else it will be considered plagiarism. But referring to other sources that directly solve the problems given in the assignment is not allowed. There is a limit to which you can refer to outside material.
- This is an individual assignment.
- In case your solution is found to have an overlap with solution by someone else (including external sources), all the parties involved will get zero in this and all future assignments plus further more penalties in the overall grade. We will check not just for lexical but also semantic overlap. Same applies for the code as well. Even an iota of cheating would NOT be tolerated. If you cheat one line or cheat one page the penalty would be same.
- Be a smart agent, think long term, if you cheat we will discover it somehow, the price you would be paying is not worth it.
- In case you are struggling with the assignment, seek help from TAs. Cheating is not an option! I respect honesty and would be lenient if you are not able to solve some questions due to difficulty in understanding. Remember we are there to help you out, seek help if something is difficult to understand.
- The deadline for the submission is given above. Submit at least 30 minutes before the deadline, lot can happen at the last moment, your internet can fail, there can be a power failure, you can be abducted by aliens, etc.
- You have to submit your assignment via the Google Form (link above)
- The form would close after the deadline and we will not accept any solution. No reason what-so-ever would be accepted for not being able to submit before the deadline.
- Since the assignment involves experimentation, reporting your results and observations, there is a lot of scope for creativity and innovation and presenting new perspectives. Such efforts would be highly appreciated and accordingly well rewarded. Be an exploratory agent!
- Your code should be very well documented, there are marks for that.
- In your plots, have a clear legend and clear lines, etc. Of course you would be generating the plots in your code but you must also put these plots in your notebook. Generate high resolution pdf/svg version of the plots so that it doesn't pixelate on zooming.
- For all experiments, report about the seed used in the code documentation, write about the seed used.
- In your notebook write about all things that are not obvious from the code e.g., if you have made any assumptions, references/sources, running time, etc.
- **DO NOT Forget to write name, roll no and email details above**
- For each of the sub-part in the question create a new cell below the question and put your answer in there

- For each of the sub-part in the question create a new cell below the question and put your answer in there.
This includes the plots as well

In [1224]:

```
# all imports go in here
import gym
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from tqdm import tqdm
import time as time
```

OpenAI Gym Environments

In this assignment we will be exploring Deep RL algorithms and for this we will be using environments provided by OpenAI Gym. In particular we will be exploring "CartPole-v0" and "MountainCar-v0" environments (<https://gym.openai.com/docs/>). The code to instantiate the environments are given in the cells below. Run these cells and play with the environments to learn more details about the environments.

In [3]:

```
# Create CartPole environment
#https://github.com/openai/gym/wiki/CartPole-v0

env = gym.make('CartPole-v0')
env.seed(34)
s = env.reset()
print("Observation Space = ")
print(env.observation_space)
print("Action Space = ")
print(env.action_space)
done = False
# for episode in range(20):
#     print("In episode {}".format(episode))
#     for i in range(100):
#         env.render()
#         print(s)
#         a = env.action_space.sample()
#         s, r, done, _ = env.step(a)
#         if done:
#             print("Finished after {} timestep".format(i+1))
env.close()
```

Observation Space =
Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
Action Space =
Discrete(2)

In [4]:

```
# Create MountainCar environment:
# https://github.com/openai/gym/wiki/MountainCar-v0

env = gym.make('MountainCar-v0')
env.seed(45)
s = env.reset()
print("Observation Space = ")
print(env.observation_space)
print("Action Space = ")
print(env.action_space)
done = False
# for episode in range(20):
```

```
#         print("In episode {}".format(episode))
#         for i in range(100):
#             env.render()
#             print(s)
#             a = env.action_space.sample()
#             s, r, done, _ = env.step(a)
#             if done:
#                 print("Finished after {} timestep".format(i+1))
env.close()
```

```
Observation Space =
Box([-1.2 -0.07], [0.6 0.07], (2,), float32)
Action Space =
Discrete(3)
```

In [5]:

```
mcar = gym.make('MountainCar-v0')
cpole = gym.make('CartPole-v0')

print(mcar.observation_space)
print(mcar.action_space)

print(cpole.observation_space)
print(cpole.action_space)

Box([-1.2 -0.07], [0.6 0.07], (2,), float32)
Discrete(3)
Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.40282
35e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
Discrete(2)
```

Deep Value Based RL agents.

The purpose of this assignment is to learn about different Deep Value Based RL agents.

In this assignment you will be implementing Deep RL algorithms we learnt in Lectures 12, 13, 14 and 15. Namely, we will be implementing NFQ, DQN, Double DQN (DDQN), Duelling Double DQN (D3QN), and Duelling Double DQN with Prioritized Experience Replay (D3QN-PER). For all the algorithms below, this time we will not be specifying the hyper-parameters, please play with the hyper-params to come up with the best values. This way you will learn to tune the model. Some of the values were specified in the lecture, that would be a good starting point. Your aim is to develop the best NFQ/DQN/DDQN/D3QN/D3QN-PER agent for each of the setting.

For those of you who follow TEDEd, here is an interesting video by TED on DQN and Atari Games:

<https://www.youtube.com/watch?v=PP8Zc778B8s>

Also since these environments are available in OpenAI Gym, there are public leaderboards

(<https://github.com/openai/gym/wiki/Leaderboard>) for each of these environments. Compare where does your agent stand on these leaderboard for each of these environments, try to tune your agents so that it is on the top of the leaderboard. In fact, if your agent performs well on these environments, you can also make your entry on the leaderboard.

All your hyperparameters should be stated here. We will change their value here and your code should work accordingly.

all hyperparameters provided in the individual code block used for testing

In []:

```
#
```

```
# gamma =
# epsilon = #epsilon greedy strategy
# temp = #softmax strategy
# delta = #huber loss
# tau = #D3QN
# alpha = #D3QN-PER
# beta = #D3QN-PER
# beta_rate = #D3QN-PER
# MAX_TRAIN_EPISODES =
# MAX_EVAL_EPISODES =
```

In [1248]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Helper Functions

In [1249]:

```
def selectGreedyAction(net, state):
    #this function gets q-values via the network and selects greedy action from q-values and returns it

    Q_values = net(torch.from_numpy(state).float()).detach().numpy()
    greedyAction = np.argmax(Q_values)
    return greedyAction
```

In [1276]:

```
def selectEpsilonGreedyAction(net, state):
    #this function gets q-values via the network and selects an action from q-values using epsilon greedy strategy
    #and returns it
    #note this function can be used for decaying epsilon greedy strategy,
    #you would need to create a wrapper function that will handle decaying epsilon
    #you can create this wrapper in this helper function section
    #for the agents you would be implementing it would be nice to play with decaying parameter to get optimal results

    Q_values = net(torch.from_numpy(state).float()).detach().numpy()
    n_actions = np.shape(Q_values)[0]
    if np.random.rand() < epsilon:
        eGreedyAction = np.random.randint(n_actions)
    else:
        eGreedyAction = np.argmax(Q_values)

    return eGreedyAction
```

In [1251]:

```
def selectSoftMaxAction(net, state, temp=0.2):
    #this function gets q-values via the network and selects an action from q-values using softmax strategy
    #and returns it
    #note this function can be used for decaying temperature softmax strategy,
    #you would need to create a wrapper function that will handle decaying temperature
    #you can create this wrapper in this helper function section
    #for the agents you would be implementing it would be nice to play with decaying parameter to get optimal results

    Q_values = net(torch.from_numpy(state).float()).detach().numpy()
    probs = np.exp(Q_values/temp)/np.sum(np.exp(Q_values/temp))
    softAction = np.random.choice(np.arange(len(probs)), p=probs)
    return softAction
```

In [1252]:

```
#Value Network
def createValueNetwork(inDim, outDim, hDim = [64,64], activation = F.relu):
```

```

#this creates a Feed Forward Neural Network class and instantiates it and returns the class
#the class should be derived from torch nn.Module and it should have init and forward method at the very least
#the forward function should return q-value for each possible action

class ValueNetwork(nn.Module):
    def __init__(self):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(inDim, hDim[0])
        self.fc2 = nn.Linear(hDim[0], hDim[1])
        self.fc3 = nn.Linear(hDim[1], outDim)

    def forward(self, x):
        x = x.to(device)
        x = activation(self.fc1(x))
        x = activation(self.fc2(x))
        x = self.fc3(x)
        return x

valueNetwork = ValueNetwork().to(device)
return valueNetwork

```

In [1253]:

```

#Dueling Network
def createDuelingNetwork(inDim, outDim, hDim = [64,64], activation = F.relu):
    #this creates a Feed Forward Neural Network class and instantiates it and returns the class
    #the class should be derived from torch nn.Module and it should have init and forward method at the very least
    #the forward function should return q-value which is derived
    #internally from action-advantage function and v-function,
    #Note we center the advantage values, basically we subtract the mean from each state-action value

    class DuelingNetwork(nn.Module):
        def __init__(self):
            super(DuelingNetwork, self).__init__()
            self.fc1 = nn.Linear(inDim, hDim[0])
            self.fc2 = nn.Linear(hDim[0], hDim[1])
            self.value = nn.Linear(hDim[1], 1)
            self.advantage = nn.Linear(hDim[1], outDim)

        def forward(self, x):
            x = x.to(device)
            x = activation(self.fc1(x))
            x = activation(self.fc2(x))
            value = self.value(x)
            advantage = self.advantage(x)
            q_value = value + advantage - advantage.mean()
            return q_value

    duelNetwork = DuelingNetwork().to(device)
    return duelNetwork

```

Function completed in the last cells with the plots and explanations.

In [1296]:

```

def plotQuantity(quantityListDict, totalEpisodeCount, descriptionList):
    # this function takes in the quantityListDict and plots quantity vs episodes.
    # quantityListDict = {envInstanceCount: quantityList}
    # quantityList is list of the quantity per episode,
    # for example it could be mean reward per episode, traintime per episode, etc.
    #
    # NOTE: len(quantityList) == totalEpisodeCount
    #
    # Since we run multiple instances of the environment, there will be variance across environments
    # so in the plot, you will plot per episode maximum, minimum and average value across all env instances
    # Basically, you need to envelop (e.g., via color) the quantity between max and min

```

```

with mean value in between
#
#
#     #use the descriptionList parameter to put legends, title, etc.
#     #For each of the plot, create the legend on the left/right side so that it doesn't
#     #overlay on the plot lines/envelop.
#
#     #this is a generic function and can be used to plot any of the quantity of interest
#     #In particular we will be using this function to plot:
#     #         mean train rewards vs episodes
#     #         mean evaluation rewards vs episodes
#     #         total steps vs episode
#     #         train time vs episode
#     #         wall clock time vs episode
#
#
#     #this function doesn't return anything
#     for description in descriptionList:
#         quantityListList = quantityListDict[description]
#         plt.plot(np.arange(totalEpisodeCount), np.mean(quantityListList,axis=0), label=
description)
#         # plt.plot(np.arange(totalEpisodeCount), np.min(quantityListList,axis=0), label
=description)
#         # plt.plot(np.arange(totalEpisodeCount), np.max(quantityListList,axis=0), label
=description)
#         plt.fill_between(np.arange(totalEpisodeCount), np.max(quantityListList, axis=0)
, np.min(quantityListList, axis=0), alpha=0.5)
#         plt.legend()

```

In [1291]:

```

def huberLoss(error):
    #this function calculates the huber loss for the error using the delta parameter
    if torch.linalg.norm(error) <= delta:
        hLoss = 0.5*error**2
    else:
        hLoss = error*delta - 0.5*delta**2
    return torch.mean(hLoss)

```

In [1256]:

```

def copyNetworks(nnOnline, nnTarget):
    nnTarget.load_state_dict(nnOnline.state_dict())

```

Do not change any Class/Methods definition. We have split the class methods across cells for code readability purposes. This requires to inherit the same class, please do not change it.

ReplayBuffer

In next few cells, you will implement replaybuffer class.

This class creates a buffer for storing and retrieving experiences. This is a generic class and can be used for different agents like NFQ, DQN, DDQN, PER_DDQN, etc. following are the methods for this class which are implemented in subsequent cells

```

class ReplayBuffer():
    def __init__(self, bufferSize, **kwargs):
    def store(self, experience)
    def update(self, indices, priorities)
    def collectExperiences(env, state, explorationStrategy, net = None)
    def sample(self, batchSize, **kwargs)
    def splitExperiences(self, experiences)
    def length(self)

```

In [1257]:

```
from collections import deque

class ReplayBuffer():
    def __init__(self, bufferSize, bufferType = 'DQN', **kwargs):
        # this function creates the relevant data-structures, and initializes all relevant variables
        # it can take variable number of parameters like alpha, beta, beta_rate (required for PER)
        # here the bufferType variable can be used to maintain one class for all types of agents
        # using the bufferType parameter in the methods below, you can implement all possible functionalities
        # that could be used for different types of agents
        # permissible values for bufferType = NFQ, DQN, DDQN, D3QN and PER-D3QN

        self.buffer = deque(maxlen = bufferSize)
        self.bufferSize = bufferSize
        self.bufferType = bufferType
        self.alpha = kwargs.get('alpha', 0.6)
        self.beta = kwargs.get('beta', 0.4)
        self.beta_rate = kwargs.get('beta_rate', 0.001)
```

In [1258]:

```
class ReplayBuffer(ReplayBuffer):
    def store(self, experience):
        #stores the experiences, based on parameters in init it can assign priorities, etc.

        if self.length() >= self.bufferSize:
            self.buffer.popleft()
        self.buffer.append(experience+(1,))
```

In [1489]:

```
class ReplayBuffer(ReplayBuffer):
    def update(self, indices, priorities):
        #this is mainly used for PER-DDQN
        #otherwise just have a pass in this method
        #
        #this function does not return anything
        #
        if self.bufferType == 'PER-D3QN':
            for i in range(len(indices)):
                temp = list(self.buffer[indices[i]])
                temp[5] = priorities[i]
                self.buffer[indices[i]] = tuple(temp)
```

In [1490]:

```
class ReplayBuffer(ReplayBuffer):
    def collectExperiences(self, env, state, explorationStrategy, countExperiences, net = None):
        #this method allows the agent to interact with the environment starting from a state and it collects
        #experiences during the interaction, it uses network to get the value function and uses exploration strategy
        #to select action. It collects countExperiences and in case the environment terminates before that it returns
        #the function calling this method needs to handle early termination accordingly.
        #
        #this function does not return anything
        #
        r, steps = 0,0
        for _ in range(countExperiences):
            action = explorationStrategy(net, state)
            nextState, reward, done, _ = env.step(action)
            experience = (state, action, reward, nextState, done)
            self.store(experience)
```

```

        r += reward
        steps += 1
        if done:
            state = env.reset()
            break
        else:
            state = nextState
    return r, steps

```

In [1520]:

```

class ReplayBuffer(ReplayBuffer):
    def sample(self, batchSize, **kwargs):
        # this method returns batchSize number of experiences
        # based on extra arguments, it could do sampling or it could return the latest batchSize experiences or
        # via some other strategy
        #
        # in the case of Prioritized Experience Replay (PER) the sampling needs to take into account the priorities
        #
        # this function returns experiences samples
        #
        if self.bufferType == 'PER-D3QN':
            probs = np.asarray([float(x[5])**self.alpha for x in self.buffer])
            indices = np.random.choice(len(self.buffer), batchSize, p = probs/float(np.sum(probs)))
            experiencesList = []
            maxi = -1
            for i in indices:
                temp = list(self.buffer[i])
                temp[5] = torch.pow(temp[5], torch.from_numpy(np.asarray(-self.beta)).float().to(device))
                temp.append(i)
                maxi = max(maxi, temp[5])
                experiencesList.append(temp)
            for i in range(len(experiencesList)):
                experiencesList[i][5] = torch.div(experiencesList[i][5], maxi)
        else:
            indices = np.random.choice(len(self.buffer), batchSize)
            experiencesList = [self.buffer[i] for i in indices]

        return experiencesList

```

In [1503]:

```

class ReplayBuffer(ReplayBuffer):
    def splitExperiences(self, experiences):

        states = torch.from_numpy(np.vstack([experience[0] for experience in experiences])).float().to(device)
        actions = torch.from_numpy(np.vstack([experience[1] for experience in experiences])).int().to(device)
        rewards = torch.from_numpy(np.vstack([experience[2] for experience in experiences])).float().to(device)
        nextStates = torch.from_numpy(np.vstack([experience[3] for experience in experiences])).float().to(device)
        dones = torch.from_numpy(np.vstack([experience[4] for experience in experiences])).long().to(device)
        if self.bufferType == 'PER-D3QN':
            priorities = torch.from_numpy(np.vstack([experience[5] for experience in experiences])).float().to(device)
            indices = torch.from_numpy(np.vstack([experience[6] for experience in experiences])).int().to(device)
            return states, actions, rewards, nextStates, dones, priorities, indices
        else:
            return states, actions, rewards, nextStates, dones

```

In [1504]:

```

class ReplayBuffer(ReplayBuffer):

```



```
def length(self):
    #tells the number of experiences stored in the internal buffer

    return len(self.buffer)
```

Neural Fitted Q (NFQ)

Implement the Neural Fitted Q algorithm. We have studied about NFQ algorithm in Lecture 12. Use the function definitions (given below).

This class implements the NFQ Agent, you are required to implement the various methods of this class as outlined below. Note this class is generic and should work with any permissible OpenAI Gym environment. Also please feel free to play with different exploration strategies with decaying paramters (epsilon/temperature)

```
class NFQ():
    def __init__(env, seed, gamma, epochs,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn)

    def initBookKeeping(self)
    def performBookKeeping(self, train = True)
    def runNFQ(self)
    def trainAgent(self)
    def trainNetwork(self, experiences, epochs)
    def evaluateAgent(self)
```

In [1265]:

```
class NFQ():
    def __init__(self, env, seed, gamma, epochs,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn):

        #this NFQ method
        # 1. creates and initializes (with seed) the environment, train/eval episodes, gamma, etc.
        # 2. creates and intializes all the variables required for book-keeping values via the initBookKeeping method
        # 3. creates Q-network using the createValueNetwork above
        # 4. creates and initializes (with network params) the optimizer function
        # 5. sets the explorationStartegy variables/functions for train and evaluation
        # 6. sets the batchSize for the number of experiences
        # 7. Creates the replayBuffer

        env = gym.make(env)
        self.env = env
        self.seed = seed
        self.env.seed(self.seed)
        self.gamma = gamma
        self.epochs = epochs

        self.MAX_TRAIN_EPISODES = MAX_TRAIN_EPISODES
        self.MAX_EVAL_EPISODES = MAX_EVAL_EPISODES
        self.explorationStrategyTrainFn = explorationStrategyTrainFn
        self.explorationStrategyEvalFn = explorationStrategyEvalFn
```

```

        self.q_net = createValueNetwork(inDim=env.observation_space.shape[0], outDim=env
.action_space.n, hDim=[64,128])

        self.optimizerLR = optimizerLR
        self.optimizerFn = optimizerFn(self.q_net.parameters(), lr=self.optimizerLR)

        self.bufferSize = bufferSize
        self.batchSize = batchSize
        self.rBuffer = ReplayBuffer(bufferSize=self.bufferSize, bufferType='NFQ')

```

In [1266]:

```

class NFQ(NFQ):
    def initBookKeeping(self):
        #this method creates and initializes all the variables required for book-keeping v
alues and it is called
        #init method
        self.trainRewardsList = []
        self.trainTimeList = []
        self.evalRewardsList = []
        self.wallClockTimeList = []
        self.loss = []

```

In [1267]:

```

class NFQ(NFQ):
    def performBookKeeping(self, train = True):
        #this method updates relevant variables for the bookKeeping, this can be called
#multiple times during training
        #if you want you can print information using this, so it may help to monitor prog
ress and also help to debug
        #
        if train:
            self.wallClockTimeList.append(time.time())

```

In [1298]:

```

class NFQ(NFQ):
    def runNFQ(self):
        #this is the main method, it trains the agent, performs bookkeeping while trainin
g and finally evaluates
        #the agent and returns the following quantities:
        #1. episode wise mean train rewards
        #2. episode wise mean eval rewards
        #2. episode wise trainTime (in seconds): time elapsed during training since the s
tart of the first episode
        #3. episode wise wallClockTime (in seconds): actual time elapsed since the start
of training,
        #
        note this will include time for BookKeeping and
evaluation
        # Note both trainTime and wallClockTime get accumulated as episodes proceed.
        #
        #Your code goes in here
        #

        self.initBookKeeping()
        self.trainAgent()
        self.finalEvalReward = self.evaluateAgent()
        return self.trainRewardsList, self.trainTimeList, self.evalRewardsList, self.wal
lClockTimeList, self.finalEvalReward, self.loss

```

In [1616]:

```

class NFQ(NFQ):
    def trainAgent(self):
        #this method collects experiences and trains the agent and does BookKeeping whil
e training.
        #this calls the trainNetwork() method internally, it also evaluates the agent per
episode
        #it trains the agent for MAX_TRAIN_EPISODES
        #

```

```

        for e in tqdm(range(self.MAX_TRAIN_EPISODES)):
            s = self.env.reset()
            r, steps = self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTrainFn, self.batchSize*100, net = self.q_net)
            self.trainRewardsList.append(r)
            self.trainTimeList.append(steps)
            while self.rBuffer.length() < self.batchSize:
                self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTrainFn, self.batchSize*100, net = self.q_net)
            experiences = self.rBuffer.sample(self.batchSize)
            self.loss.append(self.trainNetwork(experiences))
            self.performBookKeeping(train = True)
            self.evalRewardsList.append(np.mean(self.evaluateAgent()))

```

In [1617]:

```

class NFQ(NFQ):
    def trainNetwork(self, experiences, epochs=None):
        # this method trains the value network epoch number of times and is called by the trainAgent function
        # it essentially uses the experiences to calculate target, using the targets it calculates the error, which
        # is further used for calculating the loss. It then uses the optimizer over the loss
        # to update the params of the network by backpropagating through the network
        # this function does not return anything
        # you can try out other loss functions other than MSE like Huber loss, MAE, etc.
        #
        s, a, r, s_, t = self.rBuffer.splitExperiences(experiences)
        r = (r-r.mean())/(r.std()+1e-7)
        # s = (s-s.mean(dim=0))/(s.std(dim=0)+1e-7)
        max_a_qs = self.q_net(s_).max(1)[0].unsqueeze(1)
        tdTargets = r + self.gamma * max_a_qs * (1 - t)
        tdErrors = tdTargets - self.q_net(s).gather(1,a.long())
        loss = huberLoss(error=tdErrors).to(device)
        # loss = F.smooth_l1_loss(tdErrors, torch.zeros_like(tdErrors)).to(device)
        self.optimizerFn.zero_grad()
        loss.backward()
        for param in self.q_net.parameters():
            param.grad.data.clamp_(-1, 1)
        self.optimizerFn.step()
        return loss.item()

```

In [1618]:

```

class NFQ(NFQ):
    def evaluateAgent(self):
        #this function evaluates the agent using the value network, it evaluates agent for MAX_EVAL_EPISODES
        #typically MAX_EVAL_EPISODES = 1
        #
        finalEvalRewardsList = []
        for _ in range(self.MAX_EVAL_EPISODES):
            s = self.env.reset()
            done = False
            rs = 0
            while not done:
                a = self.explorationStrategyEvalFn(self.q_net, s)
                s, r, done, _ = self.env.step(a)
                rs += r
            finalEvalRewardsList.append(rs)
        return finalEvalRewardsList

```

In [1619]:

```

epsilon = 0.5
delta = 1

```

```

nfq = NFQ(env='CartPole-v0', seed=0, gamma=0.99, epochs=10**3, bufferSize=10**4, batchSize=256,

```

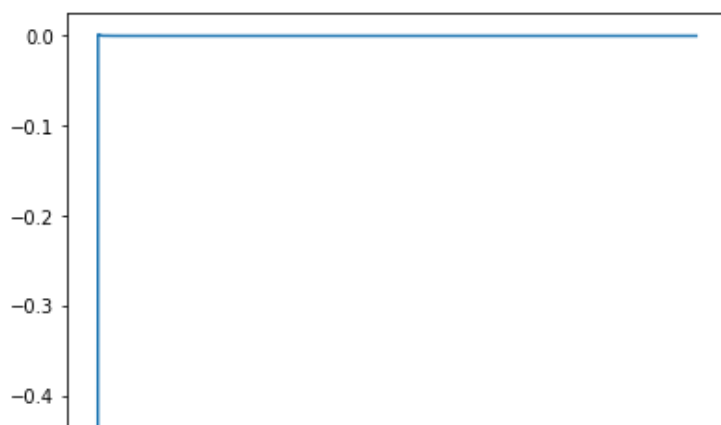
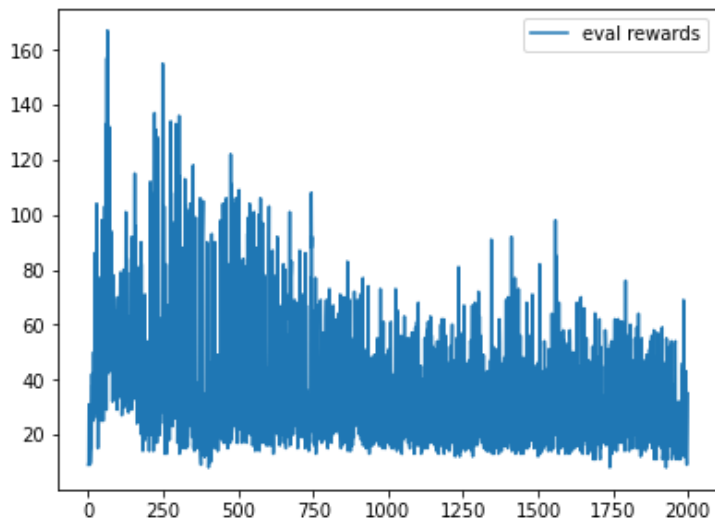
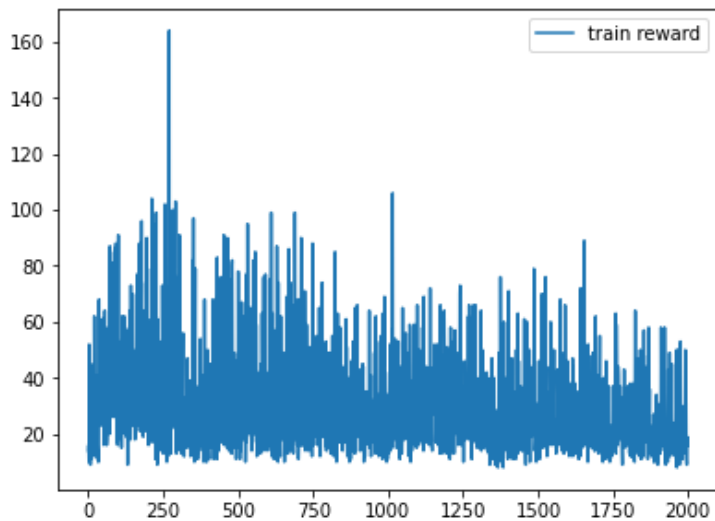
```
optimizerFn=optim.Adam, optimizerLR=7*1e-4,  
MAX_TRAIN_EPISODES=2000, MAX_EVAL_EPISODES=10**0,  
explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=  
selectGreedyAction)
```

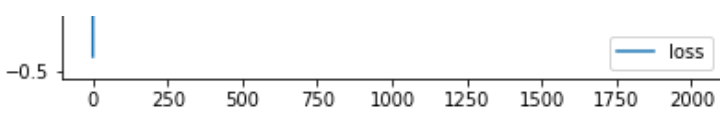
```
trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, los  
s = nfq.runNFQ()
```

100%|██████████| 2000/2000 [00:32<00:00, 60.98it/s]

In [1620]:

```
plt.plot(np.arange(len(trainRewardsList)), trainRewardsList, label='train reward')  
plt.legend()  
plt.show()  
plt.plot(np.arange(len(evalRewardsList)), evalRewardsList, label='eval rewards')  
plt.legend()  
plt.show()  
plt.plot(np.arange(len(loss)), loss, label='loss')  
plt.legend()  
plt.show()
```





Deep Q-Network (DQN)

Implement the Deep Q algorithm. We have studied about DQN algorithm in Lecture 13. Use the function definitions (given below).

This class implements the DQN Agent, you are required to implement the various methods of this class as outlined below. Note this class is generic and should work with any permissible OpenAI Gym environment

```
class DQN():
    def __init__(env, seed, gamma,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency)
    def initBookKeeping(self)
    def performBookKeeping(self, train = True)
    def runDQN(self)
    def trainAgent(self)
    def trainNetwork(self, experiences)
    def updateNetwork(self, onlineNet, targetNet)
    def evaluateAgent(self)
```

In [1308]:

```
class DQN():
    def __init__(self, env, seed, gamma,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency):
        #this DQN method
        # 1. creates and initializes (with seed) the environment, train/eval episodes, gamma, etc.
        # 2. creates and intializes all the variables required for book-keeping values via the initBookKeeping method
        # 3. creates traget and online Q-networks using the createValueNetwork above
        # 4. creates and initializes (with network params) the optimizer function
        # 5. sets the explorationStartegy variables/functions for train and evaluation
        # 6. sets the batchSize for the number of experiences
        # 7. Creates the replayBuffer

        env = gym.make(env)
        self.env = env
        self.seed = seed
        self.env.seed(self.seed)
        self.gamma = gamma
        self.tau = 1

        self.MAX_TRAIN_EPISODES = MAX_TRAIN_EPISODES
        self.MAX_EVAL_EPISODES = MAX_EVAL_EPISODES
        self.explorationStrategyTrainFn = explorationStrategyTrainFn
```

```

        self.explorationStrategyEvalFn = explorationStrategyEvalFn
        self.updateFrequency = updateFrequency

        self.nnTarget = createValueNetwork(inDim=env.observation_space.shape[0], outDim=
env.action_space.n, hDim=[64,128])
        self.nnTarget.eval()
        self.nnOnline = createValueNetwork(inDim=env.observation_space.shape[0], outDim=
env.action_space.n, hDim=[64,128])

        self.optimizerLR = optimizerLR
        self.optimizerFn = optimizerFn(self.nnOnline.parameters(), lr=self.optimizerLR)

        self.bufferSize = bufferSize
        self.batchSize = batchSize
        self.rBuffer = ReplayBuffer(bufferSize=self.bufferSize, bufferType='DQN')

```

In [1309]:

```

class DQN(DQN):
    def initBookKeeping(self):
        #this method creates and intializes all the variables required for book-keeping v
alues and it is called
        self.trainRewardsList = []
        self.trainTimeList = []
        self.evalRewardsList = []
        self.wallClockTimeList = []
        self.loss = []

```

In [1310]:

```

class DQN(DQN):
    def performBookKeeping(self, train = True):
        #this method updates relevant variables for the bookKeeping, this can be called
#multiple times during training
#if you want you can print information using this, so it may help to monitor prog
ress and also help to debug
        #
        if train:
            self.wallClockTimeList.append(time.time())

```

In [1311]:

```

class DQN(DQN):
    def runDQN(self):
        #this is the main method, it trains the agent, performs bookkeeping while trainin
g and finally evaluates
#the agent and returns the following quantities:
#1. episode wise mean train rewards
#2. episode wise mean eval rewards
#2. episode wise trainTime (in seconds): time elapsed during training since the s
tart of the first episode
#3. episode wise wallClockTime (in seconds): actual time elapsed since the start
of training,
#
note this will include time for BookKeeping and
evaluation
# Note both trainTime and wallClockTime get accumulated as episodes proceed.
#
#Your code goes in here
#
        self.initBookKeeping()
        self.trainAgent()
        self.finalEvalReward = self.evaluateAgent()
        return self.trainRewardsList, self.trainTimeList, self.evalRewardsList, self.wal
lClockTimeList, self.finalEvalReward, self.loss

```

In [1654]:

```

class DQN(DQN):
    def trainAgent(self):
        #this method collects experiences and trains the agent and does BookKeeping whil
e training.

```

```

#this calls the trainNetwork() method internally, it also evaluates the agent per
episode
#it trains the agent for MAX_TRAIN_EPISODES
#
self.updateNetwork(self.nnOnline, self.nnTarget)
for e in tqdm(range(self.MAX_TRAIN_EPISODES)):
    s = self.env.reset()
    r, steps = self.rBuffer.collectExperiences(self.env, s, self.explorationStra
tegyTrainFn, self.batchSize*10, net = self.nnOnline)
    self.trainRewardsList.append(r)
    self.trainTimeList.append(steps)
    while self.rBuffer.length() < self.bufferSize:
        self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTra
inFn, self.batchSize*10, net = self.nnOnline)
    experiences = self.rBuffer.sample(self.batchSize)
    self.loss.append(self.trainNetwork(experiences))
    self.performBookKeeping(train = True)
    self.evalRewardsList.append(np.mean(self.evaluateAgent()))

    if e%self.updateFrequency == 0:
        self.updateNetwork(self.nnOnline, self.nnTarget)

```

In [1655]:

```

class DQN(DQN):
    def trainNetwork(self, experiences, epochs=None):
        # this method trains the value network epoch number of times and is called by the
trainAgent function
        # it essentially uses the experiences to calculate target, using the targets it c
alculates the error, which
        # is further used for calculating the loss. It then uses the optimizer over the lo
ss

        # to update the params of the network by backpropagating through the network
        # this function does not return anything
        # you can try out other loss functions other than MSE like Huber loss, MAE, etc.
        #

        s, a, r, s_, t = self.rBuffer.splitExperiences(experiences)
        r = (r-r.mean())/(r.std()+1e-7)
        # s = (s-s.mean(dim=0))/(s.std(dim=0)+1e-7)
        with torch.no_grad():
            max_a_qs = self.nnTarget(s_).max(1)[0].unsqueeze(1).detach()
            tdTargets = r + self.gamma * max_a_qs * (1 - t)
            tdErrors = tdTargets - self.nnOnline(s).gather(1,a.long())
            # loss = huberLoss(error=tdErrors).to(device)
            loss = F.smooth_l1_loss(tdErrors, torch.zeros_like(tdErrors)).to(device)
            self.optimizerFn.zero_grad()
            loss.backward()
            for param in self.nnOnline.parameters():
                param.grad.data.clamp_(-1, 1)
            self.optimizerFn.step()
        return loss.item()

```

In [1656]:

```

class DQN(DQN):
    def updateNetwork(self, onlineNet, targetNet):
        #this function updates the onlineNetwork with the target network using Polyak ave
raging
        #
        if self.tau is None:
            self.tau = 1.0
        online = onlineNet.state_dict()
        target = targetNet.state_dict()

        for k in online.keys():
            target[k] = self.tau * online[k] + (1 - self.tau) * target[k]

        onlineNet.load_state_dict(target)

```

In [1657]:

```

class DQN(DQN):
    def evaluateAgent(self):
        #this function evaluates the agent using the value network, it evaluates agent for
        MAX_EVAL_EPISODES
        #typically MAX_EVAL_EPISODES = 1
        #
        finalEvalRewardsList = []
        for _ in range(self.MAX_EVAL_EPISODES):
            s = self.env.reset()
            done = False
            rs = 0
            while not done:
                a = self.explorationStrategyEvalFn(self.nnOnline, s)
                s, r, done, _ = self.env.step(a)
                rs += r
            finalEvalRewardsList.append(rs)
        return finalEvalRewardsList

```

In [1660]:

```

epsilon = 0.5
delta = 1

dqn = DQN(env='CartPole-v0', seed=0, gamma=0.99, bufferSize=10**4, batchSize=128,
          optimizerFn=optim.Adam, optimizerLR=5*1e-4,
          MAX_TRAIN_EPISODES=1000, MAX_EVAL_EPISODES=10**0,
          explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=
selectGreedyAction,
          updateFrequency=15)
trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = dqn.runDQN()

```

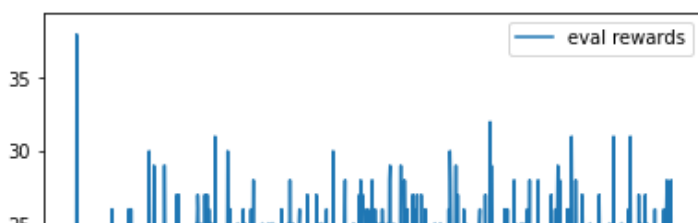
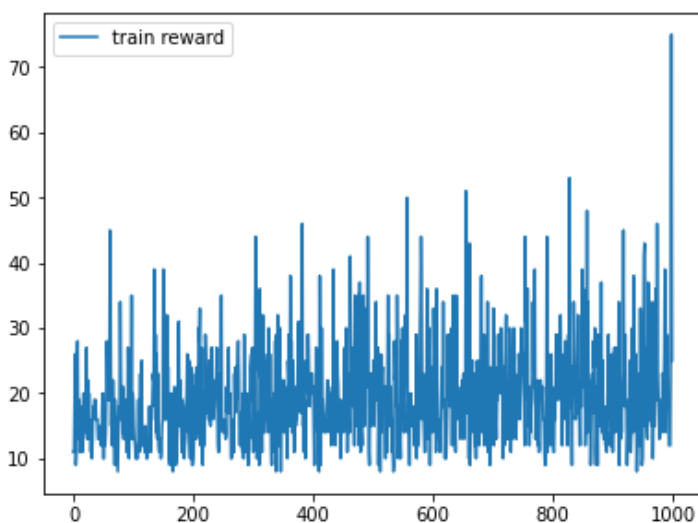
100%|██████████| 1000/1000 [00:12<00:00, 78.41it/s]

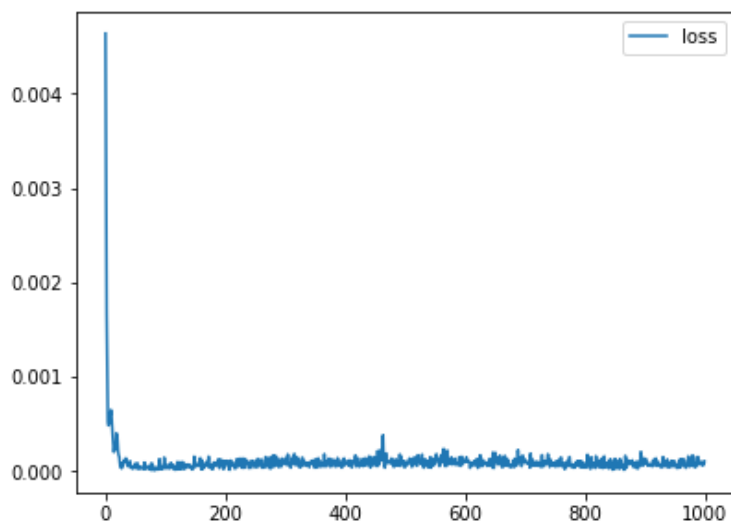
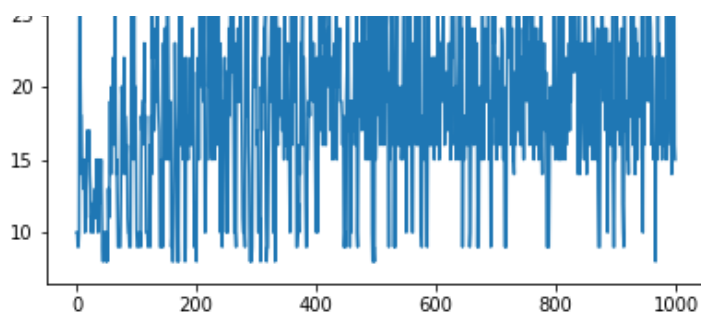
In [1661]:

```

plt.plot(np.arange(len(trainRewardsList)), trainRewardsList, label='train reward')
plt.legend()
plt.show()
plt.plot(np.arange(len(evalRewardsList)), evalRewardsList, label='eval rewards')
plt.legend()
plt.show()
plt.plot(np.arange(len(loss)), loss, label='loss')
plt.legend()
plt.show()

```





Double DQN (DDQN)

Implement the Double DQN agent. We have studied about Double DQN agent in Lecture 14. Use the function definitions (given below).

This class implements the Double DQN agent, you are required to implement the various methods of this class as outlined below. Note this class is generic and should work with any permissible OpenAI Gym environment

```
class DDQN():
    def __init__(env, seed, gamma,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency)
    def initBookKeeping(self)
    def performBookKeeping(self, train = True)
    def runDDQN(self)
    def trainAgent(self)
    def trainNetwork(self, experiences)
    def updateNetwork(self, onlineNet, targetNet)
    def evaluateAgent(self)
```

In [1662]:

```
class DDQN():
    def __init__(self, env, seed, gamma,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
```

```

        MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
        explorationStrategyTrainFn,
        explorationStrategyEvalFn,
        updateFrequency):
    #this DDQN method
    # 1. creates and initializes (with seed) the environment, train/eval episodes, gamma, etc.
    # 2. creates and initializes all the variables required for book-keeping values via the initBookKeeping method
    # 3. creates target and online Q-networks using the createValueNetwork above
    # 4. creates and initializes (with network params) the optimizer function
    # 5. sets the explorationStrategy variables/functions for train and evaluation
    # 6. sets the batchSize for the number of experiences
    # 7. Creates the replayBuffer

    env = gym.make(env)
    self.env = env
    self.seed = seed
    self.env.seed(self.seed)
    self.gamma = gamma
    self.tau = 1.0

    self.MAX_TRAIN_EPISODES = MAX_TRAIN_EPISODES
    self.MAX_EVAL_EPISODES = MAX_EVAL_EPISODES
    self.explorationStrategyTrainFn = explorationStrategyTrainFn
    self.explorationStrategyEvalFn = explorationStrategyEvalFn
    self.updateFrequency = updateFrequency

    self.nnTarget = createValueNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[128, 512])
    self.nnTarget.eval()
    self.nnOnline = createValueNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[128, 512])

    self.optimizerLR = optimizerLR
    self.optimizerFn = optimizerFn(self.nnOnline.parameters(), lr=self.optimizerLR)

    self.bufferSize = bufferSize
    self.batchSize = batchSize
    self.rBuffer = ReplayBuffer(bufferSize=self.bufferSize, bufferType='DDQN')

```

In [1663]:

```

class DDQN(DDQN):
    def initBookKeeping(self):
        #this method creates and initializes all the variables required for book-keeping values and it is called
        #init method
        #
        self.trainRewardsList = []
        self.trainTimeList = []
        self.evalRewardsList = []
        self.wallClockTimeList = []
        self.loss = []

```

In [1664]:

```

class DDQN(DDQN):
    def performBookKeeping(self, train = True):
        #this method updates relevant variables for the bookKeeping, this can be called multiple times during training
        #if you want you can print information using this, so it may help to monitor progress and also help to debug
        #
        if train:
            self.wallClockTimeList.append(time.time())

```

In [1665]:

```

class DDQN(DDQN):
    def runDDQN(self):

```

```

    #this is the main method, it trains the agent, performs bookkeeping while training and finally evaluates
    #the agent and returns the following quantities:
    #1. episode wise mean train rewards
    #2. episode wise mean eval rewards
    #2. episode wise trainTime (in seconds): time elapsed during training since the start of the first episode
    #3. episode wise wallClockTime (in seconds): actual time elapsed since the start of training,
    #
    # note this will include time for BookKeeping and evaluation
    # Note both trainTime and wallClockTime get accumulated as episodes proceed.

    self.initBookKeeping()
    self.trainAgent()
    self.finalEvalReward = self.evaluateAgent()
    return self.trainRewardsList, self.trainTimeList, self.evalRewardsList, self.wallClockTimeList, self.finalEvalReward, self.loss

```

In [1666]:

```

class DDQN(DDQN):
    def trainAgent(self):
        #this method collects experiences and trains the agent and does BookKeeping while training.
        #this calls the trainNetwork() method internally, it also evaluates the agent per episode
        #it trains the agent for MAX_TRAIN_EPISODES
        #
        self.updateNetwork(self.nnOnline, self.nnTarget)
        for e in tqdm(range(self.MAX_TRAIN_EPISODES)):
            s = self.env.reset()
            r, steps = self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTrainFn, self.batchSize*100, net = self.nnOnline)
            self.trainRewardsList.append(r)
            self.trainTimeList.append(steps)
            while self.rBuffer.length() < self.batchSize*5:
                self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTrainFn, self.batchSize*100, net = self.nnOnline)
            experiences = self.rBuffer.sample(self.batchSize)
            self.loss.append(self.trainNetwork(experiences))
            self.performBookKeeping(train = True)
            self.evalRewardsList.append(np.mean(self.evaluateAgent()))

            if e%self.updateFrequency == 0:
                self.updateNetwork(self.nnOnline, self.nnTarget)

```

In [1667]:

```

class DDQN(DDQN):
    def trainNetwork(self, experiences):
        # this method trains the value network epoch number of times and is called by the trainAgent function
        # it essentially uses the experiences to calculate target, using the targets it calculates the error, which
        # is further used for calculating the loss. It then uses the optimizer over the loss
        # to update the params of the network by backpropagating through the network
        # this function does not return anything
        # you can try out other loss functions other than MSE like Huber loss, MAE, etc.
        #
        s, a, r, s_, t = self.rBuffer.splitExperiences(experiences)
        r = (r-r.mean())/(r.std()+1e-7)
        argmax_a_qs = self.nnOnline(s_).argmax(dim=1).unsqueeze(1)
        with torch.no_grad():
            q_targets = self.nnTarget(s_).gather(1, argmax_a_qs).detach()
            tdTargets = r + self.gamma * q_targets * (1 - t)
            tdErrors = tdTargets - self.nnOnline(s).gather(1,a.long())
            loss = F.smooth_l1_loss(tdErrors, torch.zeros_like(tdErrors)).to(device)
            # loss = huberLoss(error=tdErrors)
            self.optimizerFn.zero_grad()

```

```

loss.backward()
torch.nn.utils.clip_grad_norm_(self.nnOnline.parameters(), max_norm=1)
self.optimizerFn.step()
return loss.item()

```

In [1668]:

```

class DDQN(DDQN):
    def updateNetwork(self, onlineNet, targetNet):
        #this function updates the onlineNetwork with the target network using Polyak averaging
        #
        online = onlineNet.state_dict()
        target = targetNet.state_dict()

        for k in online.keys():
            target[k] = self.tau * online[k] + (1 - self.tau) * target[k]

        onlineNet.load_state_dict(target)

```

In [1669]:

```

class DDQN(DDQN):
    def evaluateAgent(self):
        #this function evaluates the agent using the value network, it evaluates agent for MAX_EVAL_EPISODES
        #typically MAX_EVAL_EPISODES = 1
        finalEvalRewardsList = []
        for _ in range(self.MAX_EVAL_EPISODES):
            s = self.env.reset()
            done = False
            rs = 0
            while not done:
                a = self.explorationStrategyEvalFn(self.nnOnline, s)
                s, r, done, _ = self.env.step(a)
                rs += r
            finalEvalRewardsList.append(rs)
        # self.performBookKeeping(train=False)
        return finalEvalRewardsList

```

In [1535]:

```

delta = 1
epsilon = 0.3

ddqn = DDQN(env='CartPole-v0', seed=0, gamma=0.99, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=2*10**3, MAX_EVAL_EPISODES=10**0,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=
selectGreedyAction,
            updateFrequency=15)
trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = ddqn.runDDQN()

```

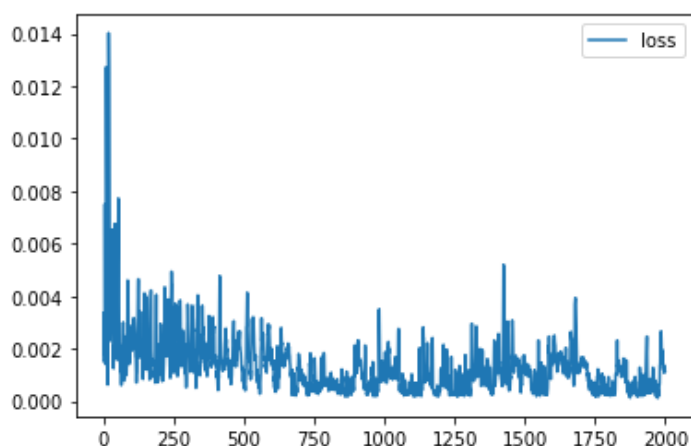
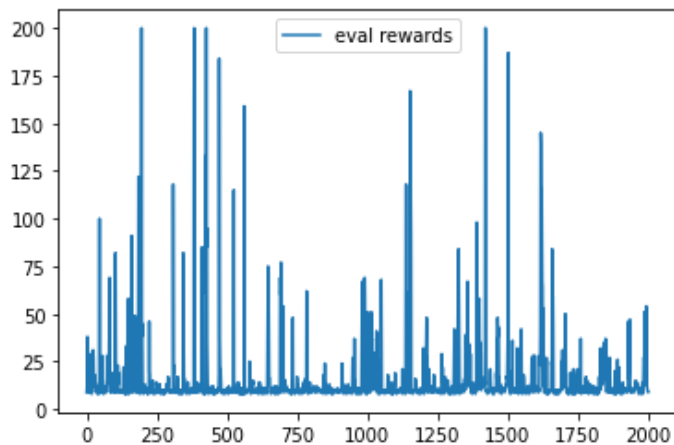
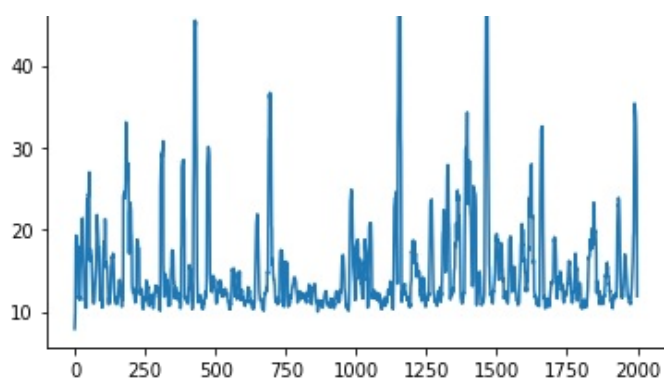
100%|██████████| 2000/2000 [00:17<00:00, 112.49it/s]

In [1536]:

```

plt.plot(np.arange(len(trainRewardsList)), np.convolve(trainRewardsList, np.ones(10)/10,
mode='same'), label='train reward')
plt.legend()
plt.show()
plt.plot(np.arange(len(evalRewardsList)), evalRewardsList, label='eval rewards')
plt.legend()
plt.show()
plt.plot(np.arange(len(loss)), loss, label='loss')
plt.legend()
plt.show()

```



Dueling DDQN

Implement the Dueling Double Deep Q algorithm. We have studied about Dueling Double DQN agent in Lecture 14. Use the function definitions (given below).

This class implements the Dueling Double DQN agent, you are required to implement the various methods of this class as outlined below. Note this class is generic and should work with any permissible OpenAI Gym environment

```
class D3QN():
    def __init__(self, env, seed, gamma, tau,
                  bufferSize,
                  batchSize,
                  optimizerFn,
                  optimizerLR,
                  MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                  explorationStrategyTrainFn,
                  explorationStrategyEvalFn,
                  updateFrequency):
    def initBookKeeping(self)
    def performBookKeeping(self, train = True)
```

```

def runD3QN(self)
def trainAgent(self)
def trainNetwork(self, experiences)
def updateNetwork(self, onlineNet, targetNet)
def evaluateAgent(self)

```

In [1537]:

```

class D3QN():
    def __init__(self, env, seed, gamma, tau,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency):
        #this D3QN method
        # 1. creates and initializes (with seed) the environment, train/eval episodes, gamma, etc.
        # 2. creates and intializes all the variables required for book-keeping values via the initBookKeeping method
        # 3. creates tareget and online Q-networks using the createValueNetwork above
        # 4. creates and initializes (with network params) the optimizer function
        # 5. sets the explorationStartegy variables/functions for train and evaluation
        # 6. sets the batchSize for the number of experiences
        # 7. Creates the replayBuffer

        env = gym.make(env)
        self.env = env
        self.seed = seed
        self.env.seed(self.seed)
        self.gamma = gamma
        self.tau = tau

        self.MAX_TRAIN_EPISODES = MAX_TRAIN_EPISODES
        self.MAX_EVAL_EPISODES = MAX_EVAL_EPISODES
        self.explorationStrategyTrainFn = explorationStrategyTrainFn
        self.explorationStrategyEvalFn = explorationStrategyEvalFn
        self.updateFrequency = updateFrequency

        self.nnTarget = createDuelingNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[64,128])
        self.nnTarget.eval()
        self.nnOnline = createDuelingNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[64,128])

        self.optimizerLR = optimizerLR
        self.optimizerFn = optimizerFn(self.nnOnline.parameters(), lr=self.optimizerLR)

        self.bufferSize = bufferSize
        self.batchSize = batchSize
        self.rBuffer = ReplayBuffer(bufferSize=self.bufferSize, bufferType='D3QN')

```

In [1538]:

```

class D3QN(D3QN):
    def initBookKeeping(self):
        #this method creates and intializes all the variables required for book-keeping values and it is called
        #init method
        #
        self.trainRewardsList = []
        self.trainTimeList = []
        self.evalRewardsList = []
        self.wallClockTimeList = []
        self.loss = []

```

In [1539]:

In [153]:

```
class D3QN(D3QN):
    def performBookKeeping(self, train = True):
        #this method updates relevant variables for the bookKeeping, this can be called
        #multiple times during training
        #if you want you can print information using this, so it may help to monitor prog
        #ress and also help to debug
        #
        if train:
            self.wallClockTimeList.append(time.time())
```

In [1540]:

```
class D3QN(D3QN):
    def runD3QN(self):
        #this is the main method, it trains the agent, performs bookkeeping while trainin
        #g and finally evaluates
        #the agent and returns the following quantities:
        #1. episode wise mean train rewards
        #2. episode wise mean eval rewards
        #2. episode wise trainTime (in seconds): time elapsed during training since the s
        #tart of the first episode
        #3. episode wise wallClockTime (in seconds): actual time elapsed since the start
        #of training,
        #
        # note this will include time for BookKeeping and
        # evaluation
        # Note both trainTime and wallClockTime get accumulated as episodes proceed.

        self.initBookKeeping()
        self.trainAgent()
        self.finalEvalReward = self.evaluateAgent()
        return self.trainRewardsList, self.trainTimeList, self.evalRewardsList, self.wal
        lClockTimeList, self.finalEvalReward, self.loss
```

In [1541]:

```
class D3QN(D3QN):
    def trainAgent(self):
        #this method collects experiences and trains the agent and does BookKeeping whil
        #e training.
        #this calls the trainNetwork() method internally, it also evaluates the agent per
        #episode
        #it trains the agent for MAX_TRAIN_EPISODES
        #
        self.updateNetwork(self.nnOnline, self.nnTarget)
        for e in tqdm(range(self.MAX_TRAIN_EPISODES)):
            s = self.env.reset()
            r, steps = self.rBuffer.collectExperiences(self.env, s, self.explorationStra
            tegyTrainFn, self.batchSize*100, net = self.nnOnline)
            self.trainRewardsList.append(r)
            self.trainTimeList.append(steps)
            while self.rBuffer.length() < self.batchSize*5:
                self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTra
                inFn, self.batchSize*100, net = self.nnOnline)
            experiences = self.rBuffer.sample(self.batchSize)
            self.loss.append(self.trainNetwork(experiences))
            self.performBookKeeping(train = True)
            self.evalRewardsList.append(np.mean(self.evaluateAgent()))

            if e%self.updateFrequency == 0:
                self.updateNetwork(self.nnOnline, self.nnTarget)
```

In [1542]:

```
class D3QN(D3QN):
    def trainNetwork(self, experiences):
        # this method trains the value network epoch number of times and is called by the
        # trainAgent function
        # it essentially uses the experiences to calculate target, using the targets it c
        #alculates the error, which
        # is further used for calulating the loss. It then uses the optimizer over the lo
```

ss

```
# to update the params of the network by backpropagating through the network
# this function does not return anything
# you can try out other loss functions other than MSE like Huber loss, MAE, etc.

s, a, r, s_, t = self.rBuffer.splitExperiences(experiences)
r = (r-r.mean())/(r.std()+1e-7)
argmax_a_qs = self.nnOnline(s_).argmax(dim=1).unsqueeze(1)
with torch.no_grad():
    q_targets = self.nnTarget(s_).gather(1, argmax_a_qs).detach()
    tdTargets = r + self.gamma * q_targets * (1 - t)
    tdErrors = tdTargets - self.nnOnline(s).gather(1,a.long())
    # loss = F.smooth_l1_loss(tdErrors, torch.zeros_like(tdErrors)).to(device)
    # loss = huberLoss(error=tdErrors)
    loss = F.mse_loss(tdErrors, torch.zeros_like(tdErrors)).to(device)
    self.optimizerFn.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(self.nnOnline.parameters(), max_norm=1)
    self.optimizerFn.step()
return loss.item()
```

In [1543]:

```
class D3QN(D3QN):
    def updateNetwork(self, onlineNet, targetNet):
        #this function updates the onlineNetwork with the target network using Polyak averaging
        #
        online = onlineNet.state_dict()
        target = targetNet.state_dict()

        for k in online.keys():
            target[k] = self.tau * online[k] + (1 - self.tau) * target[k]

        onlineNet.load_state_dict(target)
```

In [1544]:

```
class D3QN(D3QN):
    def evaluateAgent(self):
        #this function evaluates the agent using the value network, it evaluates agent for MAX_EVAL_EPISODES
        #typically MAX_EVAL_EPISODES = 1
        finalEvalRewardsList = []
        for _ in range(self.MAX_EVAL_EPISODES):
            s = self.env.reset()
            done = False
            rs = 0
            while not done:
                a = self.explorationStrategyEvalFn(self.nnOnline, s)
                s, r, done, _ = self.env.step(a)
                rs += r
            finalEvalRewardsList.append(rs)
        # self.performBookKeeping(train=False)
        return finalEvalRewardsList
```

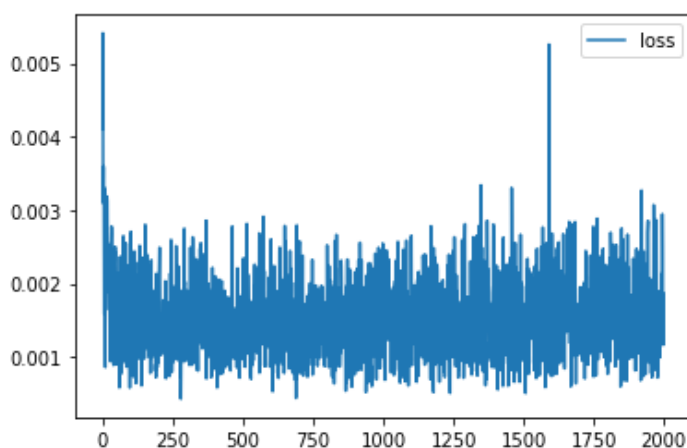
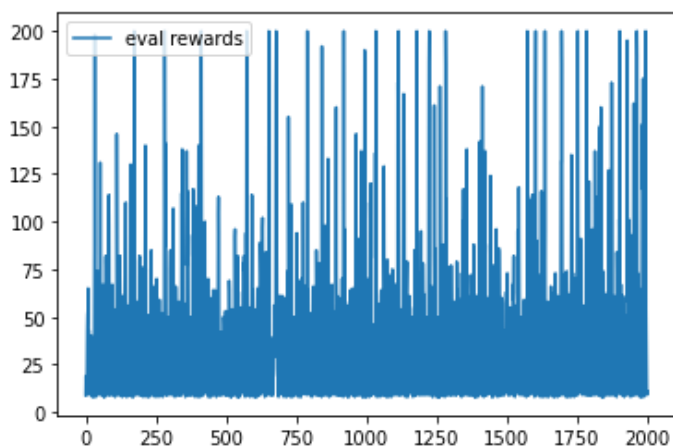
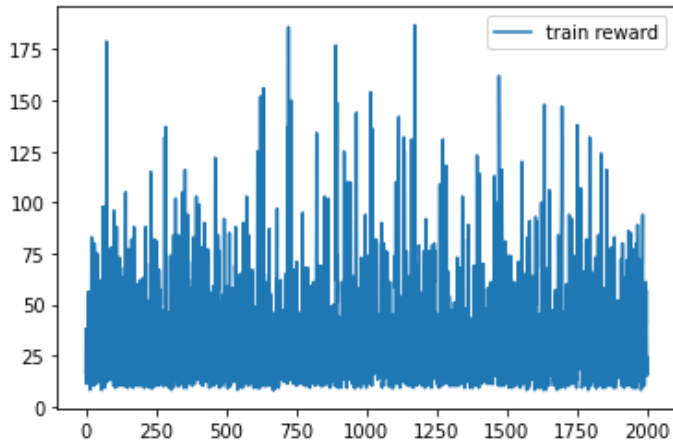
In [1464]:

```
delta = 1
epsilon = 0.5

d3qn = D3QN(env='CartPole-v0', seed=0, gamma=0.99, tau=0.3, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=2*10**3, MAX_EVAL_EPISODES=10**0,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction,
            updateFrequency=10)
trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = d3qn.runD3QN()
```


In [1465]:

```
plt.plot(np.arange(len(trainRewardsList)), trainRewardsList, label='train reward')
plt.legend()
plt.show()
plt.plot(np.arange(len(evalRewardsList)), evalRewardsList, label='eval rewards')
plt.legend()
plt.show()
plt.plot(np.arange(len(loss)), loss, label='loss')
plt.legend()
plt.show()
```



Dueling Double Deep Q Network with Prioritized Experience Replay (D3QN-PER)

Implement the Dueling Double DQN with Prioritized Experience Replay (D3QN-PER) agent. We have studied about D3QN-PER agent in Lecture 15. Use the function definitions (given below).

This class implements the D3QN-PER agent, you are required to implement the various methods of this class as outlined below. Note this class is generic and should work with any permissible OpenAI Gym environment

```

class D3QN_PER():
    def __init__(env, seed, gamma, tau, alpha, beta, beta_rate,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency)
    def initBookKeeping(self)
    def performBookKeeping(self, train = True)
    def runD3QN_PER(self)
    def trainAgent(self)
    def trainNetwork(self, experiences)
    def updateNetwork(self, onlineNet, targetNet)
    def evaluateAgent(self)

```

In [1545]:

```

class D3QN_PER():
    def __init__(self, env, seed, gamma, tau, alpha, beta, beta_rate,
                 bufferSize,
                 batchSize,
                 optimizerFn,
                 optimizerLR,
                 MAX_TRAIN_EPISODES, MAX_EVAL_EPISODES,
                 explorationStrategyTrainFn,
                 explorationStrategyEvalFn,
                 updateFrequency):

        env = gym.make(env)
        self.env = env
        self.seed = seed
        self.env.seed(self.seed)
        self.gamma = gamma
        self.tau = tau
        self.alpha = alpha
        self.beta = beta
        self.beta_rate = beta_rate

        self.MAX_TRAIN_EPISODES = MAX_TRAIN_EPISODES
        self.MAX_EVAL_EPISODES = MAX_EVAL_EPISODES
        self.explorationStrategyTrainFn = explorationStrategyTrainFn
        self.explorationStrategyEvalFn = explorationStrategyEvalFn
        self.updateFrequency = updateFrequency

        self.nnTarget = createDuelingNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[64,128])
        self.nnTarget.eval()
        self.nnOnline = createDuelingNetwork(inDim=env.observation_space.shape[0], outDim=env.action_space.n, hDim=[64,128])

        self.optimizerLR = optimizerLR
        self.optimizerFn = optimizerFn(self.nnOnline.parameters(), lr=self.optimizerLR)

        self.bufferSize = bufferSize
        self.batchSize = batchSize
        self.rBuffer = ReplayBuffer(bufferSize=self.bufferSize, bufferType='PER-D3QN', alpha=self.alpha, beta=self.beta, beta_rate=self.beta_rate)

```

In [1546]:

```

class D3QN_PER(D3QN_PER):
    def initBookKeeping(self):

```

```

        #this method creates and initializes all the variables required for book-keeping v
        alues and it is called
        #init method
        #
        self.trainRewardsList = []
        self.trainTimeList = []
        self.evalRewardsList = []
        self.wallClockTimeList = []
        self.loss = []

```

In [1547]:

```

class D3QN_PER(D3QN_PER):
    def performBookKeeping(self, train = True):
        #this method updates relevant variables for the bookKeeping, this can be called
        #multiple times during training
        #if you want you can print information using this, so it may help to monitor prog
        ress and also help to debug
        #
        if train:
            self.wallClockTimeList.append(time.time())

```

In [1564]:

```

class D3QN_PER(D3QN_PER):
    def runD3QN(self):
        #this is the main method, it trains the agent, performs bookkeeping while trainin
        g and finally evaluates
        #the agent and returns the following quantities:
        #1. episode wise mean train rewards
        #2. epsiode wise mean eval rewards
        #2. episode wise trainTime (in seconds): time elapsed during training since the s
        tart of the first episode
        #3. episode wise wallClockTime (in seconds): actual time elapsed since the start
        of training,
        #
        note this will include time for BookKeeping and
        evaluation
        # Note both trainTime and wallClockTime get accumulated as episodes proceed.

        self.initBookKeeping()
        self.trainAgent()
        self.finalEvalReward = self.evaluateAgent()
        return self.trainRewardsList, self.trainTimeList, self.evalRewardsList, self.wal
        lClockTimeList, self.finalEvalReward, self.loss

```

In [1598]:

```

class D3QN_PER(D3QN_PER):
    def trainAgent(self):
        #this method collects experiences and trains the agent and does BookKeeping whil
        e training.
        #this calls the trainNetwork() method internally, it also evaluates the agent per
        episode
        #it trains the agent for MAX_TRAIN_EPISODES
        #
        self.updateNetwork(self.nnOnline, self.nnTarget)
        for e in tqdm(range(self.MAX_TRAIN_EPISODES)):
            s = self.env.reset()
            r, steps = self.rBuffer.collectExperiences(self.env, s, self.explorationStra
            tegyTrainFn, self.batchSize*100, net = self.nnOnline)
            self.trainRewardsList.append(r)
            self.trainTimeList.append(steps)
            while self.rBuffer.length() < self.batchSize:
                self.rBuffer.collectExperiences(self.env, s, self.explorationStrategyTra
                inFn, self.batchSize*100, net = self.nnOnline)
                experiences = self.rBuffer.sample(self.batchSize)
                self.loss.append(self.trainNetwork(experiences))
                self.performBookKeeping(train = True)
                self.evalRewardsList.append(np.mean(self.evaluateAgent()))

            if e%self.updateFrequency == 0:

```

```
self.updateNetwork(self.nnOnline, self.nnTarget)
```

In [1599]:

```
class D3QN_PER(D3QN_PER):
    def trainNetwork(self, experiences):
        # this method trains the value network epoch number of times and is called by the
        trainAgent function
        # it essentially uses the experiences to calculate target, using the targets it c
        alculates the error, which
        # is further used for calulating the loss. It then uses the optimizer over the lo
        ss

        # to update the params of the network by backpropagating through the network
        # this function does not return anything
        # you can try out other loss functions other than MSE like Huber loss, MAE, etc.
        #
        #Your code goes in here

        s, a, r, s_, t, weights, indices = self.rBuffer.splitExperiences(experiences)
        r = (r-r.mean())/(r.std()+1e-7)
        argmax_a_qs = self.nnOnline(s_).argmax(dim=1).unsqueeze(1)
        with torch.no_grad():
            q_targets = self.nnTarget(s_).gather(1, argmax_a_qs).detach()
            tdTargets = r + self.gamma * q_targets * (1 - t)
            tdErrors = tdTargets - self.nnOnline(s).gather(1,a.long())
            # loss = F.smooth_l1_loss(weights*tdErrors, torch.zeros_like(tdErrors)).to(device
        )

        # loss = huberLoss(error=weights*tdErrors)
        loss = F.mse_loss(weights*tdErrors, torch.zeros_like(tdErrors)).to(device)
        self.optimizerFn.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.nnOnline.parameters(), max_norm=1)
        self.optimizerFn.step()
        priorities = torch.abs(tdErrors).detach()
        self.rBuffer.update(indices, priorities)
        return loss.item()
```

In [1600]:

```
class D3QN_PER(D3QN_PER):
    def updateNetwork(self, onlineNet, targetNet):
        #this function updates the onlineNetwork with the target network using Polyak ave
        raging
        #
        online = onlineNet.state_dict()
        target = targetNet.state_dict()

        for k in online.keys():
            target[k] = self.tau * online[k] + (1 - self.tau) * target[k]

        onlineNet.load_state_dict(target)
```

In [1601]:

```
class D3QN_PER(D3QN_PER):
    def evaluateAgent(self):
        #this function evaluates the agent using the value network, it evaluates agent fo
        r MAX_EVAL_EPISODES
        #typically MAX_EVAL_EPISODES = 1
        finalEvalRewardsList = []
        for _ in range(self.MAX_EVAL_EPISODES):
            s = self.env.reset()
            done = False
            rs = 0
            while not done:
                a = self.explorationStrategyEvalFn(self.nnOnline, s)
                s, r, done, _ = self.env.step(a)
                rs += r
            finalEvalRewardsList.append(rs)
        # self.performBookKeeping(train=False)
        return finalEvalRewardsList
```

In [1602]:

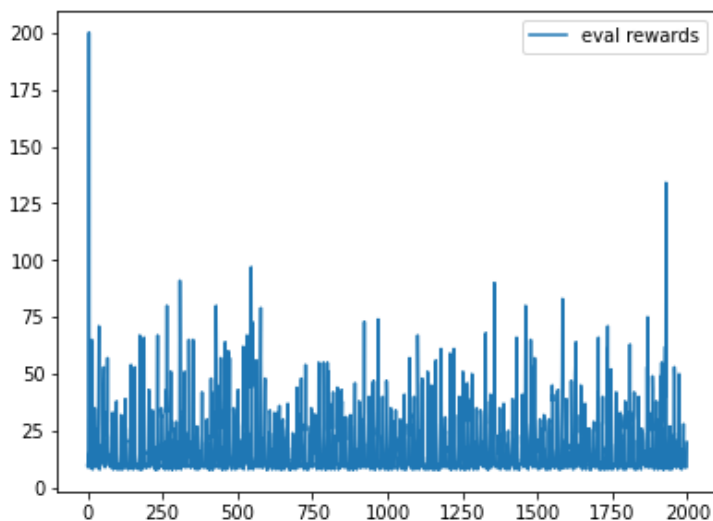
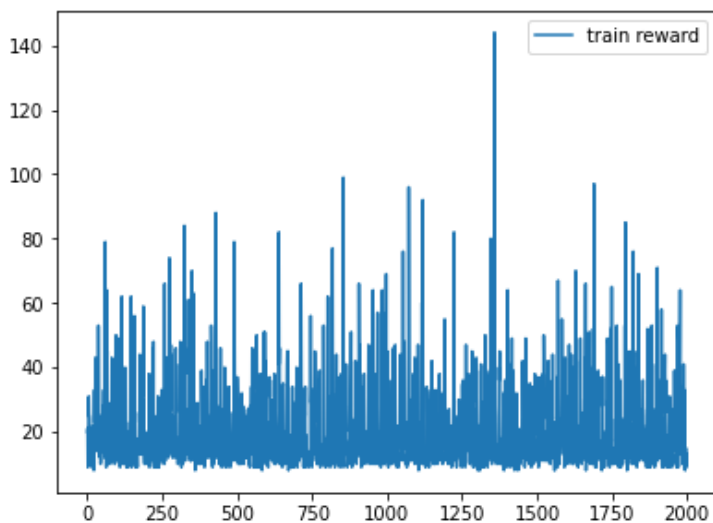
```
epsilon = 0.5
delta = 1

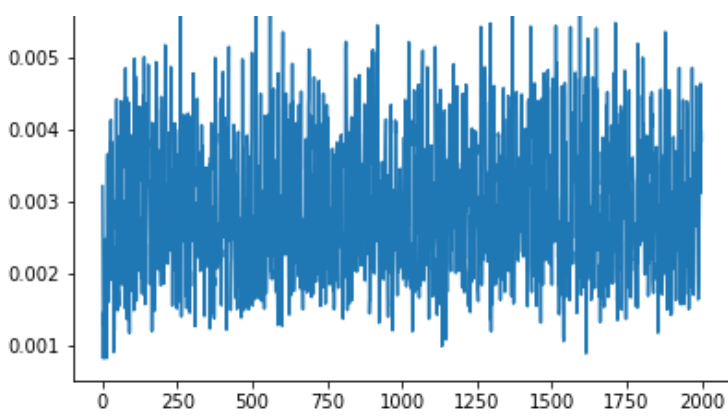
d3qn_per = D3QN_PER(env='CartPole-v0', seed=0, gamma=0.99, tau=0.3, alpha=0.7, beta=0.1,
beta_rate=0.9992,
    bufferSize=10**4, batchSize=64,
    optimizerFn=optim.Adam, optimizerLR=5*1e-4,
    MAX_TRAIN_EPISODES=2*10**3, MAX_EVAL_EPISODES=10**0,
    explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=
selectGreedyAction,
    updateFrequency=15)
trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, los
s = d3qn_per.runD3QN()
```

100%|██████████| 2000/2000 [00:59<00:00, 33.76it/s]

In [1604]:

```
plt.rcParams["figure.figsize"] = (6.4,4.8)
plt.plot(np.arange(len(trainRewardsList)), trainRewardsList, label='train reward')
plt.legend()
plt.show()
plt.plot(np.arange(len(evalRewardsList)), evalRewardsList, label='eval rewards')
plt.legend()
plt.show()
plt.plot(np.arange(len(loss)), loss, label='loss')
plt.legend()
plt.show()
```





Experiments and Plots

Run the NFQ, DQN, Double DQN, Dueling Double DQN, and Dueling Double Deep Q Network with Prioritized Experience Replay agent on CartPole environment and MountainCar environment.

Plot the following for each of the environment separately. Note based on different hyper-parameters and strategies you use, can you have multiple plots for each of the below.

As you are aware from your past experience, single run of the agent over the environment results in plots that have lot of variance and look very noisy. One way to overcome this is to create several different instances of the environment using different seeds and then average out the results across these and plot these. For all the plots below, you this strategy. You need to run 5 different instances of the environment for each agent. As you have seen in the lecture slides, we plot the maximum and minimum values around the mean in the plots, so this gives us the shaded plot with the mean curve in the between. In this assignment, you are required to do the same. Generate plots with envelop between maximum and minimum value (check the plotQuantity() function in the helper functions).

For each of the quantity of interest, plot each of the agent within the same plot using different colors for the envelop. Choose colors such that that there is clear contrast between the plots corresponding to different agents.

1. Plot mean train rewards vs episodes for Cartpole environment.
 2. Plot mean train rewards vs episodes for MountatinCar environment.
 3. Plot mean evaluation rewards vs episodes
 4. Plot mean evaluation rewards vs episodes
 5. Plot total steps vs episode for Cartpole environment.
 6. Plot total steps vs episode for MountatinCar environment.
 7. Plot train time vs episode for Cartpole environment.
 8. Plot train time vs episode for MountatinCar environment.
 9. Plot wall clock time vs episode for Cartpole environment.
 10. Plot wall clock time vs episode for MountatinCar environment.
1. Based on plots for CartPole environment, what are your observations about different agents. Compare different agents.
 2. Based on plots for MountainCar environment, what are your observations about different agents. Compare different agents. Do these observations concur with the ones for CartPole environment?
 3. Based on both the environments, can you generalize some of the findings for the value-based agents? If yes what are those findings?

In [1582]:

```
def storeData(quant, model, trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss):
    quant['reward'][model].append(trainRewardsList)
    quant['clock time'][model].append([x - wallClockTimeList[0] for x in wallClockTimeList])
    quant['eval reward'][model].append(evalRewardsList)
    quant['loss'][model].append(loss)
    quant['time step'][model].append(trainTimeList)
```

```
return quants
```

In [1583]:

```
def runDeepValueBasedAgents(env, n_seeds, EPISODES):
    models = ['NFQ', 'DQN', 'DDQN', 'D3QN', 'PER-D3QN']
    plots = ['reward', 'loss', 'clock time', 'eval reward', 'time step']
    quants = {}
    for plot in plots:
        q_plot = {}
        for model in models:
            q_plot[model] = []
        quants[plot] = q_plot
    for i in tqdm(range(n_seeds)):
        seed = i
        nfq = NFQ(env=env, seed=seed, gamma=0.99, epochs=10**3, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=EPISODES, MAX_EVAL_EPISODES=10**1,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction)

        trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = nfq.runNFQ()
        quants = storeData(quants, 'NFQ', trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss)

        dqn = DQN(env=env, seed=seed, gamma=0.99, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=EPISODES, MAX_EVAL_EPISODES=10**1,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction,
            updateFrequency=15)
        trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = dqn.runDQN()
        quants = storeData(quants, 'DQN', trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss)

        ddqn = DDQN(env=env, seed=seed, gamma=0.99, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=EPISODES, MAX_EVAL_EPISODES=10**1,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction,
            updateFrequency=15)
        trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = ddqn.runDDQN()
        quants = storeData(quants, 'DDQN', trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss)

        d3qn = D3QN(env=env, seed=seed, gamma=0.99, tau=0.3, bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=EPISODES, MAX_EVAL_EPISODES=10**1,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction,
            updateFrequency=15)
        trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = d3qn.runD3QN()
        quants = storeData(quants, 'D3QN', trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss)

        d3qn_per = D3QN_PER(env=env, seed=seed, gamma=0.99, tau=0.3, alpha=0.7, beta=0.1, beta_rate=0.9992,
            bufferSize=10**4, batchSize=64,
            optimizerFn=optim.Adam, optimizerLR=7*1e-4,
            MAX_TRAIN_EPISODES=EPISODES, MAX_EVAL_EPISODES=10**1,
            explorationStrategyTrainFn=selectEpsilonGreedyAction, explorationStrategyEvalFn=selectGreedyAction,
            updateFrequency=15)
        trainRewardsList, trainTimeList, evalRewardsList, wallClockTimeList, finalEvalReward, loss = d3qn_per.runD3QN()
```

```

    quants = storeData(quants, 'PER-D3QN', trainRewardsList, trainTimeList, evalRwa
rdsList, wallClockTimeList, finalEvalReward, loss)

```

```

return quants

```

In [1676]:

```

quants100 = runDeepValueBasedAgents(env='MountainCar-v0', n_seeds=5, EPISODES=10**2)

```

```

100%|██████████| 100/100 [00:39<00:00, 2.55it/s]
100%|██████████| 100/100 [00:41<00:00, 2.43it/s]
100%|██████████| 100/100 [00:40<00:00, 2.47it/s]
100%|██████████| 100/100 [00:51<00:00, 1.94it/s]
100%|██████████| 100/100 [00:50<00:00, 2.00it/s]
100%|██████████| 100/100 [00:37<00:00, 2.68it/s]
100%|██████████| 100/100 [00:38<00:00, 2.63it/s]
100%|██████████| 100/100 [00:39<00:00, 2.54it/s]
100%|██████████| 100/100 [00:49<00:00, 2.01it/s]
100%|██████████| 100/100 [00:53<00:00, 1.87it/s]
100%|██████████| 100/100 [00:39<00:00, 2.56it/s]
100%|██████████| 100/100 [00:40<00:00, 2.46it/s]
100%|██████████| 100/100 [00:41<00:00, 2.43it/s]
100%|██████████| 100/100 [00:50<00:00, 1.99it/s]
100%|██████████| 100/100 [00:50<00:00, 1.99it/s]
100%|██████████| 100/100 [00:38<00:00, 2.61it/s]
100%|██████████| 100/100 [00:38<00:00, 2.62it/s]
100%|██████████| 100/100 [00:41<00:00, 2.43it/s]
100%|██████████| 100/100 [00:51<00:00, 1.94it/s]
100%|██████████| 100/100 [00:52<00:00, 1.90it/s]
100%|██████████| 100/100 [00:38<00:00, 2.63it/s]
100%|██████████| 100/100 [00:39<00:00, 2.51it/s]
100%|██████████| 100/100 [00:43<00:00, 2.29it/s]
100%|██████████| 100/100 [00:49<00:00, 2.01it/s]
100%|██████████| 100/100 [00:50<00:00, 1.97it/s]
100%|██████████| 5/5 [18:34<00:00, 222.95s/it]

```

In [1678]:

```

with open('quants100_mountainCar.pkl', 'wb') as f:
    pickle.dump(quants100, f)

```

In [1677]:

```

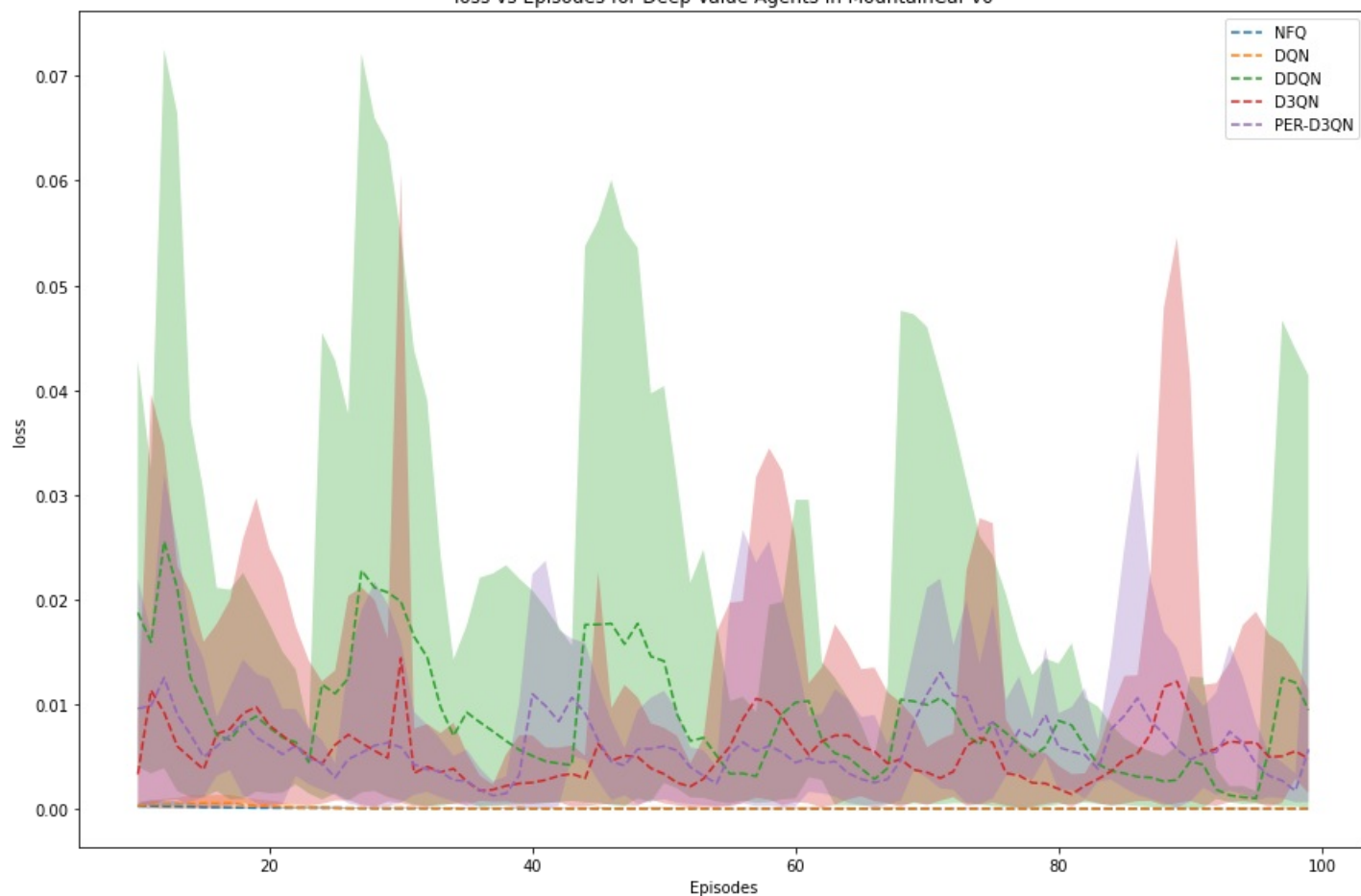
plotQuantity(quants=quants100, env='MountainCar-v0')

```

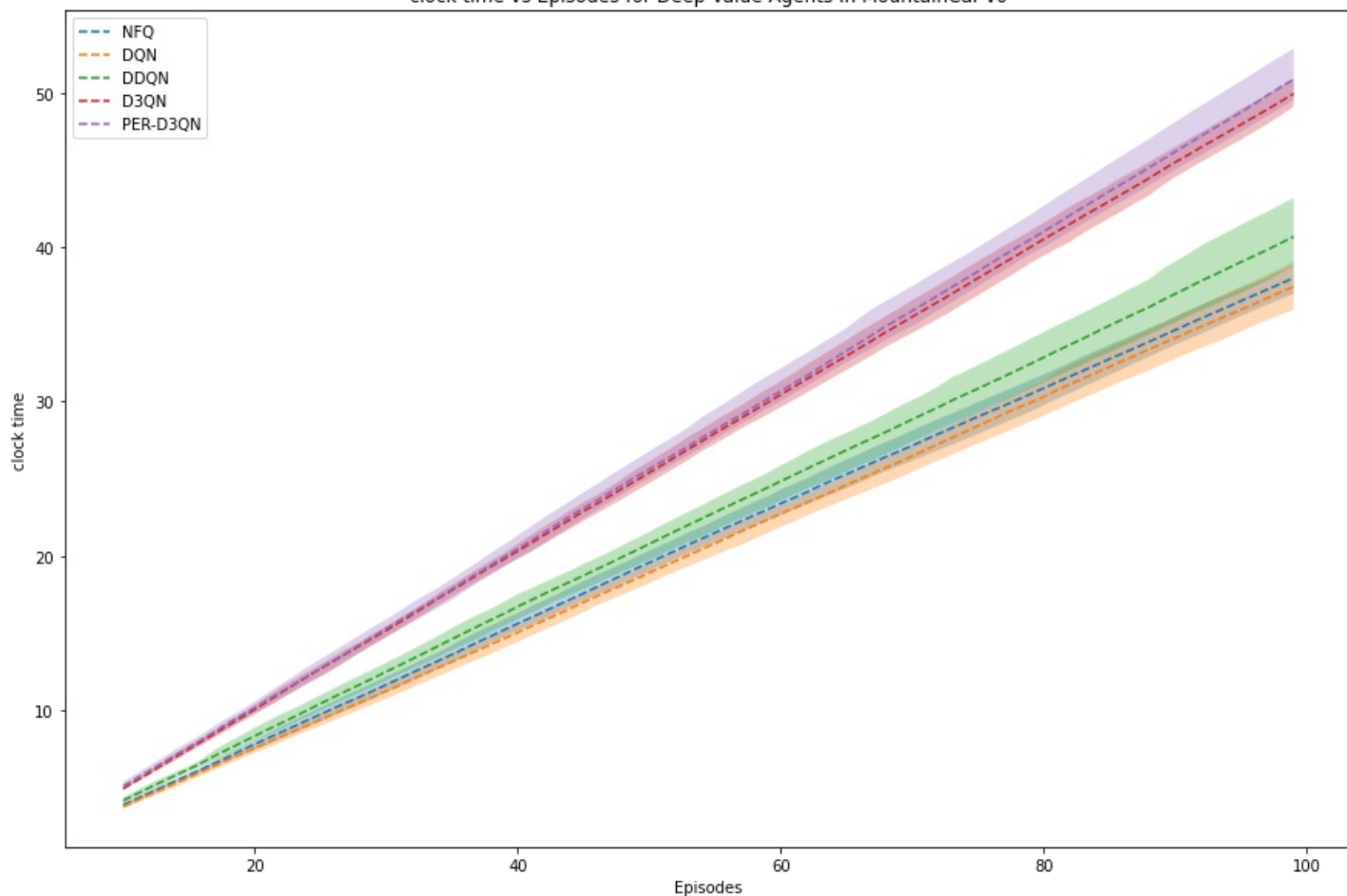




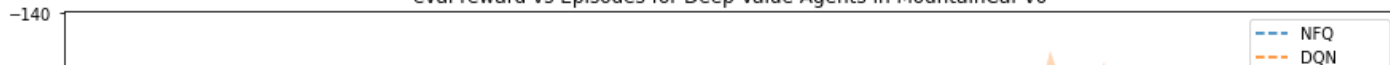
loss vs Episodes for Deep Value Agents in MountainCar-v0

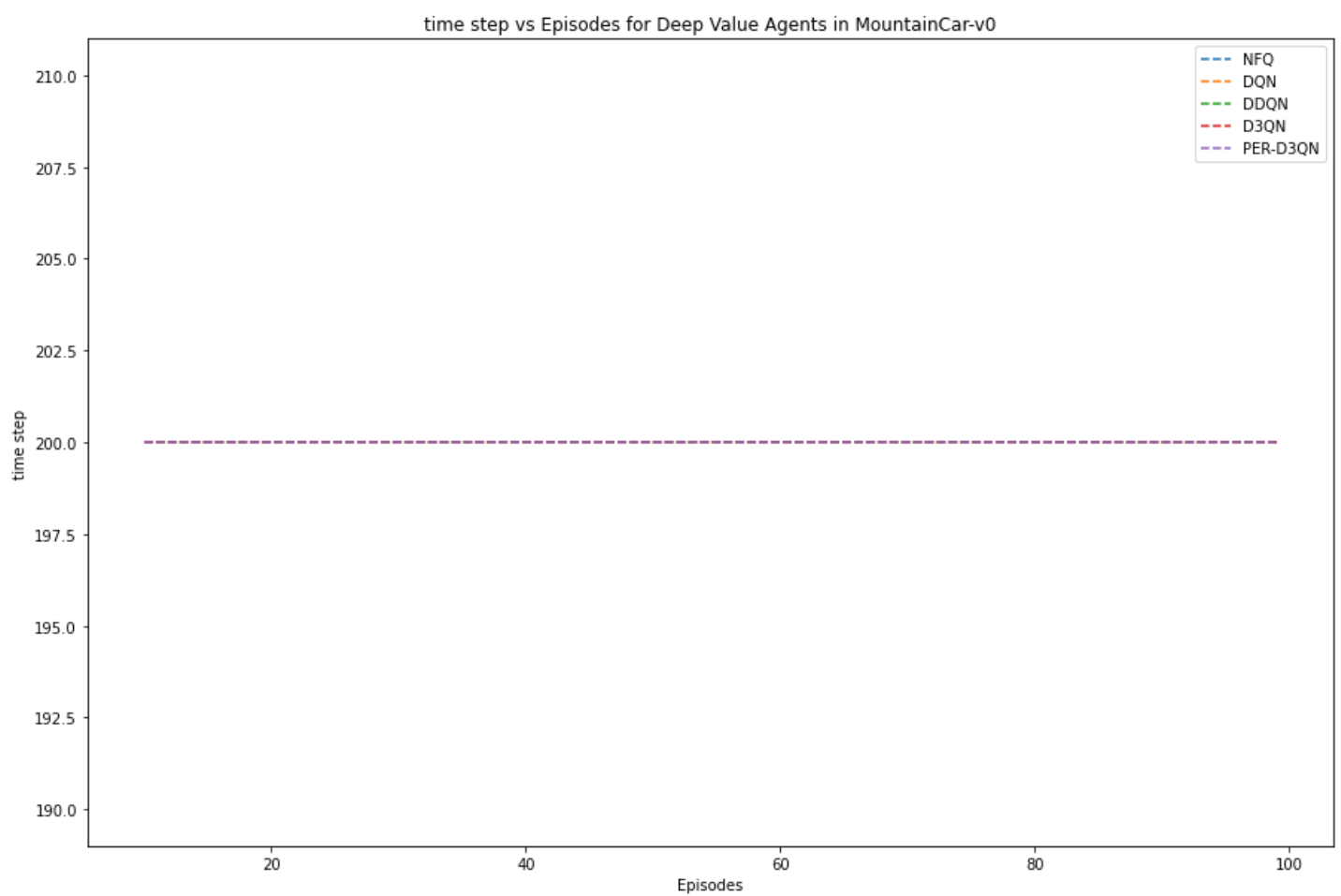
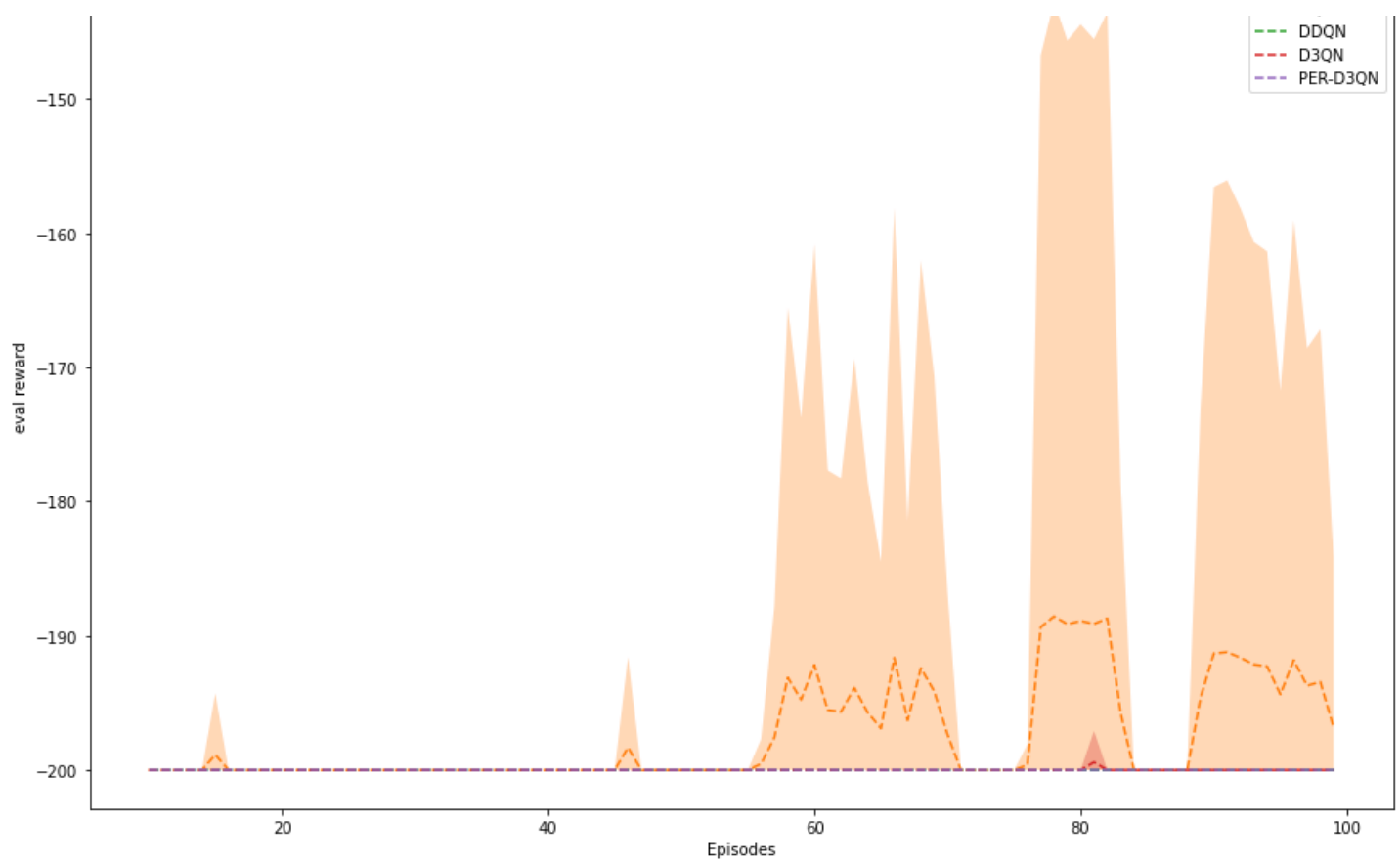


clock time vs Episodes for Deep Value Agents in MountainCar-v0



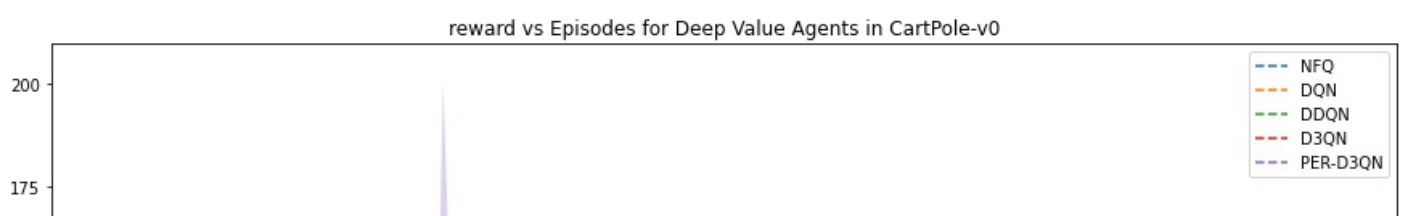
eval reward vs Episodes for Deep Value Agents in MountainCar-v0

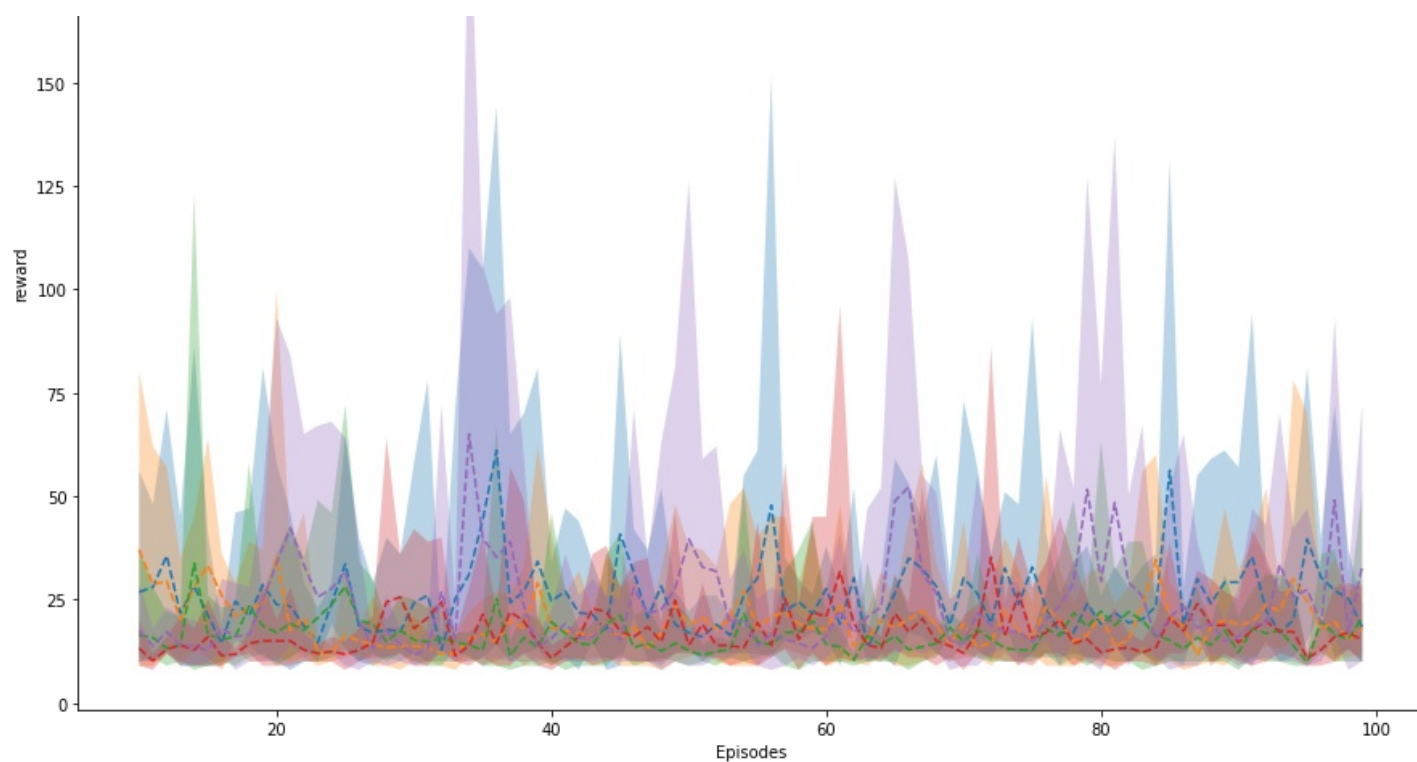




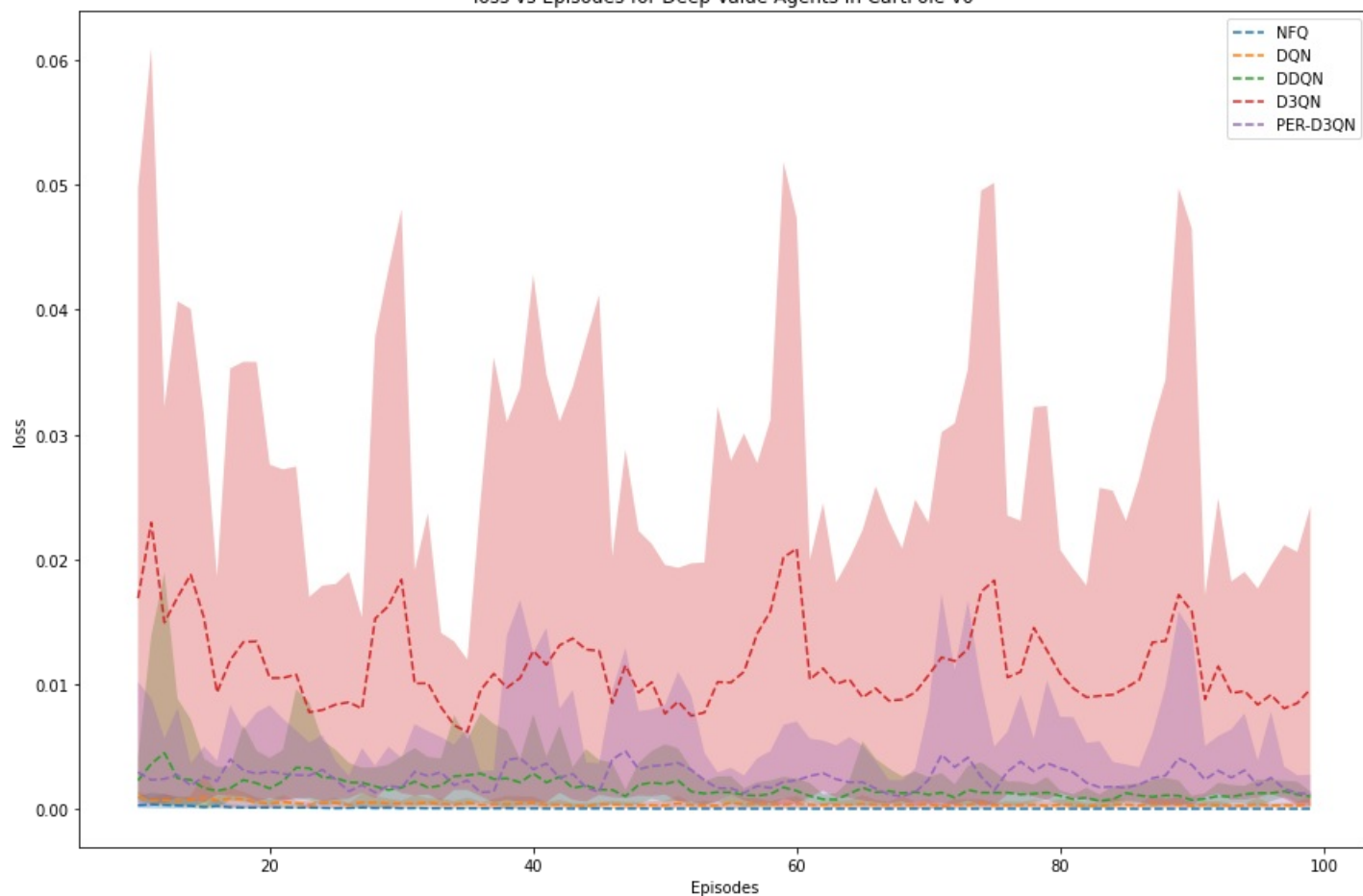
In [1675]:

```
plotQuantity(quants=quants100,env='MountainCar-v0')
```

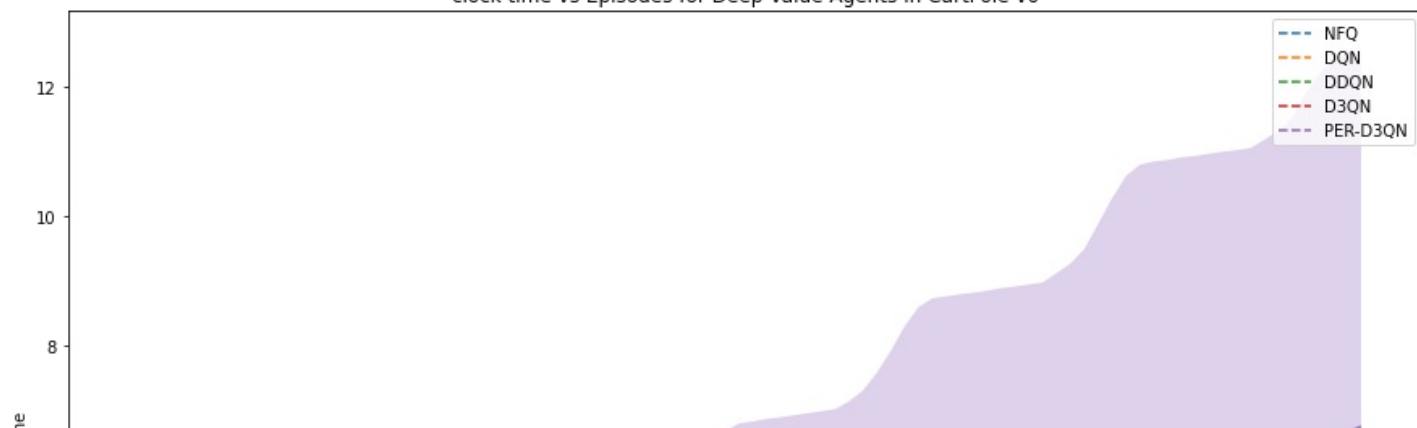


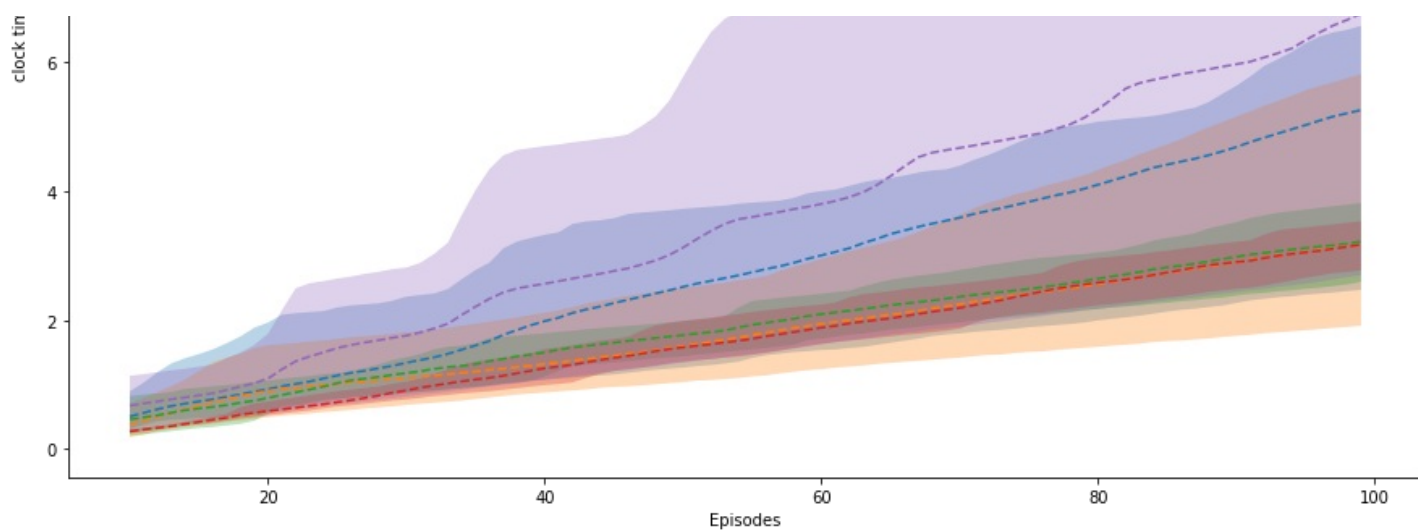


loss vs Episodes for Deep Value Agents in CartPole-v0

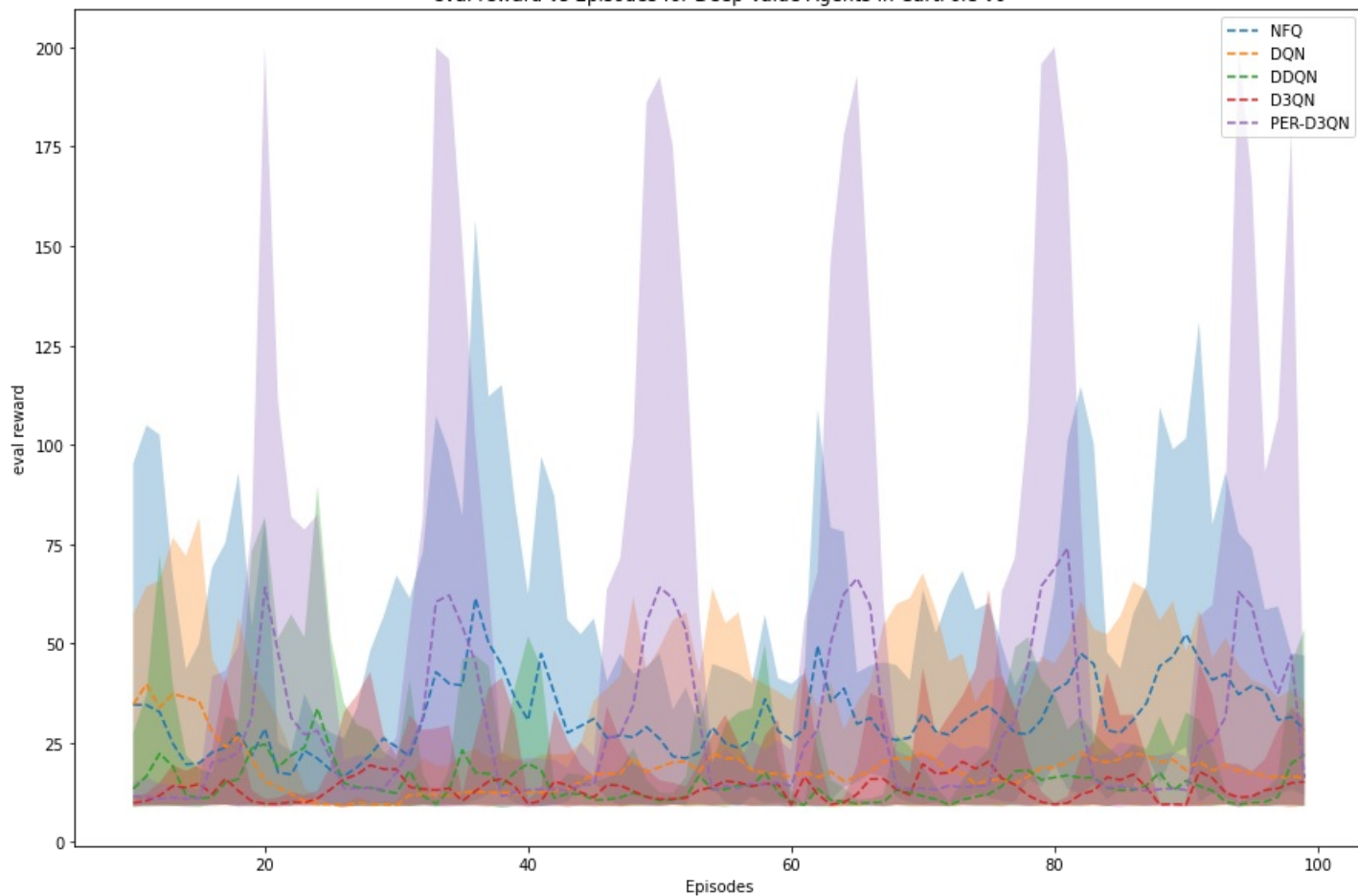


clock time vs Episodes for Deep Value Agents in CartPole-v0

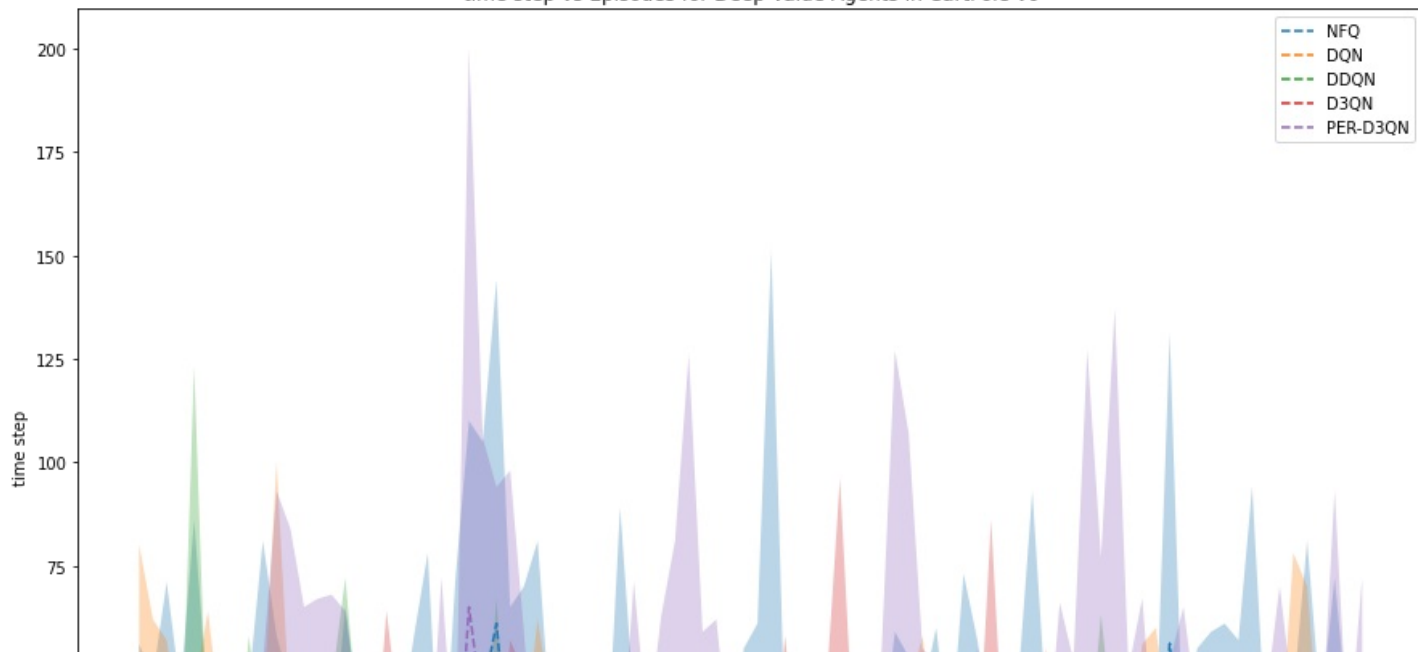


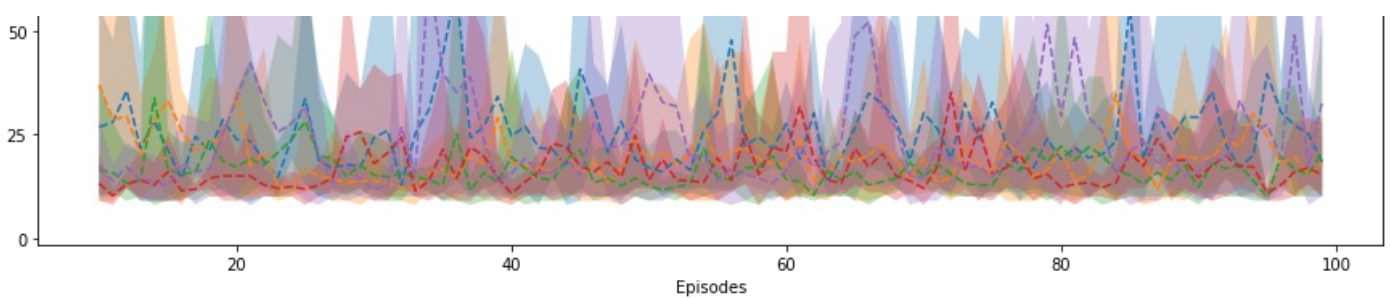


eval reward vs Episodes for Deep Value Agents in CartPole-v0



time step vs Episodes for Deep Value Agents in CartPole-v0





In [1673]:

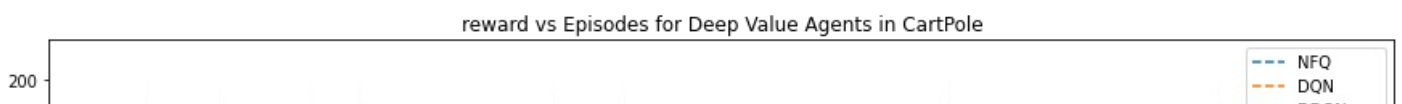
```
import pickle
with open('quants_1k_cartpole.pkl','rb') as f:
    quantslk = pickle.load(f)
```

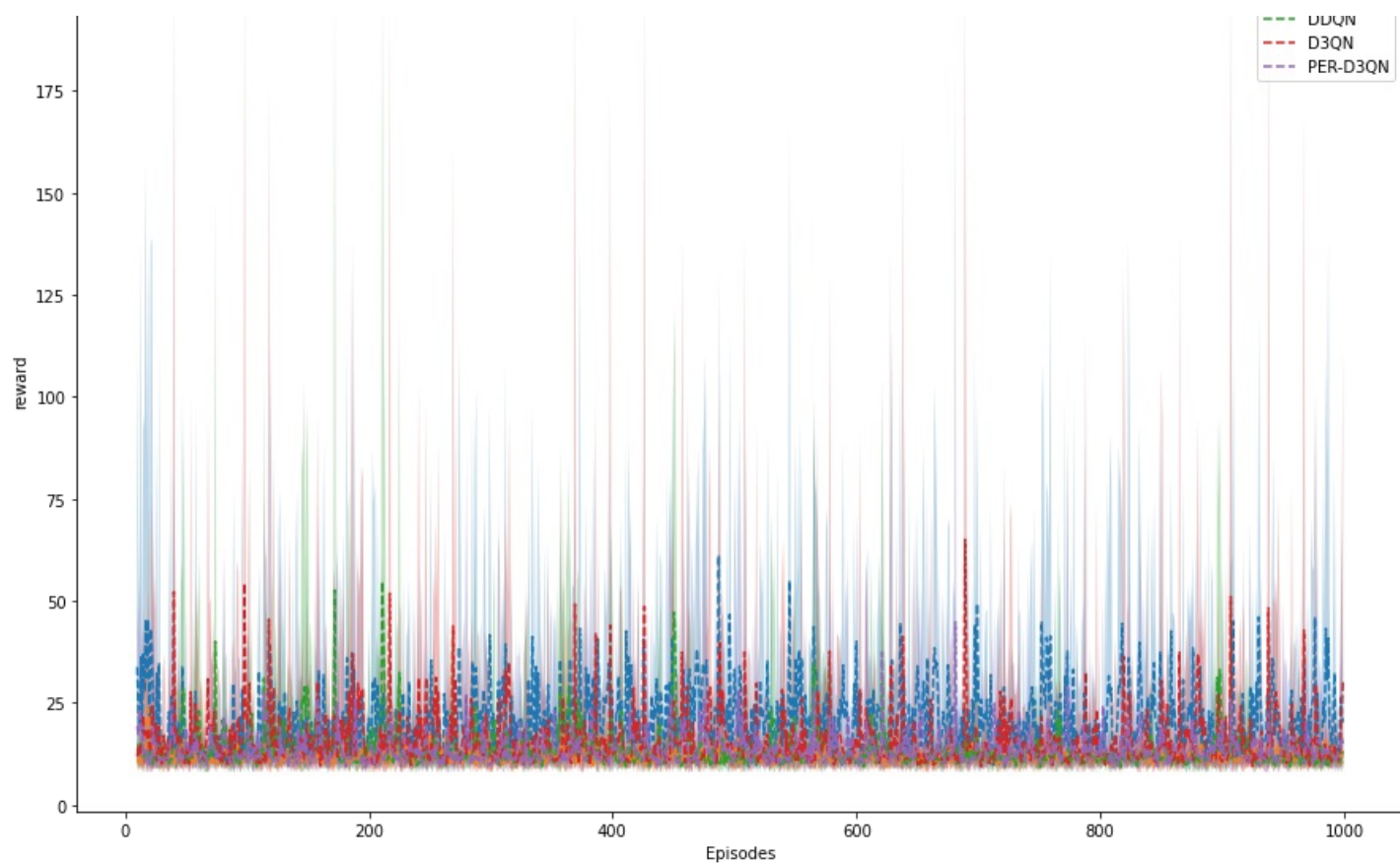
In [1596]:

```
def plotQuantity(env, quants):
    #this function takes in the quantityListDict and plots quantity vs episodes.
    #quantityListDict = {envInstanceCount: quantityList}
    #quantityList is list of the quantity per episode,
    #for example it could be mean reward per episode, traintime per episode, etc.
    #
    #NOTE: len(quantityList) == totalEpisodeCount
    #
    #Since we run multiple instances of the environment, there will be variance across en
vironments
    #so in the plot, you will plot per episode maximum, minimum and average value across
all env instances
    #Basically, you need to envelop (e.g., via color) the quantity between max and min wi
th mean value in between
    #
    #use the descriptionList parameter to put legends, title, etc.
    #For each of the plot, create the legend on the left/right side so that it doesn't ov
erlay on the plot lines/envelop.
    #
    #this is a generic function and can be used to plot any of the quantity of interest
    #In particular we will be using this function to plot:
    #    mean train rewards vs episodes
    #    mean evaluation rewards vs episodes
    #    total steps vs episode
    #    train time vs episode
    #    wall clock time vs episode
    #
    #
    #this function doesn't return anything
    models = ['NFQ', 'DQN', 'DDQN', 'D3QN', 'PER-D3QN']
    plots = ['reward', 'loss', 'clock time', 'eval reward', 'time step']
    plt.rcParams["figure.figsize"] = (15,10)
    for plot in plots:
        plt.xlabel('Episodes')
        plt.ylabel(plot)
        plt.title(plot+' vs Episodes for Deep Value Agents in '+env)
        for model in models:
            quantityList = quants[plot][model]
            episodes = np.shape(quantityList)[1]
            plt.plot(np.arange(episodes)[10:], np.mean(quantityList,axis=0)[10:], label=
model, ls='--')
            plt.fill_between(np.arange(episodes)[10:], np.min(quantityList,axis=0)[10:],
np.max(quantityList,axis=0)[10:], alpha=0.3)
            plt.legend()
            plt.show()
```

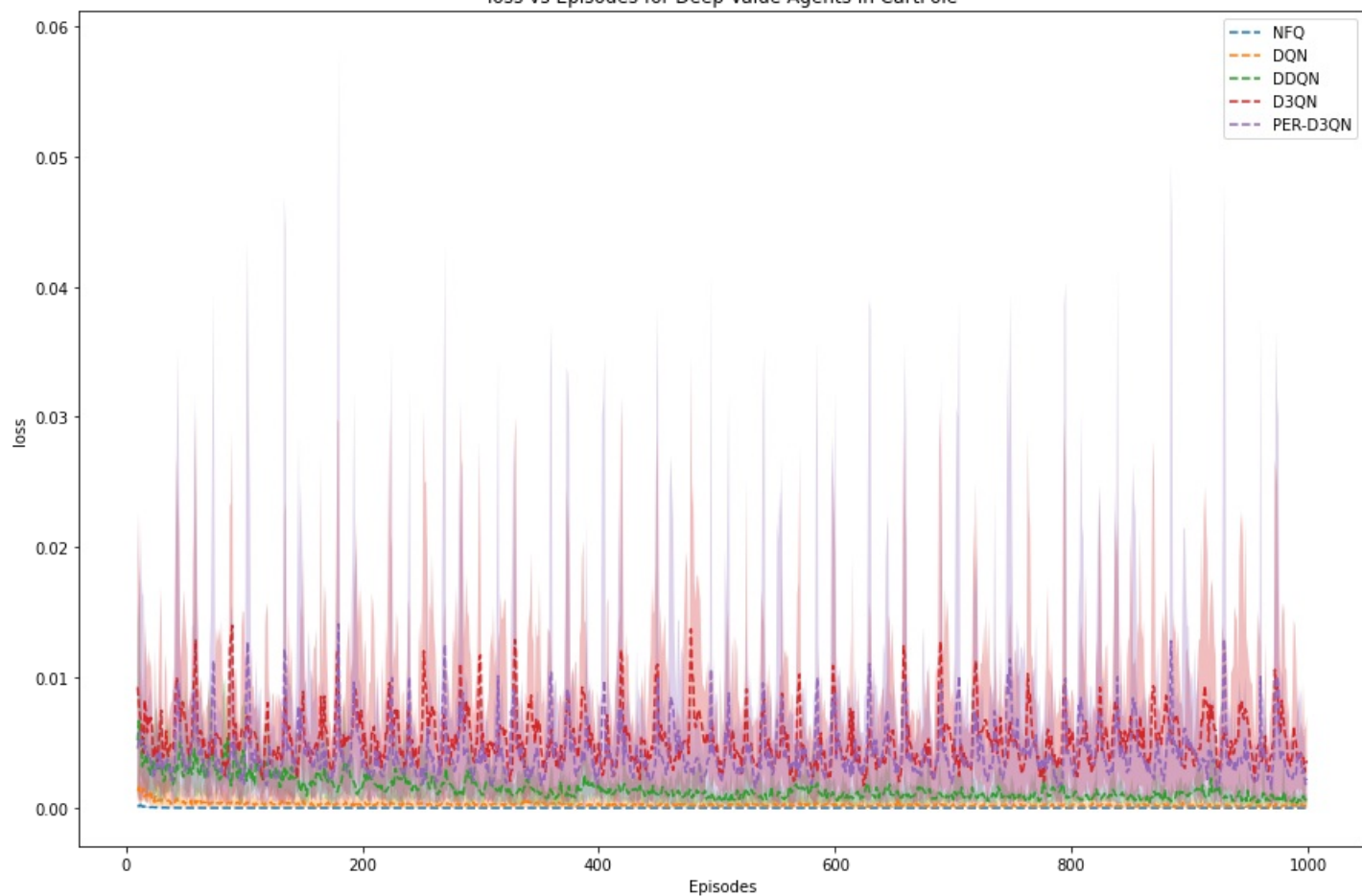
In []:

```
plotQuantity(env='CartPole',quants=quants)
```



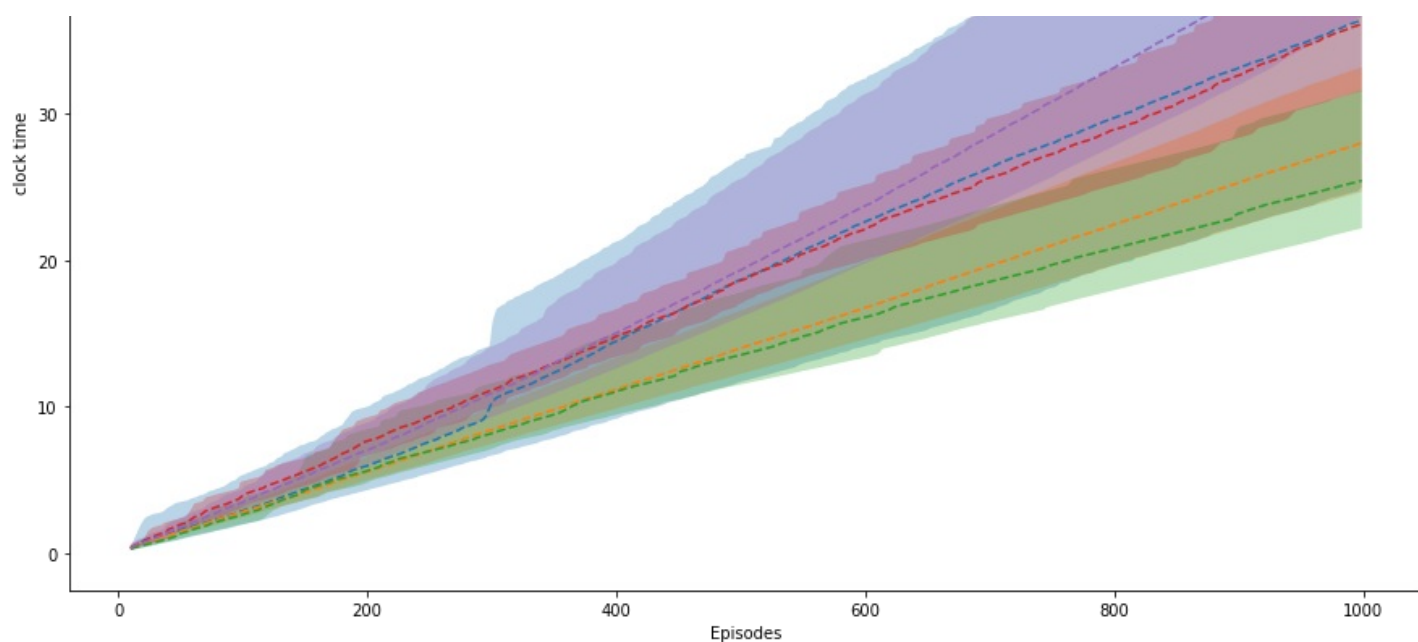


loss vs Episodes for Deep Value Agents in CartPole

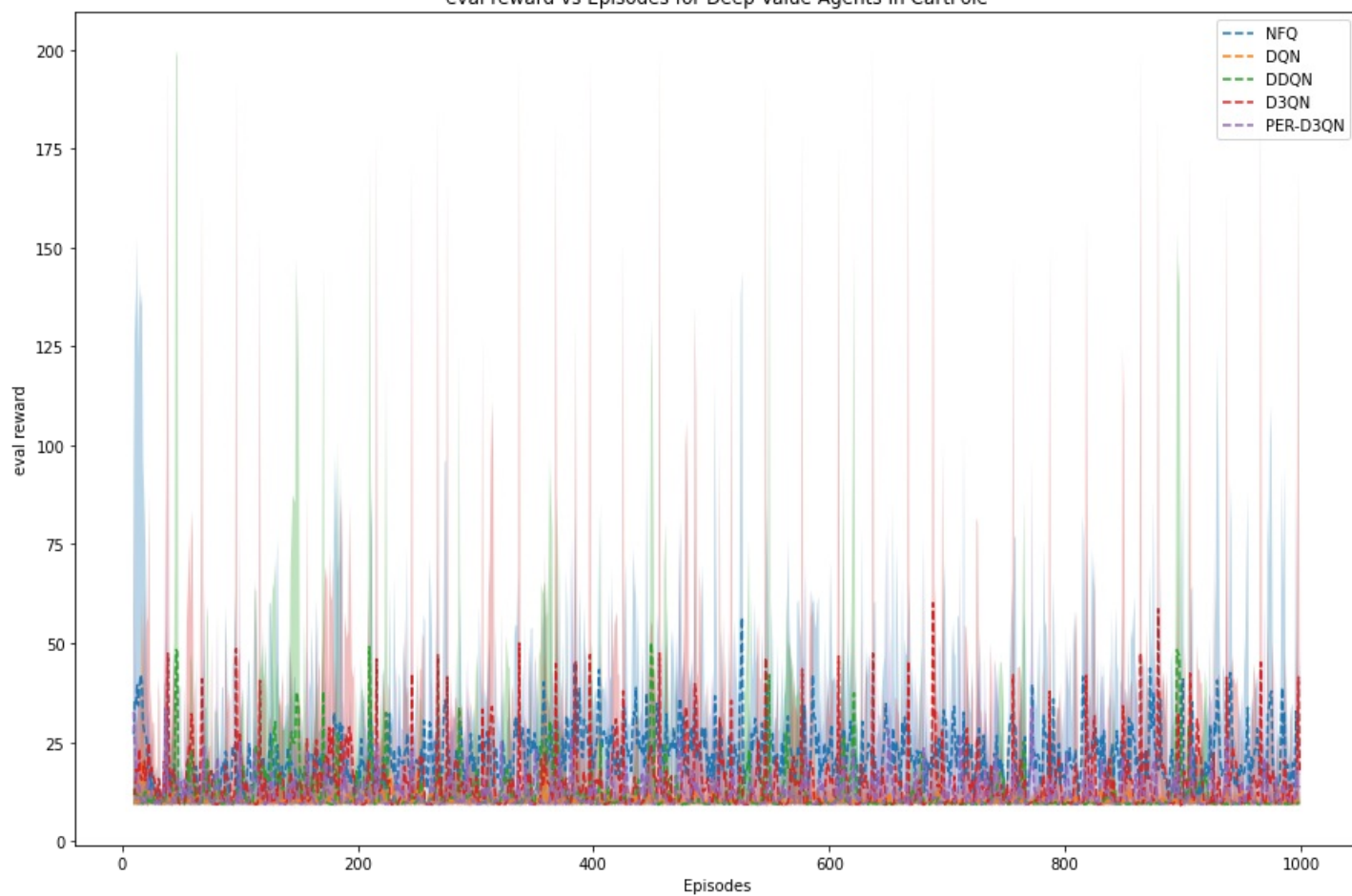


clock time vs Episodes for Deep Value Agents in CartPole

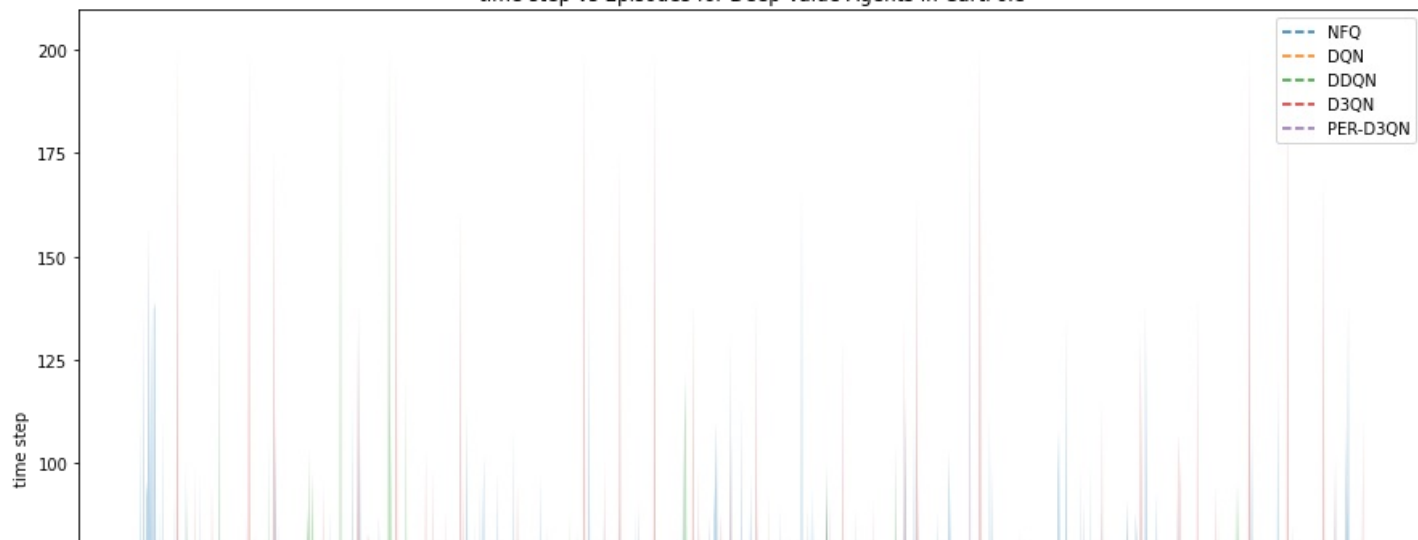


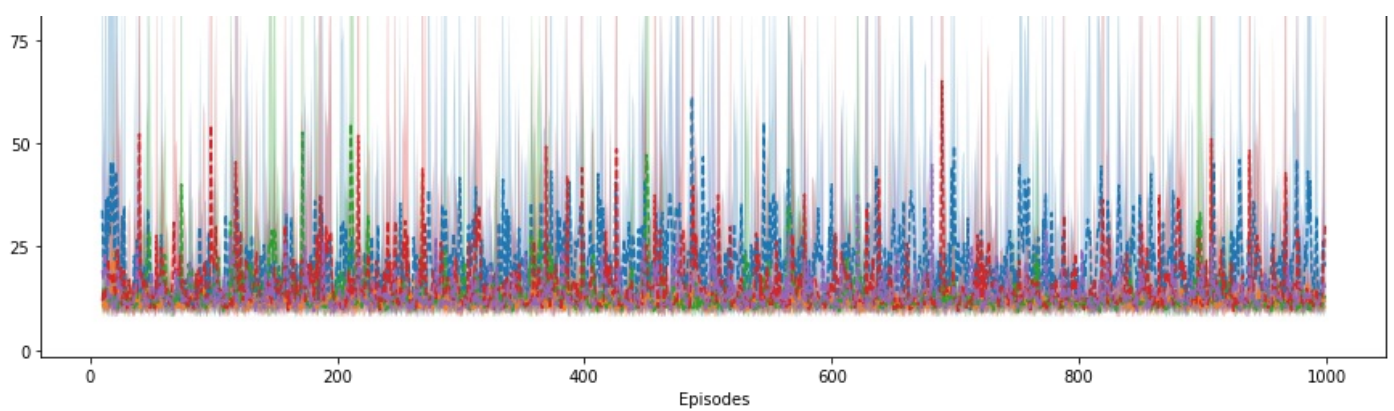


eval reward vs Episodes for Deep Value Agents in CartPole



time step vs Episodes for Deep Value Agents in CartPole





Observations from CartPole

1. Though full convergence hasn't been attained and due to noise in training rewards, it is difficult to make exact rankings on performances, we can get some ideas using the number of peaks in the training rewards to ascertain agent most likely to be most successful. D3QN and DDQN have consistent peaks, while NFNQ remains near the bottom of the graph. PER D3QN performs worse on this particular environment.
2. Reasons for the above can be ascertained from the loss plots, which indicate gradient updates for PER D3QN are very noisy and the added complications to the agent are not actually helping. Perhaps a better hyperparameter tuning regimen might aid results.
3. Clock time tells us that PER D3QN requires the most computations on the average. This is expected since the sampling and update steps require significantly more computations while prioritising and weighting samples. This benchmark cannot be used to make hard rankings since the system was not idle while running these tasks.
4. Evaluations rewards plot makes the case for performance of D3QN and DDQN even clearer. Surprisingly DQN performs even worse than NFNQ. I suspect that the simplicity of the environment takes away the advantage of using TD learning, and having 2 networks actually makes convergence hard for DQN. This is also apparent from loss plots which have spikes roughly corresponding to update frequencies.
5. Superlative performance of PER D3QN only becomes clearer in the extremely low training regime as with very low episode counts, only it is able to consistently hit the winning score of 200. These subtle differences are hard to discern in the 1000 episode runs. Also, DQN which appeared to be worse than NFNQ earlier, can now be seen to be much better. (this is clearer from the reward vs episode plots, which are for the training rewards)
6. Time steps plot does not tell us anything more meaningful since the reward formulation of the environment already is proportional to the time steps agent survives before terminating.

Observations from MountainCar

1. These plots are available in the other notebook (some available in this also in the above code cells). Heavy compute was needed to generate small training run plots, which took over 3 hours. Unfortunately the reward formulation of the environment which only gives a positive signal on winning prevents any learning before an agent reaches the top after hundreds of thousands of episodes. Estimates on running time indicate my CPU would only be able to complete 10000 episodes in 24 hours, making a large scale training infeasible.
2. In absence of training to reach a good policy since reward signal is very weak, we see that the only variation in plots arises from variations in epsilon values. For some lucky agents the random value took them to the win, resulting in spikes.
3. For comparison I ran PPO from OpenAI on this environment and the first time it managed to reach the top was after 600,000 episodes, making large scale training infeasible for value based DRL agents.
4. Training curves look completely flat indicating agent has yet to see winning positions in training.
5. Wall clock time once again shows the huge computational burden of PER D3QN and D3QN.
6. DQN seems to be lucky! It ended up reaching some win states in evaluation runs. Perhaps it did learn something from the environment in training?

Environment Comparison

1. Since CartPole is a simpler environment with rich reward signals, even simple agents like NFNQ are competitive on it, providing meaningful comparisons. MountainCar is too tough and for limited training, all

competitive on it, providing meaningful comparison. MountainCar is too tough and for limited training, all agents perform poorly, providing no comparisons.

2. Overall we can claim PER D3QN is the best performing agent, followed by D3QN as they take the least amount of time for winning results, and have least variance in training. They do have high variance in loss curves, which deserves further investigation. Single net agent NFQ and then DDQN are show low variance due to their inherent design choices.
3. Even though D3QN and PER D3QN take less episodes, on my system they showed higher wall clock time owing to their greater compute requirements. This is also due to the fact that my implementation of PER was naive and took $O(n \log(n))$ time to update the priorities.
4. Once again, the reward formulation of the environment plays a huge role in training and convergence of DRL agents. Sparse signal of only reward on winning makes MountainCar challenging in the low training regime.

Generalisations and Findings

1. Very large experience replay buffers are not desirable since they contain old experiences, more so in environments like MountainCar in which we desperately need good samples.
2. Having 2 networks working in tandem is good and helps convergence. DDQN and D3QN are can be called almost exclusively better than DQN.
3. Deep RL agents even after training are noisy, and consistent winning rewards are difficult to attain
4. Feature Normalisation is very important for training. CartPole would not train at all until rewards were standard scaled. Similarly for the state spaces.
5. There is a tradeoff between learning consistency and loss consistenct. DDQN has stabler training convergence to high rewards, but its loss has a higher variance. This becomes more apparent in Mountain Car environment.

for reproducibility, data used for generating plots has been saved as pickle dumps

due to large number of plots, they have not been saved as pngs in images folder, but instead renderings of the notebook have been included for reference