```
import time
from collections import deque

import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

class DQN:
    '''
    Baseline Deep Q-Network with experience replay
    '''

    def __init__(self, state_size, action_size, policy, learning_delay, loss_fn, epsilon,
 gamma,
        learning_rate, n_units, buffer_size, l2=0, learning_freq=1, verbose=False, **kwar
gs):
        '''
        Initialize necessary fields
        '''
        self.type = "DQN"

        self.state_size = state_size
        self.action_size = action_size
        self.policy = policy
        self.learning_delay = learning_delay
        self.learning_freq = learning_freq
        self.loss_fn = loss_fn
        self.epsilon = epsilon
        self.gamma = gamma
        self.optimizer = keras.optimizers.Adam(lr=learning_rate)
        self.setup_model(n_units, l2=l2)

        self.epsilon_log = []
        self.reward_log = []
        self.loss_log = []
        self.deque_log = []
        self.verbose = verbose
        self.replay_buffer = {
                    "states": deque(maxlen=buffer_size),
                    "actions": deque(maxlen=buffer_size),
                    "rewards": deque(maxlen=buffer_size),
                    "next_states": deque(maxlen=buffer_size),
                    "dones": deque(maxlen=buffer_size)
        }
        self.episode = 0

    def build_model(self, n_units, activation="elu", l2=0):
        '''
        Build a simple sequential model.
        '''
        print("L2: {}".format(l2))

        model = Sequential()
        i = 0

        # Input layer
        model.add(InputLayer(input_shape=(self.state_size,)))

        # Loop over hidden layers
        for n in n_units:
            model.add(Dense(n,
                        activation=activation,
                        kernel_regularizer=keras.regularizers.l2(l2),
                        name = "D"+str(i)))
            i=i+1

        # model.add(BatchNormalization())
```

```
        # Output layer
        model.add(Dense(self.action_size,
                        activation=None,
                        name = "D"+str(i)))

        return model

    def setup_model(self, n_units, l2=0):
        '''
        Compile a simple sequential model
        '''

        model = self.build_model(n_units=n_units, l2=l2)
        self.model = model
        model.summary()

    def get_epsilon(self):
        try:
            return self.epsilon(self.episode)
        except TypeError as e:
            return self.epsilon

    def play_one_step(self, env, state):
        '''
        Take one step in the environment based on the agent parameters
        '''

        action = self.policy(state, self.model, self.get_epsilon()) # Query policy

        next_state, reward, done, info = env.step(action) # Query environment
        self.memorize(state, action, reward, next_state, done) # Log

        return next_state, reward, done, info

    def memorize(self, state, action, reward, next_state, done):
        '''
        Log the experience from one step into the replay buffer as a dictionary
        '''

        state =  np.array(state, ndmin=2)
        next_state =  np.array(next_state, ndmin=2)

        self.replay_buffer["states"].append(state)
        self.replay_buffer["actions"].append(action)
        self.replay_buffer["rewards"].append(reward)
        self.replay_buffer["next_states"].append(next_state)
        self.replay_buffer["dones"].append(done)

    def sample_experience_inds(self, batch_size):
        '''
        Sample batch_size number of experience indices from the replay buffer
        '''

        # If batch size greater than current length of buffer, give all indices for buffe
r.
        # Otherwise, get random sampling of batch_size indices.
        choice_range = len(self.replay_buffer["states"])
        if batch_size is None or batch_size > choice_range:
            indices = np.random.choice(choice_range, size=choice_range, replace=False)
        else:
            indices = np.random.choice(choice_range, size=batch_size, replace=False)

        return indices

    def sample_experience_inds_old(self, batch_size):
        '''
        Sample batch_size number of experience indices from the replay buffer
        '''

        # If batch size greater than current length of buffer, give all indices for buffe
```

```
r.
        # Otherwise, get random sampling of batch_size indices.
        if batch_size > len(self.replay_buffer["states"]):
            indices = list(range(len(self.replay_buffer["states"])))
        else:
            indices = np.random.randint(len(self.replay_buffer["states"]), size=batch_siz
e)

        return indices

    def sample_experience(self, inds):
        '''
        Sample experiences with indices from replay buffer
        '''

        batch = {}
        for key in self.replay_buffer.keys():
            batch[key] = [self.replay_buffer[key][index] for index in inds]


        batch["states"] = np.concatenate(batch["states"], axis=0)
        batch["next_states"] = np.concatenate(batch["next_states"], axis=0)

        return batch

    def get_current_Q_values(self, states):
        return self.model(states)

    def get_next_Q_values(self, next_states):
        return self.model.predict(next_states)

    def learning_step(self, batch_size=100):
        '''
        Train the model with one batch by sampling from replay buffer
        Use the gradient tape method
        '''

        batch_time_start = time.time()
        # Fetch batch
        batch_inds = self.sample_experience_inds(batch_size)
        batch = self.sample_experience(batch_inds)

        # Create target q values, with mask to disregard irrelevant actions
        next_Q_values = self.get_next_Q_values(batch["next_states"]) # Get subsequent Q v
alues
        max_next_Q_values = np.max(next_Q_values, axis=1) # Get max of subsequent Q value
s
        target_Q_values = (batch["rewards"] + (1 - np.asarray(batch["dones"])) * self.gam
ma * max_next_Q_values) # Define Q targets
        mask = tf.one_hot(batch["actions"], self.action_size) # Create mask to mask actio
ns not taken

        # Use optimizer to apply gradient to model
        with tf.GradientTape() as tape:
            all_Q_values = self.get_current_Q_values(batch["states"]) # Get all possible
q values from the states
            masked_Q_values = all_Q_values * mask # Mask the actions which were not taken
            Q_values = tf.reduce_sum(masked_Q_values, axis=1) # Get the sum to reduce to
action taken
            loss = tf.reduce_mean(self.loss_fn(target_Q_values, Q_values)) # Compute the
losses
            self.loss_log.append(loss) # Append to log
            grads = tape.gradient(loss, self.model.trainable_variables) # Compute the gra
dients
            self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables)) #
Apply the gradients to the model

    def execute_episode(self, env, n_steps=None, render_flag=False, batch_size=100, verbo
se=False,
        train=True):
```

```
        '''
        Execute one episode, which terminates when done if flagged or step limit is reach
ed
        '''

        # Initialize vars
        reward_total = 0
        step = 0
        done = False
        state = env.reset()
        while (n_steps is None or step < n_steps) and not done: # Continue till step coun
t, or until done

            if render_flag: # Create visualization for environment
                env.render()

            state, reward, done, info = self.play_one_step(env, state) # Custom step func
tion
            reward_total += reward
            step += 1
            if done:
                break

        # If train flag and episode above some threshold (to fill buffer), train
        if train and self.episode >= self.learning_delay and self.episode % self.learning
_freq == 0:
            print("\tLearning")
            self.learning_step(batch_size=batch_size)
        else:
            print("\tCollecting")

        self.reward_log.append(reward_total)
        self.epsilon_log.append(self.get_epsilon())
        self.deque_log.append(len(self.replay_buffer["states"]))
        self.episode += 1

        if verbose:
            print("\tReward: {}".format(reward_total))

    def execute_episodes(self, env, n_episodes, n_steps, render_flag=False, batch_size=10
0, verbose=False,
        train=True):
        '''
        Execute multiple episodes
        '''

        for episode in range(n_episodes):
            if verbose:
                print("Episode: {}".format(self.episode))

            self.execute_episode(
                env=env,
                n_steps=n_steps,
                render_flag=render_flag,
                batch_size=batch_size,
                verbose=verbose,
                train=train)

            if render_flag:
                env.close()
```

```
import tensorflow.keras as keras

from .DQN import DQN

class TargetDQN(DQN):

    def __init__(self, target_update_freq, **kwargs):
        super().__init__(**kwargs)

        self.type = "TargetDQN"

        self.setup_target_model()
        self.target_update_freq = target_update_freq

    def setup_target_model(self):
        self.target_model = keras.models.clone_model(self.model)
        self.update_target_model()

    def update_target_model(self):
        self.target_model.set_weights(self.model.get_weights())

    def get_next_Q_values(self, next_states):
        return self.target_model.predict(next_states)

    def execute_episode(self, **kwargs):
        super().execute_episode(**kwargs)

        if self.episode % self.target_update_freq == 0:
            self.update_target_model()
            if "verbose" in kwargs.keys():
                if kwargs["verbose"]:
                    print("\tUpdate target model")
```

```
#!/usr/bin/env python3

import sys
import glob
import pickle

import numpy as np
import matplotlib.pyplot as plt

def plot_agent(agent_folder):
    # Plot results
    print("Agent folder: {}".format(agent_folder))

    trial_folders = glob.glob("{}/*".format(agent_folder))
    print("Trial folders: {}".format(trial_folders))
    all_trials_results = []
    for trial_folder in trial_folders:
        with open("{}/results_dict.pkl".format(trial_folder), "rb") as f:
            all_trials_results.append(pickle.load(f))

    rewards = [trial_results["rewards"] for trial_results in all_trials_results]
    rewards = np.stack(rewards, axis=0)
    avgs = np.mean(rewards[:, rewards.shape[1]-100-1:], axis=1)
    print("Rewards shape: {}".format(rewards.shape))
    print("Averages shape: {}".format(avgs.shape))
    print("Averages: {}".format(avgs))

    fig, axs = plt.subplots(1)
    for i in range(rewards.shape[0]):
        axs.plot(rewards[i])

    axs.set_title("Performance")
    axs.set_xlabel("Episode")
    axs.set_ylabel("Total Reward")
    axs.set_ylim([-550, 50])
    axs.legend()

    plt.show()

if __name__ == "__main__":
    plot_agent(sys.argv[1])
```

```
import numpy as np

def epsilon_greedy_policy_generator(action_low, action_high_plus):

    def epsilon_greedy_policy(state, model, epsilon = 0):
        '''
        Simple policy which either gets the next action or random action with chance epsi
lon in cartpole
        '''

        if np.random.rand() < epsilon:
            return np.random.randint(action_low, high=action_high_plus)
        else:
            Q_values = model.predict(state[np.newaxis])
            return np.argmax(Q_values[0])

    return epsilon_greedy_policy

def random_policy(*args):
    '''
    Randomly chooses an action in cartpole
    '''

    return np.random.randint(2)

def epsilon_episode_decay(initial_epsilon, min_epsilon, rate=500):
    '''
    Linearly decays epsilon
    '''

    # Decay epsilon to some min value according to episode
    return lambda episode: max(initial_epsilon - episode / rate, min_epsilon)

def acrobot_epsilon_decay(initial_epsilon, dropoff):

    def func(episode):

        if episode >= dropoff:
            return .01
        else:
            return initial_epsilon

    return func
```

```
#!/usr/bin/env python3

import statistics
import os
import pickle
import glob
import sys

import gym
import matplotlib.pyplot as plt
from yacs.config import CfgNode as CN
import tensorflow.keras as keras

from agents import DQN, TargetDQN
from policies import epsilon_episode_decay, random_policy, epsilon_greedy_policy_generato
r, acrobot_epsilon_decay

### Hyperparameter options
# gamma = [.99, 1]
# n_units = [[16, 8], [32, 16], [40]]
# learning_rate = [.01, .001]
# target_freq = [25, 50]

### Experiments: 1000 episodes; epsilon decay 300; batch size 2000; learning delay 50;
# --Gamma--
# 1: DQN; gamma=.99; n_units=[16, 8]; learning_rate=.01
# 2: DQN; gamma=1; n_units=[16, 8]; learning_rate=.01
# 3: DQN; gamma=.98; n_units=[16, 8]; learning_rate=.01
# 4: DQN; gamma=.97; n_units=[16, 8]; learning_rate=.01
# 5: DQN; gamma=.96; n_units=[16, 8]; learning_rate=.01 <--
# 6: DQN; gamma=.95; n_units=[16, 8]; learning_rate=.01
# 7: DQN; gamma=.94; n_units=[16, 8]; learning_rate=.01
# 8: DQN; gamma=.93; n_units=[16, 8]; learning_rate=.01
# --> Pick best
# --Learning rate--
# 9: DQN; gamma=.96; n_units=[16, 8]; learning_rate=.001 <--
# --> Pick best
# --Network--
# 10: DQN; gamma=.96; n_units=[32, 16]; learning_rate=.001
# 11: DQN; gamma=.96; n_units=[40]; learning_rate=.001 <--
# --> Pick best
# Target update frequency
# 12: TargetDQN; gamma=.96; n_units=[40]; learning_rate=.001; target_freq=25 <--
# 13: TargetDQN; gamma=.96; n_units=[40]; learning_rate=.001; target_freq=50
# --> Pick best
# Batch size :::: TRASHED BC TARGET FREQ 50
# 14: TargetDQN; gamma=.96; n_units=[40]; learning_rate=.001; target_freq=25; batch size
64
# 15: TargetDQN; gamma=.96; n_units=[30]; learning_rate=.001; target_freq=25
# 16: TargetDQN; gamma=.96; n_units=[20]; learning_rate=.001; target_freq=25
# 17: TargetDQN; gamma=.96; n_units=[60]; learning_rate=.001; target_freq=25
# 18: Experiment 12 for 3000 episodes
# 19: TargetDQN; gamma=.99; n_units=[40]; learning_rate=.001; target_freq=25 <--
# 20: Experiment 19 for 3000 episodes
# 21: TargetDQN; gamma=1; n_units=[40]; learning_rate=.001; target_freq=25
# 22: TargetDQN; gamma=.99; n_units=[40]; learning_rate=.001; target_freq=25; Epsilon dec
ay to .01 <--
# 23: TargetDQN; gamma=1; n_units=[40]; learning_rate=.001; target_freq=25; Epsilon decay
 to .01
# 24: TargetDQN; gamma=1; n_units=[40]; learning_rate=.001; target_freq=50; Epsilon decay
 to .01
# 25: TargetDQN; gamma=.99; n_units=[40]; learning_rate=.001; target_freq=50; Epsilon dec
ay to .01
# 26: Experiment 22 with L2 .1
# 27: Experiment 22 with L2 .01
# 28: Experiment 22 with L2 .001
# 29: Experiment 22 with L2 .0001
# Final choice: Experiment 22. Before final test, might need to remove the regularizer st
uff
```

```
# Figure 1: 5 independent runs with experiment 22
# Figure 2: 5 independent runs with experiment 22 modified with learning rate of .01

def save_results_and_models(agent, agent_folder, trial_name):
    fbase = "{}/".format(agent_folder)
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    fbase = "{}/".format(fbase + trial_name)
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    results = {}
    results["rewards"] = agent.reward_log
    results["losses"] = agent.loss_log
    print("Reward log length: {}".format(len(results["rewards"])))
    print("Loss log length: {}".format(len(results["losses"])))

    # Save full results binary
    with open("{}results_dict.pkl".format(fbase), "wb") as f:
        pickle.dump(results, f)

    if agent.type == "DQN":
        agent.model.save("{}model.h5".format(fbase))
    elif agent.type == "TargetDQN":
        agent.model.save("{}model.h5".format(fbase))
        agent.target_model.save("{}target_model.h5".format(fbase))

def main():

    agent_folder = sys.argv[1]
    trial_name = sys.argv[2]

    keras.backend.clear_session()

    # Create environment
    env = gym.make('Acrobot-v1')
    print("State space: {}".format(env.observation_space))
    print("Action space: {}".format(env.action_space))

    # Create agent configuration
    agent_class = DQN
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    policy = epsilon_greedy_policy_generator(-1, 2)
    loss_fn = keras.losses.mean_squared_error
    epsilon = epsilon_episode_decay(1, .1, 300)
    gamma = .99
    buffer_size = 10000
    n_units = [16, 8]
    l2 = 0
    learning_rate = .01
    learning_delay = 50
    learning_freq = 1
    verbose = True
    target_update_freq = 25

    # Create silent episode configuration
    silent_episodes = CN()
    silent_episodes.n_episodes = 1000
    silent_episodes.n_steps = 500
    silent_episodes.render_flag = False
    silent_episodes.batch_size = 2000
    silent_episodes.verbose = True

    # Create visible episodes configuration
    visible_episodes = CN()
    visible_episodes.n_episodes = 1
    visible_episodes.n_steps = 500
    visible_episodes.render_flag = False
```

```
    visible_episodes.batch_size = 2000
    visible_episodes.verbose = True

    # Build agent
    agent = agent_class(
        state_size=state_size,
        action_size=action_size,
        policy=policy,
        loss_fn=loss_fn,
        epsilon=epsilon,
        gamma=gamma,
        buffer_size=buffer_size,
        n_units=n_units,
        l2=l2,
        learning_rate=learning_rate,
        learning_delay=learning_delay,
        learning_freq=learning_freq,
        verbose=verbose,
        target_update_freq=target_update_freq
        )

    print("--Training--")
    print("\tAgent type: {}".format(agent.type))

    # Run silent episodes
    agent.execute_episodes(
        env=env,
        n_episodes=silent_episodes.n_episodes,
        n_steps=silent_episodes.n_steps,
        render_flag=silent_episodes.render_flag,
        batch_size=silent_episodes.batch_size,
        verbose=silent_episodes.verbose
        )

    # Run visible episodes
    agent.execute_episodes(
        env=env,
        n_episodes=visible_episodes.n_episodes,
        n_steps=visible_episodes.n_steps,
        render_flag=visible_episodes.render_flag,
        batch_size=visible_episodes.batch_size,
        verbose=visible_episodes.verbose,
        train=False
        )

    save_results_and_models(agent, agent_folder, trial_name)

if __name__ == "__main__":
    main()
```

```python
#!/usr/bin/env python3

import sys
import os
import subprocess

from solver import main

def start_job(agent_folder, trial_name):
    """
    Starts a job for the fed arguments. This takes the form of a subprocess,
    whether on a normal computer or supercomputer
    """

    print("Starting job: folder -> {}, trial -> {}".format(agent_folder, trial_name))

    # Decide which script to run
    if "-s" in sys.argv:
        script_to_run = ["sbatch", "supercomputer_job.sh"]
    else:
        script_to_run = ["./standard_job.sh"]

    # Build script with hyperparameters
    full_command = [
        *script_to_run,
        "{}".format(agent_folder),
        "{}".format(trial_name)
    ]

    # Run chosen script with correct arguments
    process = subprocess.Popen(full_command)

    # Wait if not parallel
    if "-p" not in sys.argv:
        process.wait()

def loop_agents_and_average(agent_folder="exp_1", n_trials=5):

    if os.path.exists(agent_folder):
        raise Exception("Folder {} already exists.".format(agent_folder))

    for trial in range(n_trials):
        start_job(agent_folder=agent_folder, trial_name="trial_{}".format(trial))

if __name__ == "__main__":

    n_trials = 5
    for arg in sys.argv:
        if "-agent_folder=" in arg:
            agent_folder = arg.replace("-agent_folder=", "")

        if "-n_trials=" in arg:
            n_trials = int(arg.replace("-n_trials=", ""))

    loop_agents_and_average(agent_folder=agent_folder, n_trials=n_trials)
```

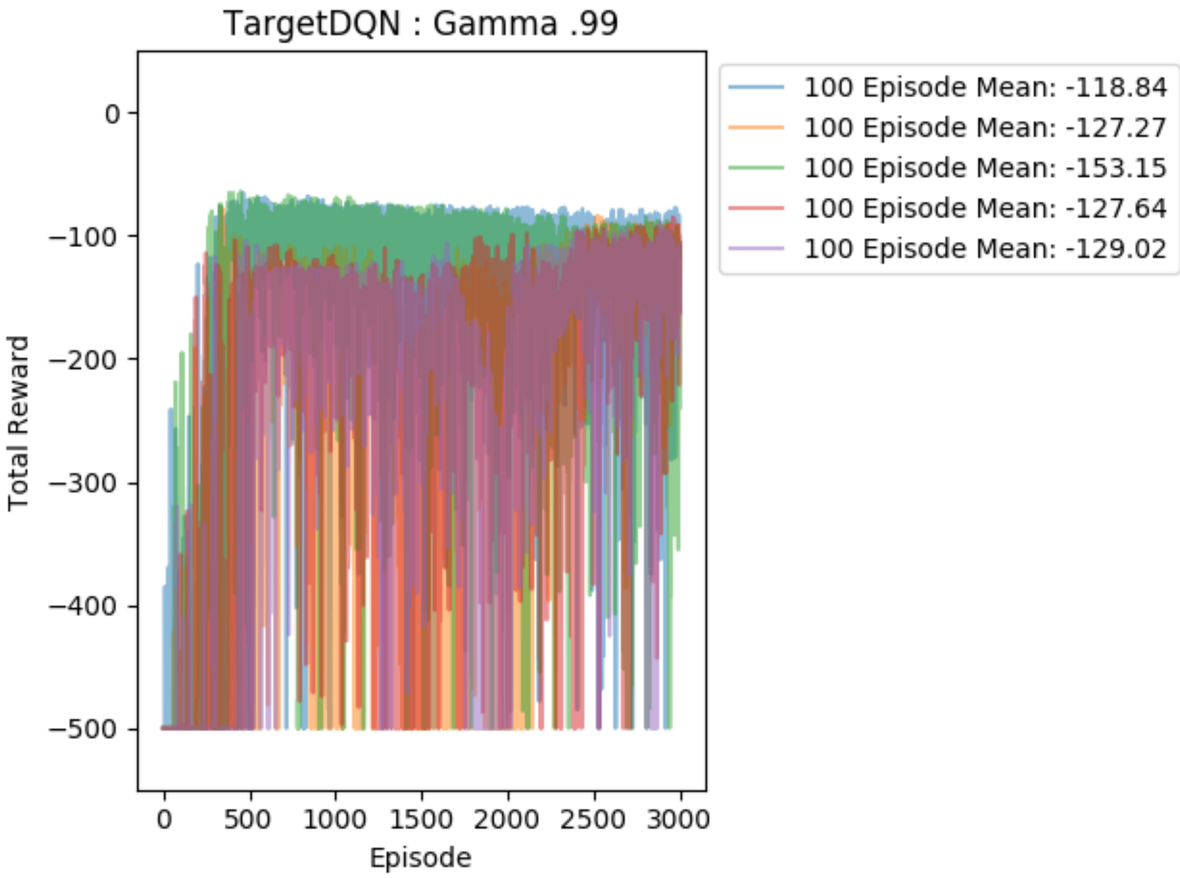
```
#!/bin/bash

python3 solver.py $@
```

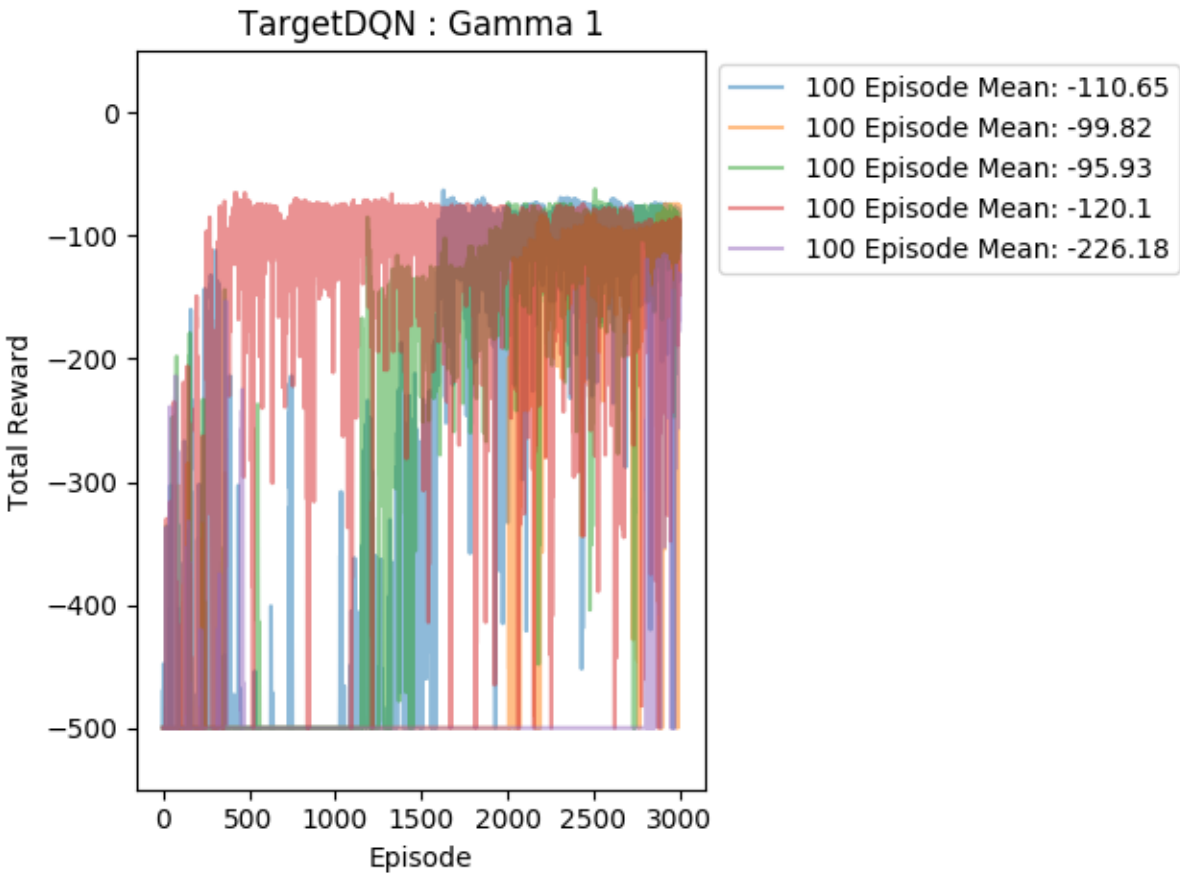
```
#!/bin/bash

#SBATCH --partition=normal
#SBATCH --ntasks=1
#SBATCH --mem=5000
#SBATCH --output=job_output/subprocess-%j-stdout.txt
#SBATCH --error=job_output/subprocess--%j-stderr.txt
#SBATCH --time=7:00:00
#SBATCH --job-name=subprocess_%j
#SBATCH --mail-user=john.w.spaeth-1@ou.edu
#SBATCH --mail-type=ALL
#SBATCH --chdir=/home/jwspaeth/workspaces/advanced-ml/homework_5/

python3 solver.py $@
```



TargetDQN : Gamma .99
Total Reward
Episode
0
−100
−200
−300
−400
−500
0
500
1000
1500
2000
2500
3000
100 Episode Mean: -118.84
100 Episode Mean: -127.27
100 Episode Mean: -153.15
100 Episode Mean: -127.64
100 Episode Mean: -129.02


TargetDQN : Gamma 1
100 Episode Mean: -110.65
100 Episode Mean: -99.82
100 Episode Mean: -95.93
100 Episode Mean: -120.1
100 Episode Mean: -226.18
Total Reward
Episode
0
−100
−200
−300
−400
−500
500
1000
1500
2000
2500
3000