PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
ELECTRICAL ENGINEERING DEPARTMENT
DOCTORATE IN DECISION SUPPORT METHODS

REINFORCEMENT LEARNING FINAL PROJECT

Gabriel Baruque

2012325

Rio de Janeiro

2020

GABRIEL BARUQUE

REINFORCEMENT LEARNING FINAL PROJECT

Work presented to class ELE2394 – Reinforcement Learning, of Post Graduate program in Support Decision Methods from Electrical Engineering Coordination of PUC-Rio as partial request of evaluation.

Professor: Wouter Caarls

Rio de Janeiro

2020

INTRODUCTION

In order to solve a task with reinforcement learning, many algorithms can be used. Their performance may vary depending on various aspects of the problem, or task, that the agent has to learn. For some tasks the best choice is value iteration, while for others, a more direct approach, using policy improvement, will give better results. Actor-critic methods try to merge the best of both worlds, having a critic, that evaluate the value function, and an actor, that decides the best action to take. There is also the option to use a model of the environment to help learning the task using fewer real episodes. It all depends on the environment and task to be learned.

One of the approaches to implement reinforcement learning takes account of the Q-value, which is the value of being in a given state and taking a given action. The Q-learning seeks to improve the policy minimizing the error of estimated Q function and a target. This can be made online, using each sample of information (transition) individually, or using a whole batch of samples, in an off-policy way to learn.

Two Q learning algorithms are used in this work: Deep Deterministic Policy Gradient (DDPG) and Clipped Q-learning. DDPG uses an approximation of the Q function with deep neural networks, considering the optimal policy as deterministic. The clipped Q-learning is implemented in order to prevent overestimation bias of the Q-function, due to the error in approximation. A version using a Deep Q network (DQN) and a DDPG is used.

The objective of this work is to implement both algorithms, and explain in detail its differences, comparing the results obtained in training an agent. Features like time of convergence, end performance and rise time will be evaluated.

The experiments will be made in the OpenAI HalfCheetah v3 environment. This is a simulation of a 2D bipedal agent, and the goal is to learn to “walk” as fast as possible.

ALGORITHMS USED

Deep Deterministic Policy Gradient

DDPG is a method that uses a deterministic policy gradient, and with a compatible approximation (analytical or through experience replay) is able to learn the task. It is an implementation of actor-critic method, that combines a critic to evaluate the state-action value function, and an actor to perform an action, based on the values found by the critic. In this work, the experience replay is used in order to train the approximator (deep network), meaning that this algorithm is trained off-policy.

Figure 1 shows DDPG pseudo code:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 1: DDPG pseudo code

To use DDPG, the base script from class practical exercise 5 (ex5.py) was used. Some minor changes were made in order to adjust to the new environment. Other than that, the script used to implement the learning of DDPG was based on exercise 4 solution, were we used DDPG to solve the pendulum problem. Again, some changes were made to accommodate the new environment, and a new feature of rollouts (skip network training for *K* episodes, and train *K* times, the next one) was added. This feature ended up slowing training, therefore, it was not used in this work, as each training run took about 4 hours to finish without using rollouts.

This algorithm was used as a baseline, as we covered it in class, knowing that it has a satisfactory performance in continuous action and state spaces. It is also known that convergence is not guaranteed for DDPG, which may influence its performance.

Clipped Deep Q-learning

Deep Q-learning (DQL) uses a general Q function approximator, a deep network, to approximate the value of each state and action. Also, it makes use of a target network that slowly tracks the original network, in order to reduce the influence of an update on the value of the next state. But using this target network does not prevent overestimation of the values.

The clipped Deep Q-learning is an updated version of the vanilla Deep Q-learning seeking to prevent overestimation of the errors in the state-action value function

approximation, and therefore, suboptimal policies. To accomplish that, the minimum state-action value of two network approximation is taken to update the targets.

Before the clipped version of Deep Q-learning, the Double Deep Q Networks (DDQN) was one of the state of art algorithms [Hasset et. al, 2015]. It used two deep networks in order to update the target. The "main" network was responsible to evaluate the action chosen by the second network, the target network.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 2: Double Deep Q Network pseudo code

The clipped update to this implementation came in 2018 with Fujimoto et. al. We can see in Figure 4 the pseudo code of the clipped version of DDQN.

Algorithm 1 : Clipped Double Q-learning (Fujimoto et al., 2018)

Initialize networks Q_{θ_1}, Q_{θ_1}, replay buffer $\mathcal{D}$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma min_{i=1,2} Q_{\theta_i}(s_{t+1}, argmax_{a'} Q_{\theta_i}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 3: Clipped Double Q Learning pseudo code

The Clipped DQL implementation used in this work uses a DDPG and a Deep Q network (DQN), respectively acting as the first and second network. The actions were taken based on the actor of DDPG.

The ex5.py was also used as a base script, and the implementation of the Clipped DQL had some inspiration in the DDPG code, changing the learning to in fact, apply the Clipped algorithm. It is important to say that, in this work, rather than implementing two Deep Q network, as in the original implementation of Clipped DQL [Fujimoto et. al, 2018], a DDPG network and a DQN network were implemented and used to clip the approximation of the value function.

ENVIRONMENT

As mentioned, the environment used to experiment the implementations of DDPG and Clipped DQL was the openAI gym HalfCheetah-v3. Similar to a feline, this environment offers a 2D agent that has 2 legs, with 3 junction each, and the goal is to learn to walk as fast as possible.

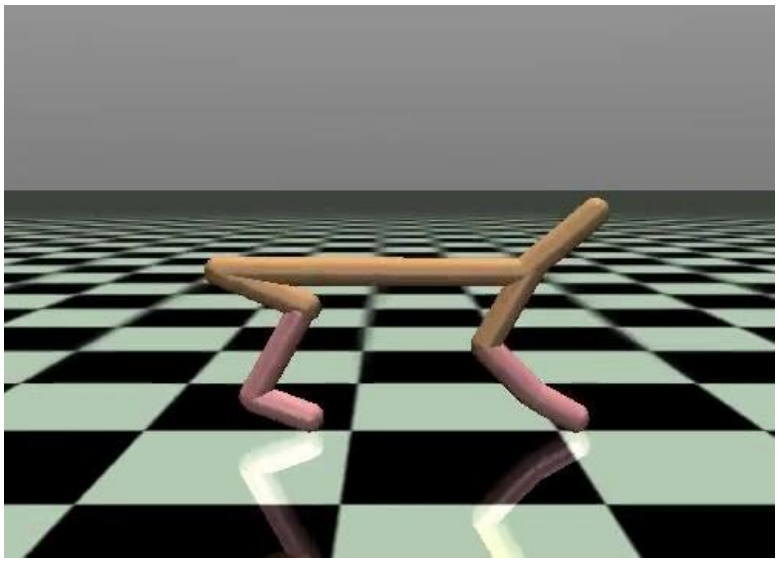

Figure 4: HalfCheetah-v3 environment

Unfortunately, there was few information about this environment provided either by openAI or MuJoCo.

Through experimentation, the range of the 6 actions was learned to be in the continuous range [0,1]. Each of them represents a joint of the robot, being 3 in rear leg and 3 in front leg.

I was not able to find the range for states or even what each one of them represented for the model. The closest information found online was from openAI repository: https://github.com/openai/gym/blob/345c65973fc7160d8be374745a60c36869d8accc/gym/envs/mujoco/assets/half_cheetah.xml

In this script, an explanation of 18 states is given, but since the gym environment requires 17 states, the information gets a bit blurry:

```
The state space is populated with joints in the order that they are
      defined in this file. The actuators also operate on joints.

      State-Space (name/joint/parameter):
          - rootx     slider      position (m)
          - rootz     slider      position (m)
          - rooty     hinge       angle (rad)
          - bthigh    hinge       angle (rad)
          - bshin     hinge       angle (rad)
          - bfoot     hinge       angle (rad)
          - fthigh    hinge       angle (rad)
          - fshin     hinge       angle (rad)
          - ffoot     hinge       angle (rad)
          - rootx     slider      velocity (m/s)
          - rootz     slider      velocity (m/s)
          - rooty     hinge       angular velocity (rad/s)
          - bthigh    hinge       angular velocity (rad/s)
          - bshin     hinge       angular velocity (rad/s)
          - bfoot     hinge       angular velocity (rad/s)
          - fthigh    hinge       angular velocity (rad/s)
          - fshin     hinge       angular velocity (rad/s)
          - ffoot     hinge       angular velocity (rad/s)
      Actuators (name/actuator/parameter):
          - bthigh    hinge       torque (N m)
          - bshin     hinge       torque (N m)
          - bfoot     hinge       torque (N m)
          - fthigh    hinge       torque (N m)
          - fshin     hinge       torque (N m)
          - ffoot     hinge       torque (N m)
```

Also, this environment is considered unsolved, or at least, there is no accumulated return set to represent it as “solved”. Some implementations found on internet considered it solved with a return of about 4800-5000.

HYPERPARAMETERS

In this section, we will go through the hyperparameters used in each experiment and discuss why they were chosen. In total, four experiments were made, two with DDPG and two with Clipped DQL. This is not, by all means, an extensive amount of experiments needs to make a performance comparison between two given algorithms. More experiments need to be made, but due to high time-consuming experiments, these were not possible.

- Experiment 1 – DDPG with small exploration:

Table 1 – DDPG: experiment 1 hyperparameters

Hyperparameters		
Discount Rate:		0.99
Exploration Schedule	Start:	0.1
	Min:	0.01
	Rate:	0.98
Batch Size:		256
Update Target Network Interval:		800
Network Hidden Layers:		[400,300]

The hyperparameters chosen in all experiments were based on what we saw on classes. Usually discount rate is near one, to give credit to future rewards too. Exploration schedule was set starting from 0.1 because of a known "problem" with the cheetah environment. Usually, the agent tends to flip upside down when exploring different actions, and keep that way, trying to move its legs in order to "propel" it forward (maybe the upside down position is somewhat more stable, or can be considered a local minimum). This can be mitigated limiting how much the agent can explore. So, this approach, starting exploration schedule with a small error, was taken.

The update interval to copy parameters from the "original" network to the target network was set to 800 time-steps. This was chosen based on the information that delaying this copy would make the learning more stable, so the fast updates would not influence the next state so quickly. More values have to be tested for improvements.

Finally, the number of hidden layers and neurons in each one was chosen following Fujimoto et. al in "Addressing Function Approximation Error in Actor-Critic Methods".

- Experiment 2 – DDPG with big exploration:

Table 2 – DDPG: experiment 2 hyperparameters

Hyperparameters		
Discount Rate:		0.99
Exploration Schedule	Start:	0.8
	Min:	0.01
	Rate:	0.98
Batch Size:		256
Update Target Network Interval:		800
Network Hidden Layers:		[400,300]

In order to compare results, the second experiment with DDPG was made with the same hyperparameter, but exploration schedule was chosen like if there was no "local minimum" upside down problem. Therefore, a start value of 0.8 was chosen, to – almost only – explore in the beginning episodes, and over time, keep choosing the best actions.

- Experiment 3 – Clipped DQL with small exploration:

Table 3 – Clipped DQL: experiment 3 hyperparameters

Hyperparameters		
Discount Rate:		0.99
Exploration Schedule	Start:	0.1
	Min:	0.01
	Rate:	0.98
Batch Size:		256
Update Target Network Interval:		800
Network Hidden Layers:		[400,300]

The parameters were kept unchanged to make it possible to compare both algorithms. In this first run of Clipped DQL, the exploration schedule started with 0.8.

- Experiment 4 – Clipped DQL with big exploration

Table 4 – Clipped DQL: experiment 4 hyperparameters

Hyperparameters		
Discount Rate:		0.99
Exploration Schedule	Start:	0.8
	Min:	0.01
	Rate:	0.98
Batch Size:		256
Update Target Network Interval:		800
Network Hidden Layers:		[400,300]

The last experiment, following the previous pattern, was applying Clipped DQL with a big exploration schedule.

RESULTS

Trying to get statistically meaningful results, each experiment consisted in 3 runs of 500 episodes of training, followed by 100 episodes of testing. For each experiment, the mean values of the training phase were plotted in a graph, and the mean of all end performances (testing part) was calculated. Also, the total time taken to finish each experiment is shown.

- Experiment 1 – DDPG with small exploration:

Average End-performance:	3019.37
Average training time:	15682.6 (s)
Total time:	48072 (s)

Discount rate (gamma):0.99
Exploration schedule: 0.1 (min = 0.01 | rate = 0.98)
Batch Size:256
Update interval (t_net): 800
Network:[400, 300]
Rollouts —> Time-steps skip: 1
Network trainings:1

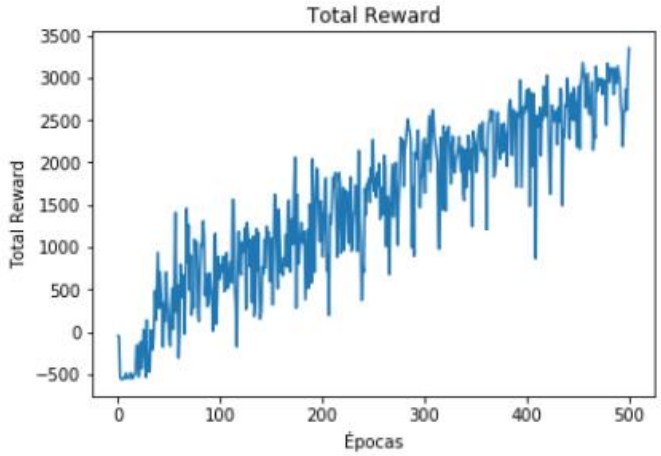

Figure 5: Total Reward of experiment 1: DDPG with small exploration

The first experiment showed that DDPG could learn a good walking pattern. This was not optimal, since many other algorithms could achieve more than 4000 as total reward, but the training was capped at 500 episodes. It is clear that the algorithm was still learning to improve performance, since it was increasing the accumulated reward consistently. Maybe if given more time, DDPG would perform better than showed in the graph.

Also, we can observe that the graph is not so stable. The variation in total reward from one episode to another is evident. This is something that clipped version of DQL try to minimize too.

- Experiment 2 – DDPG with big exploration:

Average End-performance:	2326.3
Average training time:	15053.40 (s)
Total time:	46297.60 (s)

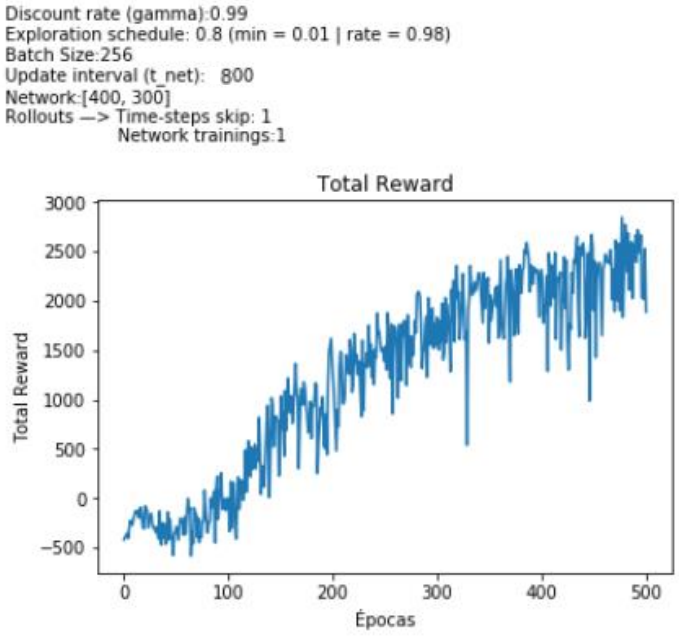

Figure 6:Total Reward of experiment 1: DDPG with big exploration

In this experiment, we observe that training time and total time are about the same, as expected. One interesting thing to note is that the average End Performance is smaller than the previous experiment. This is probably due to the mentioned “local minimum”. When exploring more, the agent flips and start trying to improve its movement while upside down. For some reason (maybe this position offers more stability) the agent tries to improve the “walk” in this position. As mentioned, to prevent that in the previous experiment, less exploration was adopted.

About stability, it is similar to experiment 1, since no changes other than exploration schedule were made.

- Experiment 3 – Clipped DQL with small exploration:

Average End-performance:	3952.8
Average training time:	21156.40 (s)
Total time:	64392.82 (s)

Discount rate (gamma):0.99
Exploration schedule: 0.1 (min = 0.01 | rate = 0.98)
Batch Size:256
Update interval (t_net): 800
Network:[400, 300]
Rollouts —> Time-steps skip: 1
Network trainings:1

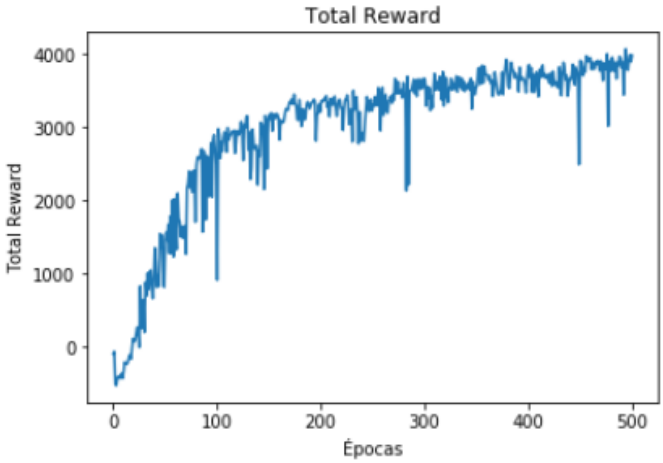

Figure 7: Total Reward of Clipped DQL with small exploring

This experiment shows how the clipped version of DQL is able to perform better than DDPG. The choosing of the lower value of Q value function avoid the overestimation error, increasing performance in training. Also, the learning is more stable.

While training, one run ended up having a “bad” end performance of about 1800. Again, this was probably caused by the flipping problem of this environment, but the exploration schedule using 0.1 as initial sigma value was able to avoid this problem in 2 of 3 runs.

Average end performance could achieve near 4000, even when the agent was flipped in one run. The two other runs achieved around 5000.

- Experiment 4 – Clipped DQL with big exploration

Average End-performance:	2933
Average training time:	20314.74 (s)
Total time:	61323.60 (s)

Discount rate (gamma):0.99
Exploration schedule: 0.8 (min = 0.01 | rate = 0.98)
Batch Size:256
Update interval (t_net): 800
Network:[400, 300]
Rollouts —> Time-steps skip: 1
Network trainings:1

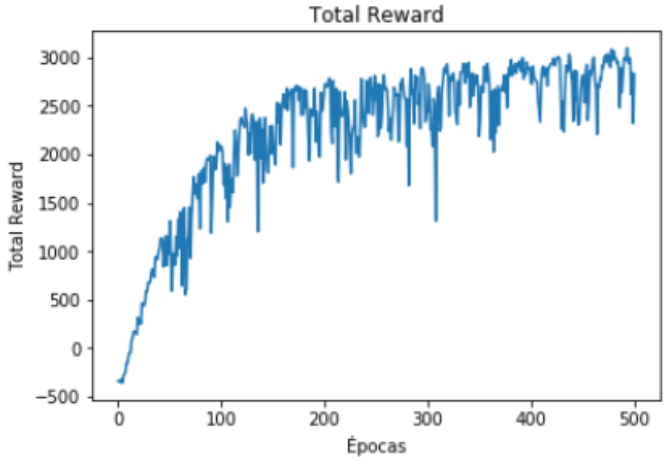

Figure 8: Total Reward of Clipped DQL with big exploring

In the last experiment, we can see that as in experiment 3, stability in learning was improved. In this case, the average End Performance was smaller than previous experiment. Again, it is probably due to the mentioned tendency the agent has to flip around. It was observed that 2 runs had end performance around 2500-2800, while one run had End Performance of about 5000. This means that for two runs, agent leant the best way to move forward while flipped around, decreasing the End performance.

We can see that Clipped DQL is more consistent while learning, converging faster (less episodes) and more stable, at the cost of computation complexity increased, since both training time and full time are bigger than in DDPG. More runs would be crucial to have more statistical meaningful results, but this work can give a prelude of the expected performance of both approaches.

PROBLEMS FOUND

Two were the main problems encountered in these experiments. The first one was time. Both algorithms were very time-consuming, since they used deep neural networks to learn the task. The second was the particularity of the environment. with the tendency of flipping upside down. This problem could be solved implementing the rollouts, but some tests were going to be needed and, with the rollouts, the execution time would be even slower.

IMPROVEMENTS AND OBSERVATIONS

To improve results another technique could be implemented, the target policy smoothing regularization. This technique aims to reduce the variance of the target, caused by function approximation errors, through regularization.

CODES

DDPG:

```
env = Cheetah()
rodadas = 3
train_episodes = 500
test_episodes = 100

runs_total_reward = []
runs_computation_time = []
runs_end_performance = []

full_start_time = time()

for rodada in range(rodadas):


    K = 1 # rollout
    lista_rewards = np.empty(train_episodes)

    camadas = [400,300]

    #17 elementos em cada estado
    #6 ações ao mesmo tempo entre -1 e +1
    # DDPG
    ddpg = DDPG(17,6,camadas)
    # target DDPG
    t_ddpg = DDPG(17,6,camadas)
    # Create replay memory
    memory = Memory(17,6)

    # Parameters
    gamma = 0.99
    sigma = 0.8
    batch_size = 256
    target_delay = 800


    training_start_time = time()

    print('----------------------------Episodes start----------------------------')
    for e in range(train_episodes):
        # reset total reward
        total_reward = 0

        # Reset environment
        s = env.reset()
```

```
#----------------------------Time Steps start----------------------------
for t in range(2000):
    # valor do estado dada uma ação (ação padrão é ação do actor)
    Q = ddpg.critic(s)

    # Select action ( sample a from policy(s,actor) + Noise )
    a = np.random.normal(ddpg.actor(s), sigma)

    #clip actions in range [-1,1]
    for i in range(6):
        if a[i] < -1:
            a[i] = -1
        elif a[i] > 1:
            a[i] = 1

    # Step environment
    s_p, r, done, info = env.step(a)

    # Add transition to replay memory
    memory.add(s, a, r, s_p ,done)

    ########## total reward ########
    total_reward = total_reward + r
    ################################

    #----------------------------Training start----------------------------
    # If at least 2000 transitions in memory, sample minibatch and learn
    if memory.n >= 2000 and t%K == 0:
        for k in range(K):
            # choose random minibatch
            b_s, b_a, b_r, b_s_p, b_done = memory.sample(batch_size)


            #train minibatch
            Q_sp_ap = t_ddpg.critic(b_s_p)


            targets = np.empty(batch_size)
            for i in range(batch_size):
                if b_done[i]:
                    targets[i] = b_r[i]
                else:
                    targets[i] = b_r[i] + gamma * Q_sp_ap[i]

            ddpg.train(b_s, b_a, targets)
    #----------------------------Training finish----------------------------

    s = np.copy(s_p)
```

```
            #Render current state
#           if t%1 == 0:
#               env.render()

            # Every 200 timesteps, copy weights to target network
            if t%target_delay == 0:
                t_ddpg <<= ddpg

            if done:
                break

        #---------------------------Time Steps finish---------------------------
        sigma = sigma * 0.98
        if sigma < 0.01:
            sigma = 0.01


        lista_rewards[e] = total_reward
        print(str(e) + 'Episódio  ––>   total reward: ' + str(total_reward))

        training_end_time = time()
        training_computation_time = training_end_time - training_start_time

    #---------------------------Episodes finish---------------------------

    # TODO: Close environment
    # env.close()

    runs_computation_time.append(training_computation_time)
    runs_total_reward.append(lista_rewards)
    print()
    print('Computation time: ' + str(training_computation_time))

    #================================= Start Evaluation
================================
    print()
    print('-----------------Iniciating Evaluation-----------------')

    lista_rewards = np.empty(test_episodes)

    #Parameters
    gamma = 0.99
    sigma = 0
    batch_size = 256

    #-------------------------Episodes start---------------------------------
    for e in range(test_episodes):
```

```
        total_reward = 0

        s = env.reset()

        #--------------------Time Step Start---------------------------------
        for t in range(2000):
            Q = ddpg.critic(s)

            a = ddpg.actor(s)

            s_p, r, done, info = env.step(a)

            ########## total reward ########
            total_reward = total_reward + r
            ################################

            s = np.copy(s_p)

            if done:
                break
        #--------------------Time Step Finish---------------------------------
        lista_rewards[e] = total_reward
        print(str(e) + 'Episódio  ––>   total reward: ' + str(total_reward))
        #-------------------------Episodes finish--------------------------------
    #================================= Finish Evaluation
================================
    runs_end_performance.append(np.mean(lista_rewards))
    print('End performance: ' + str(runs_end_performance[rodada]))
    print()
    print()

full_end_time = time()
full_computation_time = full_end_time - full_start_time
print('Full Computation time: ' + str(full_computation_time))
```

Clipped Double Q-Learning:

```
env = Cheetah()
rodadas = 3
train_episodes = 500
test_episodes = 100

runs_total_reward = []
runs_computation_time = []
runs_end_performance = []

full_start_time = time()

for rodada in range(rodadas):


    K = 1 # rollout
    lista_rewards = np.empty(train_episodes)

    camadas = [400,300]

    #17 elementos em cada estado
    #6 ações ao mesmo tempo entre -1 e +1
    # DDPG
    ddpg = DDPG(17,6,camadas)
    t_ddpg = DDPG(17,6,camadas)
    dqn = DQN(17,6,camadas)
    t_dqn = DQN(17,6,camadas)
    # Create replay memory
    memory = Memory(17,6)

    # Parameters
    gamma = 0.99
    sigma = 0.1
    batch_size = 256
    target_delay = 800


    training_start_time = time()

    print('----------------------------Episodes start----------------------------')
    for e in range(train_episodes):
        # reset total reward
        total_reward = 0

        # Reset environment
        s = env.reset()

        #----------------------------Time Steps start----------------------------
```

```
for t in range(2000):
    # Select action ( sample a from policy(s,actor) + Noise )
    a = np.random.normal(ddpg.actor(s), sigma)

    #clip actions in range [-1,1]
    for i in range(6):
        if a[i] < -1:
            a[i] = -1
        elif a[i] > 1:
            a[i] = 1

    Q1 = ddpg.critic(s,a)
    Q2 = dqn(s,a)

    # Step environment
    s_p, r, done, info = env.step(a)

    # Add transition to replay memory
    memory.add(s, a, r, s_p ,done)

    ########## total reward ########
    total_reward = total_reward + r
    ################################

    #---------------------------Training start---------------------------
    # If at least 2000 transitions in memory, sample minibatch and learn
    if memory.n >= 2000 and t%K == 0:
        for k in range(K):
            # choose random minibatch
            b_s, b_a, b_r, b_s_p, b_done = memory.sample(batch_size)


            #train minibatch
            ap = np.random.normal(ddpg.actor(b_s_p), sigma)
            Q1_sp_ap = t_ddpg.critic(b_s_p, ap)
            Q2_sp_ap = t_dqn(b_s_p, ap)


            targets = np.empty(batch_size)
            for i in range(batch_size):
                if b_done[i]:
                    targets[i] = b_r[i]
                else:
                    targets[i] = b_r[i] + gamma * min(Q1_sp_ap[i], Q2_sp_ap[i])

            ddpg.train(b_s, b_a, targets)
            dqn.train(b_s, b_a, targets)
    #---------------------------Training finish---------------------------
```

```
            s = np.copy(s_p)

            #Render current state
#             if t%1 == 0:
#                 env.render()

            # Every 200 timesteps, copy weights to target network
            if t%target_delay == 0:
                t_ddpg <<= ddpg
                t_dqn <<= dqn

            if done:
                break

        #----------------------------Time Steps finish----------------------------
        sigma = sigma * 0.98
        if sigma < 0.01:
            sigma = 0.01


        lista_rewards[e] = total_reward
        print(str(e) + 'Episódio  ——>   total reward: ' + str(total_reward))

        training_end_time = time()
        training_computation_time = training_end_time - training_start_time

    #----------------------------Episodes finish----------------------------

    # TODO: Close environment
    # env.close()

    runs_computation_time.append(training_computation_time)
    runs_total_reward.append(lista_rewards)
    print()
    print('Computation time: ' + str(training_computation_time))

    #================================= Start Evaluation
================================
    print()
    print('-----------------Iniciating Evaluation-----------------')

    lista_rewards = np.empty(test_episodes)

    #Parameters
    gamma = 0.99
    sigma = 0
    batch_size = 256
```

```
    #-------------------------Episodes start---------------------------------
    for e in range(test_episodes):
        total_reward = 0

        s = env.reset()

        #---------------------Time Step Start---------------------------------
        for t in range(2000):
            a = ddpg.actor(s)

            s_p, r, done, info = env.step(a)

            ########## total reward ########
            total_reward = total_reward + r
            ################################

            s = np.copy(s_p)

            if done:
                break
        #---------------------Time Step Finish--------------------------------
        lista_rewards[e] = total_reward
        print(str(e) + 'Episódio  ––>   total reward: ' + str(total_reward))
        #-------------------------Episodes finish---------------------------------
    #================================= Finish Evaluation
================================
    runs_end_performance.append(np.mean(lista_rewards))
    print('End performance: ' + str(runs_end_performance[rodada]))
    print()
    print()

full_end_time = time()
full_computation_time = full_end_time - full_start_time
print('Full Computation time: ' + str(full_computation_time))
```

base.py:

```
"""ELE2761 Deep Reinforcement Learning helper functinos

Implements the provided functionality to be used in your solution.

CLASSES
   Model       -- Dynamics model approximator
   Memory      -- Replay Memory
   Pendulum    -- OpenAI Gym Pendulum-v0 environment

FUNCTIONS
   rbfprojector -- Gaussian RBF projector factory
"""

from math import pi
import numpy as np
import scipy.stats
import tensorflow as tf
import matplotlib.pyplot as plt
import gym

def __gaussrbf(s, p, v, sigma):
   """Gaussian radial basis function activation.

      f = gaussrbf(s, p, v, sigma) returns the activation for the
      radial basis functions specified by (`p`, `v`, `sigma`) calculated at
      `s`. `p` is a list of position centers, `v` is a list of velocity centers,
      and `sigma` is the basis function width. The return value f is a vector
      with activations.

      `s` is a vector containing the state, or may be a matrix in which each
      row specifies a state. In that case, `f` is a matrix where each row
      contains the activation for a row in `s`.
   """

   s = np.atleast_2d(s)
   pd = np.arctan2(s[:, None, 1], s[:, None, 0]) - p.flatten()
   pd = abs((pd-pi)%(2*pi)-pi)

   dist = np.sqrt(pd**2 + ((s[:, None, 2] - v.flatten())/(8/pi))**2)
   return np.squeeze(scipy.stats.norm.pdf(dist, 0, sigma))

def rbfprojector(nbasis, sigma):
   """Returns function that projects states onto Gaussian radial basis function features.

      feature = rbfprojector(nbasis, sigma) returns a function
         f = feature(s)
      that projects a state `s` onto a Gaussian RBF feature vector `f`. `nbasis` is the number
```

```
      of basis functions per dimension, while `sigma` is their width.

      If `s` is a matrix where each row is a state, `f` is a matrix where each row
      contains the feature vector for a row of `s`.

      EXAMPLE
         >>> feature = rbfprojector(3, 2)
         >>> print(feature([0, 0, 0]))
         [0.01691614 0.05808858 0.05808858 0.19947114 0.01691614 0.05808858]
   """

   p, v = np.meshgrid(np.linspace(-pi, pi-(2*pi)/(nbasis-1), nbasis-1), np.linspace(-8, 8,
nbasis))
   return lambda x: __gaussrbf(x, p, v, sigma)


"""Base functions for Deep RL networks

  METHODS
     __ilshift__ -- Copy network weights.
     combine     -- Combine state and action vectors.
"""
class Network:
   def __init__(self, states, actions):
      self.states = states
      self.actions = actions

   def __ilshift__(self, other):
      """Copies network weights.

         network2 <<= network1 copies the weights from `network1` into `network2`. The
         networks must have the same structure.
      """

      if isinstance(self, DQN) or isinstance(self, Model):
         self.__model.set_weights(other.__model.get_weights())

      if isinstance(self, DDPG):
         self._DDPG__actor.set_weights(other._DDPG__actor.get_weights())
         self._DDPG__critic.set_weights(other._DDPG__critic.get_weights())

      return self

   def combine(self, s, a, force=False):
      """Combines state and action vectors into single network input.

         m, reshape = Network.combine(s, a) has five cases. In all cases,
         `m` is a matrix and `reshape` is a shape to which the network Q output
```

```
  should be reshaped. The shape will be such that states are in
  rows and actions are in columns of `m`.

   1) `s` and `a` are vectors. They will be concatenated.
   2) `s` is a matrix and `a` is a vector. `a` will be replicated for
      each `s`.
   3) `s` is a vector and `a` is a matrix. `s` will be replicated for
      each `a`.
   4) `s` and `a` are matrices with the same number of rows. They will
      be concatenated.
   5) `s` and `a` are matrices with different numbers of rows or
      force=True. Each `s` will be replicated for each `a`.

  EXAMPLE
     >>> print(network.combine([1, 2], 5))
     (array([[1., 2., 5.]], dtype=float32), (1, 1))
     >>> print(network.combine([[1, 2], [3, 4]], 5))
     (array([[1., 2., 5.],
             [3., 4., 5.]], dtype=float32), (2, 1))
     >>> print(network.combine([1, 2], [5, 6])) # single action only
     (array([[1., 2., 5.],
             [1., 2., 6.]], dtype=float32), (1, 2))
     >>> print(network.combine([1, 2], [[5], [6]]))
     (array([[1., 2., 5.],
            [1., 2., 6.]], dtype=float32), (1, 2))
     >>> print(network.combine([[1, 2], [3, 4]], [5, 6])) # single action only
     (array([[1., 2., 5.],
             [3., 4., 6.]], dtype=float32), (2, 1))
     >>> print(network.combine([[1, 2], [3, 4]], [[5], [6]]))
     (array([[1., 2., 5.],
             [3., 4., 6.]], dtype=float32), (2, 1))
     >>> print(network.combine([[1, 2], [3, 4]], [[5], [6]], force=True))
     (array([[1., 2., 5.],
             [1., 2., 6.],
             [3., 4., 5.],
             [3., 4., 6.]], dtype=float32), (2, 2))
"""

# Convert scalars to vectors
s = np.atleast_1d(np.asarray(s, dtype=np.float32))
a = np.atleast_1d(np.asarray(a, dtype=np.float32))

# Convert vectors to matrices for single-state environments
if self.states == 1 and len(s.shape) == 1 and s.shape[0] > 1:
   s = np.atleast_2d(s).transpose()

# Convert vectors to matrices for single-action environments
if self.actions == 1 and len(a.shape) == 1 and a.shape[0] > 1:
```

```
            a = np.atleast_2d(a).transpose()

        # Normalize to matrices
        s = np.atleast_2d(s)
        a = np.atleast_2d(a)

        # Sanity checking
        if len(s.shape) > 2 or len(a.shape) > 2:
            raise ValueError("Input dimensionality not supported")

        if s.shape[1] != self.states:
            raise ValueError("State dimensionality does not match network")

        if a.shape[1] != self.actions:
            raise ValueError("Action dimensionality does not match network")

        # Replicate if necessary
        if s.shape[0] != a.shape[0] or force:
            reshape = (s.shape[0], a.shape[0])
            s = np.repeat(s, np.repeat(reshape[1], reshape[0]), axis=0)
            a = np.tile(a, (reshape[0], 1))
        else:
            reshape = (s.shape[0], 1)

        m = np.hstack((s, a))

        return m, reshape

class DQN(Network):
    """Deep learning-based Q approximator.

       METHODS
          train      -- Train network.
          __call__   -- Evaluate network.
    """

    def __init__(self, states, actions=1, hiddens=[25, 25]):
        """Creates a new Q approximator.

           DQN(states, actions) creates a Q approximator with `states`
           observation dimensions and `actions` action dimensions. It has
           two hidden layers with 25 neurons each. All layers except
           the last use ReLU activation."

           DQN(states, actions, hiddens) additionally specifies the
           number of neurons in the hidden layers.

           EXAMPLE
```

```
        >>> dqn = DQN(2, 1, [10, 10])
    """

    super(DQN, self).__init__(states, actions)

    inputs = tf.keras.Input(shape=(states+actions,))
    layer = inputs
    for h in hiddens:
      layer = tf.keras.layers.Dense(h, activation='relu')(layer)
    outputs = tf.keras.layers.Dense(1, activation='linear')(layer)

    self.__model = tf.keras.Model(inputs, outputs)
    self.__model.compile(loss=tf.keras.losses.MeanSquaredError(),
                         optimizer=tf.keras.optimizers.Adam())

  def train(self, s, a, target):
    """Trains the Q approximator.

      DQN.train(s, a, target) trains the Q approximator such that
      it approaches DQN(s, a) = target.

      `s` is a matrix specifying a batch of observations, in which
      each row is an observation. `a` is a vector specifying an
      action for every observation in the batch. `target` is a vector
      specifying a target value for each observation-action pair in
      the batch.

      EXAMPLE
        >>> dqn = DQN(2, 1)
        >>> dqn.train([[0.1, 2], [0.4, 3], [0.2, 5]], [-1, 1, 0], [12, 16, 19])
    """

    self.__model.train_on_batch(self.combine(s, a), np.atleast_1d(target))

  def __call__(self, s, a):
    """Evaluates the Q approximator.

      DQN(s, a) returns the value of the approximator at observation
      `s` and action `a`.

      `s` is either a vector specifying a single observation, or a
      matrix in which each row specifies one observation in a batch.
      If `a` is the same size as the number of rows in `s`, it specifies
      the action at which to evaluate each observation in the batch.
      Otherwise, it specifies the action(s) at which the evaluate ALL
      observations in the batch.

      EXAMPLE
```

```
            >>> dqn = DQN(2, 1)
            >>> # single observation and action
            >>> print(dqn([0.1, 2], -1))
            [[ 12 ]]
            >>> # batch of observations and actions
            >>> print(dqn([[0.1, 2], [0.4, 3]], [-1, 1]))
            [[12]
             [16]]
            >>> # evaluate single observation at multiple actions
            >>> print(dqn([0.1, 2], [-1, 1]))
            [[12  -12]]
        """

        inp, reshape = self.combine(s, a)
        return np.reshape(np.asarray(self.__model(inp)), reshape)

    def __ilshift__(self, other):
        """Copies network weights.

          network2 <<= network1 copies the weights from `network1` into `network2`. The
          networks must have the same structure.
        """

        self.__model.set_weights(other.__model.get_weights())

        return self

class DDPG(Network):
    """Deep Deterministic Policy Gradient

      METHODS
          train       -- Train network.
          critic      -- Evaluate critic network.
          actor       -- Evaluate actor network.
    """

    def __init__(self, states, actions=1, hiddens=[25, 25]):
        """Creates a new DDPG network.

          DDPG(states, actions) creates a DDPG network with `states`
          observation dimensions and `actions` action dimensions. It has
          two hidden layers with 25 neurons each. All layers except
          the last use ReLU activation. The last actor layer uses the
          hyperbolic tangent. As such, all actions are scaled to [-1, 1]."

          DDPG(states, actions, hiddens) additionally specifies the
          number of neurons in the hidden layers.
```

```
        EXAMPLE
            >>> ddpg = DDPG(2, 1, [10, 10])
        """

        super(DDPG, self).__init__(states, actions)

        # Actor
        inputs = tf.keras.Input(shape=(states,))
        layer = inputs
        for h in hiddens:
            layer = tf.keras.layers.Dense(h, activation='relu')(layer)
        outputs = tf.keras.layers.Dense(actions, activation='tanh')(layer)
        self.__actor = tf.keras.Model(inputs, outputs)
        self.__opt = tf.keras.optimizers.Adam()

        # Critic
        inputs = tf.keras.Input(shape=(states+actions,))
        layer = inputs
        for h in hiddens:
            layer = tf.keras.layers.Dense(h, activation='relu')(layer)
        outputs = tf.keras.layers.Dense(1, activation='linear')(layer)

        self.__critic = tf.keras.Model(inputs, outputs)
        self.__critic.compile(loss=tf.keras.losses.MeanSquaredError(),
                              optimizer=tf.keras.optimizers.Adam())

    def train(self, s, a, target):
        """Trains both critic and actor.

            DDPG.train(s, a, target) trains the critic such that
            it approaches DDPG.critic(s, a) = target, and the actor to
            approach DDPG.actor(s) = max_a'(DDPG.critic(s, a'))

            `s` is a matrix specifying a batch of observations, in which
            each row is an observation. `a` is a vector specifying an
            action for every observation in the batch. `target` is a vector
            specifying a target value for each observation-action pair in
            the batch.

            EXAMPLE
                >>> ddpg = DDPG(2, 1)
                >>> ddpg.train([[0.1, 2], [0.4, 3], [0.2, 5]], [-1, 1, 0], [12, 16, 19])
        """

        # Critic
        self.__critic.train_on_batch(self.combine(s, a), np.atleast_1d(target))

        # Actor
```

```
    s = tf.convert_to_tensor(s, dtype=tf.float32)
    with tf.GradientTape() as tape:
        q = -self.__critic.call(tf.concat([s, self.__actor(s)], 1))
    grad = tape.gradient(q, self.__actor.variables)
    self.__opt.apply_gradients(zip(grad, self.__actor.variables))

def critic(self, s, a=None):
    """Evaluates the value function (critic).

        DDPG.critic(s) returns the value of the approximator at observation
        `s` and the actor's action.

        DDPG.critic(s, a) returns the value of the approximator at observation
        `s` and action `a`.

        `s` is either a vector specifying a single observation, or a
        matrix in which each row specifies one observation in a batch.
        If `a` is the same size as the number of rows in `s`, it specifies
        the action at which to evaluate each observation in the batch.
        Otherwise, it specifies the action(s) at which the evaluate ALL
        observations in the batch.

        EXAMPLE
            >>> ddpg = DQN(2, 1)
            >>> # single observation and action
            >>> print(ddpg.critic([0.1, 2], -1))
            [[ 12 ]]
            >>> # batch of observations and actions
            >>> print(ddpg.critic([[0.1, 2], [0.4, 3]], [-1, 1]))
            [[12]
             [16]]
            >>> # evaluate single observation at multiple actions
            >>> print(ddpg.critic([0.1, 2], [-1, 1]))
            [[12  -12]]
    """

    if a is None:
        s = tf.convert_to_tensor(np.atleast_2d(s), dtype=tf.float32)
        out = self.__critic(tf.concat([s, self.__actor(s)], 1)).numpy()
        return out
    else:
        inp, reshape = self.combine(s, a)
        return np.reshape(np.asarray(self.__critic(inp)), reshape)

def actor(self, s):
    """Evaluates the policy(actor).

        DDPG.actor(s) returns the action to take in state `s`.
```

```
        `s` is either a vector specifying a single observation, or a
        matrix in which each row specifies one observation in a batch.

        EXAMPLE
          >>> ddpg = DDPG(2, 1)
          >>> # single observation
          >>> print(ddpg.actor([0.1, 2]))
          [-0.23]
          >>> # batch of observations
          >>> print(dqn([[0.1, 2], [0.4, 3]]))
          [[-0.23]
           [0.81]]
      """

      single = len(np.asarray(s).shape) == 1

      s = tf.convert_to_tensor(np.atleast_2d(s), dtype=tf.float32)
      out = self.__actor(s).numpy()

      if single:
        out = out[0]

      return out

    def __ilshift__(self, other):
      self._DDPG__actor.set_weights(other._DDPG__actor.get_weights())
      self._DDPG__critic.set_weights(other._DDPG__critic.get_weights())

      return self


class Model(Network):
  """Deep learning-based dynamics model approximator.

    METHODS
      train      -- Train network on minibatch.
      fit        -- Fit network on memory.
      __call__   -- Evaluate network.
  """

  def __init__(self, states, actions=1, hiddens=[25, 25]):
    """Creates a new dynamics model approximator.

      Model(states, actions) creates a dynamics approximator with `states`
      observation dimensions and `actions` action dimensions. It has
      two hidden layers with 25 neurons each. All layers except
      the last use ReLU activation."
```

```
        Model(states, actions, hiddens) additionally specifies the
        number of neurons in the hidden layers.

        EXAMPLE
            >>> model = Model(2, 1, [10, 10])
        """

        super(Model, self).__init__(states, actions)

        inputs = tf.keras.Input(shape=(states+actions,))
        layer = inputs
        for h in hiddens:
            layer = tf.keras.layers.Dense(h, activation='relu')(layer)
        outputs = tf.keras.layers.Dense(states+1, activation='linear')(layer)

        self.model = tf.keras.Model(inputs, outputs)
        self.model.compile(loss=tf.keras.losses.MeanSquaredError(),
                           optimizer=tf.keras.optimizers.Adam())

        self.diff = False

    def train(self, s, a=None, r=None, sp=None):
        """Trains the dynamics approximator.

        Model.train(s, a, sp) trains the dynamics approximator such that
        it approaches Model(s, a) = r, sp.

        `s` is a matrix specifying a batch of observations, in which
        each row is an observation. `a` is a matrix specifying an
        action for every observation in the batch. `sp` is a matrix
        specifying the next state for each observation-action pair in
        the batch.

        EXAMPLE
            >>> bs, ba, br, bsp, bd = memory.sample(batch)
            >>> model.fit(bs, ba, bsp)
        """

        r = np.atleast_2d(r)
        if r.shape[1] > 1:
            r = r.T

        if self.diff:
            self.model.train_on_batch(self.combine(s, a), np.hstack((np.atleast_2d(sp-s), r)))
        else:
            self.model.train_on_batch(self.combine(s, a), np.hstack((np.atleast_2d(sp), r)))
```

```
  def fit(self, memory):
    """Fits the dynamics approximator.

      Model.fit(memory) fits the dynamics approximator such that it
      approaches Model(s, a) = r, sp for all `s`, `a`, `r`, `sp` in `memory`.

      EXAMPLE
        >>> model.fit(memory)
        Epoch 00393: early stopping
        Model validation loss  0.003967171715986397
    """

    p = np.random.permutation(len(memory))

    es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', mode='min', verbose=1,
patience=20)

    if self.diff:
       history = self.model.fit(np.hstack((memory.s[p], memory.a[p])),
np.hstack((memory.sp[p]-memory.s[p], memory.r[p])), verbose=0, validation_split=0.1,
epochs=1000, callbacks=[es])
    else:
       history = self.model.fit(np.hstack((memory.s[p], memory.a[p])),
np.hstack((memory.sp[p], memory.r[p])), verbose=0, validation_split=0.1, epochs=1000,
callbacks=[es])

    print('Model validation loss ', history.history['loss'][-1])

  def __call__(self, s, a):
    """Evaluates the dynamics approximator.

      r, sp = Model(s, a) returns the approximated reward `r` and next
      state `sp` at observation `s` and action `a`.

      `s` is either a vector specifying a single observation, or a
      matrix in which each row specifies one observation in a batch.
      `a` is the same size as the number of rows in `s`, and specifies
      the action at which to evaluate each observation in the batch.

      EXAMPLE
        >>> model = Model(2, 1, [10, 10])
        >>> print(model([0, 0], 0))
        (0.0, [0., 0.])
        >>> print(model([[0, 0], [1, 1]], [0, 1]))
        ([0.       , 0.12079704], [[ 0.       ,  0.       ],
                                   [-0.04185265,  0.16854128]])
    """
```

```
        inp, reshape = self.combine(s, a)
        if reshape[0] != inp.shape[0] or reshape[1] != 1:
            raise ValueError("Input does not describe one action per state")

        reshape = (reshape[0], self.states+1)
        out = np.reshape(np.asarray(self.model(inp)), reshape)

        r = out[:,-1]
        if self.diff:
            sp = out[:, :-1] + np.atleast_2d(s)
        else:
            sp = out[:, :-1]

        if len(np.asarray(s).shape) == 1:
            r = r[0]
            sp = sp[0]

        return r, sp

    def __ilshift__(self, other):
        self.__model.set_weights(other.__model.get_weights())

        return self


class Memory:
    """Replay memory

       METHODS
          add    -- Add transition to memory.
          sample -- Sample minibatch from memory.
    """
    def __init__(self, states, actions, size=1000000):
        """Creates a new replay memory.

          Memory(states, action) creates a new replay memory for storing
          transitions with `states` observation dimensions and `actions`
          action dimensions. It can store 1000000 transitions.

          Memory(states, actions, size) additionally specifies how many
          transitions can be stored.
        """

        self.s = np.ndarray([size, states])
        self.a = np.ndarray([size, actions])
        self.r = np.ndarray([size, 1])
        self.sp = np.ndarray([size, states])
        self.done = np.ndarray([size, 1])
```

```
        self.n = 0

    def __len__(self):
        """Returns the number of transitions currently stored in the memory."""

        return self.n

    def add(self, s, a, r, sp, done):
        """Adds a transition to the replay memory.

          Memory.add(s, a, r, sp, done) adds a new transition to the
          replay memory starting in state `s`, taking action `a`,
          receiving reward `r` and ending up in state `sp`. `done`
          specifies whether the episode finished at state `sp`.
        """

        self.s[self.n, :] = s
        self.a[self.n, :] = a
        self.r[self.n, :] = r
        self.sp[self.n, :] = sp
        self.done[self.n, :] = done
        self.n += 1

    def sample(self, size):
        """Get random minibatch from memory.

        s, a, r, sp, done = Memory.sample(batch) samples a random
        minibatch of `size` transitions from the replay memory. All
        returned variables are vectors of length `size`.
        """

        idx = np.random.randint(0, self.n, size)

        return self.s[idx], self.a[idx], self.r[idx], self.sp[idx], self.done[idx]

"""OpenAI Gym Environment wrapper.

  METHODS
    reset   -- Reset environment
    step    -- Step environment
    render  -- Visualize environment
    close   -- Close visualization

  MEMBERS
    states  -- Number of state dimensions
    actions -- Number of action dimensions
"""
class Environment():
```

```
    def reset(self):
        """Reset environment to start state.

          obs = env.reset() returns the start state observation.
        """
        return self.env.reset()

    def step(self, u):
        """Step environment.

          obs, r, done, info = env.step(u) takes action u and
          returns the next state observation, reward, whether
          the episode terminated, and extra information.
        """
        return self.env.step(u)

    def render(self):
        """Render environment.

          env.render() renders the current state of the
          environment in a separate window.

          NOTE
            You must call env.close() to close the window,
            before creating a new environment; otherwise
            the kernel may hang.
        """
        return self.env.render()

    def close(self):
        """Closes the rendering window."""
        return self.env.close()

"""OpenAI Gym Pendulum-v0 environment."""
class Cheetah(Environment):
    """Creates a new Pendulum environment."""
    def __init__(self):
        """Creates a new Pendulum environment.

          EXAMPLE
            >>> env = Pendulum()
            >>> print(env.states)
            3
            >>> print(env.actions)
            1
        """
        self.env = gym.make("HalfCheetah-v3")
        self.states = self.env.observation_space.shape[0]
```

```
    self.actions = self.env.action_space.shape[0]

def discretize_actions(divisions = 3):
    discrete = np.linspace(env.action_space.low, env.action_space.high, divisions).T
    return discrete.T

def step(self, u):
    """Step environment."""
    return self.env.step(np.atleast_1d(u))

def normalize(self, s):
    """Normalize state to unit circle.

       s = env.normalize(s) normalizes `s` such that its cosine-sine
       angle representation falls on the unit circle.

       EXAMPLE
          >>> env = Pendulum()
          >>> print(env.normalize([1, 1, 2])
          [0.70710678 0.70710678 2.        ]
    """

    single = len(np.asarray(s).shape) == 1

    s = np.atleast_2d(s)
    ang = np.arctan2(s[:,None,1], s[:,None,0])
    s = np.hstack((np.cos(ang), np.sin(ang), s[:,None,2]))

    if single:
       s = s[0]

    return s

def plotlinear(self, w, theta, feature=None):
    """Plot value function and policy.

       plot(w, feature) plots the function approximated by
       w^T feature(x) .

       plot(w, theta, feature) plots the functions approximated by
       w^T * feature(x) and theta^T * feature(x) .
    """
    ac = True
    if feature is None:
       feature = theta
       ac = False

    p, v = np.meshgrid(np.linspace(-pi, pi, 64), np.linspace(-8, 8, 64))
```

```
    s = np.vstack((np.cos(p.flatten()), np.sin(p.flatten()), v.flatten())).T
    f = feature(s)
    c = np.reshape(np.dot(f, w), p.shape)

    if ac:
        a = np.reshape(np.dot(f, theta), p.shape)

        fig, axs = plt.subplots(1,2)
        fig.subplots_adjust(right=1.2)

        h = axs[0].contourf(p, v, c, 256)
        fig.colorbar(h, ax=axs[0])

        h = axs[1].contourf(p, v, a, 256)
        fig.colorbar(h, ax=axs[1])

        axs[0].set_title('Critic')
        axs[1].set_title('Actor')
    else:
        fig, ax = plt.subplots(1,1)
        h = ax.contourf(p, v, c, 256)
        fig.colorbar(h, ax=ax)

        ax.set_title('Approximator')

def plotnetwork(self, network):
    """Plot network.

      plot(dqn) plots the value function and induced policy of DQN network `dqn`.
      plot(ddpg) plots the value function and policy of DDPG network `ddpg`.
      plot(model) plots the dynamics approximation of Model network `model`.
    """
    if network.states != 3 or network.actions != 1:
        raise ValueError("Network is not compatible with Pendulum environment")

    pp, vv = np.meshgrid(np.linspace(-np.pi,np.pi, 64), np.linspace(-8, 8, 64))
    obs = np.hstack((np.reshape(np.cos(pp), (pp.size, 1)),
                     np.reshape(np.sin(pp), (pp.size, 1)),
                     np.reshape(       vv , (vv.size, 1))))

    aval = np.linspace(-2, 2, 3)

    if isinstance(network, DQN):
        qq = network(obs, aval)
        vf = np.reshape(np.amax(qq, axis=1), pp.shape)
        pl = np.vectorize(lambda x: aval[x])(np.reshape(np.argmax(qq, axis=1), pp.shape))

        fig, axs = plt.subplots(1,2)
```

```
            fig.subplots_adjust(right=1.5)

            h = axs[0].contourf(pp, vv, vf, 256)
            fig.colorbar(h, ax=axs[0])
            h = axs[1].contourf(pp, vv, pl, 256)
            fig.colorbar(h, ax=axs[1])

            axs[0].set_title('Value function')
            axs[1].set_title('Policy')
        elif isinstance(network, DDPG):
            vf = np.reshape(network.critic(obs), pp.shape)
            pl = np.reshape(network.actor(obs), pp.shape)

            fig, axs = plt.subplots(1,2)
            fig.subplots_adjust(right=1.5)

            h = axs[0].contourf(pp, vv, vf, 256)
            fig.colorbar(h, ax=axs[0])
            h = axs[1].contourf(pp, vv, pl, 256)
            fig.colorbar(h, ax=axs[1])

            axs[0].set_title('Critic')
            axs[1].set_title('Actor')
        elif isinstance(network, Model):
            fig, axs = plt.subplots(len(aval), 3)
            fig.subplots_adjust(top=2,right=1.5)

            for aa in range(len(aval)):
                r, sp = network(obs, aval[aa])
                pd = np.reshape(np.arctan2(sp[:, 1], sp[:,0]), pp.shape)-pp
                vd = np.reshape(sp[:,2], vv.shape)-vv
                r = np.reshape(r, pp.shape)

                h = axs[aa, 0].contourf(pp, vv, pd, np.linspace(-0.5, 0.5, 256))
                fig.colorbar(h, ax=axs[aa, 0])
                h = axs[aa, 1].contourf(pp, vv, vd, np.linspace(-1.75, 1.75, 256))
                fig.colorbar(h, ax=axs[aa, 1])
                h = axs[aa, 2].contourf(pp, vv, r, 256)
                fig.colorbar(h, ax=axs[aa, 2])

                axs[aa, 0].set_title('Position derivative (a={})'.format(aval[aa]))
                axs[aa, 1].set_title('Velocity derivative (a={})'.format(aval[aa]))
                axs[aa, 2].set_title('Reward (a={})'.format(aval[aa]))
        else:
            raise ValueError("Input should be either DQN, DDPG or Model, not
{}".format(type(network).__name__))
```