All pages

Homework 3 - Reinforcement Learning

3.1 Base Implementation

a) Deep Q-Network (2)

> Implement a Deep Q-Network and its forward pass in the DQN class in model.py. Your network should take a single frame as input. In addition, you may again utilize the extract sensor values function. Describe your architecture in your report.

`> i) Would it be a problem if we only trained our network on a crop of the game pixels, which would not include the bottom status bar and would not use the extracted sensor values as an additional input to the network? Hint: Think about the Markov assumption.`

Yes! If we crop out the image, and lose the bottom status bar, it would be a problem. The bottom status bar gives us the following information, current steering values, and acceleration. To construct a markovian state, we need this information from either the bottom status bar in the image or from the extracted sensor values. If not, a single frame state would not be markovian. To make the state markovian in that case, we could use a stack of frames as state.

`> ii) Why do we not need to use a stack of frames as input to our network as proposed to play Atari games in the original Deep Q-Learning papers?`

In the original DL papers, they create a single agent that can play multiple Atari games, and so a stack of frames is a general way to achieve a markovian state for all those games. In our case, the sensor values of acceleration and steer along with the image are sufficient to construct a markovian state.

b) Deep Q-Learning (2)

`> i) Why do we utilize fixed targets with a separate policy and target network?`

According to the Nature DQN paper, a fixed target network makes the training more stable.

`> ii) Why do we sample training data from a replay memory instead of using a batch of past consecutive frames?`

There’s two reasons for this:

1. The memory is constrained, it is not reasonable to expect to store all the experience of the agent.
2. Reduce correlation amongs frames. For e.g. in our task, there is a lot of frames for which the agent is driving straight. To handle all the aspects of driving in the environment, the training data should sample well from various destributions, i.e. driving straight, turning left, sharpturns, slight turns, etc. Randomly samling from the replay memory helps us achieve this.

c) Action selection (2)

`> i) Why do we need to balance exploration and exploitation in a reinforcement learning agent and how does the e-greedy algorithm accomplish this?`

We want the agent to exploit so that the agent chooses the action which can yield the most reward. Exploration is needed so that the agent can try out new states, which may yield a reward better than the currently known best reward. Also, the environment can be stochastic, and the reward distribution may change over time. The RL agent needs to strike a goo dbalance between eploitation and exploration.

epsilon-greedy algorithm is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.

A(t) = maxQ(a) with probability 1-e, any action (a) with probability e

d) Training (2)

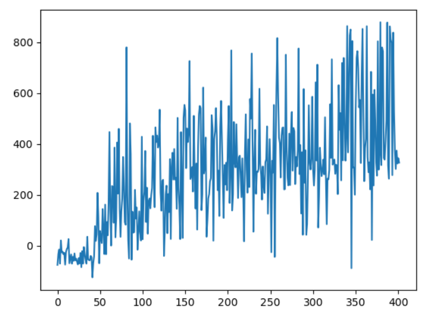
`> Train a Deep Q-Learning agent using the train racing.py file with the provided default parameters. Describe your observations of the training process.`

We had a few observations of the training process: 1. The feedback loop is very long, it takes very long. Cannot determine whether the rewards are not increasing because of the slow learning of the agent, or due to an error in the loss function.

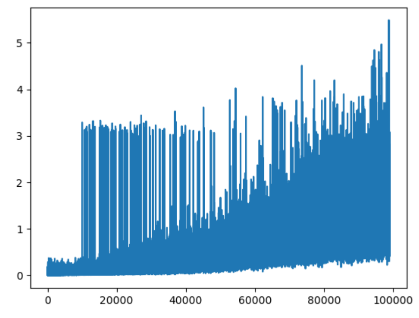
1. It is hard to determine how long we must allow the agent to explore.
2. Changing the hyperparameters and running the training all over again is also very time consuming.

> > In particular, show the generated loss and reward curves and describe how they develop over the course of training. Some points of interest to describe should be: How quickly is the agent able to consistently achieve positive rewards? What is the relationship between the e-greedy exploration schedule and the development of the cumulative reward which the agent achieves over time?

Agent rewards plot



Agent training loss plot



`> How does the loss curve compare to the loss curve that you would expect to see on a standard supervised learning problem?`

The loss curve in our experiment is not smooth. In a supervised learning problem, you expect the loss to come down in a smooth manner. This makes sense, because in a supervised learning setting, you know what the ground truth is, and the model is trained on the loss from this ground truth. In a Reinforcement Learning setting, we do not know what the best actions are, the agent has to figure out the best actions by exploring the environment, and using the rewards from the environment as feedback.

Also, the loss function here is the loss between the target q network (which is only updated slower) and the current policy network. The rewards are a better estimate of whether the agent is learning.

e) Evaluation (2)

> *Evaluate the trained Deep Q-Learning agent by running the evaluate racing.py script. Observe the performance of the agent by running the script on your local machine. Where does the agent do well and where does it struggle?*

The agent performs well in a straight track most of the times. For tracks with slight curves, the agent takes sharp turns at high speeds. For very sharp turns, the agent brakes a lot, but manages to get through eventually.

In some episodes, however, the agent just keeps braking, accumulating a negative reward, and occasionally, also skids while taking a turn at high speeds.

We feel this is because of the very small action space, and with a richer action space, the agent can learn better policies (and possible quicker). We will try this in problem 3.2 ©.

> *How does its performance compare to the imitation learning agent you have trained for Exercise 1? Discuss possible reasons for the observed improvement/decline in performance compared to your imitation learning agent from Exercise 1.*

1. Compared to the imitation learning agent in Exercise 1, this agent has the ability to recover from failures like getting off track. The imitation learning agent had no way to figure out how to recover from failures unless such datasets were explicitly fed to it.
2. This agent also learns to drive better than human imitations in exercise 1, aka cuts corners well. Coincidentally, it probably learns to do turns similar to how we implemented the spline smoothness for turns in problem 2.
3. This agent also tends to get stuck in some tracks, i.e. it does not take any actions, and also has the tendency to skid off the tracks while taking turns. This is similar to our Imitation learning agent. We believe this may again be due to the data distribution available to the agent. Even though we pick random samples from the replay memory, the distribution it picks from is skewed. There is far more samples of driving straight than turning.

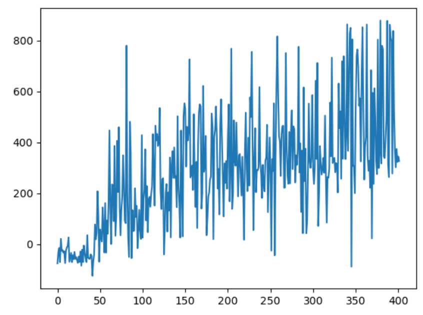
3.1 Further Investigations and Extensions

a) Discount factor γ (gamma)

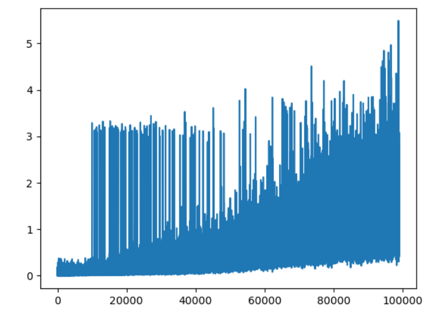
> *Investigate the influence of the discount factor γ. Show reward curves that demonstrate the effect of an increase/decrease of γ from its default of 0.99 on your agent*

The discount factor controls the calculation of the future expected reward of the agent. It directly affects the loss reward calculation.

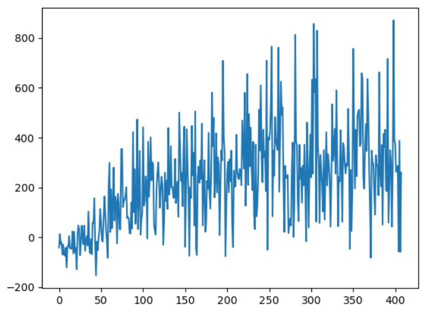
Agent rewards plot gamma 0.99



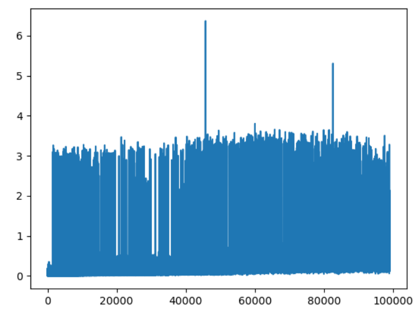
Agent training loss plot gamma 0.99



Agent rewards plot gamma 0.90



Agent training loss plot gamma 0.90



i) Why do we typically use a discount factor in reinforcement learning problems and in which cases would it be a problem not to use a discount factor (i.e. γ = 1)?

We use a discount factor so as to make the futire reward estimation useful mathematically. For tasks that are continuing, or non-episodic, the estimate of sum of future rewards quickly becomes infinite.

Tasks like game playing (chess, go, etc) are typically episodic. Tasks such as driving and a process control robot in a factory are non-episodic.

To manage this notion of expected future rewards, we introduce *discounting*. The agent now tries to select actions such that the sum of discounted rewards it receives over the future is maximized.

b) Action repeat parameter

Describe the reasoning behind the use of an action repeat parameter. By default, this value is set to 4. What is the effect on the training progress (look at your training plots) and your evaluation performance if this parameter is increased or decreased? Discuss and interpret your findings.

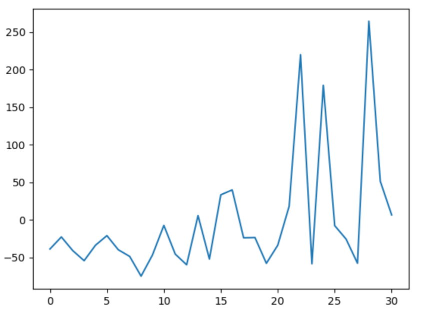
The action repeat parameter defines how many times the agent takes an action, after deciding which action to take. A good action repeat parameter helps the agent learn a smooth policy. On decreasing this parameter, and keeping rest of the variables the same, the training performance decreases significantly. It gets much less rewards.

With action repeat parameter = 1, the agent has to decide on an action every frame. Running one step in the environment is much cheaper than making inference forthat step. Hence, this also significantly impacts training time. Also, we think with such a small repeat parameter, it will take very long for the agent to achieve good rewards. For a higher action repeat parameter (8 or above), we think it would be very hard for the agent to learn a good policy.

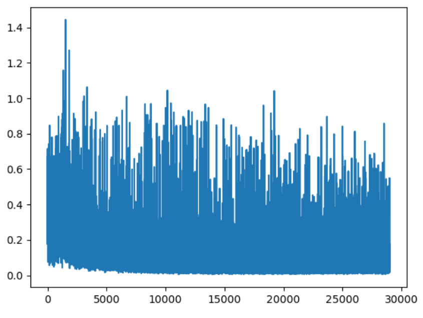
We present our plots for action repeat parameter = 1

repeat action 1 time

Agent rewards plot



Agent loss plot



i) Why might it be helpful to repeat each action several times?

A good action repeat parameter helps the agent learn a smooth policy. Repeating the same action multiple times enables transitions between different advantageous states.

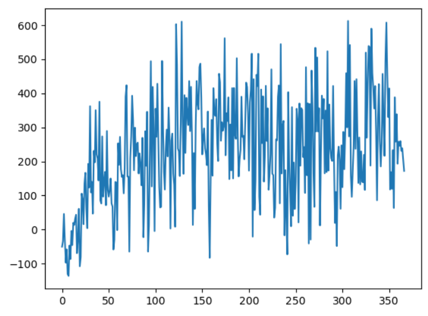
c) Action space

By default, the agent uses a 4-dimensional action set of left-turn, right-turn, brake and acceleration (see get action set function in action.py). Investigate the addition of a null action ([0,0,0]) as well as more fine-grained turning, braking or acceleration actions. What is the effect on the agent's driving style as well as its evaluation score? Which additional actions lead to an improvement in the agent's performance and which do not?

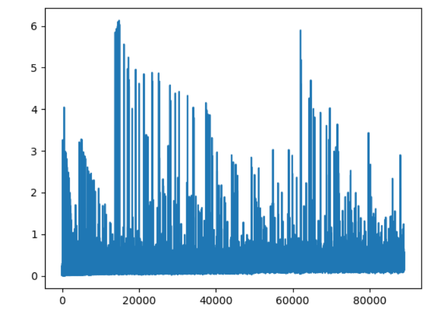
We tested a 7 action space, the results of which are:

7 action space

Agent rewards plot



Agent loss plot



> *i) Why might it not always be helpful to increase an agent’s action space?*

Increasing tha action space makes increases the dimensions of our Q-value function. However, this also makes the learning harder, and more difficult to converge to a good solution. Also, it will require more training time.

With more training time though, a bigger action space can theoretically achieve better performance.

> *ii) In general, why are Deep Q-Networks limited to a discrete set of actions and what solutions exist to overcome this limitation?*

Q learning becomes unstable at very high dimensions. Hence we approximate the action space to a discrete set to achieve stable learning. For continuous action space, we will also need to change our loss function accordingly. There are some solutions to overcome this limitation. Deep Deterministic Policy Gradient (DDPG), and actor-critic methods are algorithms for learning continous actions.

d) Double Q-learning

> *One problem with the standard Deep Q-Learning approach is an overestimation of Q-values. A proposed solution to this problem is the double Q-learning algorithm [3]. Read this paper, implement the double Q-learning algorithm and evaluate its effect on the training and evaluation performance of your agent. Important: Make sure to include your double Q-learning implementation in your submitted code.*

The main change in our learning code is:

```
q_values = policy_net.forward(obs_batch)
q_values = q_values[torch.arange(q_values.size(0)), torch.from_numpy(act_batch)].unsqueeze(1)

# double q learning
target_q =  target_net(next_obs_batch)
policy_net_best_id = policy_net(next_obs_batch).max(1)[1].detach()
max_target_q = target_q[torch.arange(target_q.size(0)), policy_net_best_id]

# mask done episodes
max_target_q *= torch.from_numpy(done_mask==0).to(device)
q_target = torch.from_numpy(rew_batch).to(device) + gamma*max_target_q

# calculate loss
loss = F.smooth_l1_loss(q_values, q_target.unsqueeze(1))
```

> *i) Shortly summarize the reason for the overestimation of Q-values in a standard DQN*

The max operator in standard Q-learning and DQN, uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. van Hasselt in 2010 argued that noise in the environment can lead to overestimations even when using tabular representation, and proposed Double Q-learning as a solution.

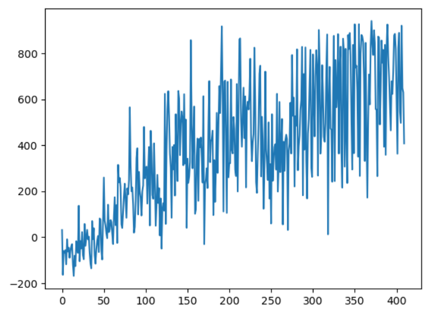
> *ii) How does double Q-learning algorithm solve this issue?*

The idea of Double Q-learning is to reduce overestimationsby decomposing the max operation in the target into actionselection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks.

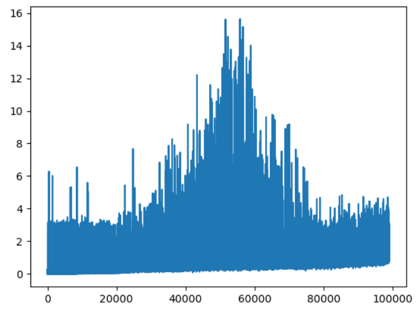
Our plots for double deep q learning are as follows:

Double deep-q learning

Agent rewards plot



Agent loss plot



d) Best Solution

How is this agent constructed and trained?

This agent is trained using double deep-q learning with a 7 action space. The other hyperparameters are: Gamma: 0.99 Learning rate: 10e-4 timesteps: 100,000

In which aspects has its performance improved over your baseline agent from Section 3.1 and where does it still exhibit sub-optimal behavior?

The agent driving is much smoother, and it does not get stuck on braking on straight roads. This agent also seems to recover better from failures. Agent still has the tendency to skid off the road when turning at high speeds. Also, when it recovers from a skid, it tends to drive in the reverse direction on the road.

Briefly sketch out an idea for overcoming the main remaining issue with your agent's training or performance (this does not need to be implemented).

We could train for longer. Also, we could try a smarter sampling from the replay memory to fix data distribution skew.