Driving a car on a hill with machine learning

Quinty van Dijk 14086530

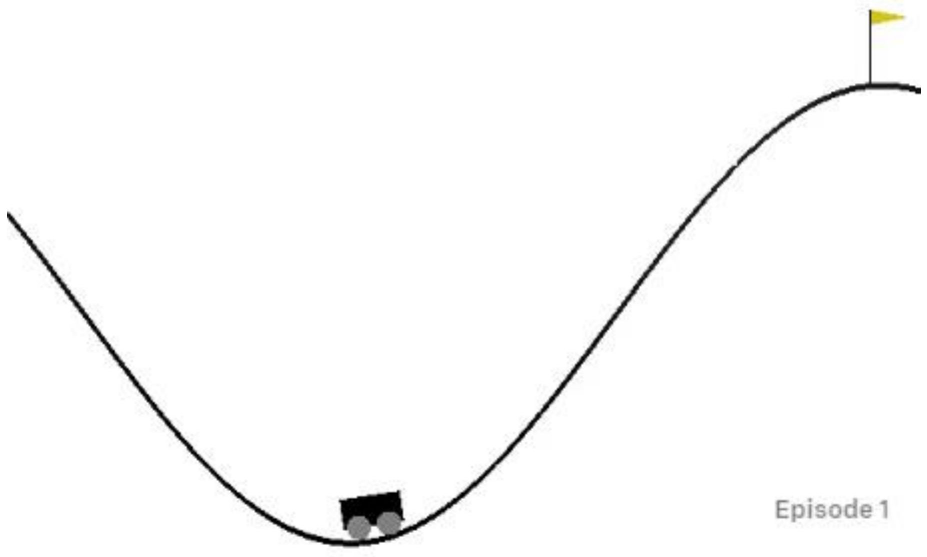


Figure 1 Visual representation of the learning task. The cart has to reach the flag.

Introduction

In this report we will discuss a machine learning algorithm. The task of the algorithm is to drive a car to the top of mountain hill, but its engine is too weak to drive directly on the hill, it must build op momentum by driving forward and backward.

This report will first go through the environment (dataset), then discuss the algorithm and its results and finally the conclusions and recommendations

Environment

The chosen environment is the MountainCar-v0 by OpenAIs Gym [1]. This environment has a car that is placed in a valley (between two mountains). The mountains are created by a sine wave between x = -1.2 and x = 0.6. The car spawns between x = -0.6 and x = -0.4. This is visualized in figure 1. The robot is controlled by three commands, a 0 for driving backwards, a 1 for doing nothing and a 2 for driving forward. That means that this problem is classification problem. The goal is to drive the car to x = 0.5. The car is not powerful enough to drive directly to the top of the hill, therefore it must drive forwards and backwards to build op momentum. The car doesn't experience friction.

After the car has been controlled by on of thee commands the environment returns the state of the car, existing of position and velocity, the reward and a done Boolean. If the done Boolean is true the car has reached the top of the hill or the episode has taken longer than 200 steps. The reward is always -1. This means that the reward will always be the same until the car has reached the top.

The algorithm

Since this problem doesn’t have targets but only a goal that must be achieved on a most optimal way this problem is a Reinforcement learning problem. This means that the algorithm will be a Neural Network since this isn’t a linear problem. The neural network has three outputs, so it can be used for classification. It uses two layers with ReLU and softmax activation functions respectively and with 200 hidden nodes. This was adopted by other reinforcement algorithms found [2] and changing them either didn’t give a noticeable difference or a worse result, so they weren’t changed. To receive a classification result the three output are seen as probabilities. So, the first output returns the change the action will be to drive backwards, the second to do nothing and the third to drive forward. The output with the highest probability is chosen. The Adam optimization function is used since it’s the most optimal one according to literature [3].

The learning rate is set at 0.01 and has been found by trial and error. This learning rate seams (very close to) the optimal learning rate since the algorithm learns fast and lowering it doesn’t create better results. The maximum iterations aren’t fixed since it can take a while before the algorithm learns something, as will be explained below, but a maximum of 1000 iterations should give some result most of the times.

As explained above the reward is -1 for every action the environment takes. This means that the reward is always -200 (because the environment only runs for a maximum of 200 steps) until the car reaches the top, because then the number of steps is lower van 200. This means that the algorithm keeps randomly guessing its weights. The problem is that the algorithm never gets the car to the top of the hill, so the reward is always the same and the algorithm doesn’t really learn. To make the algorithm learn the closest position the car has gotten in the episode is added to the reward. Because the normal reward is relatively big in comparison with the position (-200 vs -0.5) you might think that the reward based on the position should be made bigger, but by trial and error the opposite was found. This is good because once the car has reached the goal the goal of the algorithm is to optimize this process and not to get further.

The result

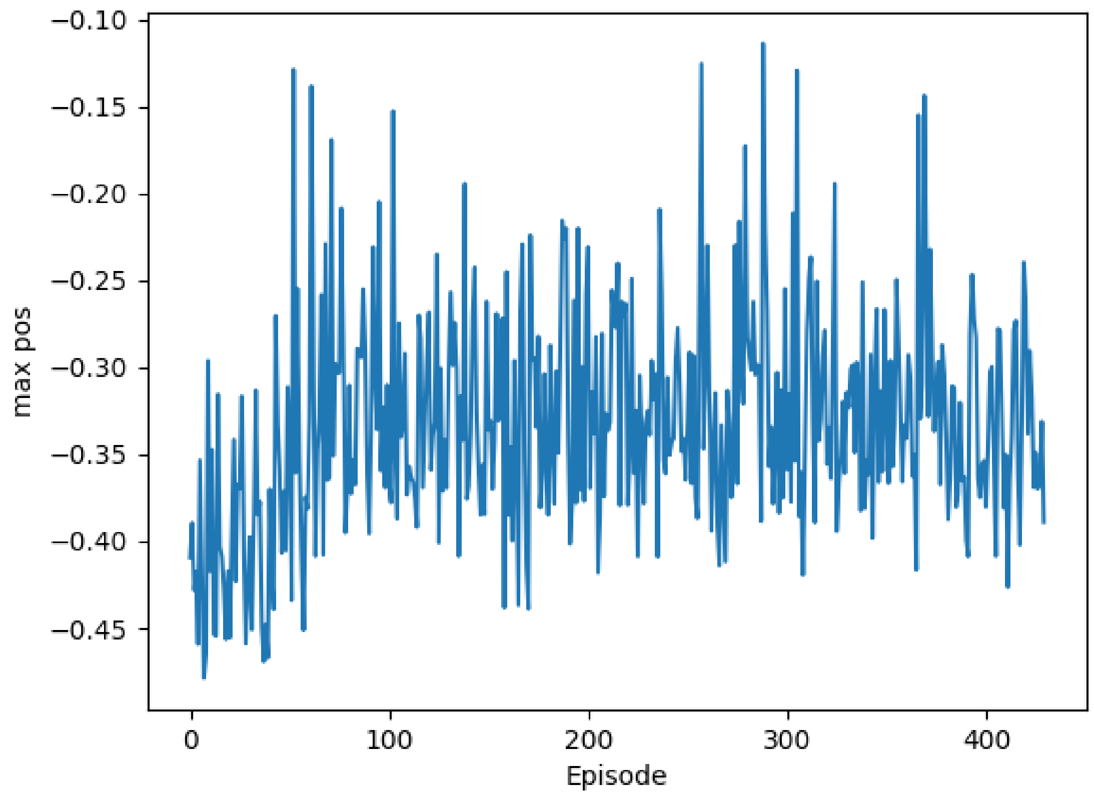


Figure 2 Result

As shown in figure 2 the algorithm isn't able to reach the top of the mountain (flag position is 0.5). It's clearly visible that the algorithm learns, since the average maximum position is clearly increasing after about 50 episodes but stops at 100 episodes. Other weight initializations show that this window can also happen way later, even after more than 1000 episodes.

It's not directly clear why this is. There are different examples of reinforcement learning that use the same neural network [2] and the algorithm does clearly learn, only it seems to stop learning to early. This might be because it has found a local minimum. To check this the following actions have been taken to find a better, or even the global, minimum, but didn't have success:

- Adjusting the learning rate
- Adjusting the reward:
  - Magnifying the max position reward
  - Add a reward for higher velocities
  - Add a reward for greater distances travel (since the car needs to travel more distance to build op speed)
- Changing the Neural Network:
  - Adjusting the amount of hidden nodes
  - Adjusting the amount of output to one (so that when 0<x<1/3 means driving backwards and 1/3<x<2/3 means standing still and x>2/3 means driving forward)
  - Adjusting the activation and optimization functions

How to improve

Others have had more luck designing a machine learning algorithm for this specific task. They mostly used Q-learning [4] or SARSA and TileCoding [5]. I didn't use those methods from the start because I thought the reinforcement learning algorithm based on methods I already knew from the course would work. I wasn't able to understand these newer methods fully and didn't just want to copy someone else's code. As far as my understanding of these methods goes is that they take the previous action in consideration when calculating a new action. This might be why my Neural Network didn't work, this would surprise me however, because the current velocity and position of the car should be enough to decide the next action (when velocity is approaching 0, turn around).

Conclusion and Recommendations

A machine learning algorithm has been created that can drive a cart almost halfway up a hill. Where the goal was to drive it all the way up a hill. To improve the algorithm it is worth it to look at Q-learning or SARSA and LileCoding, since others had more luck using that approach.

Bibliografie

[1] „MountainCar-v0 Gym Environment,” OpenAI, 2018. [Online]. Available: https://gym.openai.com/envs/MountainCar-v0/. [Geopend 2019].

[2] „Pytorch reinforcement example,” Pytorch, 2018. [Online]. Available: https://github.com/pytorch/examples/blob/master/reinforcement_learning/reinforce.py. [Geopend 2019].

[3] A. Agrawal, „Loss Functions and Optimization Algorithms. Demystified.,” Medium, 29 September 2017. [Online]. Available: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c. [Geopend 2019].

[4] ts1829, „Mountain Car v0 - Q Learning,” 2018. [Online]. Available: https://gist.github.com/ts1829/244d36ea4aac872f1c3a82d3b481a99c#file-mountain-car-v0-q-learning-ipynb. [Geopend 2019].

[5] ZhiqingXiao, „Solve MoutainCar-v0 using SARSA() + TileCoding,” 2019. [Online]. Available: https://github.com/ZhiqingXiao/OpenAIGymSolution/blob/master/MountainCar-v0/mountaincar_v0_sarsa_lambda_tilecode.ipynb. [Geopend 2019].

Appendix A: Code

```python
import argparse
import gym
import numpy as np
from itertools import count
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
import matplotlib.pyplot as plt
import matplotlib

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

parser = argparse.ArgumentParser(description='PyTorch REINFORCE example')
parser.add_argument('--gamma', type=float, default=0.99, metavar='G',
                    help='discount factor (default: 0.99)')
# parser.add_argument('--seed', type=int, default=543, metavar='N',
#                       help='random seed (default: 543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='interval between training status logs (default:
10)')
args = parser.parse_args()

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
env = gym.make('MountainCar-v0')
# env.seed(args.seed)
# torch.manual_seed(args.seed)


class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(2, 200)
        # self.affine11 = nn.Linear(128, 255)
        self.affine2 = nn.Linear(200, 3)

        self.saved_log_probs = []
        self.rewards = []

    def forward(self, x):
        x = F.relu(self.affine1(x))
        # x = F.relu((self.affine11(x)))
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=-1)


policy = Policy().to(device)
optimizer = optim.Adam(policy.parameters(), lr=0.01)
eps = np.finfo(np.float32).eps.item()
```

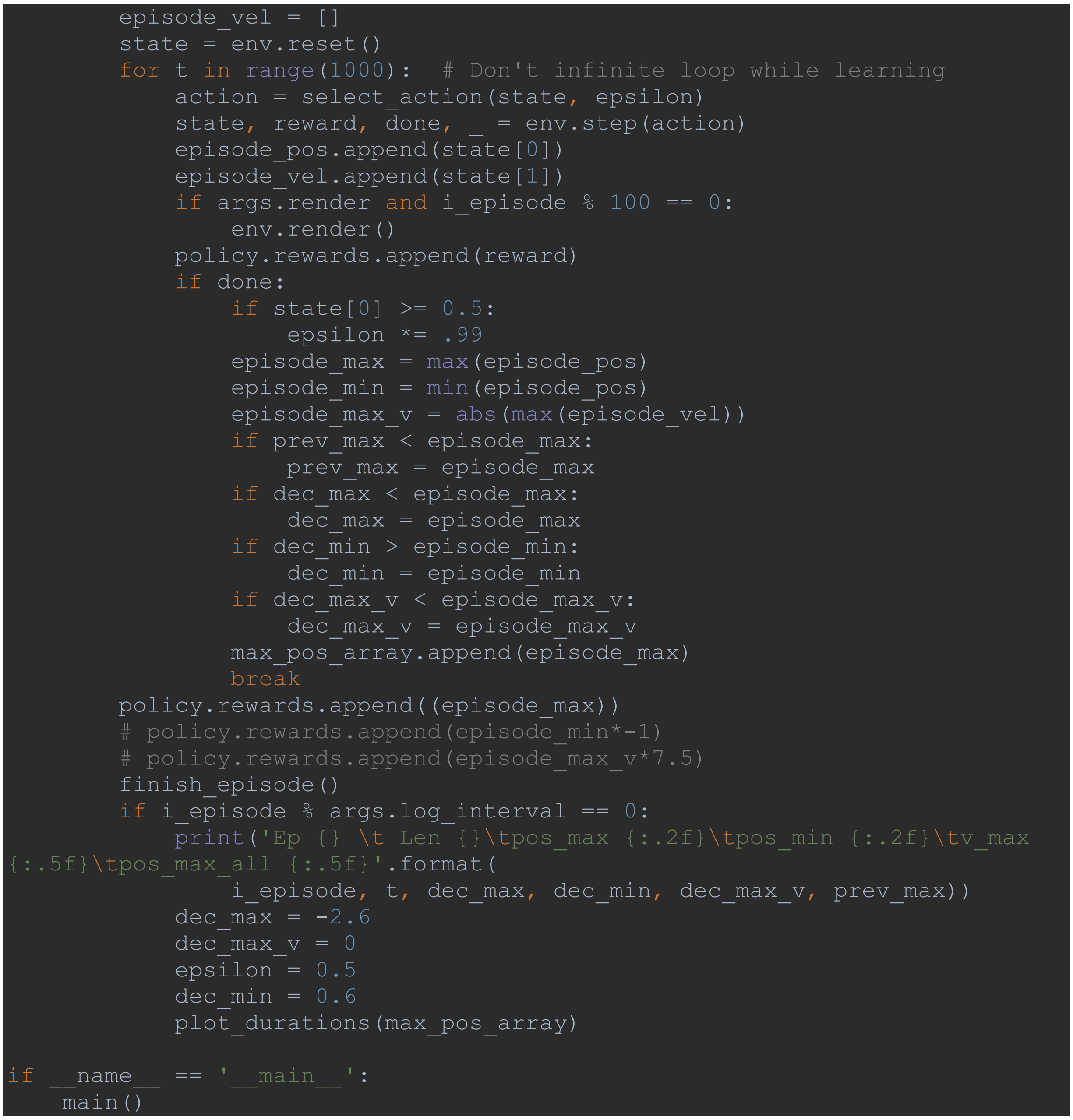
```python
def select_action(state, epsilon):
    if np.random.rand(1) < epsilon:
        action = np.random.randint(0,3)
    else:
        state = torch.from_numpy(state).float().unsqueeze(0)
        probs = policy(state)
        # print(probs)
        m = Categorical(probs)
        action = m.sample()
        # print(action)
        policy.saved_log_probs.append(m.log_prob(action))
        # print(action.item())
        action = action.item()
    return action


def finish_episode():
    R = 0
    policy_loss = []
    rewards = []
    for r in policy.rewards[::-1]:
        R = r + args.gamma * R
        rewards.insert(0, R)
    rewards = torch.tensor(rewards, device=device)
    rewards = (rewards - rewards.mean()) / (rewards.std() + eps)
    for log_prob, reward in zip(policy.saved_log_probs, rewards):
        policy_loss.append(-log_prob * reward)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()
    del policy.rewards[:]
    del policy.saved_log_probs[:]

def plot_durations(episode_max):
    plt.figure(2)
    plt.clf()
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('max pos')
    plt.plot(episode_max)
    # Take 100 episode averages and plot them too
    # if len(durations_t) >= 100:
    #     means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
    #     means = torch.cat((torch.zeros(99), means))
    #     plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        display.clear_output(wait=True)
        display.display(plt.gcf())

def main():
    prev_max = -2.6
    dec_max = -2.6
    dec_max_v = 0
    epsilon = 0.3
    dec_min = 0.6
    max_pos_array = []
    for i_episode in count(1):
        episode_pos = []
```



```
        episode_vel = []
        state = env.reset()
        for t in range(1000):  # Don't infinite loop while learning
            action = select_action(state, epsilon)
            state, reward, done, _ = env.step(action)
            episode_pos.append(state[0])
            episode_vel.append(state[1])
            if args.render and i_episode % 100 == 0:
                env.render()
            policy.rewards.append(reward)
            if done:
                if state[0] >= 0.5:
                    epsilon *= .99
                episode_max = max(episode_pos)
                episode_min = min(episode_pos)
                episode_max_v = abs(max(episode_vel))
                if prev_max < episode_max:
                    prev_max = episode_max
                if dec_max < episode_max:
                    dec_max = episode_max
                if dec_min > episode_min:
                    dec_min = episode_min
                if dec_max_v < episode_max_v:
                    dec_max_v = episode_max_v
                max_pos_array.append(episode_max)
                break
        policy.rewards.append((episode_max))
        # policy.rewards.append(episode_min*-1)
        # policy.rewards.append(episode_max_v*7.5)
        finish_episode()
        if i_episode % args.log_interval == 0:
            print('Ep {} \t Len {}\tpos_max {:.2f}\tpos_min {:.2f}\tv_max
{:.5f}\tpos_max_all {:.5f}'.format(
                i_episode, t, dec_max, dec_min, dec_max_v, prev_max))
            dec_max = -2.6
            dec_max_v = 0
            epsilon = 0.5
            dec_min = 0.6
            plot_durations(max_pos_array)

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