COMP3667: Reinforcement learning practical 7

Function Approximation

robert.lieck@durham.ac.uk

This is the version *with* answers!

1 Overview

Welcome to the seventh reinforcement learning practical. In this practical, we will build some intuition and experience for deep RL methods. Some of the details will only be explained in the next lecture, but they are not required to do this practical. In particular, we will

- use a function approximator to learn a policy in a continuous state space (but still discrete action space) with the REINFORCE method
- learn a continuous policy for an infinite-armed bandit problem
- use the SAC method that can handle continuous action spaces.

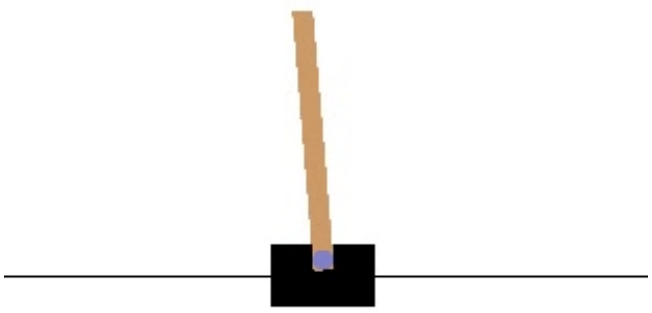
2 Setup

Start from **this colab notebook** (either locally or online). It contains the necessary boiler-plate code for you to get started with the experiments. *Note:* your NCC kernels from previous weeks should work, otherwise you may need:

```
!pip install setuptools==65.5.0 "wheel<0.40.0"
!apt update
!apt-get install python3-opengl
!apt install xvfb -y
!pip install 'swig'
!pip install 'pyglet==1.5.27'
!pip install 'gym[box2d, atari]==0.19.0'
!pip install 'autorom[accept-rom-license]==0.4.2'
```

which will take a moment to complete. If run locally it may fail if some dependencies are not installed (remove the `%%capture` to see the output).

3 REINFORCE and `CartPole`



The `CartPole` environment has a 4-dimensional continuous state/observation space (cart position, cart velocity, pole angle, pole angular velocity) and a discrete action space with two actions (push cart to the left, push cart to the right). As we cannot list all states in a table, we need to use function approximation methods to learn values or policies. We will here use a *direct policy* method called REINFORCE (details in lecture 8), which directly learns a policy instead of first learning a value function and then deriving the policy from that. The method is very simple, a complete working implementation looks like this (the one in the colab notebook is almost the same):

```
1  class REINFORCE(nn.Module):
2      def __init__(self, obs_dim, act_dim):
3          super().__init__()
4          self.data = []
5          self.fc1 = nn.Linear(obs_dim, 128)
6          self.fc2 = nn.Linear(128, act_dim)
7          self.optimizer = optim.Adam(self.parameters(), lr=0.0002)
8
9      def sample_action(self, state):
10         x = F.relu(self.fc1(torch.from_numpy(state).float()))
11         prob = F.softmax(self.fc2(x), dim=0)
12         action = Categorical(prob).sample()
13         return action.item(), torch.log(prob[action])
14
15     def put_data(self, item):
16         self.data.append(item)
17
18     def train(self):
19         R = 0
20         self.optimizer.zero_grad()
21         for _, _, r, _, _, log_prob in self.data[::-1]:
22             R = r + 0.98 * R
23             loss = -log_prob * R
24             loss.backward()
25         self.optimizer.step()
26         self.data = []
```

Question

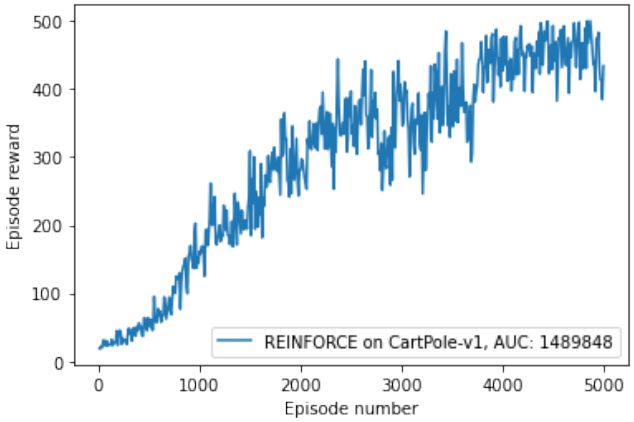
How are continuous states/observations mapped to discrete actions in the REINFORCE implementation (lines 5, 6, 10–12)?

States are first mapped to a continuous vector of the correct dimensionality via one hidden layer of size 128. Using a softmax, these are then mapped to probabilities to define a categorical distribution over actions, from which the action is then sampled.

Learning `CartPole`

With the helper functions in the notebook you can easily test the REINFORCE method on the `CartPole` environment like this:

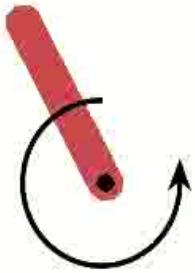
```
1  env_name = 'CartPole-v1'
2  env = gym.make(env_name)
3  agent = REINFORCE(env)
4  max_episodes = 5000
5  experiment(env, env_name, agent, plotter, max_episodes)
```



Exercise

- Improve the performance (AUC) by tweaking the network structure and learning rate.

4 SAC and `Pendulum`



The `Pendulum` environment is somewhat similar to `CartPole` but has continuous actions. The 3-dimensional state space is $(\sin(\theta), \cos(\theta), \dot{\theta})$, where $\theta \in [-\pi, \pi]$ is the pendulum's angle measured from the upright position and $\dot{\theta}$ is its angular velocity. The continuous actions $a \in [-2, 2]$ correspond to the torque exercised on the pendulum (the arrow in the picture above). The reward is defined as $-(\theta^2 + \frac{1}{10}\dot{\theta}^2 + \frac{1}{1000}a^2)$, that is, the pendulum should be in an upright position but ideally also not move and not require any torque to be applied.

Continuous actions

REINFORCE cannot handle continuous action spaces, which is why we use the soft actor-critic (SAC) method for this environment (details in lecture 8). SAC learns a policy (the "actor") and a value function approximation (the "critic"). The implementation of SAC is a bit more involved as you can see in the notebook. One important part is the `PolicyNet` class for the actor, which looks (reduced to the essentials) like this:

```
class PolicyNet(nn.Module):
    def __init__(self):
        super(PolicyNet, self).__init__()
        self.fc1 = nn.Linear(3, 128)
        self.fc_mu = nn.Linear(128,1)
        self.fc_std  = nn.Linear(128,1)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        mu = self.fc_mu(x)
        std = F.softplus(self.fc_std(x))
```

```
        dist = Normal(mu, std)
        action = dist.rsample()
        real_action = torch.tanh(action)
        return real_action
```

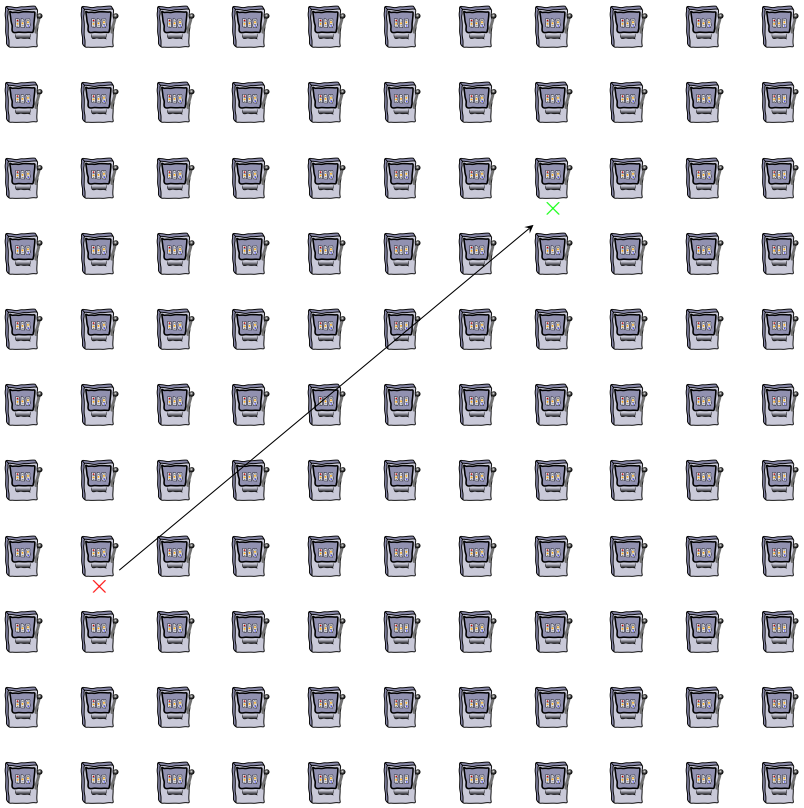
Exercise

- Try to understand each line of the code. Discuss with your neighbor(s) what each line does. In a continuous action space, there are infinitely many possible actions the agent could take. How does the policy choose a specific one to return?

  The policy network first transforms the input state to μ and σ, which define the mean and standard deviation of a normal distribution. The action is then sampled from this distribution (`rsample` uses the *reparameterisation trick* to allow for propagating gradients through the sampling step). Finally, the sampled action (which is unbounded, i.e. in the interval $[-\infty, \infty]$) it is mapped to the interval $[-1, 1]$.

Infinite-armed bandits

To get a better intuition of what happens in a fully continuous setting, we can look at a continuous/infinite version of the multi-armed bandit problem. Imagine a casino hall with many bandits set up in rows an columns:



In this environment, each episode lasts only a single step: the agent is standing in front of one of the machines and moves (relative to its location) to play another machine (or the same if it does not move). In the limit of infinitely many rows and columns, this becomes a continuous scenario. If we assume that there are "hot spots" in the continuous space where the reward is high, we can implement a continuous version of this environment as follows:

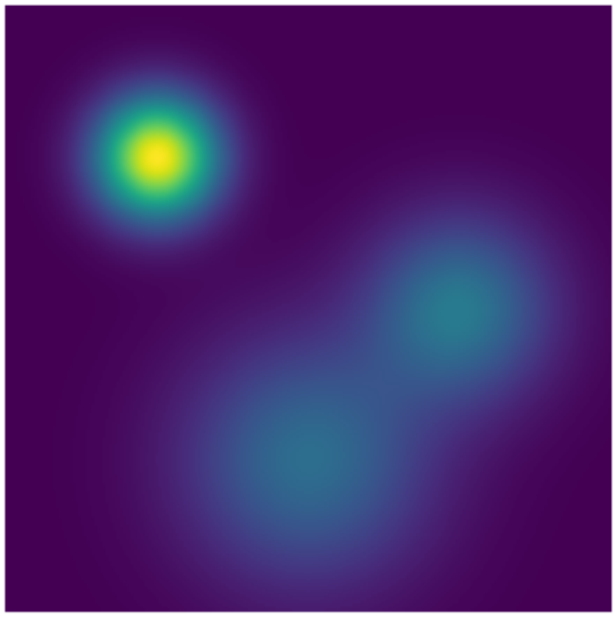
```
class Env(nn.Module):
    def __init__(self, locations, rewards, tolerances):
        self.locations = torch.tensor(locations)
        self.rewards = torch.tensor(rewards)
        self.covariances = torch.eye(self.locations.shape[-1]) * \
                           torch.tensor(tolerances)[:, None, None] ** 2

    def reward(self, loc):
```

```
        reward = 0
        d = loc[..., None, :] - self.locations
        x = -torch.einsum('...a,...ab,...b', d, torch.inverse(self.covariances), d)
        return (x.exp() * self.rewards).sum(dim=-1)
```

An instance with three "hot spots"

```
Env(locations=[[-0.5, 0.5], # hot spot 1
               [0, -0.5],   # hot spot 2
               [0.5, 0]],   # hot spot 3
    rewards=[2.5, 0.9, 1.], # maximal reward at the three hot spots
    tolerances=[0.2, 0.4, 0.3]) # size of the hot spots
```

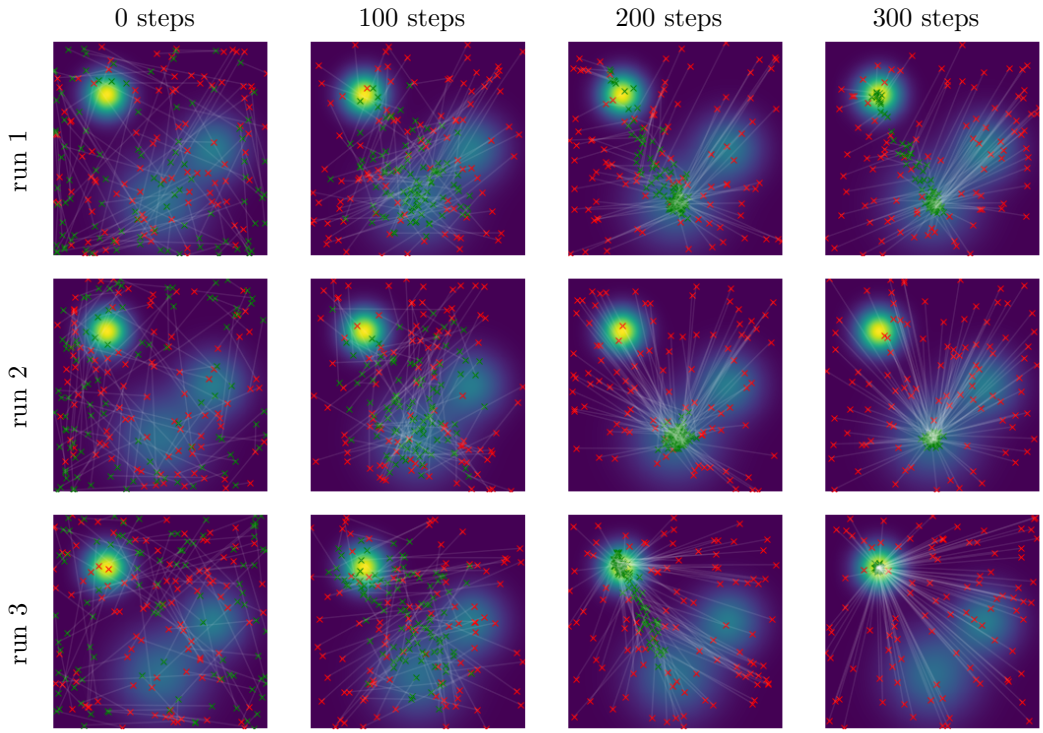
has a reward function like this (the state space is $[-1, 1]$ in both dimensions)



Exercise

Use the code from the notebook to train a `PolicyNet` on this environment. How does the initial and final policy look like? How reliably does the policy converge to the optimal solution? Can you "trick" it into always learning a sub-optimal policy by changing the environment? *Note:* In the plots, the red crosses indicate states and the green crosses indicate the state *after* taking the action (connected by the white lines).

The initial policy seems very random, there is no obvious pattern. The final policy tends to be very focused and go to one single place (with high reward) from any state in the environment. Depending on the environment, the policy does not always converge to the optimal solution. In particular, "strong but narrow" hot spots are harder to find than "weaker but broader" ones, even though the optimal solution would be to always go to the center of the strongest hot spot.



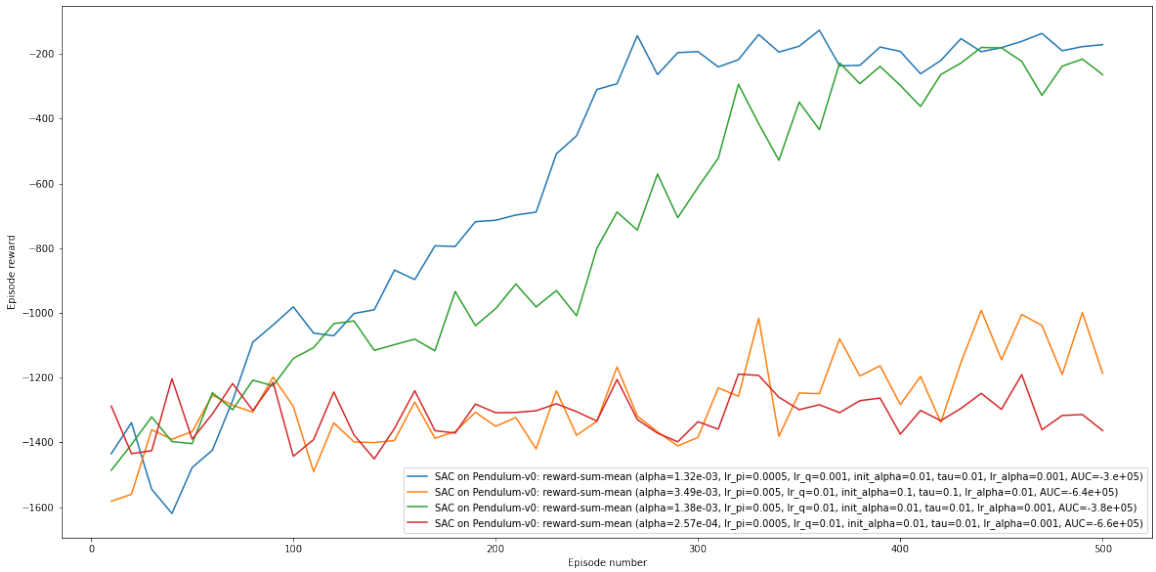
Tuning hyperparameters in SAC

The policy learned by SAC (the "actor") is very similar to the infinite-armed bandit case. The main difference is that SAC also learns action values (the "critic") to estimate the return instead of just using the immediate reward (which was only possible in the simple bandit environment, where episodes only last one step). When running an experiment, you may specify a number of hyperparameters for SAC:

```
env_name = "Pendulum-v0"
env = gym.make(env_name)
agent = SAC(lr_pi            = 0.0005, # learning rate for the policy (actor)
            lr_q             = 0.001,  # learning rate for the values (critic)
            init_alpha       = 0.01,   # initial exploration
            gamma            = 0.98,
            batch_size       = 32,
            buffer_limit     = 50000,
            tau              = 0.01,   # smoothing of value learning (lower is smoother)
            target_entropy   = -1.0,
            lr_alpha         = 0.001,  # learning rate for exploration
            reward_scale     = 10,
            action_scale     = 2)
max_episodes = 500
experiment(env, env_name, agent, plotter, max_episodes)
```

Exercise

- Tweak the hyperparameters and try to get the best performance (AUC) on the `Pendulum` environment.


Episode reward
Episode number
−200
−400
−600
−800
−1000
−1200
−1400
−1600
0
100
200
300
400
500
SAC on Pendulum-v0: reward-sum-mean (alpha=1.32e-03, lr_pi=0.0005, lr_q=0.001, init_alpha=0.01, tau=0.01, lr_alpha=0.001, AUC=-3.e+05)
SAC on Pendulum-v0: reward-sum-mean (alpha=3.49e-03, lr_pi=0.005, lr_q=0.01, init_alpha=0.1, tau=0.1, lr_alpha=0.01, AUC=-6.4e+05)
SAC on Pendulum-v0: reward-sum-mean (alpha=1.38e-03, lr_pi=0.005, lr_q=0.01, init_alpha=0.01, tau=0.01, lr_alpha=0.001, AUC=-3.8e+05)
SAC on Pendulum-v0: reward-sum-mean (alpha=2.57e-04, lr_pi=0.0005, lr_q=0.01, init_alpha=0.01, tau=0.01, lr_alpha=0.001, AUC=-6.6e+05)