10-403 Deep Reinforcement Learning and Control
Homework 6: Model-Based Reinforcement Learning
Spring 2019

April 26, 2019

Due by 11:59 PM on May 10th, 2019

Instructions

You may work in teams of **2** on this assignment. Only one person should submit the writeup and code on Gradescope. Additionally, the same person who submitted the writeup to Gradescope must upload a single file with the consolidated code to Autolab. Make sure you mark your teammates as collaborators on Gradescope (you do not need to do this in Autolab) and that all names are listed in the writeup. Writeups should be typeset in LaTeXand submitted as PDF. All code, including auxiliary scripts used for testing, should be submitted with a README. Please limit your writeup to 8 pages or less (excluding the provided instructions).

We've provided some code templates that you can use if you wish. Abiding to the function signatures defined in these templates is not mandatory; you can write your code from scratch if you wish. You can use any deep learning package of your choice (e.g., Keras, Tensorflow, PyTorch). You should not need the cluster or a GPU for this assignment. The models are small enough that you can train on a laptop CPU.

It is expected that all of the work you submit is your own. Submitting a classmate's code or code which copied from online and claiming it is your own is not allowed. Anyone who does this will be violating University policy, and risks failure of the assignment, course and possibly disciplinary action taken by the university.

Environment

For this assignment, you will work with the `Pushing2D-v1` environment, note that this environment is slightly **different** from the one you used in the previous homework. In particular, the state is now 12 dimensional and includes the velocity for the pusher and the box. To recap, in order to make the environment using `gym`, you can use the following code:

```
import gym
import envs

env = gym.make("Pushing2D-v1")
```

Then you can interact with the environment as you usually do. Similarl to Homework 5, the environment is considered "solved" once the percent successes (i.e., the box reaches the goal within the episode) reaches 95%.

Model-Based Reinforcement Learning with PETS

In Homework 5, you were asked to implement DDPG, which is a model-free policy gradient method. For this homework, you will implement a model-based reinforcement learning (MBRL) method called **PETS** which stands for probabilistic ensemble and trajectory sampling [1]. You will find the recitation slides to be very useful for this assignment. An overview of MBRL with PETS is shown in Algorithm 1.

Algorithm 1 MBRL with PETS

procedure MBRL(# of iterations I)
    Initialize empty data array D and initialize probabilistic ensemble (PE) of models.
    Sample 100 episodes from the environment using random actions and store into D.
    Train ensemble of networks for 10 epochs.
    repeat for I iterations:
        Sample 1 episode using MPC and latest PE and add to D.
        Train PE for 5 epochs over D.
end procedure

There are 3 main components to MBRL with PETS:

1. Probabilistic ensemble of networks: you will be using probabilistic networks that output a distribution over resulting state given a state and action pair.

2. Trajectory sampling: propagate hallucinated trajectories through time by passing hypothetical state-action pairs through different networks of the ensemble.

3. Model predictive control: use the trajectory sampling method along with a cost function to perform planning and select good actions.

We will go into more detail on each component below.

Part 1: Probabilistic Ensemble

You are provided starter code in `model.py` that specifies that model architecture that you will be using for each member of the ensemble. Specifically, each network is a fully connected network with 3-hidden layers, each with 400 hidden nodes. If you have trouble running this

network, a smaller network may work as well but may require additional hyperparameter tuning. The starter code also includes a method for calculating the output of each network, which will return the mean and log variance of the resulting states.

The loss that you should use to train each network is the negative log likelihood of the actual resulting state under the predicted mean and variance from the network:

$$\text{loss}_{\text{Gauss}} = \sum_{n=1}^{N} [\mu_\theta(s_n, a_n) - s_{n+1}]^T \Sigma_\theta^{-1}(s_n, a_n)[\mu_\theta(s_n, a_n) - s_{n+1}] + \log \det \Sigma_\theta(s_n, a_n) \quad (1)$$

Note that as shown in this equation, both the μ and the Σ depend on the input state and action.

The training routine for the ensemble is shown in Algorithm 2. Consider defining a list

Algorithm 2 Training the Probabilistic Ensemble

procedure TRAIN(data D, # networks N, # epochs E)
    Sample $|D|$ transitions from D for each network (sample with replacement).
    for e in $1 : E$:
        for n in $1 : N$:
            Shuffle the sampled transitions for network n.
            Form batches of size **128**.
            Loop through batches and take a gradient step of the loss for each batch.
end procedure

of operations that you can run simultaneously with a single call to `Session.run()`. This will help speed up training significantly.

Part2: Trajectory Sampling

For your implementation of PETS, you will use the TS1 sampling method which is shown in Algorithm 3. Note that this algorithm only determines which networks will be used for which time steps to perform the state prediction. It is only one component of model predictive control and will be combined with the model and action optimization method in the next section. You can think of this trajectory sampling method as returning P particles that each

Algorithm 3 Trajectory Sampling with TS1

procedure TS1(# networks N, # particles P, plan horizon T)
    Initialize array S of dimension $P \times T$ to store network assignments for each particle.
    for p in $1 : P$:
        Randomly sample a sequence s of length T where $s \in \{1, \ldots, N\}^T$.
        Set $S[p, :] = s$.
end procedure

represent a path of length T where at each time step, a random network in the ensemble is used to generate the next state.

Part 3: Action Selection with Cross Entropy Method

In order to perform action selection, we need a cost function to evaluate the fitness of different states and action pairs. Defining the right cost function is often the hardest part of getting model-based reinforcement learning to work, since the action selection and resulting trajectories from MPC depend on the cost function. For the `Pushing2D-v1` environment, you will be using the following cost function:

$$\text{cost}(\textbf{pusher}, \textbf{box}, \textbf{goal}, \textbf{a}) = d(\text{pusher}, \text{box}) + 2d(\text{box}, \text{goal}) + 5\left|\frac{\text{box}_x}{\text{box}_y} - \frac{\text{goal}_x}{\text{goal}_y}\right| \quad (2)$$

Feel free to play around with other cost functions to see if you can do better.

We can now use TS1, along with the cost function, to estimate the cost of a particular action sequence $a_{1:T}$ from state s_0. Let TS be the output of TS1 for our settings of N, P, T and let $s_{(p,t)} = \text{model}_{TS[p,t]}.\text{predict}(s_{(p,t-1)}, a_t)$; that is, the next state for a given particle is the predicted state from the model indicated by TS for the given particle and time step applied to the last state of the particle with the given action from $a_{1:T}$ (remember that we are using a probabilistic model so the output is a sample from a normal distribution). We can now calculate the cost of a state and action sequence pair as the sum of the average of the cost over all particles:

$$\text{C}(a_{1:T}, s) = \frac{1}{P}\sum_{p=1}^{P}\sum_{t=1}^{T}\text{cost}(s_{(p,t)}, a_t) \quad (3)$$

where $s_{(p,t)}$ is calculated as described above.

Finally, we can optimize the action sequence with the cross entropy method (CEM), which you implemented in Homework 2. To make things precise, we CEM in Algorithm 4. Note that we assume the actions across time are independent when updating the variance for CEM.

Algorithm 4 CEM

1: **procedure** CEM(population size M, # elites e, # iters I, μ, σ)
2: Generate M action sequences according to μ and σ from normal distribution.
3: **for** i in $1 : I$:
4: **for** m in $1 : M$:
5: Calculate the cost of $a_{m,1:T}$ according to Equation 3.
6: Update μ and σ using the top e action sequences.
7: **return:** μ
8: **end procedure**

Finally: Tying It All Together

The only missing piece left to implement Algorithm 1 is line 6, i.e., how to sample an episode using MPC; this is shown in Algorithm 5. We proceed by starting with an initial μ and σ and use that as input to CEM. Then we take the updated μ from CEM and execute the

action in the first time step in the environment to get a new state that we use for MPC in the next time step. We then update the μ that is used for the next timestep to be the μ from CEM for the remaining steps in the plan horizon and initialize the last time step to 0. Finally, we return all the state transitions gathered so that they can be appended to D.

Algorithm 5 Generating an episode using MPC

procedure MPC(env, plan horizon T)
    `transitions` $= []$
    $s =$ `env.reset()`
    $\mu = \mathbf{0}$, $\sigma = \mathbf{1}$
    while not done:
        $\mu =$CEM(200, 20, 5, μ, σ)
        $a = \mu[0, :]$
        $s' =$ `env.step(a)`
        `transitions.append`(s, a, s')
        $s = s'$
        $\mu = \mu[1 : T]$`.append(0)`
    return: `transitions`
end procedure

For this homework, please respond to the prompts below:

1. **Single probabilistic network (30 pts)** Train a single probabilistic network on transitions from 1000 randomly sampled episodes.

    (a) (10 pts) Plot the loss and RMSE vs number of epochs trained for the single network.

    Solution

    (b) (10 pts) Combine your model with planning using randomly sampled actions. Evaluate the performance of your model when planning using a time horizon of 5 and 2000 possible action sequences. Do this by reporting the percent successes on 50 episodes.

    Solution

    (c) (10 pts) Combine your model with planning using CEM. Evaluate the performance of your model when planning using a time horizon of 5, a population of 200, 20 elites, and 5 iterations. Do this by reporting the percent successes on 50 episodes.

    Solution

    m

(d) (5 pts) Which planning method performs better, random or CEM? How did MPC using this model perform in general? When is the derived policy able to succeed and when does it fail?

Solution

2. **MBRL with PETS**.

(a) (20 pts) Describe in detail your implementation of PETS. Your response should walk the reader through your submitted code so that he/she will understand the key components of your implementation and be able to run your code with different arguments.

Solution

(b) (10 pts) Run Algorithm 1 for 500 iterations (i.e., collect 100 initial episodes and then 500 episodes using MPC). Plot the loss and RMSE vs number of epochs trained for the single network.

Solution

(c) (20 pts) Every 50 epochs, test your model with MPC on 20 episodes and report the percent of successes. Plot this as a function of the number of iterations of PETS.

Solution

(d) (10 pts) Execute MPC with your trained model to generate two episodes, one where $\text{goal}_x - \text{goal}_y > 1$ and another where $\text{goal}_x - \text{goal}_y < -1$. Print the positions of the pusher, box, goal at each time step of the episode. How does your derived policy behave?

Solution

(e) (5 pts) What are some limitations of MBRL? Under which scenarios would you prefer MBRL to policy gradient methods like the ones you implemented in the previous homeworks?

Important Implementation Advice

It takes quite a while to run the experiments in this homework. Please plan accordingly

and get started early! Additionally, with the semester ending, we will be very strict with late days for this homework; please turn your homework in on time to facilitate grading! Again, to make debugging easier, you should implement your code piecewise and test each component as you build up your implementation, i.e., test the probabilistic ensemble, test the trajectory sampling, test the cross entropy method, test the cost calculation, test the MPC, etc.

References

[1] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.