Bootstrapped MPC

CS 294-112

Vladimir Feinberg, Samvit Jain, Michael Whittaker

10 October 2017

1 Smarter Sampling

We consider learning the dynamics for the `HalfCheetah-v1` environment and using those dynamics to simulate rollouts with random actions, taking the action that resulted in the lowest cost after the simulation.

Unless otherwise specified, all parameters are their defaults: 5 different seeds (TODO: for now only 1) for each setup, 60 epochs of dynamics training on the full dataset per on-policy iteration, a dynamics batch size of 512, maximum episode length of 1000, 1000 simulated paths in the MPC controller (each of which has a horizon of 15), and 10 paths sampled by the initial random agent and then by the MPC controller that are aggregated every policy iteration. The learning rate was the default 10^{-3}.

Instead of random sampling during the simulations, one might attempt to have a more intelligent MPC rollout. In particular, we can have a policy learn the from the MPC expert: every iteration we train a neural network from our aggregated policy to mimic the MPC rollout. Then, when we sample, we use MPC control, but only with the first action randomly sampled at uniform. All then next actions are taken by the learned policy. By having more reasonable simulated rollouts, MPC can choose smarter initial actions. The neural network training wasn't a noticeable additional time cost, but resulted in much-improved performance (Figure 1).

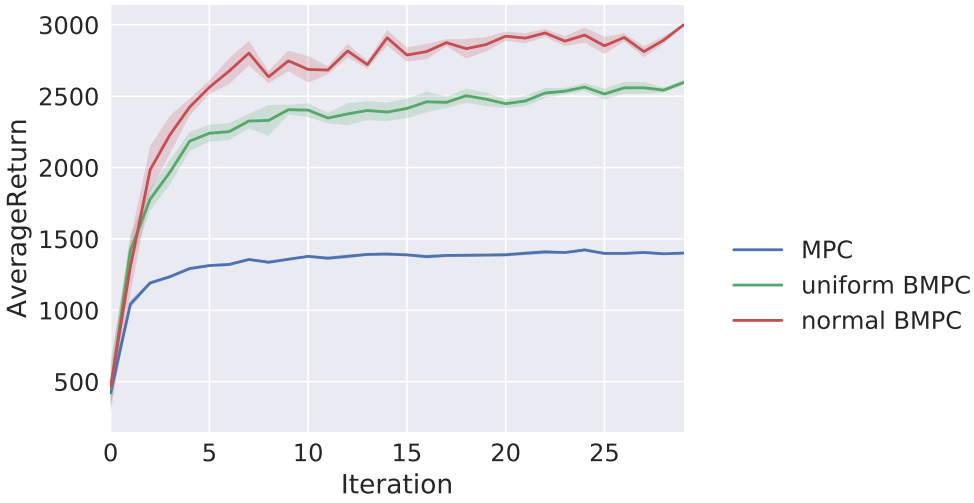

Figure 1: Average return over policy aggregation iterations for the learning-based sampling MPC agent, compared to the standard uniform sampling MPC agent.

This results in a pretty interesting "bootstrapping" effect where the MPC expert gets better by relying on what is essentially a memoized version of itself. Note that the learner neural network is only used starting

on the second step of the MPC rollout simulations, with the first one still randomly sampled. This strategy seems pretty similar to MCTS, but in a continuous setting (where we have a policy picking the action to take).

In Figure 1, we consider the first action to not just be random, but also random and sampled normally around the learned policy. Figure 1 uses a policy provided in the homework. If we don't use the homework's policy, which encodes supervision in the cost function (bad cheetah posture was penalized explicitly), we can only rely on the usual `HalfCheetah-v1` reward. This makes problem significantly harder.

In this case, bootstrapping still helps (Figure 2).

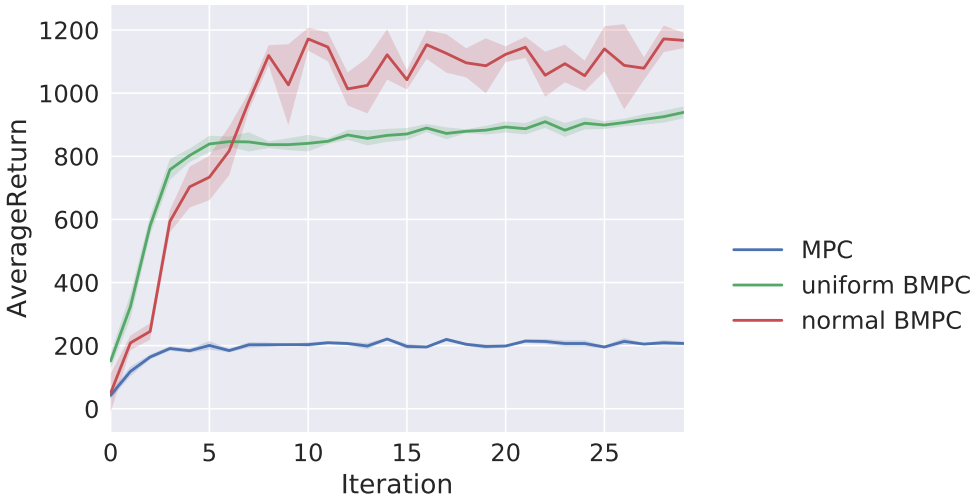

Figure 2: Average return over policy aggregation iterations for the learning-based sampling MPC agent, compared to the standard uniform sampling MPC agent, this time with a hard cost function.

2 Improving the Learner

If we query how the learner at any point during training, we find that its performance is not very good (TODO:figure).

We expect that it may be possible to speed up test-time execution by using the learned policy network mentioned in Section 1, with a DAGGER-style approach as mentioned in the paper: use the learned policy network to make sampling decisions, and only rely on the MPC controller (perhaps using the policy network for sampling again) for expert annotations.

When I tried this, I found that the policy network did not have variable-enough performance to explore sufficiently (Figure 3). Perhaps this was just a matter of messing with variance hyperparameters for the learned network's Gaussian policy, but I tried quite a few. I noticed that the paper mentioned in the assignment warm-started the dataset with (randomly-sampled) MPC samples (whereas I did a cold-start from training on expert-labelled states generated by the random agent). Perhaps the warm-start was key, but it feels like cheating, since you need to run your MPC samples anyway.

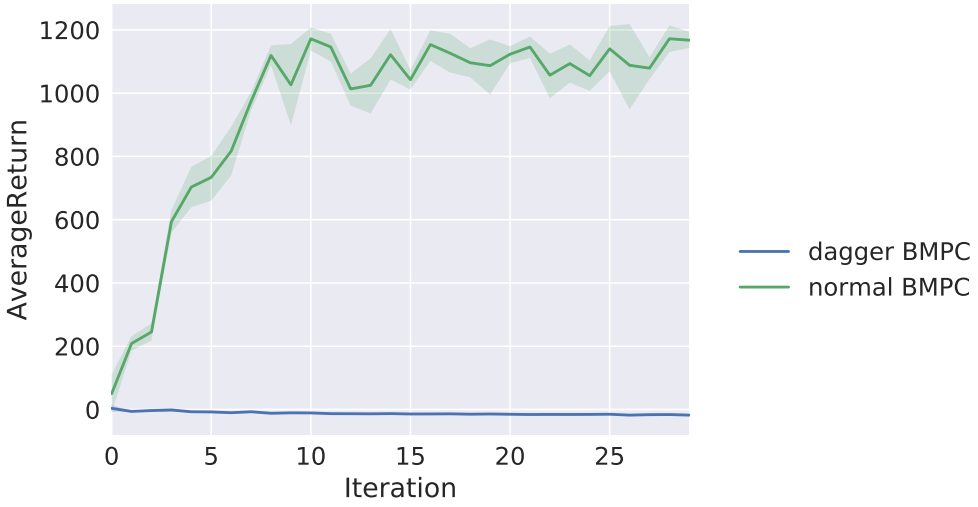

Figure 3: DAgger-augmented bootstrapped MPC vs the original BMPC.