Model-based RL Report

CS 294-112

Vladimir Feinberg

10 October 2017

1 Main Assignment

In this assignment, I implemented model-based RL through an MPC controller. In particular, 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. The cost function is manually specified for the task. All invocations are available in the `README.md` file in the submitted assignment code.

Unless otherwise specified, all parameters are their defaults: 5 different seeds 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, 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}.

Even after learning dynamics from a random sample, the MPC controller performs fairly well (without any aggregated on-policy iterations) (Fig. 1).

Iteration	0
AverageCost	-463
StdCost	31.3
MinimumCost	-513
MaximumCost	-426
AverageReturn	317
StdReturn	16.3
MinimumReturn	282
MaximumReturn	339
DynamicsMSE	0.291

Figure 1: MPC controller performance without any aggregated data.

Next, we evaluate whether we're actually learning the dynamics (Fig. 2). We note that MSE is generally low, but actually improves in the aggregating case because of the extra training data improving generalization.

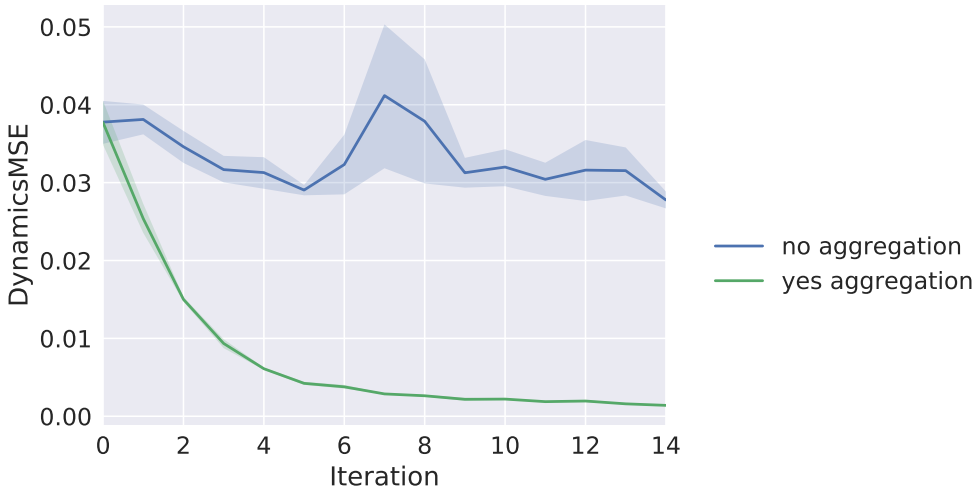

Figure 2: Unnormalized next-state prediction MSE over several iterations of 60-epoch rounds of full-dataset training. We show learning performance with data aggregation (i.e., a changing optimization target for optimization) and without data aggregation (the simpler problem of only learning only the first round of randomly-sampled policy transitions. Note that the MSE metric here is collected from a random agent sampled on unseen, newly reset environments (in the aggregation, this validation data is added into the training set after validating it).

Finally, we evaluate the effect of including on-policy aggregated data from the randomly-sampling MPC controller (Fig. 3).

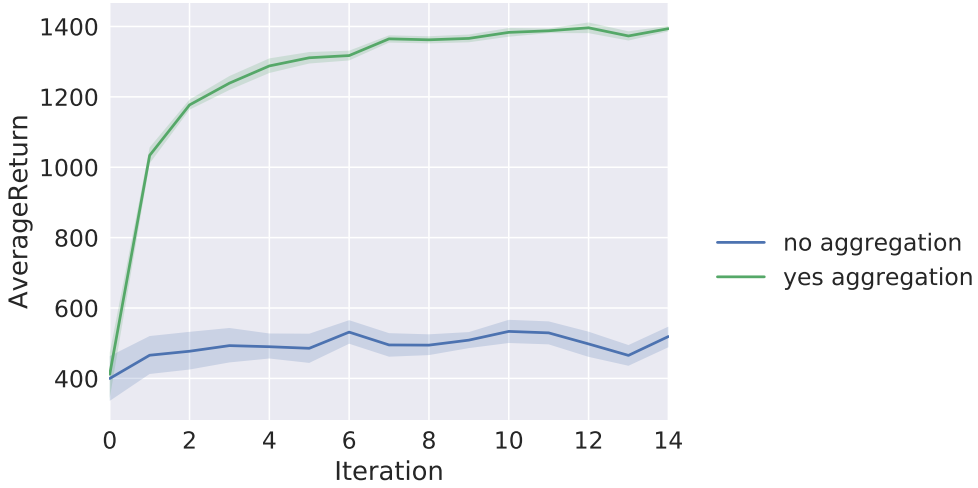

Figure 3: Comparison of average returns with and without on-policy aggregation for the randomly-sampling MPC controller. We compare over 15 such on-policy iterations with 10 sampled on-policy paths each, and display statistics only from the most recent sample. In the case of no aggregation, different iterations reflect samples of the performance of an MPC controller with a dynamics model iteratively learning from the same initial randomly-sampled set of transitions.

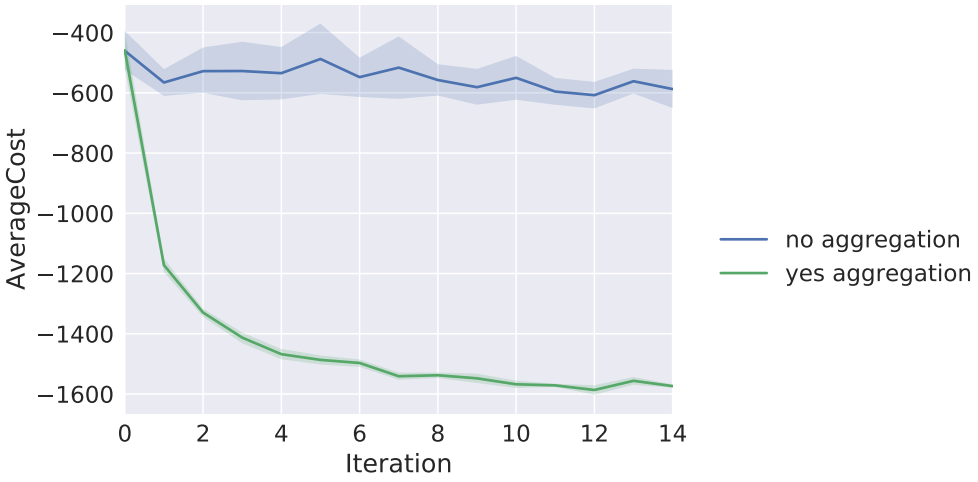

Figure 4: Comparison of average costs with and without on-policy aggregation for the randomly-sampling MPC controller in the same setting as Fig. 3.

2 Extension: Speed

To enable rapid development, I implemented several optimizations for the MPC controller. After following all of these in succession, I was able to get a single MPC iteration (i.e., with all default flags, 10 MPC rollouts total) speed **down to 104.11 seconds**, compared to **around 600 seconds** in a vanilla implementation (only batching within the MPC controller). These results are on an Nvidia K20 GPU.

1. (Vanilla) assuming the MPC controller expands p simulation paths, batch the dynamics prediction in these p simulated rollouts for each time step that is simulated (for a horizon h of simulated timesteps). At this point, the GPU is about 10% utilized.

2. When sampling i (non-simulated) rollouts total from the MPC controller, use i independently-seeded environments and request actions from the MPC controller for all i states in each of the environments simultaneously. Within the MPC controller, batch the dynamics prediction to ip simultaneous predictions for each of h time steps simulated. At this point, the GPU is about 50% utilized.

3. Push the MPC loop onto the GPU alone by using a `tf.while_loop` for the MPC rollout. This requires coupling the dynamics computation graph with the MPC one so the entire MPC rollout can stay within GPU memory in one `tf.Session.run` call (this would also be wayyy easier in PyTorch). This is by far the biggest speedup, now at about 85% GPU utilization.

4. Eke out the last bit of performance by pushing trajectory cost function evaluation into the MPC controller loop. This is a win since the cost is additive across states so the GPU no longer needs to keep track of the entire rollout state matrix. This hits about 90% utilization.

3 Extension: No Delta Normalization

The assignment specification requires training the dynamics model f on normalized outputs (where learned outputs are the changes to the dynamics, $f(\hat{s}) \approx \hat{\delta}$, with $\hat{\cdot}$ indicating normalized values and $\delta = s' - s$ being the state transition). This requires learning statistics for state transition differences δ.

This seemed a bit unstable to me, since some dimensions of δ might reasonably be concentrated around 0 for most transitions. For this reason, I considered predicting the difference in normalized states alone

$\hat{s'} - \hat{s}$, which is still invariant to state distribution location and scale. Its performance with the regular δ-normalizing approach is explored (Fig. 5).

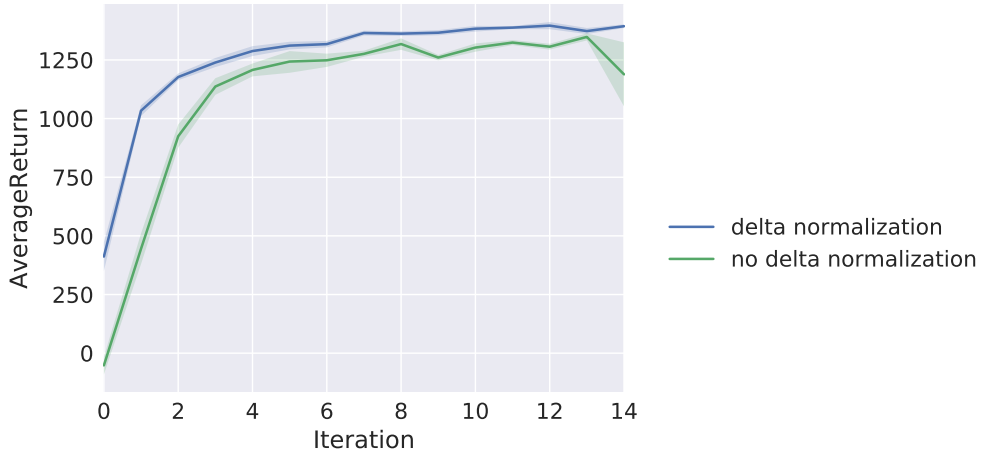

Figure 5: Comparison of average returns for on-policy aggregation with a randomly-sampling MPC controller between δ-normalization and no δ-normalization.

Unfortunately, it seems that this intuition is unsuccessful. It seems that normalizing the transitions improves learning. This may be because the transitions themselves δ have at least a partially stationary distribution (which we might model as a sum of a stationary and a nonstationary component), and normalizing this component still improves learning by keeping the neural network targets at least partially normalized.

4 Extension: BPTT

Instead of using expensive MPC rollouts, I considered training a policy to minimize cost directly, back propagating through time (BPTT), through the dynamics that I learned.

In other words, I consider the following explicit, unconstrained minimization, where θ are our policy parameters, and our dynamics f are fixed and smooth and have known cost c:

$$\operatorname*{argmin}_{\theta} \mathbb{E}_{s_0} \sum_{t=0}^{H} c(s_t, \pi_\theta(s_t), s_{t+1}); \quad s_{t+1} \triangleq f(s_t, \pi_\theta(s_t))$$

Note that the above notation s_t should not be confused with optimization variables—the full expression above is an unconstrained minimization by θ in terms of c, f alone.

The above computation graph can be expressed in space linear in H, so it is tractable to do gradient descent. However, this optimization, as warned in lecture, suffers from several issues. First, we may have the standard RNN issue of vanishing gradients. Second, we may suffer from minor inaccuracies in the dynamics, which may deviate from reality significantly at the end of a simulated rollout but are incorporated in the optimization of θ. I attempted solve these issues by (1) using a gated recurrent unit (GRU) as the policy architecture and (2) reducing H from the full 1000 to 50 and sampling the initial state s_0 from the on-policy state distribution (so the policy trains on random segments within an episode).

Nonetheless, the results were still pretty bad, but it was cool to find out that BPTT really does suck when combined with a shooter method from learned dynamics. , the training task (Fig. 6). Notably, test-time performance is an order of magnitude faster (since we have no rollouts to simulate). We still need to train the policy instead now, but total iteration time is reduced to **23.74 seconds**.

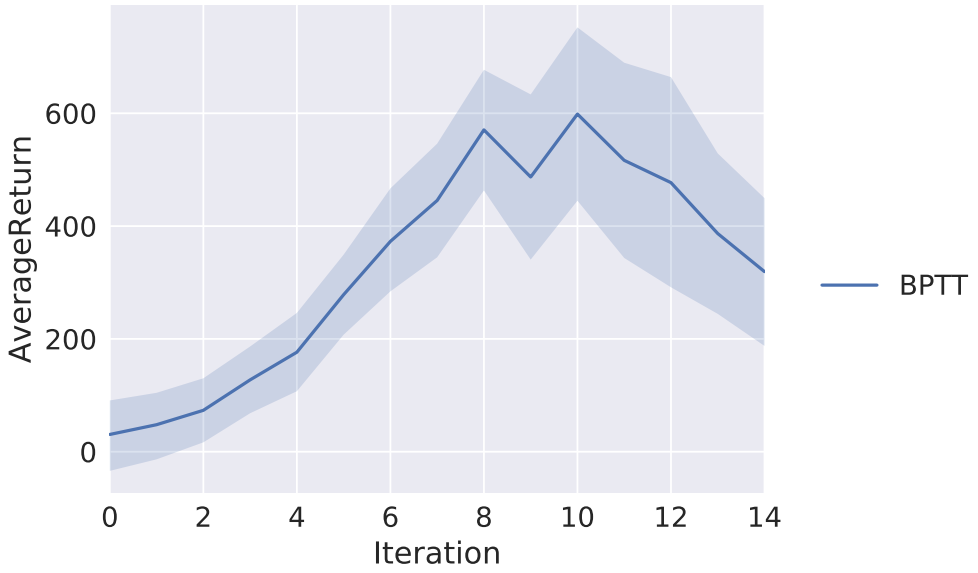

Figure 6: Average return over policy aggregation iterations for the BPTT agent. I use 2 stacked GRU units with 32 hidden units each for the BPTT policy with a learning rate of 10^{-5}, a mini-batch size of 512, and 1 epoch of training per on-policy iteration.

I tried stacked RNN units up to 5 stacked GRUs over variable sizes 8, 16, 32, each over different learning rates $10^{-5}, 10^{-4}, 10^{-3}$, with variable simulation horizons (15, 50, 100), epochs of training (1, 10, 100), and batch sizes (64, 512, 8196). Everything looked as noisy/bad as Fig. 6. I figure that this is the case even with a GRU policy because the GRU policy accepts the states as inputs, but states are not independent data—they're generated by the frozen dynamics f, which is a function of the actions generated by the policy. Perhaps gradients are exploding through this path of the computation graph (and, if so, then perhaps we can make the dynamics f a GRU as well)?

5 Extension: Smarter Sampling

Instead of random sampling, 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 (Fig. 7).

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).

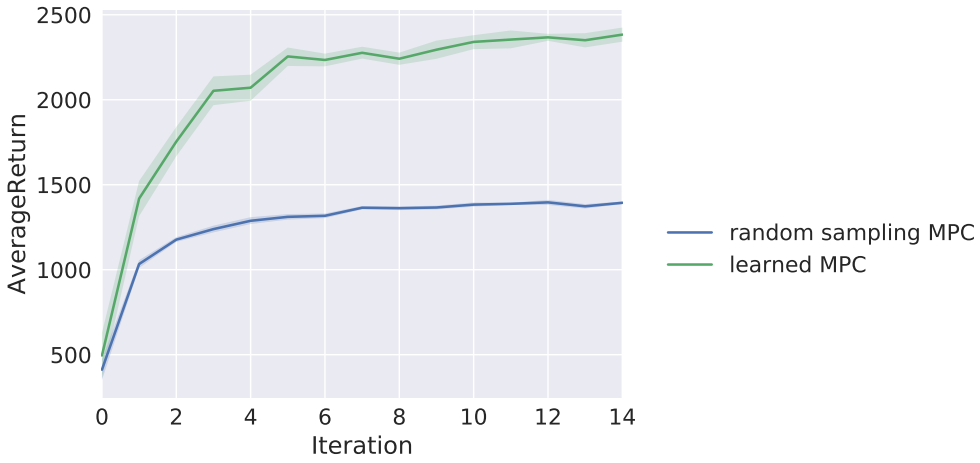

Figure 7: Average return over policy aggregation iterations for the learning-based sampling MPC agent, compared to the standard uniform sampling MPC agent.

6 Conclusion

I expect that it may be possible to speed up test-time execution by using the learned policy network mentioned in Sec. 5, 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. 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.