Micro-data policy search

Learning in a handful of trials

Jean-Baptiste Mouret & Konstantinos Chatzilygeroudis

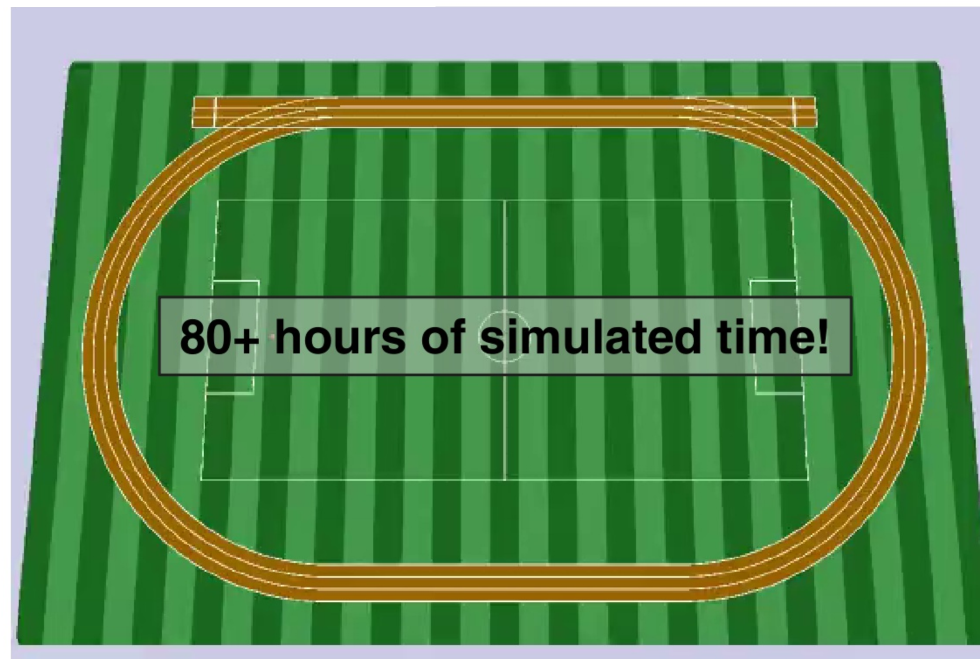
Part 3

Learning models of the dynamics

Konstantinos Chatzilygeroudis

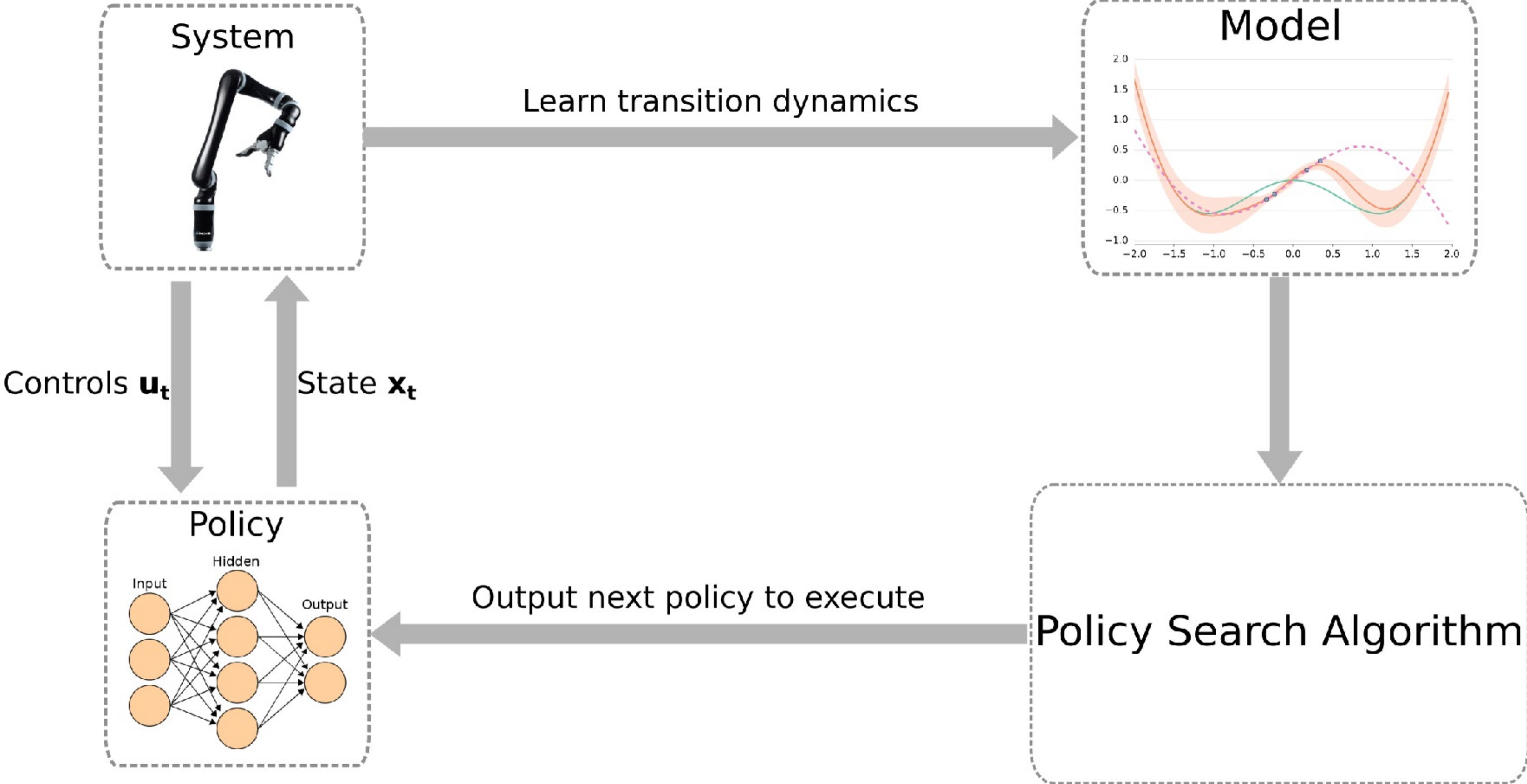
Model-free policy search is good, but ....

... needs many samples to converge.



Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

What if we interact with a model?



What does it mean exactly?

- We have the real world that we model as:

$$x_{t+1} = f(x_t, u_t) + \omega_t$$

- We want to have an estimation of the real dynamics, $\tilde{f}$

- We collect samples of the form $(x_t, u_t) \rightarrow x_{t+1}$
  - In practice this looks more like: $(x_t, u_t) \rightarrow x_{t+1} - x_t$

- And use any regression techinque to find $\tilde{f}$

Which regression model to choose?

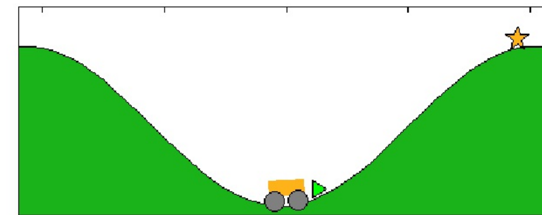
We have many regression techniques, which one to choose?

- **Desired characteristics**
  - Good generalization
  - Work with little data
  - Easy to tune
  - Easy to add “prior knowldege”
  - Scale to high-dimensional data
- **Options**
  - Neural Networks
  - Gaussian Processes
  - SVM
  - ....

Example: Continuous Mountain Car

See notebook

- Continuous Mountain Car
  - we want to “climb up” the mountain
  - we control the force applied to the car
  - but we do not have enough power to climb directly
  - state: $[x, \dot{x}]$
  - Continuous version of the problem



https://en.wikipedia.org/wiki/Mountain_car_problem

Example: Continuous Mountain Car

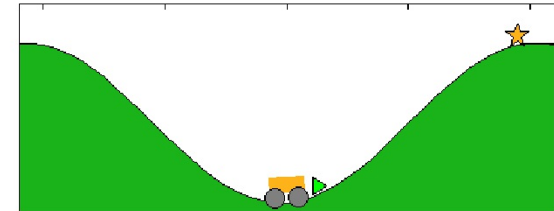
OpenAI gym environment (notebook)

```python
import gym

env = gym.make("MountainCarContinuous-v0")
max_episode_steps = 999
```

```python
steps = 0
total_steps = 0
max_steps = 1500

state = env.reset()
while True:
    action = env.action_space.sample() # sample random action
    next_state, reward, done, _ = env.step(action) # step the world
    state = next_state.copy()
    # env.render() # we could even render
    steps = steps + 1
    total_steps = total_steps + 1
    if total_steps >= max_steps:
        break
    if done or steps >= max_episode_steps:
        state = env.reset()
        steps = 0
```



https://en.wikipedia.org/wiki/Mountain_car_problem

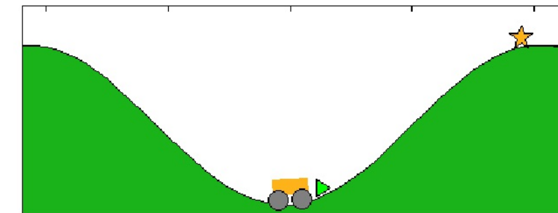
- Continuous Mountain Car
    - we want to “climb up” the mountain
    - we control the force applied to the car
    - but we do not have enough power to climb directly
    - state: $[x, \dot{x}]$

Example: Continuous Mountain Car

Collecting data for model learning (notebook)

```
data = []
steps = 0
total_steps = 0
max_steps = 1500

state = env.reset()
while True:
    action = env.action_space.sample() # sample random action
    next_state, reward, done, _ = env.step(action) # step the world
    if (total_steps % 10) == 0:  # do not collect all data to avoid too slow learning
        data += [(state.copy(), action.copy(), next_state.copy())]
    state = next_state.copy()
    # env.render() # we could even render
    steps = steps + 1
    total_steps = total_steps + 1
    if total_steps >= max_steps:
        break
    if done or steps >= max_episode_steps:
        state = env.reset()
        steps = 0
```



https://en.wikipedia.org/wiki/Mountain_car_problem

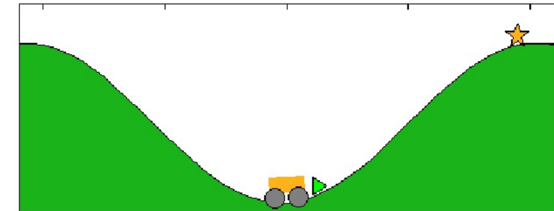
- Continuous Mountain Car
  - we want to “climb up” the mountain
  - we control the force applied to the car
  - but we do not have enough power to climb directly
  - state: $[x, \dot{x}]$

Example: Continuous Mountain Car

Prepare the data to pass to the model (notebook)

```
N = len(data)
D = state.shape[0]
A = action.shape[0]
# Let's create the dataset
X = np.zeros((N, D+A))
Y = np.zeros((N, D))

for i in range(N):
    state, action, next_state = data[i]
    X[i, :D] = state.copy()
    X[i, D:] = action.copy()
    Y[i, :] = (next_state - state)
```



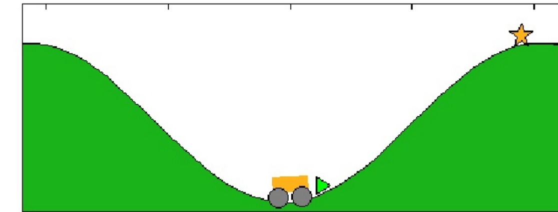
https://en.wikipedia.org/wiki/Mountain_car_problem

- Continuous Mountain Car
  - we want to “climb up” the mountain
  - we control the force applied to the car
  - but we do not have enough power to climb directly
  - state: $[x, \dot{x}]$

10

Example: Continuous Mountain Car

Learn GP models with GPy (notebook)



https://en.wikipedia.org/wiki/Mountain_car_problem

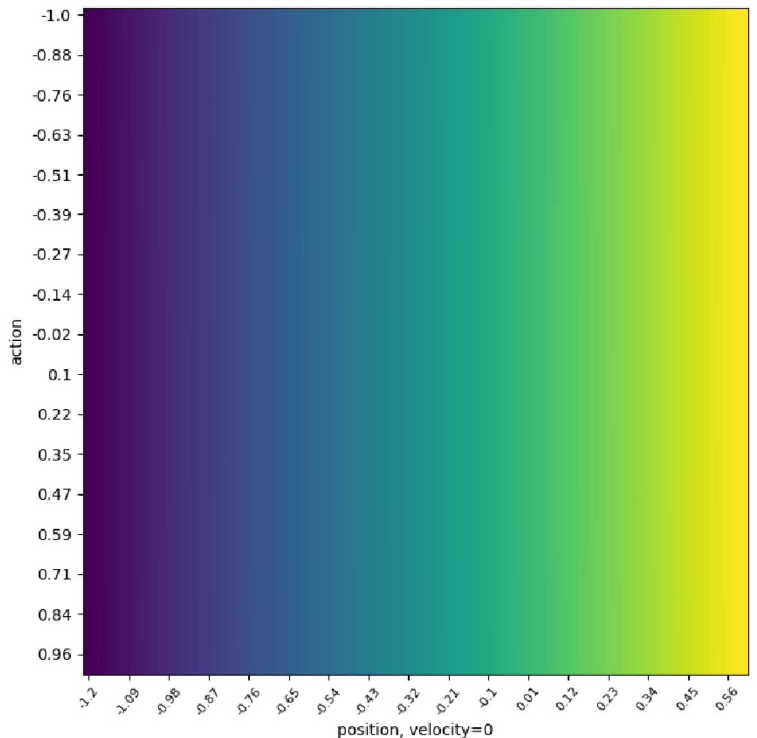
```
import GPy
# Learn the model
print("Learning GP models...")
# we need one GP per output dimension
start_time = time.time()
models = []
for i in range(D):
    # we use a simple RBF kernel
    kernel = GPy.kern.RBF(input_dim=D+A, variance=1., lengthscale=1.)
    m = GPy.models.GPRegression(X, Y[:, i].reshape((N, 1)), kernel)
    # m.optimize(messages=True) # we could optimize for the hyper-parameters also
    models += [m]

total_time = time.time() - start_time
print("Models learnt in " + str(round(total_time, 2)) + "s")
```

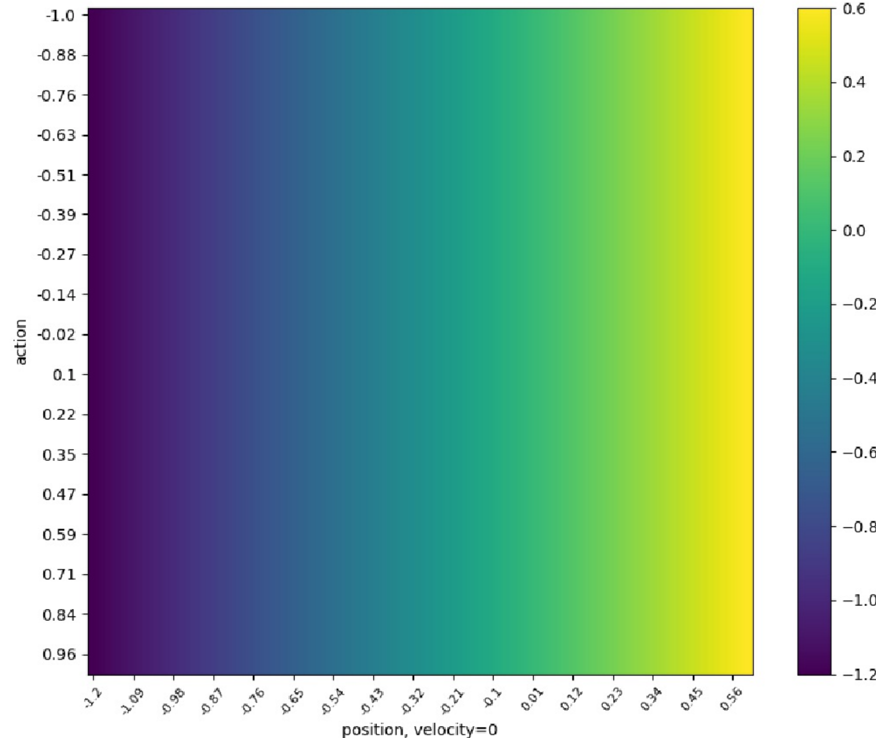
GPs for Learning Dynamical Models

- We assume each output is independent
- We learn a different GP for each output
- We use **GPy** to learn the model

How good is the learnt model?

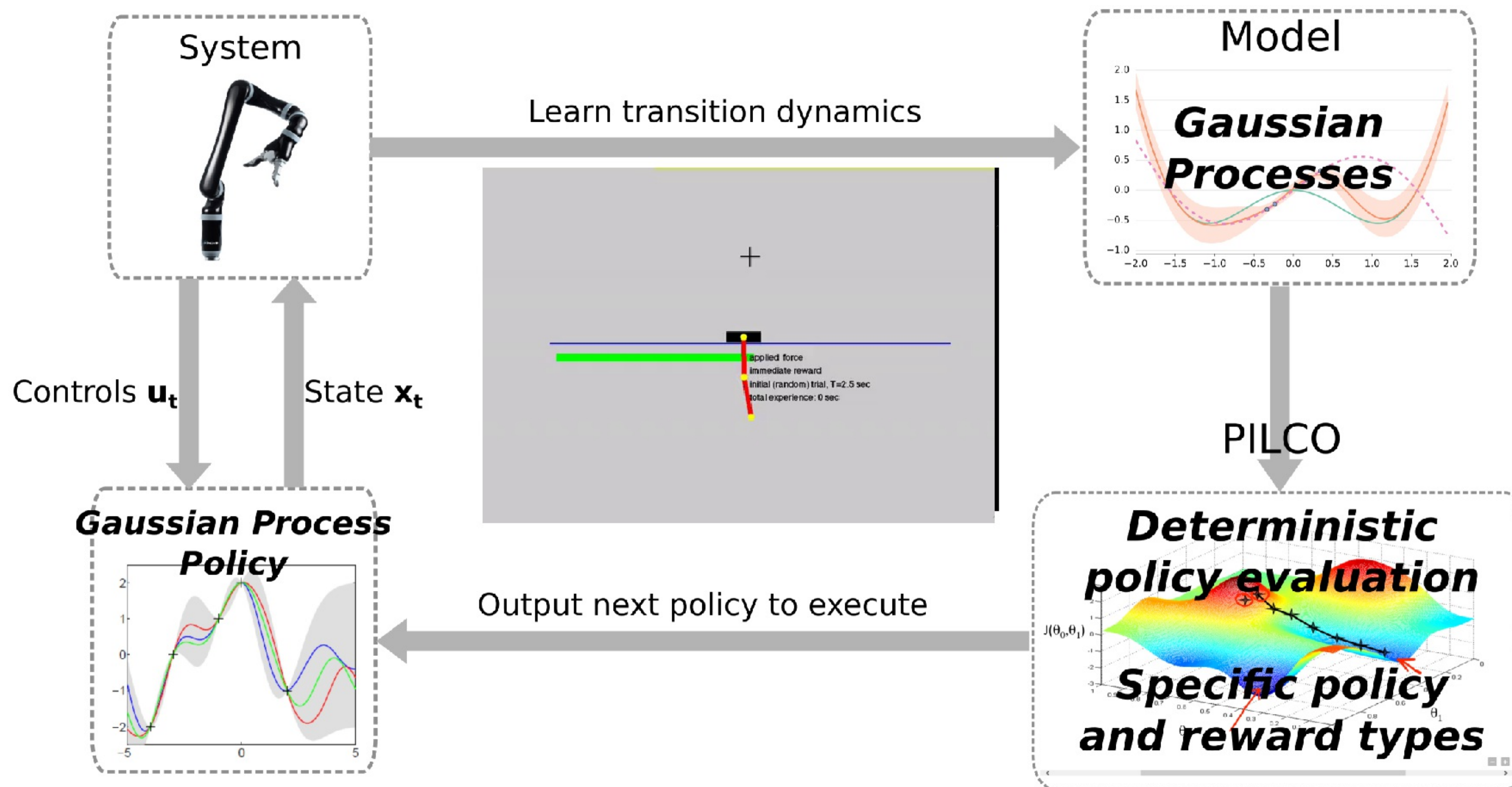


GP model



True model

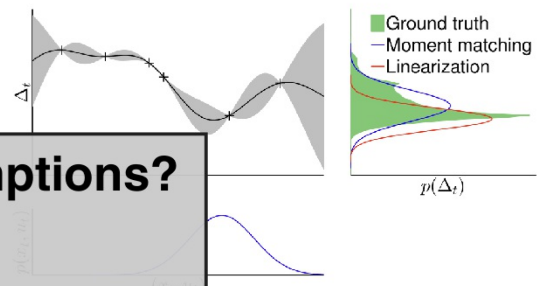
PILCO: Learning in seconds!



PILCO: Learning in seconds!

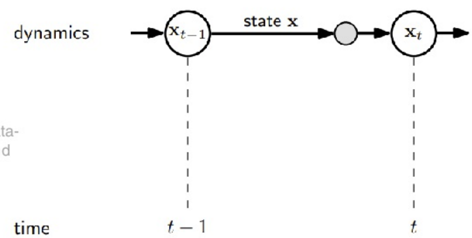
Algorithm 2 Detailed implementation of PILCO

1: **init:** $\psi, p(\mathbf{x}_0), a, b, T_{\text{init}}, T_{\max}, N, \Delta_t$ ▷ initialization
2: $T := T_{\text{init}}$ ▷ initial prediction horizon
3: generate initial training set $\mathcal{D} = \mathcal{D}_{\text{init}}$ ▷ initial interaction phase
4: **for** $j = 1$ to PS **do**
5: learn probabilistic dynamics model based on $\mathcal{D}$ ▷ update model (batch)
6: find ψ^* with $\pi^*_{\psi^*} \in \arg\min_{\pi_\psi \in \Pi} V^{\pi_\psi}(\mathbf{x}_0) \,|\, \mathcal{D}$ ▷ policy search via internal simulation
7: compute $\mathcal{A}_t := \mathbb{E}_{\mathbf{x}_t}[c(\mathbf{x}_t)|\pi^*_{\psi^*}], t = 1, \ldots, T$
8: **if** task_learned($\mathcal{A}$) **and** $T < T_{\max}$ **then** ▷ if good solution predicted
9: $T := 1.25\,T$ ▷ increase prediction horizon
10: **end if**
11: apply $\pi^*_{\psi^*}$ to the system for T time steps, o
12: $\mathcal{D} := \mathcal{D} \cup \mathcal{D}_{\text{new}}$ ▷ update data set
13: **end for**
14: **return** π^* ▷ return learned policy

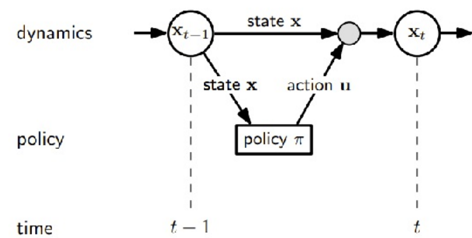


Can we relax the assumptions?

Can we parallelize?



(a) Cascading predictions without a policy.



(b) Cascading predictions with a policy.

Deisenroth, M.P., Fox, D. and Rasmussen, C.E., 2013. Gaussian processes for data-efficient learning in robotics and control. IEEE transactions on pattern analysis and machine intelligence, 37(2), pp.408-423.

Black-DROPS: Data-Efficient Black-Box Policy Search

Key Idea #1 *Implicit Policy Evaluation*; treat each rollout as a noisy measurement of the *expected long-term reward*:

$$G(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + N(\boldsymbol{\theta})$$

$$\mathbb{E}\Big[G(\boldsymbol{\theta})\Big] = \mathbb{E}\Big[J(\boldsymbol{\theta}) + N(\boldsymbol{\theta})\Big] = \mathbb{E}\Big[J(\boldsymbol{\theta})\Big] + \mathbb{E}\Big[N(\boldsymbol{\theta})\Big]$$

$$= J(\boldsymbol{\theta}) + \mathbb{E}\Big[N(\boldsymbol{\theta})\Big] (\text{since } \mathbb{E}\Big[\mathbb{E}[x]\Big] = \mathbb{E}[x])$$

Key Idea #2 Use a black-box optimizer that:

- Is suited for noisy optimization;
- Can take advantage of multi-core computers.

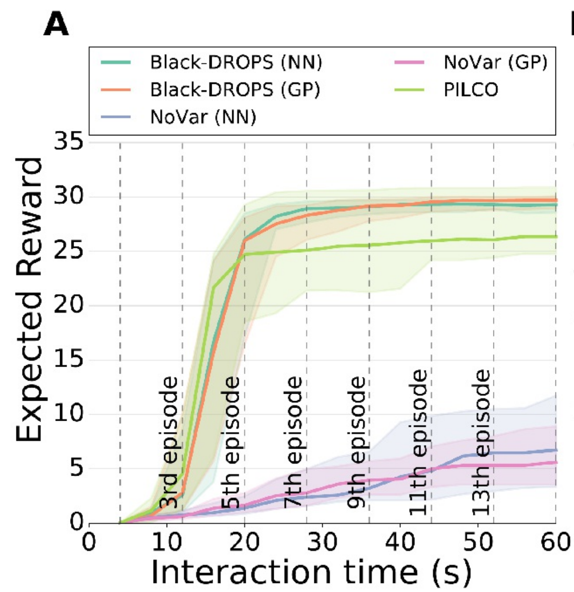
Chatzilygeroudis, K., Rama, R., Kaushik, R., Goepp, D., Vassiliades, V. and Mouret, J.B., 2017, September. Black-box data-efficient policy search for robotics. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 51-58). IEEE.

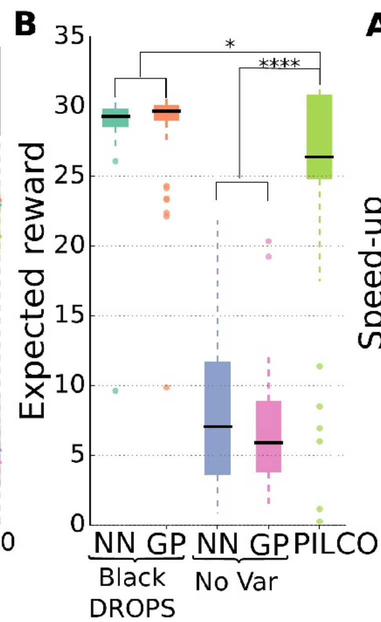
We use a modified version of IPOP-CMA-ES[1] that fulfills the above properties.

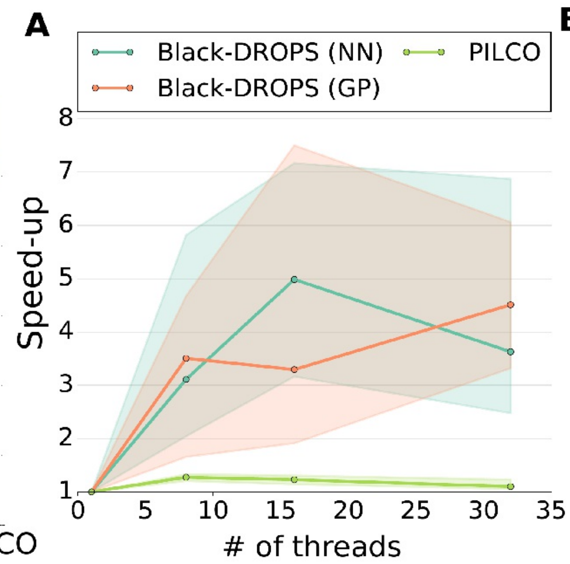
---

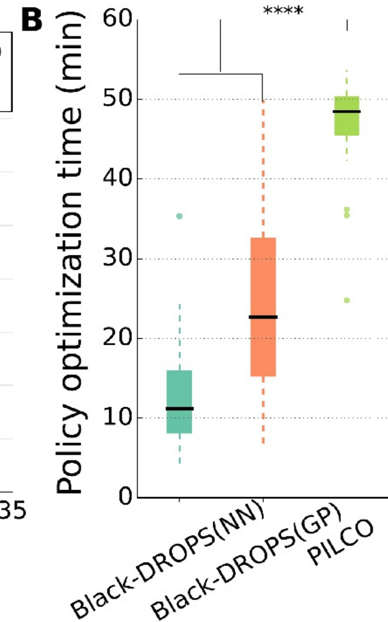
[1] Hansen, N., Ostermeier, A. "Completely derandomized self-adaptation in evolution strategies", Evolutionary Computation, 2001

Black-DROPS: How well does it work?

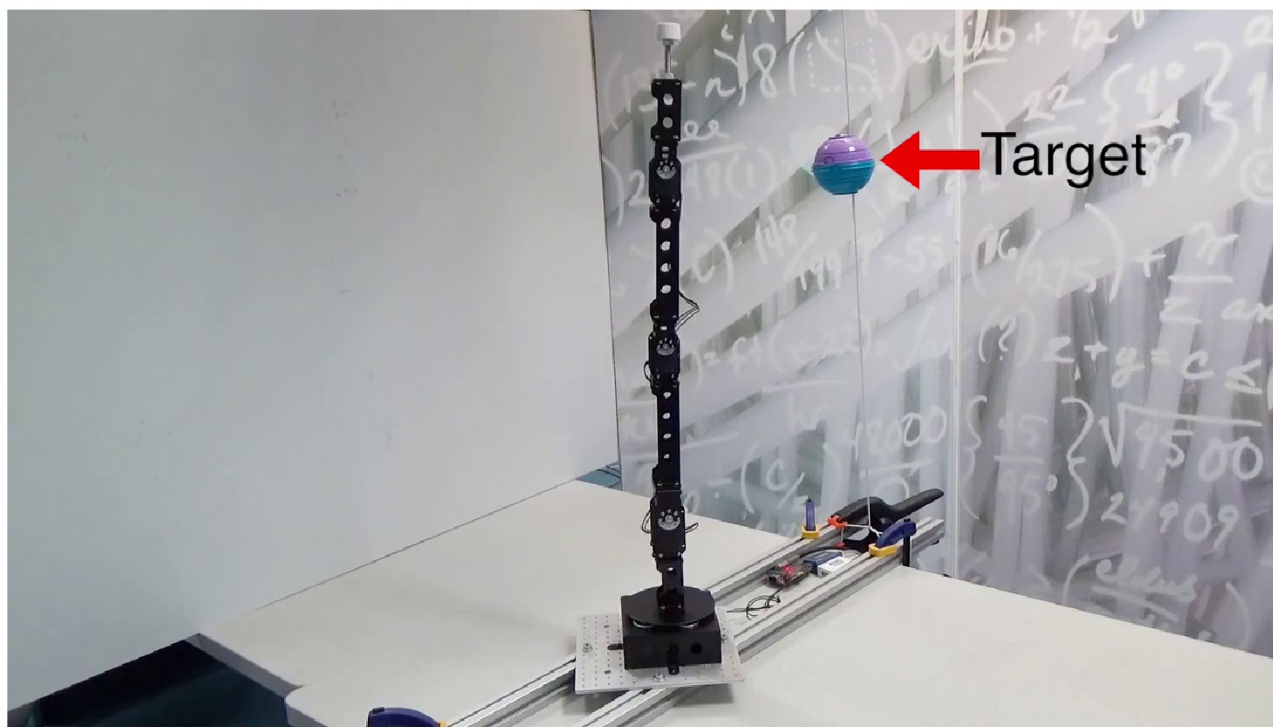




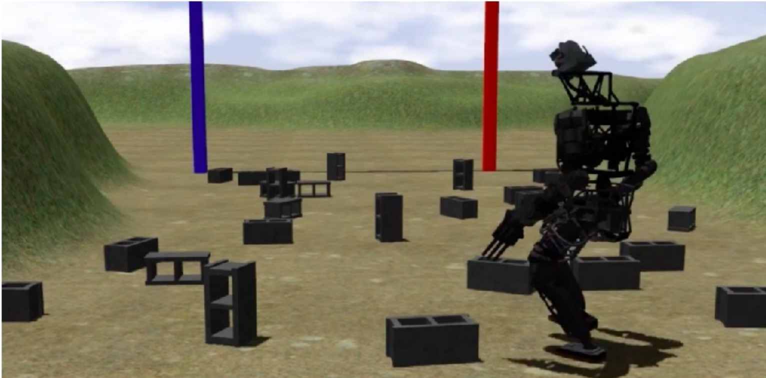


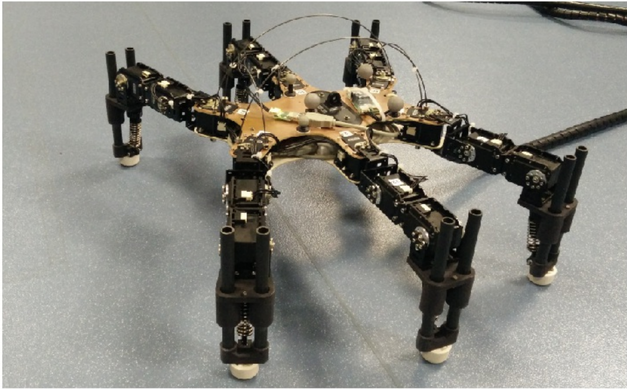


Black-DROPS: How well does it work?



How about some big robots?





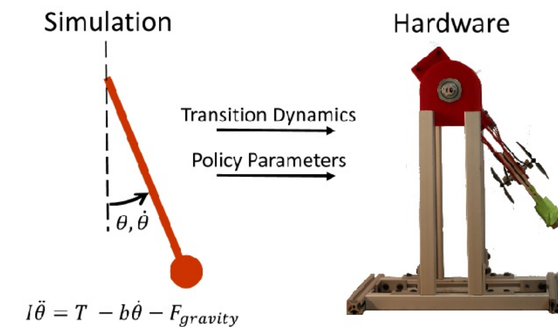
GP-MI: Combining parametric learning with model identification

Key idea:

$$x_{t+1} = M(x_t, u_t, \varphi) + F(x_t, u_t, \theta) + \omega_t$$

So we now approximate:

$$\tilde{f} = M(x_t, u_t, \varphi) + \widetilde{F}(x_t, u_t, \theta)$$



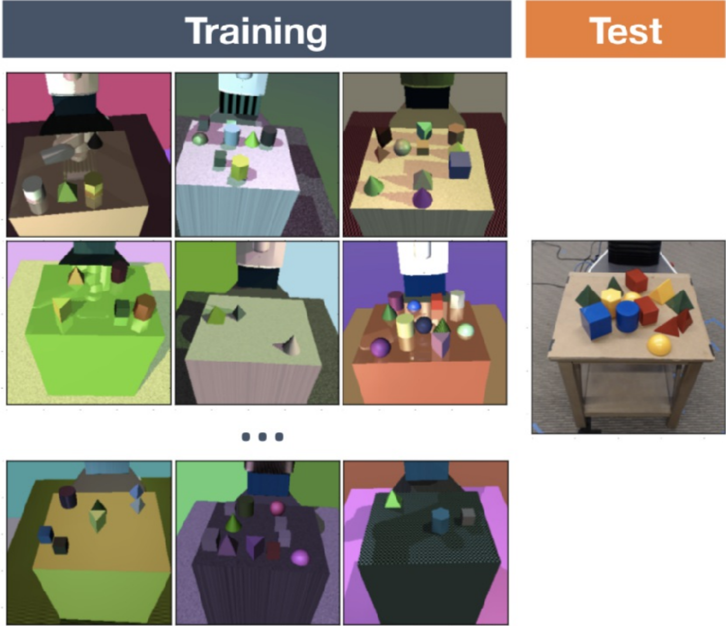
$M(x_t, u_t, \varphi)$ is known → we can further tune some parameters!

$\widetilde{F}(x_t, u_t, \theta)$ is learned → data-driven but learns only the difference!

Chatzilygeroudis, K. and Mouret, J.B., 2018, May. Using parameterized black-box priors to scale up model-based policy search for robotics. In 2018 IEEE International Conference on Robotics and Automation (ICRA) (pp. 5121-5128). IEEE.

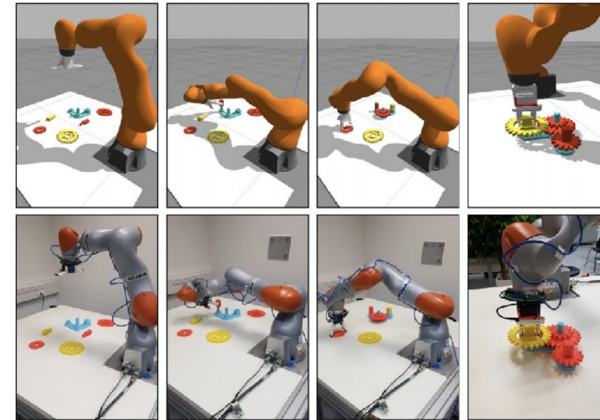
Black-DROPS with GP-MI

Sim2Real Methods



Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W. and Abbeel, P., 2017, September. Domain randomization for transferring deep neural networks from simulation to the real world. In 2017 IEEE/RSJ international conference on intelligent robots and systems (IROS) (pp. 23-30). IEEE.

Sim2Real Methods: Brief Overview



Litvak, Y., Biess, A. and Bar-Hillel, A., 2018. Learning a High-Precision Robotic Assembly Task Using Pose Estimation from Simulated Depth Images. ArXiv abs/1809.10699.

Main intuitions:

- Simulators give us the ground truth data
- Simulators can be fast
- Simulators are reproducible
- Simulators allow easy customization

Main steps:

Domain Randomization (DR)

- For each episode, select a random initialization of the world dynamics
- Optimize the policy for a few steps (e.g., via RL)
- Create new episode

Imitation Learning (IL)

- Solve/Learn the task in simulation
- Collect many many variations of the solution (e.g., different view-points)
- Learn the policy via supervised learning

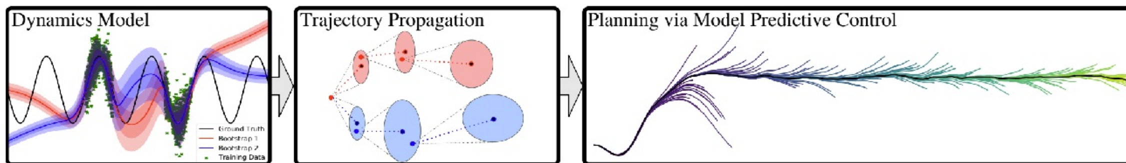
DR + IL gives best results...

Model-based Policy Search Methods with Neural Networks

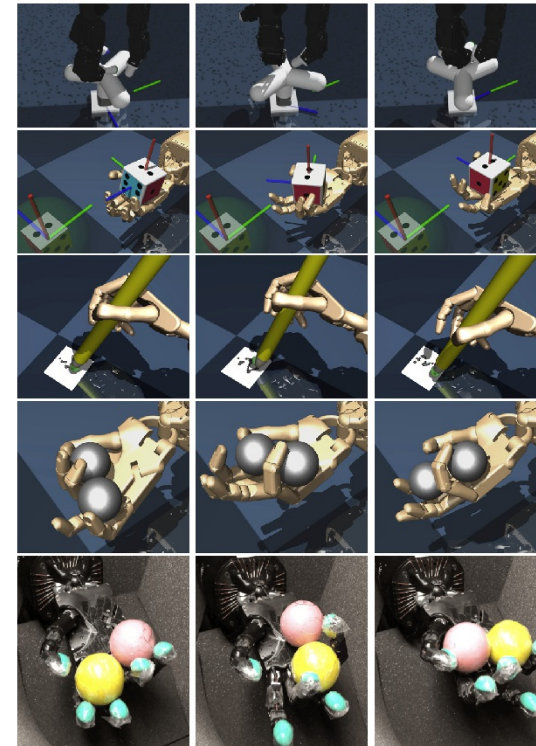
Algorithm 2 Model-Based Policy Optimization with Deep Reinforcement Learning

1: Initialize policy π_ϕ, predictive model p_θ, environment dataset $\mathcal{D}_{\text{env}}$, model dataset $\mathcal{D}_{\text{model}}$
2: **for** N epochs **do**
3:   Train model p_θ on $\mathcal{D}_{\text{env}}$ via maximum likelihood
4:   **for** E steps **do**
5:     Take action in environment according to π_ϕ; add to $\mathcal{D}_{\text{env}}$
6:     **for** M model rollouts **do**
7:       Sample s_t uniformly from $\mathcal{D}_{\text{env}}$
8:       Perform k-step model rollout starting from s_t using policy π_ϕ; add to $\mathcal{D}_{\text{model}}$
9:     **for** G gradient updates **do**
10:       Update policy parameters on model data: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi, \mathcal{D}_{\text{model}})$

Janner, M., Fu, J., Zhang, M. and Levine, S., 2019. When to trust your model: Model-based policy optimization. NeurIPS.

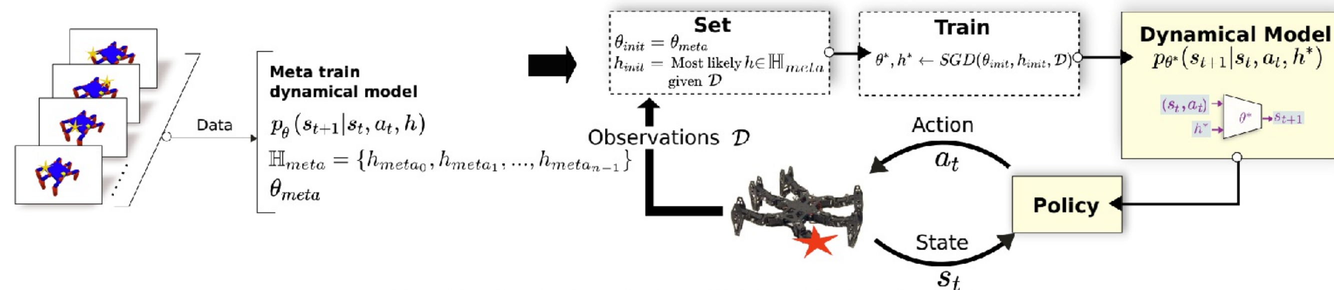


Chua, K., Calandra, R., McAllister, R. and Levine, S., 2018. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. NeurIPS.



Nagabandi, A., Konolige, K., Levine, S. and Kumar, V., 2020, May. Deep dynamics models for learning dexterous manipulation. In Conference on Robot Learning (pp. 1101-1112). PMLR.

Meta-Learning the model



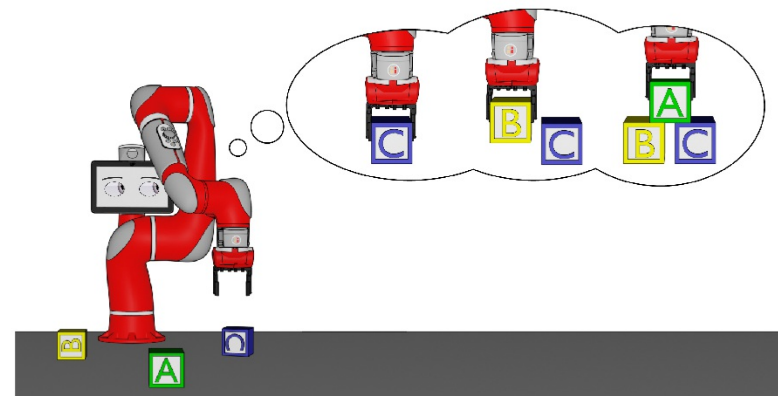
Kaushik, R., Anne, T. and Mouret, J.B., 2020. Fast online adaptation in robotics through meta-learning embeddings of simulated priors. IROS.

We can meta-learn the model using a simulator:

- Prior to deployment:
  - Use multiple instantiations of the simulator (e.g. different damage conditions)
  - Learn a meta-model
- When running:
  - Every K time-steps, select the most-appropriate model
  - Update the meta-models
  - Use MPC with the selected model

Conclusions: Learning models of the dynamics

Learn models of the transition dynamics



M. Janner, https://bair.berkeley.edu/blog/2019/12/12/mbpo/

Effective strategy to reduce interaction time, but...

- Learning a good model is difficult
- What about exploration?
- Does not scale very well to big observation spaces

Simulators and priors are key here (again!)
We can use simulators:

- as “mean” functions
- for meta-learning models
- for learning robust policies that transfer directly (Sim2Real)
- as prior models that can be tuned online