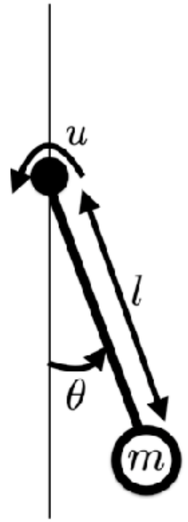
ROB-GY 6323
reinforcement learning and optimal control for robotics

Lecture XII
policy gradient II - model-based reinforcement learning

Project #2 (due December 21st - no deadline extension)

Goal: implement Q-learning to control an inverted pendulum



Deliverables:

1) report in pdf format answering all the questions that do not request code. DO NOT include code in the report
2) One (or several) Jupyter notebook(s) containing all the code used to answer the questions. The notebook(s) should be runnable as is.

Paper report (due December 21st - no deadline extension)

Goal: read one scientific paper and understand it

Pick one paper from the list of papers markers with a * on brightspace

Report (maximum 2 pages - IEEE format double column)
It should contain 3 sections:

1. A section that summarizes the paper:
   What was done? How was it done?
   Why was it worth doing? What are the results?
2. A section explaining how the paper relates to the algorithms seen in class.
   Which algorithms? What is different?
3. A section containing a critical discussion on the paper: pros and cons.
   What seems to work and what convinces you about the result What are the issues/ limitations? What could be done better? What should be done next?

Do not copy equations or figures from the paper - keep your explanations to the point

$\mathbf{x}_{n+1} = \mathbf{f}(\mathbf{x}_n, \mathbf{u}_n, \boldsymbol{\omega}_n)$

given a dynamical system

$\min_{u_k} \sum_k g_k(x_k, u_k)$

minimize a performance cost
(measuring how well a task is performed)

Bellman's principle of optimality: $J_n(x_n) = \min_{u_n} g_n(x_n, u_n) + J_{n+1}(f(x_n, u_n))$

f(x,u) is known

Global Methods
=> optimal value and policy functions

- Dynamic Programming
- Shortest path algorithms
- Linear Quadratic Regulators
- Linear Model Predictive Control
- Value and Policy Iteration

Local Methods
=> (locally) optimal policies / trajectories

- LQ problems with constraints
- DDP / iLQR
- Nonlinear trajectory optimization

f(x,u) is unknown

Learn the value function (or Q-function)

- TD learning and Q-learning
- deep Q-learning

Learn the dynamic model

- Model-based RL

Learn the policy

- Actor-critic (Deep RL)
- Policy gradients

Stochastic policies (discrete actions)

$\pi(u|x) \rightarrow$ probability of selecting u in state x

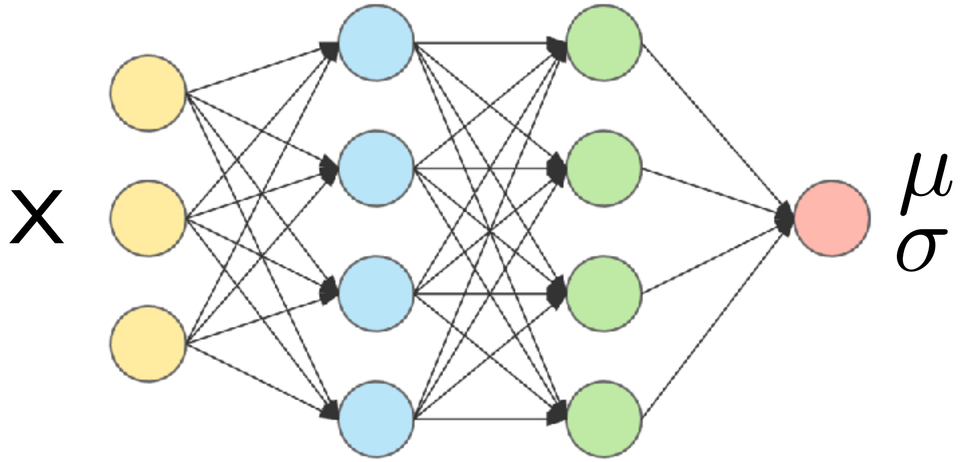
Exponential soft-max distributions: $\pi(u|x,\theta) = \frac{e^{h(x,u,\theta)}}{\sum_a e^{h(x,a,\theta)}}$

where $h(x,u,\theta)$ reflects preferences for each state-action pair

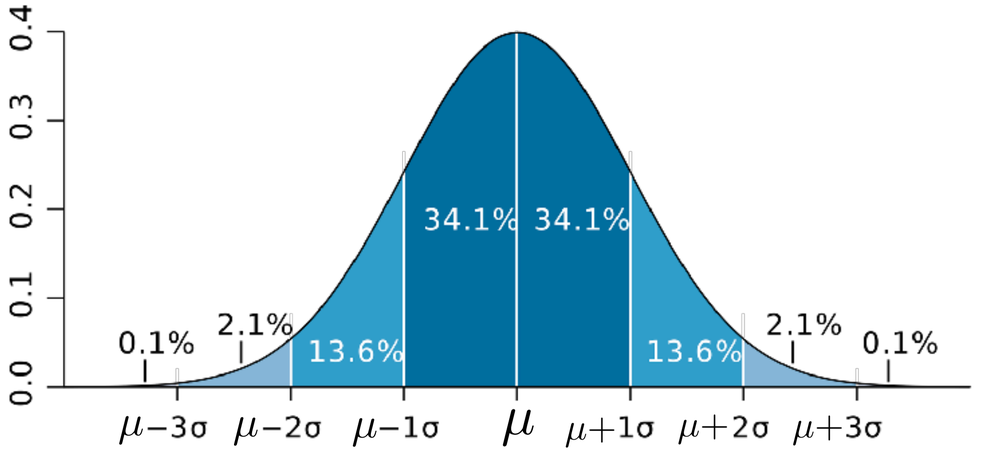
Only for discrete action space

Stochastic policies for continuous actions

Gaussian policies parametrized by a neural network



$$u \sim \mathcal{N}(\mu, \sigma^2)$$



Policy gradient methods

$\pi(u|x) \rightarrow$ probability of selecting u in state x

$$J(\theta) = \mathbb{E}_{u_n \sim \pi_\theta} \left[\sum_{n=0}^{N} \alpha^n g(x_n, u_n) \right] \qquad R(\tau) = \sum_{n=0}^{N} \alpha^n g(x_n, u_n)$$

$$\nabla_\theta J(\theta) = \mathbb{E} \left[\sum_{n=0}^{N} R(\tau) \nabla_\theta \log \pi(u_n|x_n, \theta) \right]$$

Policy gradient methods

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} R(\tau)\nabla_\theta \log \pi(u_n|x_n,\theta)\right] \quad R(\tau) = \sum_{n=0}^{N} \alpha^n g(x_n,u_n)$$

REINFORCE $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} G_n\nabla_\theta \log \pi(u_n|x_n,\theta)\right] \quad G_n = \sum_{k=n}^{N} \alpha^k g(x_k,u_k)$$

REINFORCE with baseline $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (G_n - V(x_n))\nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Actor-critic

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (g(x_n,u_n) + \alpha V(x_{n+1}) - V(x_n))\nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

REINFORCE

[Williams, 1992]

Initialize the policy parameters θ for an input policy $\pi(u|x,\theta)$

Choose a step size γ (using discount factor α)

Loop forever (for each episode):

Generate an episode $x_0, u_0, x_1, u_1, \cdots, x_N, u_N$ following π

For each step t of the episode

$$G_t = \sum_{k=t}^{T} \alpha^k g(x_k, u_k)$$

$$\theta \leftarrow \theta - \gamma G_t \nabla_\theta \left[\ln \pi(u_t|x_t,\theta)\right]$$

REINFORCE with baseline

[Williams, 1992]

Initialize parameters θ_V for value function $V(x, \theta_V)$

Initialize parameters θ_π for policy function $\pi(u|x, \theta_\pi)$

Choose step sizes $\gamma_\pi > 0$ and $\gamma_V > 0$

Loop forever (for each episode):

Generate an episode $x_0, u_0, x_1, u_1, \cdots, x_N, u_N$ following π

For each step t of the episode

$$G_t = \sum_{k=t}^{T} \alpha^k g(x_k, u_k)$$

$$\theta_V \leftarrow \theta_V - \gamma_V \Big(V(x_t) - G_t \Big) \cdot \nabla_{\theta_V} V(x_t, \theta_V)$$

$$\theta_\pi \leftarrow \theta_\pi - \gamma_\pi \Big(G_t - V(x_t) \Big) \cdot \nabla_{\theta_\pi} \left[\ln \pi(u_t|x_t, \theta_\pi) \right]$$

One-step Actor-critic algorithm

Initialize parameters θ_V for value function $V(x, \theta_V)$

Initialize parameters θ_π for policy function $\pi(u|x, \theta_\pi)$

Choose step sizes $\gamma_\pi > 0$ and $\gamma_V > 0$

Loop forever (for each episode):

$I \leftarrow 1$ Initialize the initial state x_0

Loop for the duration of the episode

$u \sim \pi(.|x, \theta)$

Apply action u and get x_{t+1}

$\delta \leftarrow g(x_t, u) + \alpha V(x_{t+1}, \theta_V) - V(x_t, \theta_V)$

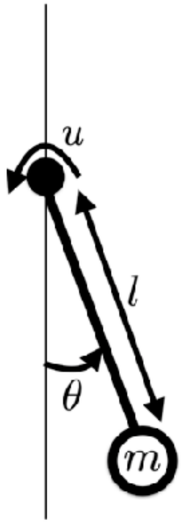
$\theta_V \leftarrow \theta_V + \gamma_V I \delta \nabla V(x, \theta_V)$

$\theta_\pi \leftarrow \theta_\pi - \gamma_\pi I \delta \nabla \ln \pi(u|x, \theta_\pi)$

$I \leftarrow \alpha I$

REINFORCE

$$\min \sum_{i=0}^{N} \alpha^i g(\theta_i, \omega_i, u_i)$$



$$g(x, v, u) = (x - \pi)^2 + 0.01v^2 + 0.00001u^2$$

$$u = [-5,\ 0,\ 5]$$

Softmax stochastic policy $\pi(u|x, \theta) = \frac{e^{h(x,u,\theta_\pi)}}{\sum_a e^{h(x,a,\theta)}}$

$$h(x, u, \theta_\pi) = \theta_\pi^T \Psi(x, u)$$

$$\psi_{k,l,c,0}(\theta, \omega, u) = \frac{e^{-\frac{(u-u_c)^2}{0.002}}}{\sqrt{2\pi 0.001}} \cos(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

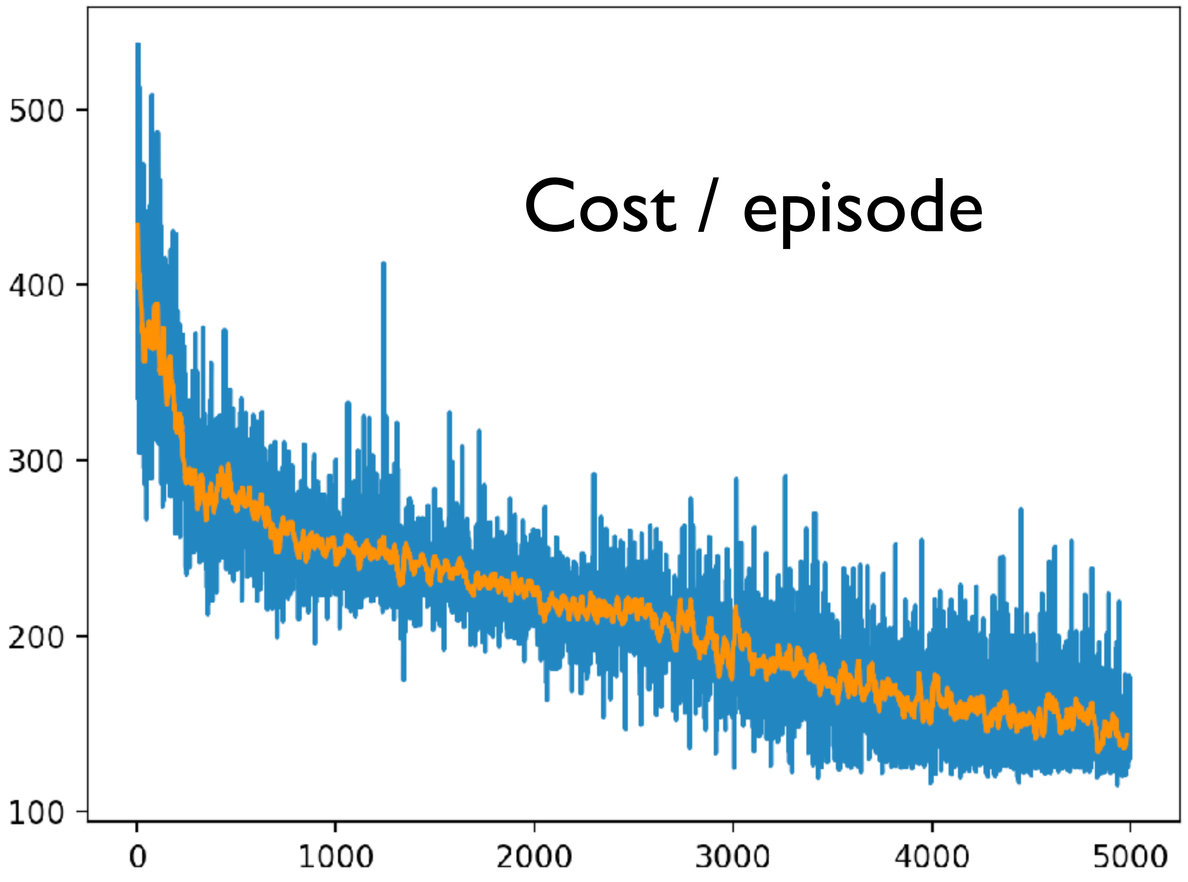
$$\psi_{k,l,c,1}(\theta, \omega, u) = \frac{e^{-\frac{(u-u_c)^2}{0.002}}}{\sqrt{2\pi 0.001}} \sin(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

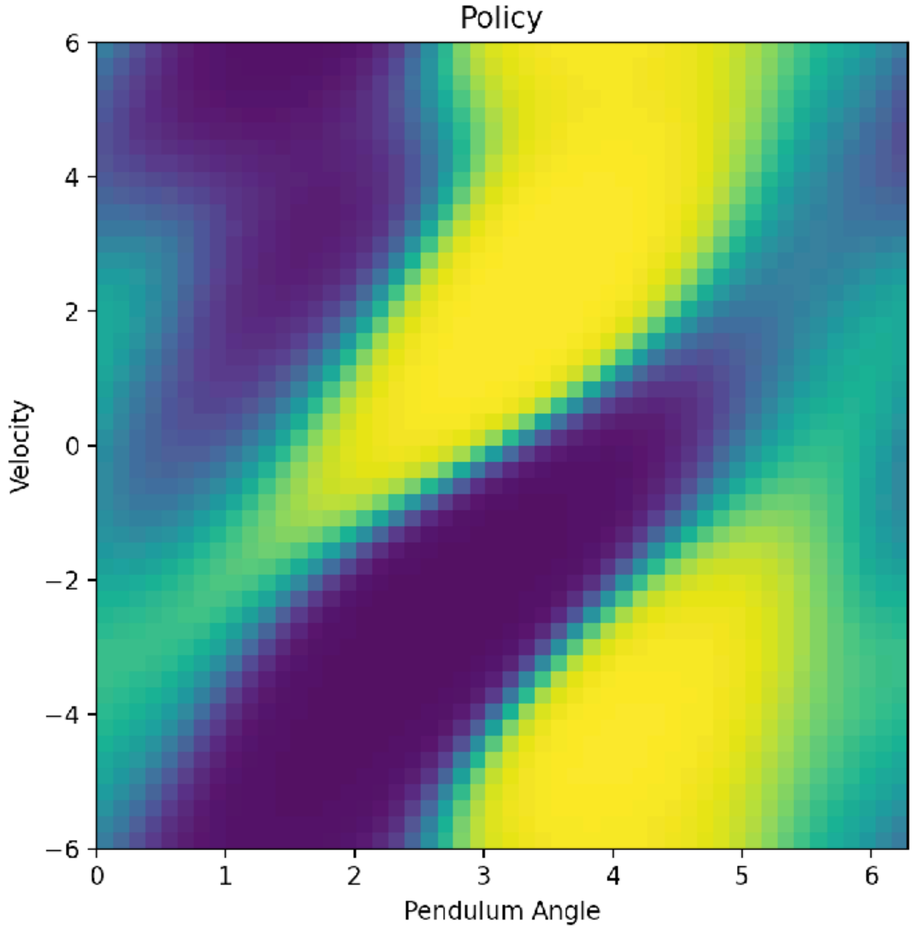
```python
class StochasticPolicyPeriodicFeatures:
    """
    This class implements a stochastic policy with linear sum of nonlinear features
    the features are periodic functions multiplied by a radial basis function of u
    """
    def __init__(self, controls, order = 2):
        """
        class constructor - controls is the array of control inputs, order is the order of the periodic basis
        """
        self.controls =  controls.copy()
        self.num_controls = len(self.controls)
        self.exp_basis = np.zeros([self.num_controls])
        self.order = order

        # the vector of basis functions
        self.basis_vector = np.zeros([2*self.num_controls*(self.order+1)**2])

        # the linear parameters to learn
        self.theta = np.zeros_like(self.basis_vector)

    def basis(self, x, u):
        """
        Returns the vector of basis functions evaluated at x,u
        """
        dx = x[0]
        dy = x[1]/6. * np.pi
        count = 0
        for c in self.controls:
            du = 1/(np.sqrt(2*np.pi*0.001)) * np.exp(-(u-c)**2/0.002)
            for j,k in itertools.product(range(self.order+1), range(self.order+1)):
                self.basis_vector[count] = du * np.cos(j*dx + k*dy)
                self.basis_vector[count+1] = du * np.sin(j*dx + k*dy)
                count += 2
        return self.basis_vector

    def get_distribution(self, x):
        """
        Computes pi(u|x) for all u
        returns an array of pi and an array of basis functions (row is the control index and column is the )
        """
        dist = np.zeros_like(self.controls)
        basis_fun = np.zeros([len(self.theta), len(self.controls)])
        for i,u in enumerate(self.controls):
            # this gives the basis function evaluated as (x,u)
            basis_fun[:,i] = self.basis(x,u)
            # dist gives exp(theta * basis_function)
            dist[i] = np.exp(self.theta.dot(basis_fun[:,i]))

        # we sum the exponentials
        sm = np.sum(dist)
        # dist is rescaled by the sum of exponentials (we now have a probability distribution)
        dist = dist/sm
        return dist, basis_fun

    def sample(self, x):
        """
        sample from the stochastic policy given x
        it returns the index of the control and its value
        """
        probs, basis = self.get_distribution(x)
        index = np.random.choice(len(self.controls), p=probs)
        return index, self.controls[index]
```

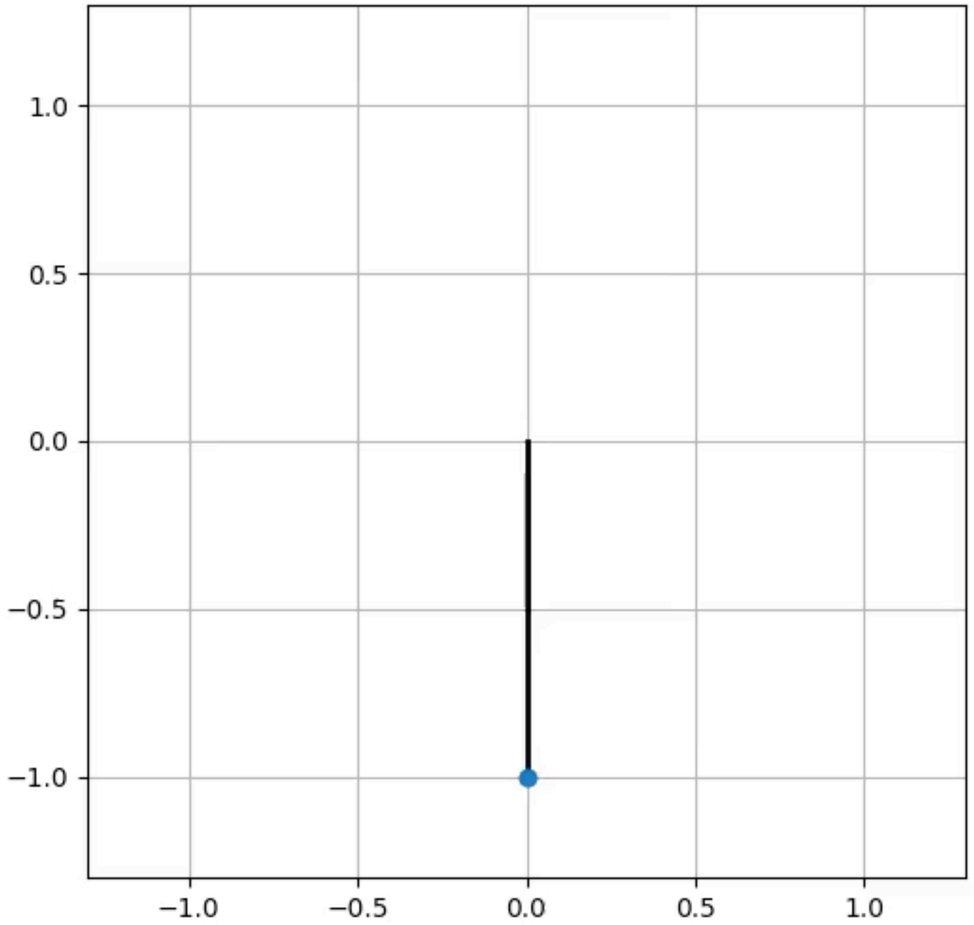
```python
class Reinforce:
    """
    An implementation of the reinforce algorithm (without baseline)
    """

    def __init__(self, model, cost, policy, discount_factor=0.99,
                 episode_length=100, policy_learning_rate = 0.000001, value_learning_rate = 0.01):

        self.model = model
        self.cost = cost

        self.policy = policy

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

    def iterate(self, num_iter=1):
        learning_progress = []

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj = np.zeros([self.episode_length+1, self.model.num_states])
            u_traj = np.zeros([self.episode_length, 1])
            u_index = np.zeros([self.episode_length], dtype=np.int)
            cost_traj = np.zeros([self.episode_length])

            for j in range(self.episode_length):
                u_index[j], u_traj[j,:] = self.policy.sample(x_traj[j,:])
                cost_traj[j] = self.cost(x_traj[j,:], u_traj[j,0])
                x_traj[j+1,:] = self.model.step(x_traj[j,:], u_traj[j,:])[:,0]

            # now we learn computing backwards
            G = 0.
            for j in range(self.episode_length-1, -1, -1):
                G = cost_traj[j] + self.discount_factor * G
                dist, basis = self.policy.get_distribution(x_traj[j,:])
                grad = basis[:,u_index[j]] - basis.dot(dist)
                self.policy.theta -= self.policy_learning_rate * (self.discount_factor**j) * G * grad

            learning_progress.append(G)

        return learning_progress
```

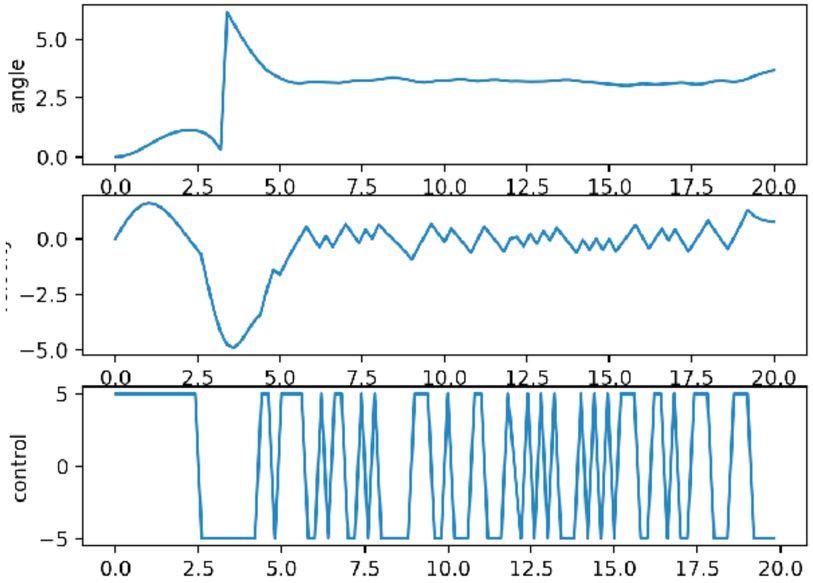
```
pendulum = Pendulum()
policy = StochasticPolicyPeriodicFeatures(controls = pendulum.controls, order = 2)
reinforce_nob = Reinforce(pendulum, cost, policy, value, episode_length=100, discount_factor=0.99,
                          policy_learning_rate = 0.0000001)
```

Learning rate 10e-7


Cost / episode
500
400
300
200
100
0
1000
2000
3000
4000
5000


Policy
Velocity
Pendulum Angle
6
4
2
0
−2
−4
−6
0
1
2
3
4
5
6


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


angle
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0.0
−2.5
−5.0
control
5
0
−5

REINFORCE with baseline

$$V(x, \theta_V) = \theta_V^T B(x)$$

$$b_{k,l,0}(\theta, \omega) = \cos(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

$$b_{k,l,1}(\theta, \omega) = \sin(k\theta + l\frac{\pi}{\omega_{max}}\omega)$$

```python
class ValueFunctionPeriodicFeatures:
    """
    This class implements a function approximator with linear sum of nonlinear features
    the features are periodic functions
    We use this to approximate the value function
    """

    def __init__(self, order = 2):
        """
        the class constructor - order is the order of the periodic basis
        """

        self.order = order
        self.basis_vector = np.zeros([2*(self.order+1)**2])

        # the parameters to learn
        self.theta = np.zeros_like(self.basis_vector)

    def basis(self, x):
        """
        Returns the vector of basis functions evaluated at x
        """
        dx = x[0]
        dy = x[1]/6. * np.pi
        count = 0
        for j,k in itertools.product(range(self.order+1), range(self.order+1)):
            self.basis_vector[count] = np.cos(j*dx + k*dy)
            self.basis_vector[count+1] = np.sin(j*dx + k*dy)
            count += 2
        return self.basis_vector

    def getValue(self, x):
        """
        returns the value at x and the basis functions evaluated at x
        """
        bs = self.basis(x)
        return bs.dot(self.theta), bs
```

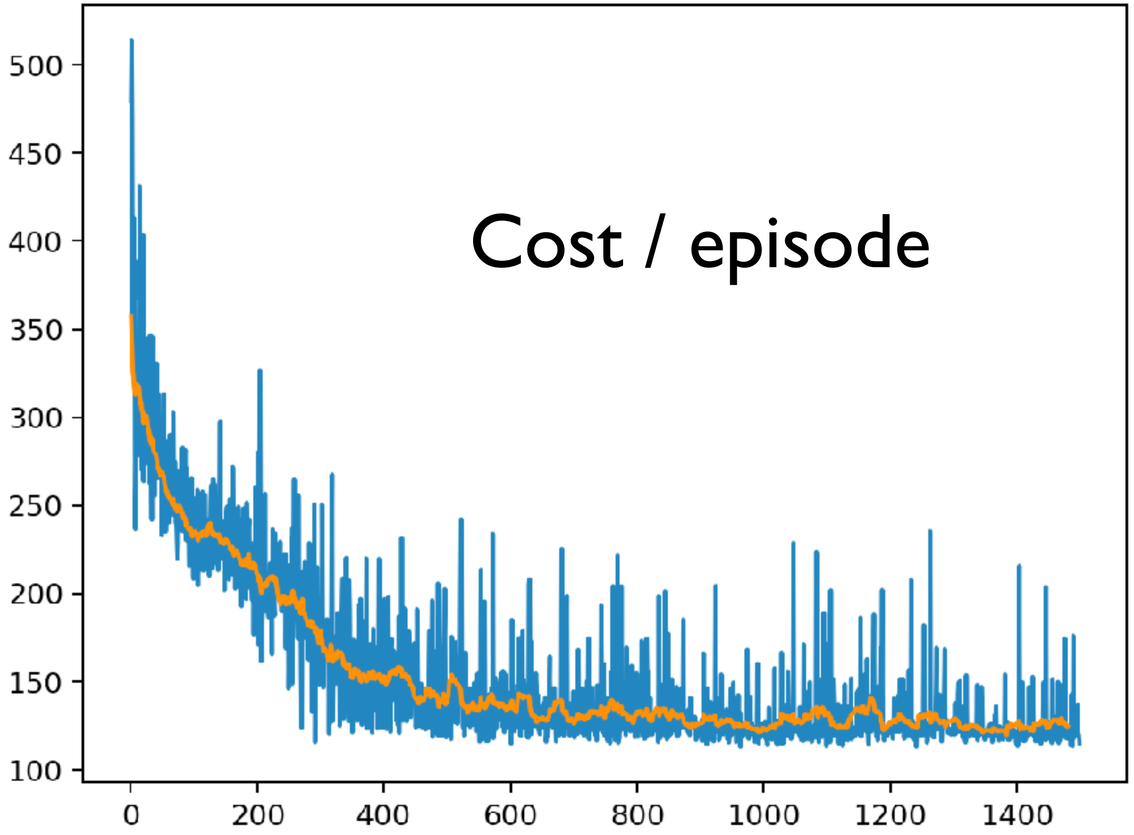
```python
class ReinforceBaseline:
    """
    An implementation of the reinforce algorithm (with or without baseline)
    """

    def __init__(self, model, cost, policy, valuefunction, discount_factor=0.99,
                 episode_length=100, policy_learning_rate = 0.000001, value_learning_rate = 0.01):

        self.model = model
        self.cost = cost

        self.policy = policy
        self.value = valuefunction

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

    def iterate(self, num_iter=1):
        learning_progress = []

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj = np.zeros([self.episode_length+1, self.model.num_states])
            u_traj = np.zeros([self.episode_length, 1])
            u_index = np.zeros([self.episode_length], dtype=np.int)
            cost_traj = np.zeros([self.episode_length])

            for j in range(self.episode_length):
                u_index[j], u_traj[j,:] = self.policy.sample(x_traj[j,:])
                cost_traj[j] = self.cost(x_traj[j,:], u_traj[j,0])
                x_traj[j+1,:] = self.model.step(x_traj[j,:], u_traj[j,:])[:,0]

            # now we learn computing backwards
            G = 0.
            for j in range(self.episode_length-1, -1, -1):
                G = cost_traj[j] + self.discount_factor * G
                dist, basis = self.policy.get_distribution(x_traj[j,:])
                grad = basis[:,u_index[j]] - basis.dot(dist)
                value, grad_value = self.value.getValue(x_traj[j,:])
                delta = (self.discount_factor**j) * (G - value)
                self.value.theta += self.value_learning_rate * delta * grad_value
                self.policy.theta -= self.policy_learning_rate * delta * grad

            learning_progress.append(G)

        return learning_progress
```

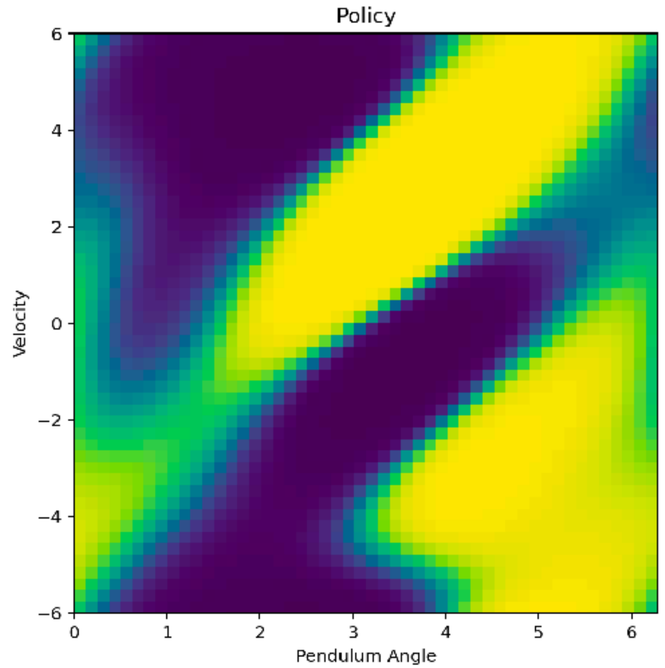
REINFORCE with baseline

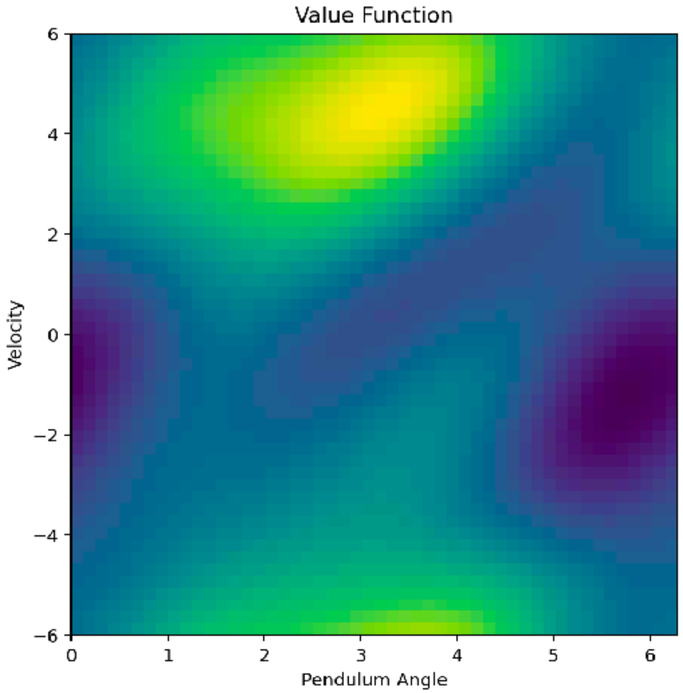
```
pendulum = Pendulum()
policy = StochasticPolicyPeriodicFeatures(controls = pendulum.controls, order = 2)
value = ValueFunctionPeriodicFeatures(order = 2)
reinforce_withb = Reinforce(pendulum, cost, policy, value, episode_length=100, discount_factor=0.99,
                      policy_learning_rate = 0.000001, value_learning_rate = 0.01)
```

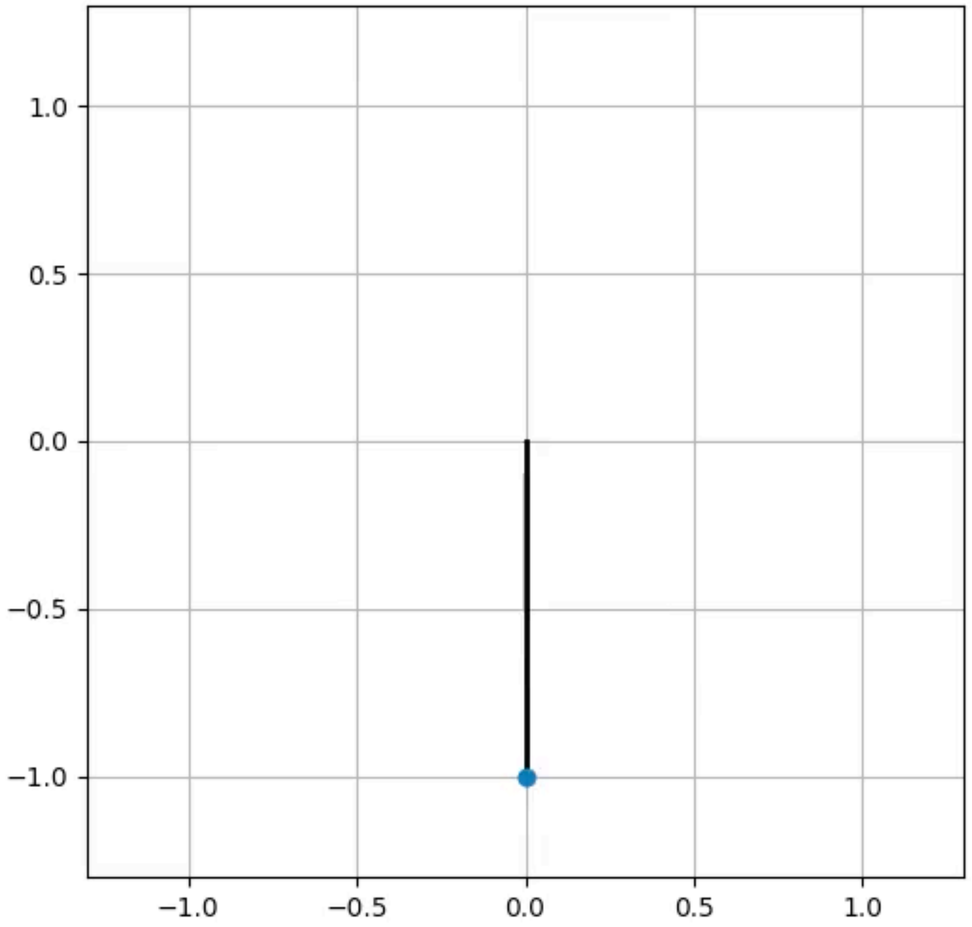
Learning rate 10e-6

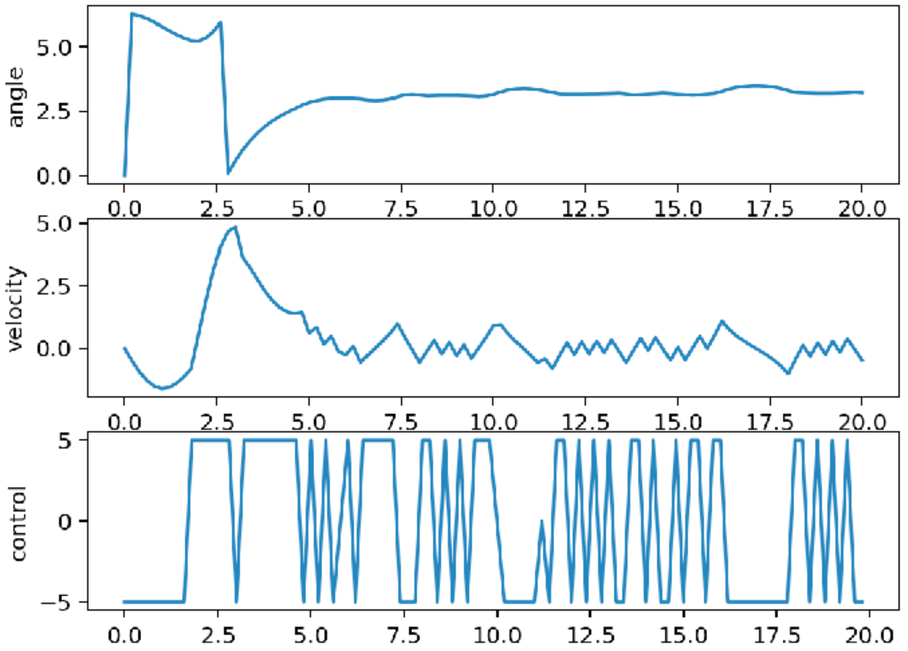
REINFORCE with baseline








1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


angle
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
velocity
5.0
2.5
0.0
control
5
0
−5

Deep Deterministic Policy Gradient

[Lillicrap et al., ICML, 2016]

The deterministic policy gradient can then be used to learn deterministic policies, which is convenient to work with continuous action spaces encoded as neural networks

$$\nabla J(\theta) = \mathbb{E}_{\pi}\left[\frac{\partial}{\partial\theta}\pi(x_t,\theta)\cdot\frac{\partial}{\partial a}Q_{\pi}(x_t,a)_{|a=\pi(x_t,\theta)}\right]$$

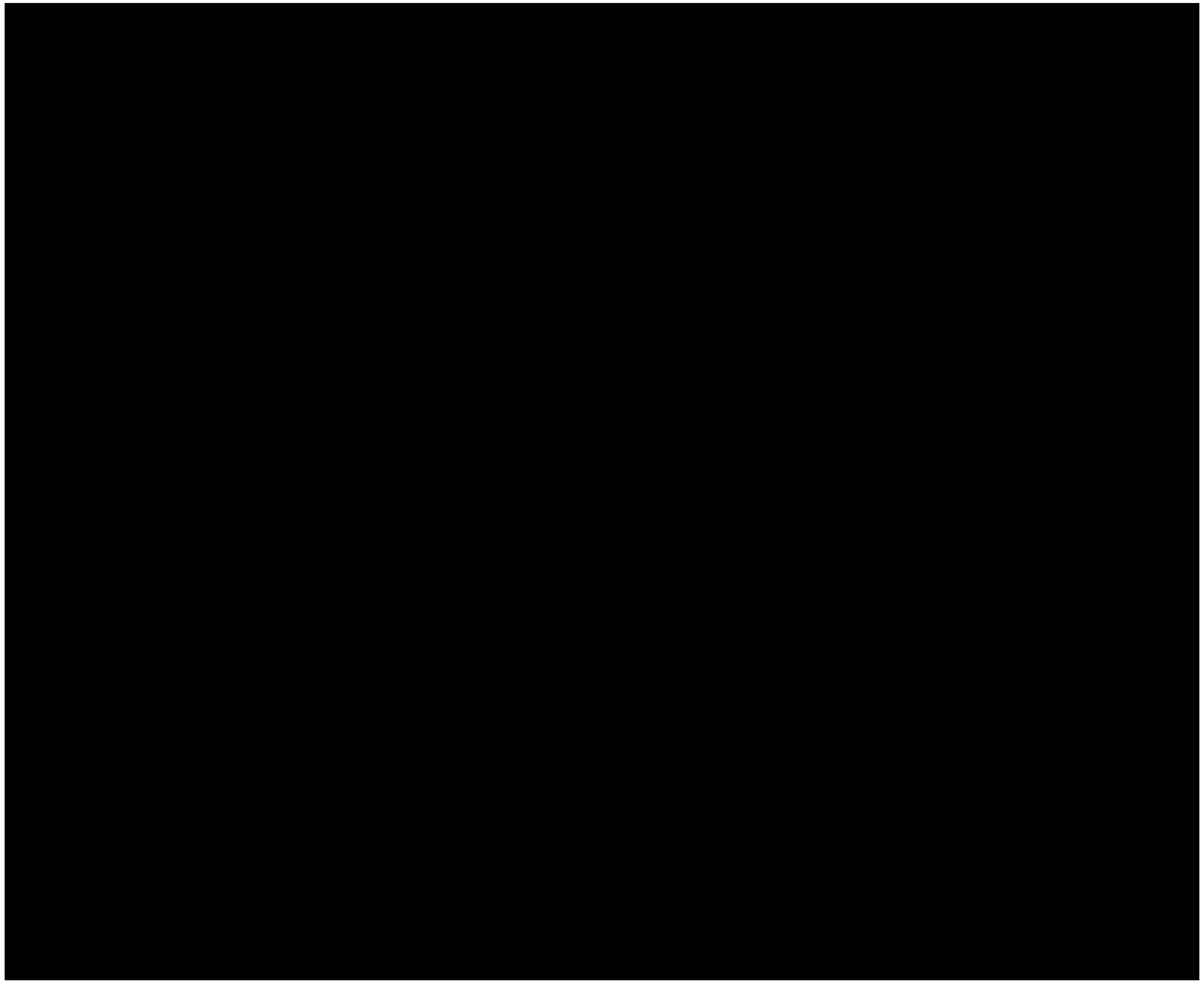
$$\simeq \frac{1}{N}\sum_{i=0}^{N-1}\left[\frac{\partial}{\partial\theta}\pi(x_i,\theta)\cdot\frac{\partial}{\partial a}Q_{\pi}(x_i,a)_{|a=\pi(x_i)}\right]$$

i.e. it is possible to get an approximation of the policy gradient by sampling N different states

We can now derive an actor-critic algorithm using deterministic policies and continuous action spaces

DDPG

[Lillicrap et al., ICML, 2016]



Policy gradient methods

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} R(\tau)\nabla_\theta \log \pi(u_n|x_n,\theta)\right] \quad R(\tau) = \sum_{n=0}^{N} \alpha^n g(x_n,u_n)$$

REINFORCE $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} G_n\nabla_\theta \log \pi(u_n|x_n,\theta)\right] \quad G_n = \sum_{k=n}^{N} \alpha^k g(x_k,u_k)$$

REINFORCE with baseline $$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (G_n - V(x_n))\nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Actor-critic

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} (g(x_n,u_n) + \alpha V(x_{n+1}) - V(x_n))\nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

Policy gradient methods

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

$$\Psi_n = \sum_{k=0}^{N} \alpha^k g(x_k,u_k)$$

$$\Psi_n = g(x_n,u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k,u_k)$$

$$\Psi_n = Q_\pi(x_n,u_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k,u_k) - b(x_n)$$

$$\Psi_n = A_n = Q(x_n,u_n) - V(x_n)$$

Proximal policy optimization (PPO)

Explicit gradient descent

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \nabla_\theta \log \pi(u_n|x_n,\theta)\right]$$

$$\rightarrow \frac{1}{\pi} \cdot \nabla\pi$$

Equivalent to

$$\min_\theta \mathbb{E}\left[\Psi_n \log \pi(u_n|x_n,\theta)\right]$$

Use the gradient of log to re-arrange the formula

$$\min_\theta \mathbb{E}\left[A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}\right] \Leftarrow$$

$\rightarrow$ fixed

Proximal policy optimization (PPO)

"Clip" the total scaling

$$\min_{\theta} \mathbb{E}\left[\min\left(A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, clip\left(\frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, 1-\epsilon, 1+\epsilon\right) A_n\right)\right]$$

Run a lot of episodes in parallel (in simulation) to improve the estimation of the gradient and expectation

Proximal policy optimization (PPO)

Evaluating the advantage An

$$\delta_n = g(x_n, u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$A_n = \sum_{k=n}^{N} (\alpha\lambda)^{k-n} \delta_k$$

Proximal policy optimization (PPO)

While not converged

For actors 1, … , P do

Run the policy in the simulator for N time steps

Collect state/action transition

Compute advantage estimates $A_n = \sum_{k=n}^{N} (\alpha\lambda)^{k-n} \delta_k$

End for

Do gradient descent on the cost

$$\min_{\theta} \mathbb{E}\left[\min\left(A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, clip\left(\frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, 1-\epsilon, 1+\epsilon\right) A_n\right)\right]$$

Update the value function estimates (e.g. TD-learning)

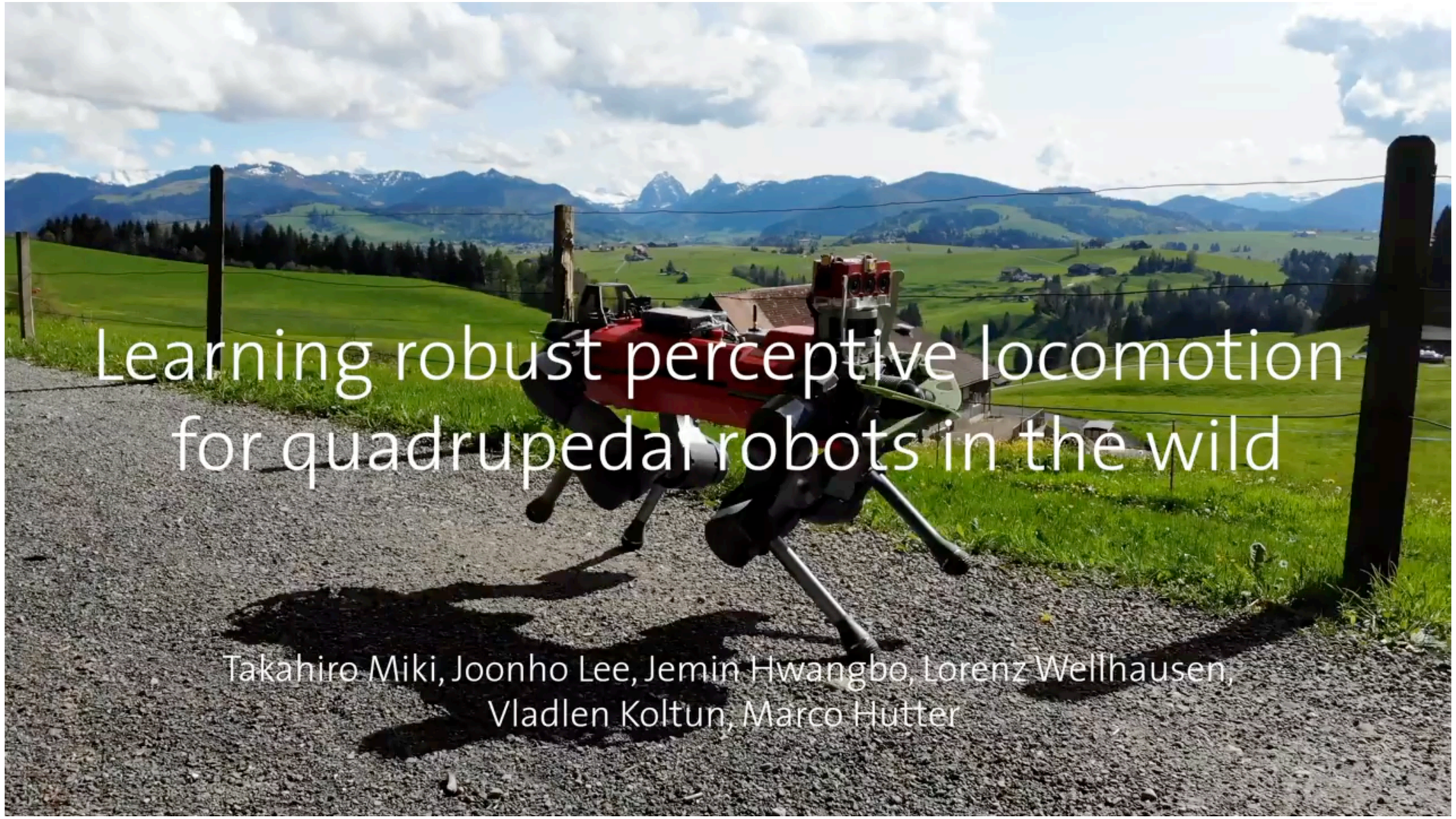
Proximal policy optimization (PPO)

Lots of heuristics but it works rather well in practice
Parallelization and clipping help a lot to get good gradient steps

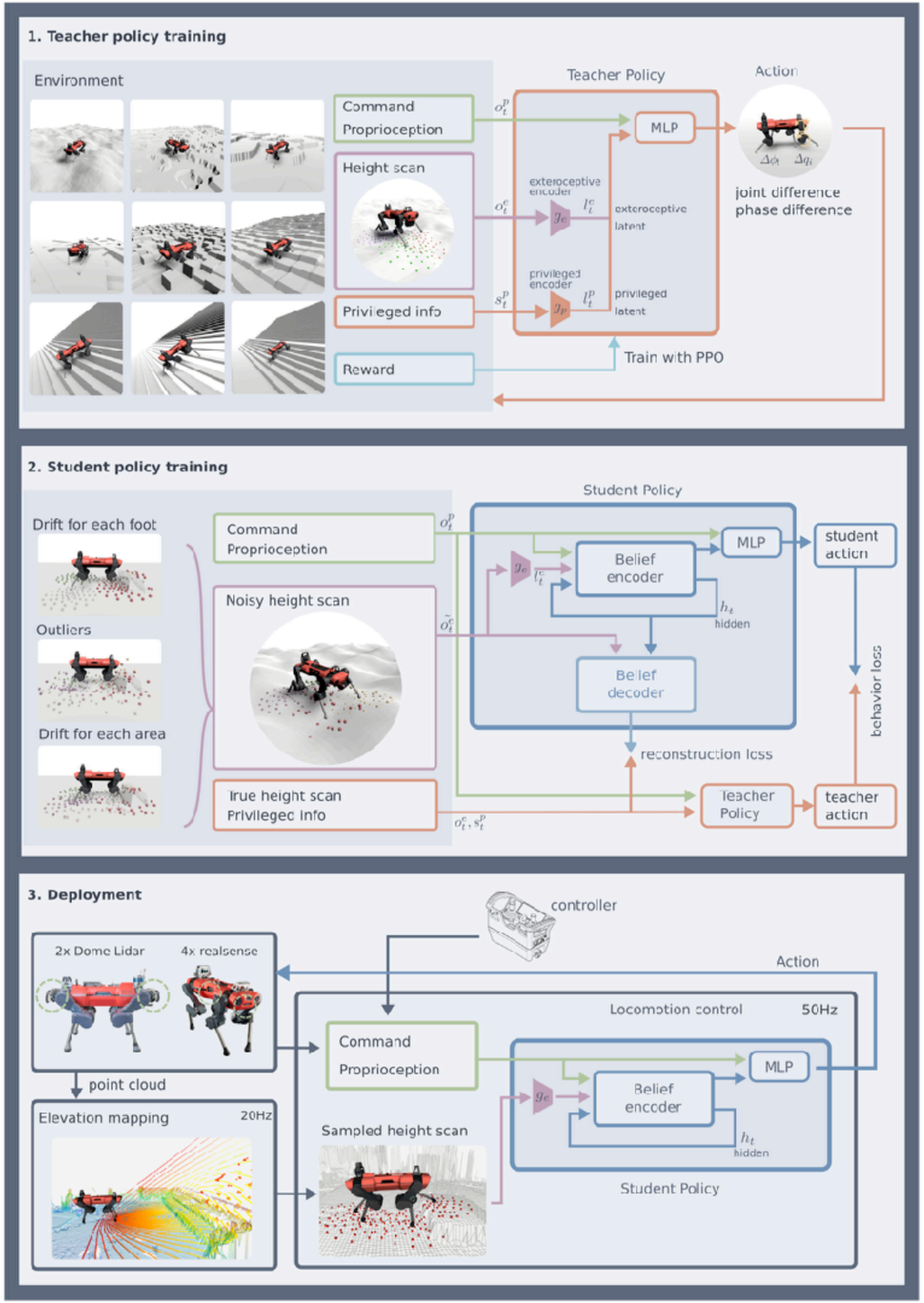
PPO is considered “state of the art” for deep RL - used a lot in robotics

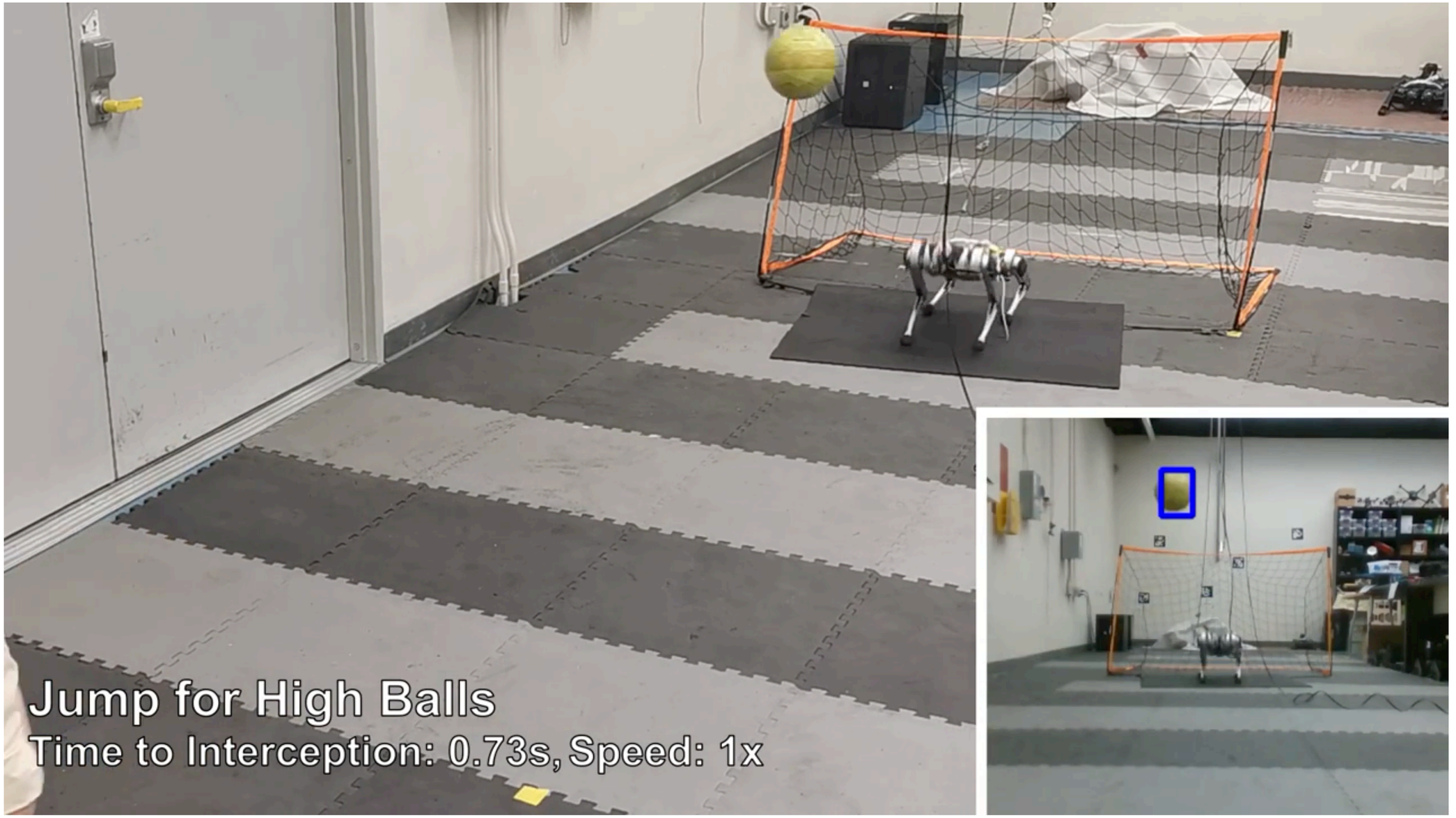
BUT it is rarely used as is - a lot of engineering around is necessary

Learning various behaviors

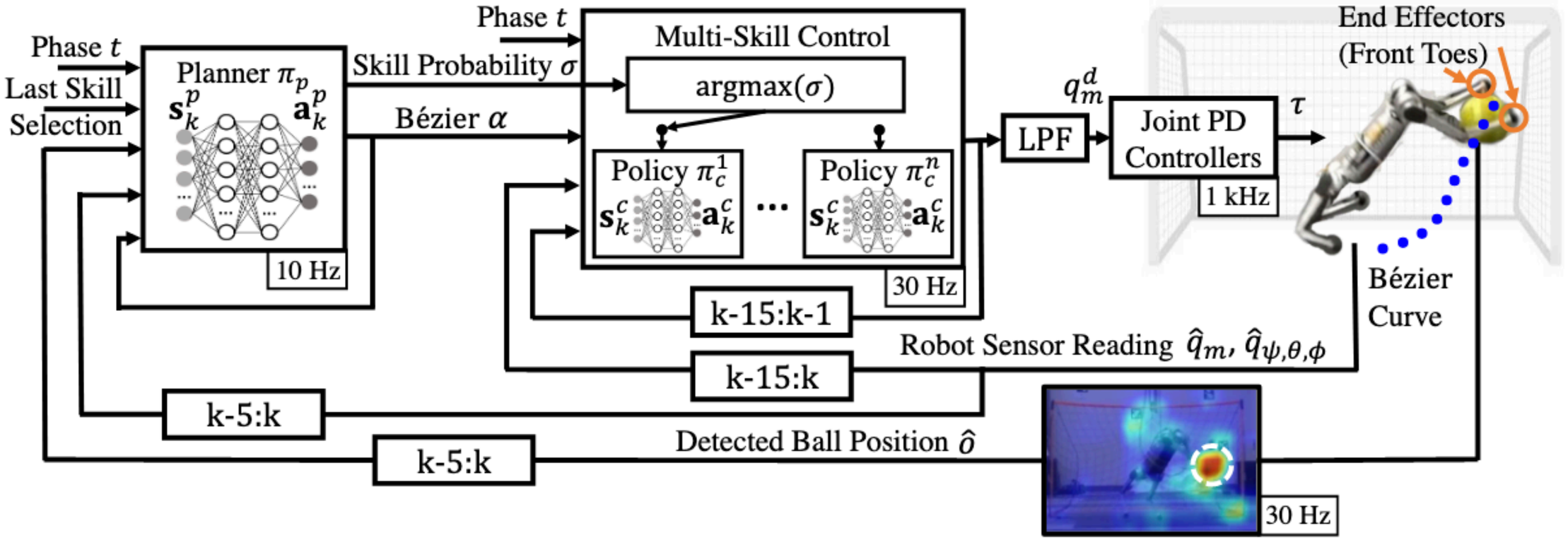


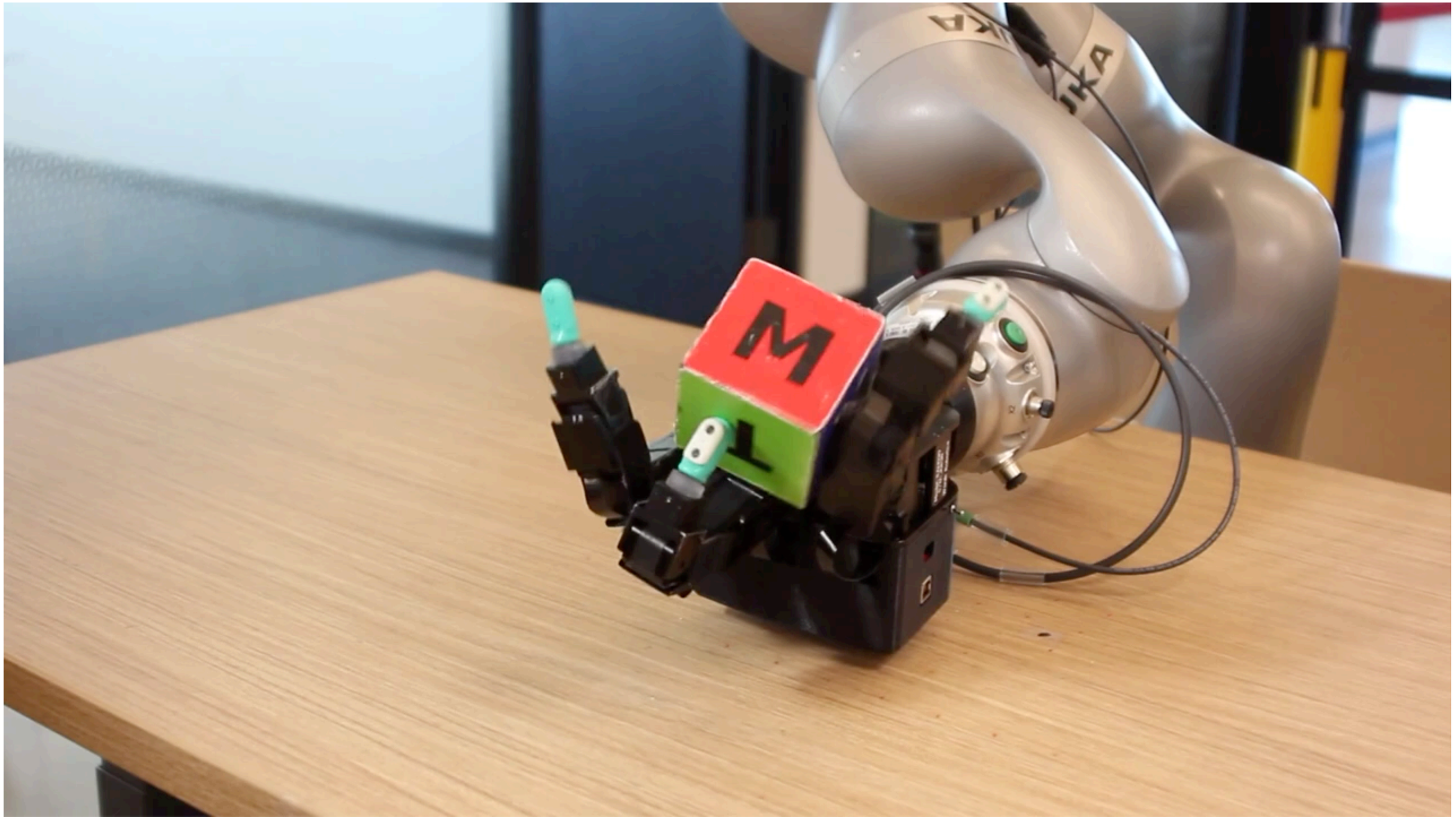
[Miki et al. Science 2022]


1. Teacher policy training
Environment
Command
Proprioception
Height scan
Privileged info
Reward
Teacher Policy
MLP
exteroceptive encoder
exteroceptive latent
privileged encoder
privileged latent
Action
joint difference
phase difference
Train with PPO
2. Student policy training
Drift for each foot
Outliers
Drift for each area
Command
Proprioception
Noisy height scan
True height scan
Privileged info
Student Policy
Belief encoder
MLP
student action
hidden
Belief decoder
reconstruction loss
behavior loss
Teacher Policy
teacher action
3. Deployment
controller
2x Dome Lidar
4x realsense
point cloud
Elevation mapping
20Hz
Action
Locomotion control
50Hz
Command
Proprioception
Sampled height scan
Belief encoder
MLP
hidden
Student Policy

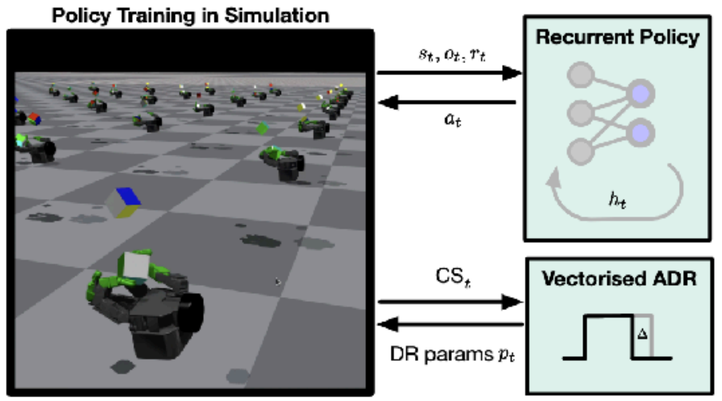


[Huang et al. 2022]

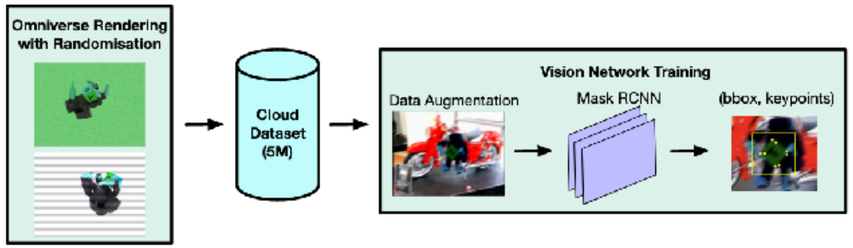

Phase t
Last Skill Selection
Planner π_p
$\mathbf{s}_k^p$
$\mathbf{a}_k^p$
10 Hz
Skill Probability σ
Bézier α
Phase t
Multi-Skill Control
argmax(σ)
Policy π_c^1
$\mathbf{s}_k^c$
$\mathbf{a}_k^c$
Policy π_c^n
30 Hz
k-15:k-1
k-15:k
k-5:k
k-5:k
q_m^d
LPF
Joint PD Controllers
1 kHz
τ
End Effectors (Front Toes)
Bézier Curve
Robot Sensor Reading $\hat{q}_m$, $\hat{q}_{\psi,\theta,\phi}$
Detected Ball Position $\hat{o}$
30 Hz



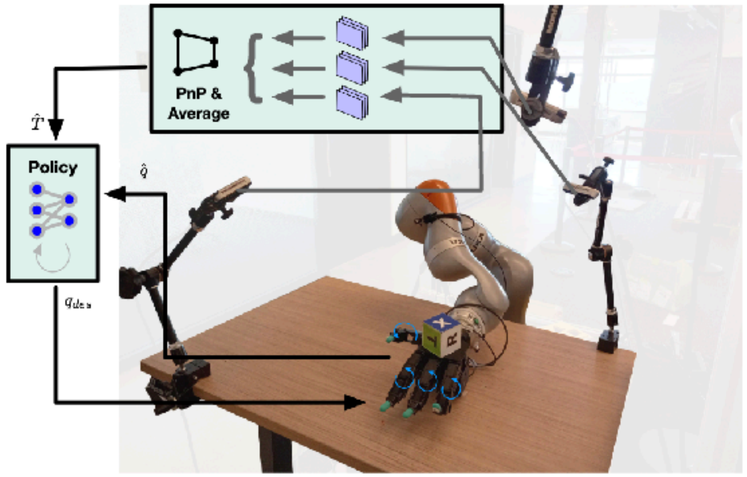
[Handa et al. 2022]



(a) Policy training.



(b) Vision data generation and training pipeline.



(c) Functioning in the real world.

Model-based reinforcement learning

We can learn:

- a value function
- a policy
- a model?

Model-based RL

=> learn a model + do optimal control with the model

Model-based reinforcement learning

How do we learn a model?

If the dynamics is linear

$$x_{n+1} = Ax_n + Bu_n$$

we can find A and B from data
=> regression problem

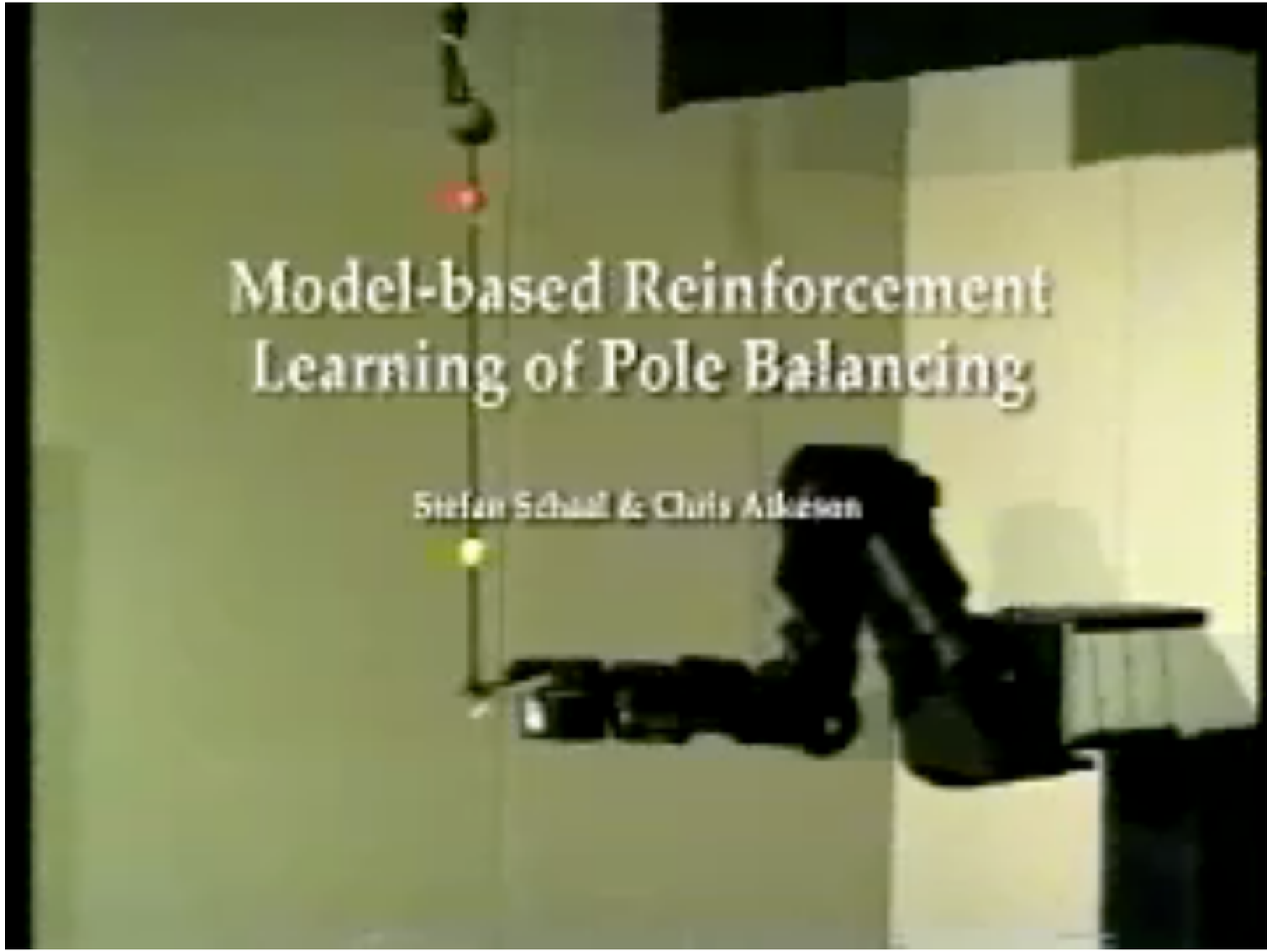
Nonlinear models

1. Use a function approximator for nonlinear functions that is "linearization" friendly
   (e.g. locally weighted regression, Schaal et al. 1997 or Gaussian Processes, Deisenroth et al. 2011)
   => good to do LQR and related, exploit linearity

2. Learn a nonlinear model
   => typically linearization is problematic - might need other techniques to solve OC problems (e.g. cross-entropy methods)

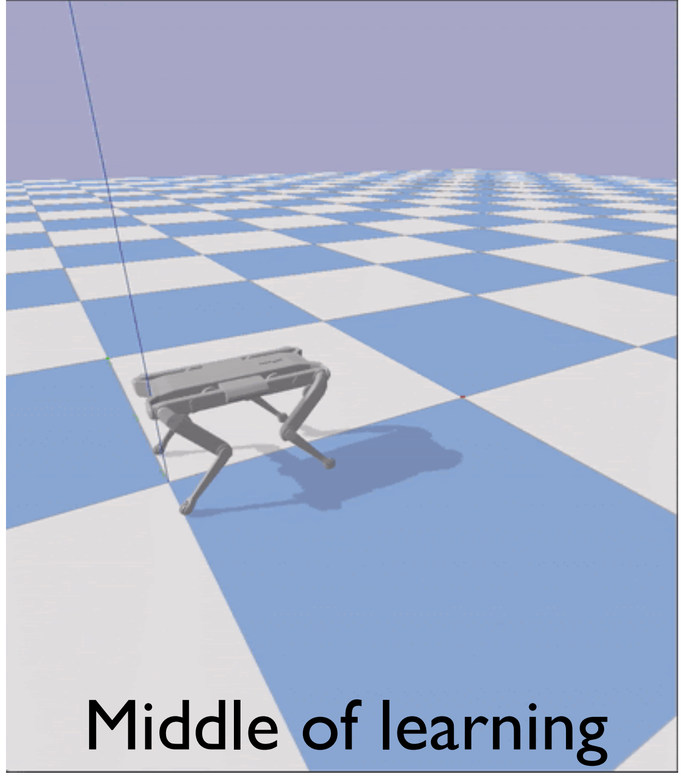
Model-based reinforcement learning

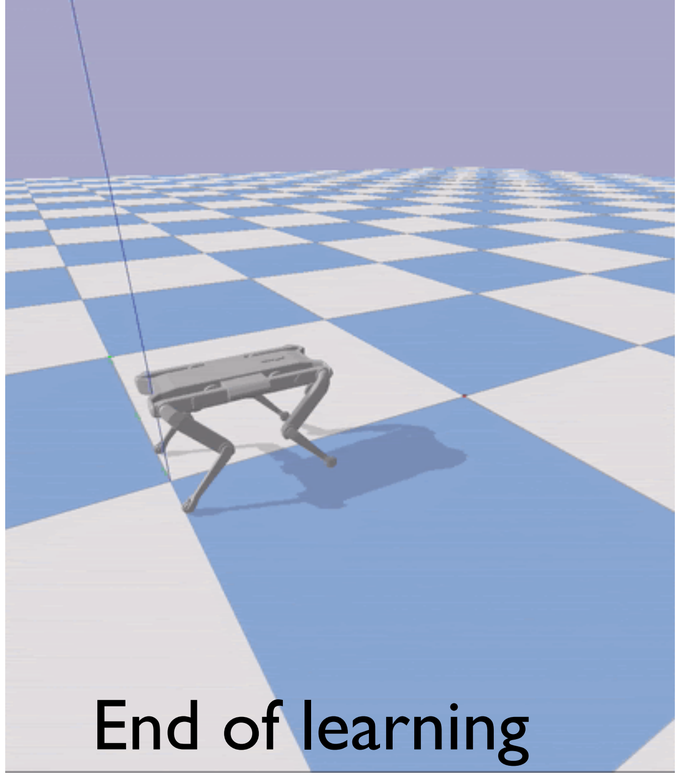
[Schaal and Atkeson ~1995]

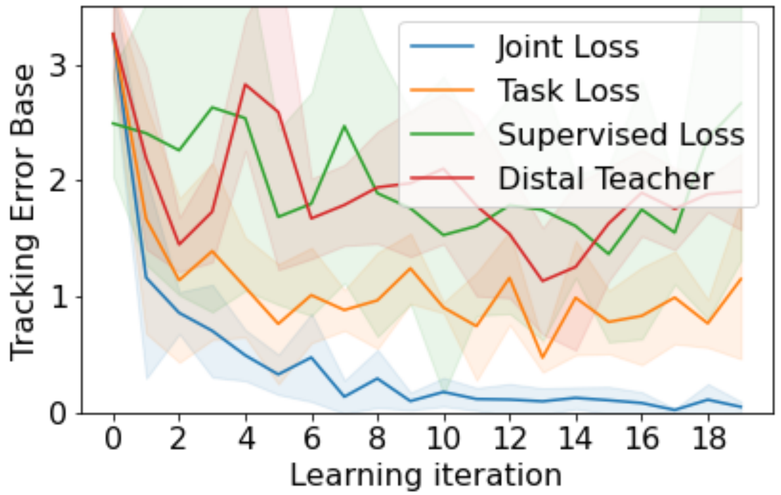


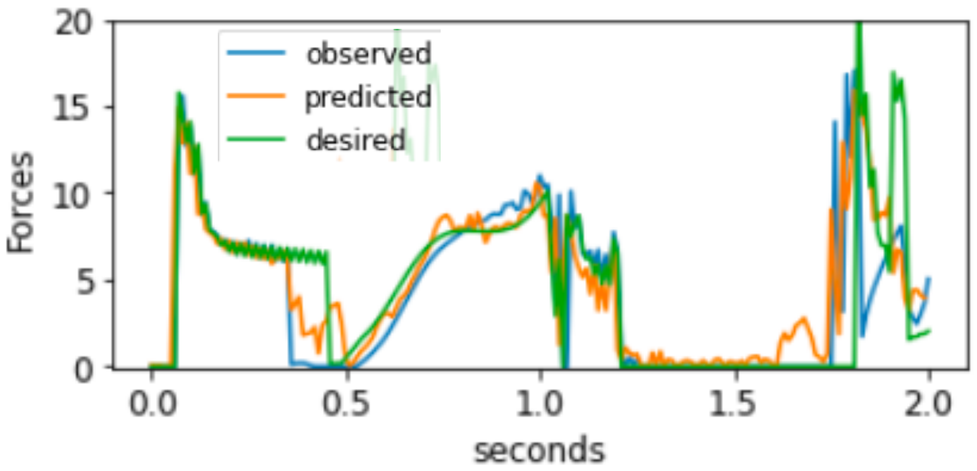
Model-based reinforcement learning

[Schaal and Atkeson ~1995]



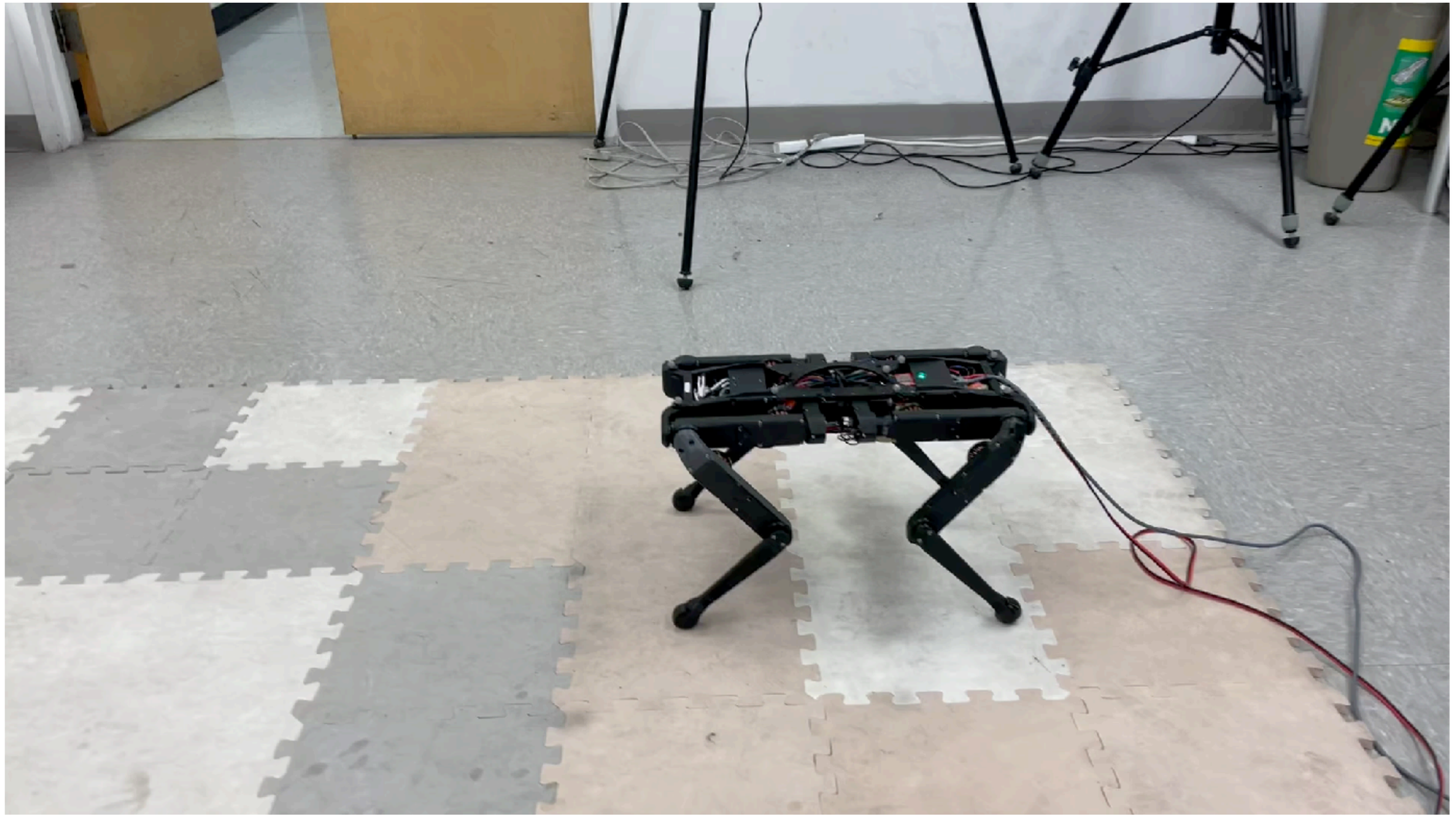






[Bechtle et al., ICRA 2021]

Efficient learning of forward models and policies



[Bechtle et al., ICRA 2021]

Advantages of model-based RL

- It tends to be sample-efficient
  => can be used on robots
- Can be used to solve other tasks (i.e. we can change the cost function and keep the model)
- It is easy to compute a locally optimal policy (trajectory optimization) while adding constraints

Issues / drawbacks

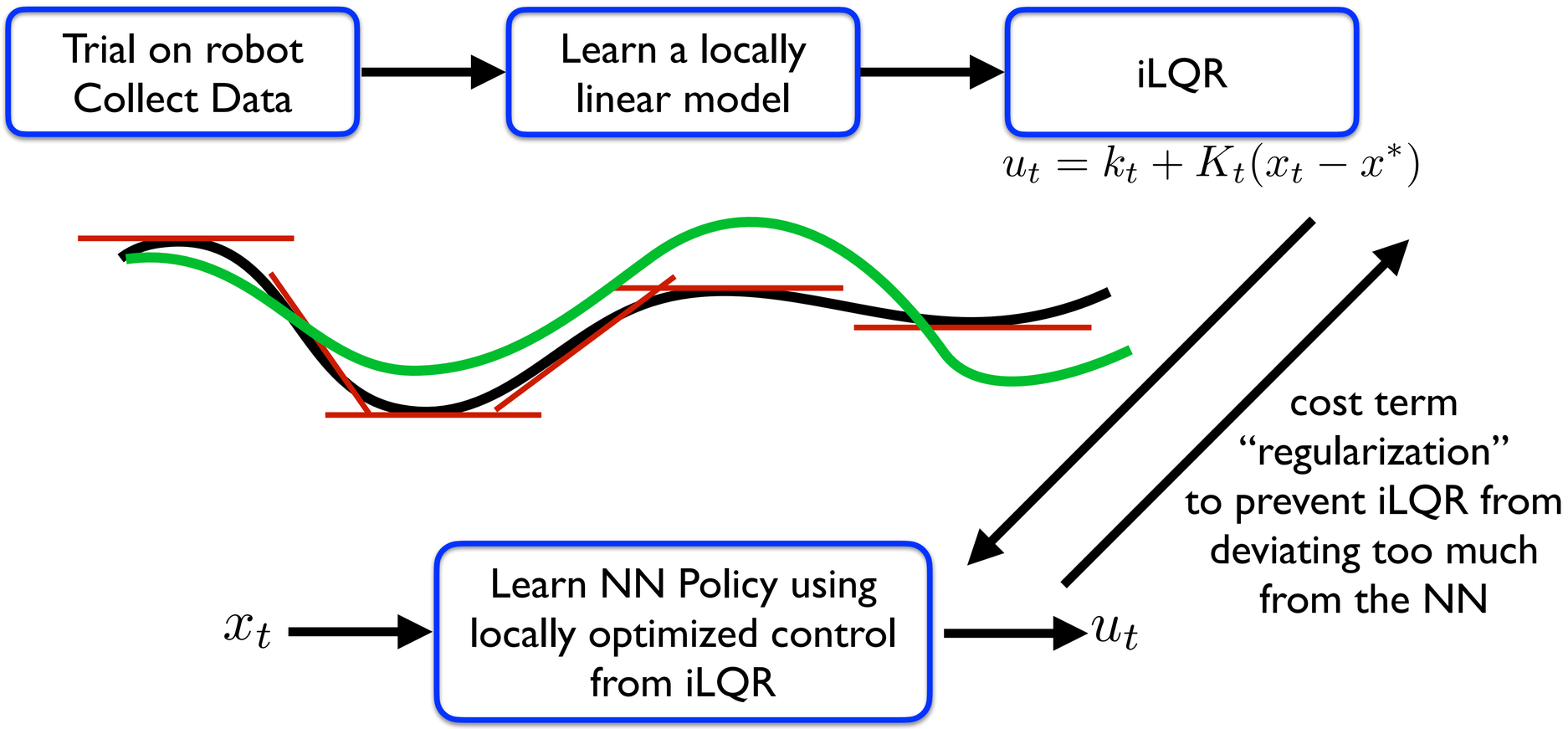
- generating enough data to learn the model
- what controller do we use to generate the first samples?
- difficulty to learn models capable to predict long in the future (nonlinear dynamics is tricky)
- mapping from model to policy might not work
- still need to solve an OC problem all the time

Can we get a policy from optimal control trajectories?

Idea: use control trajectories (e.g. coming from iLQR) to learn a “global” policy using a function approximation (e.g. a neural network)

Guided Policy Search

[Levine et al. 2015]



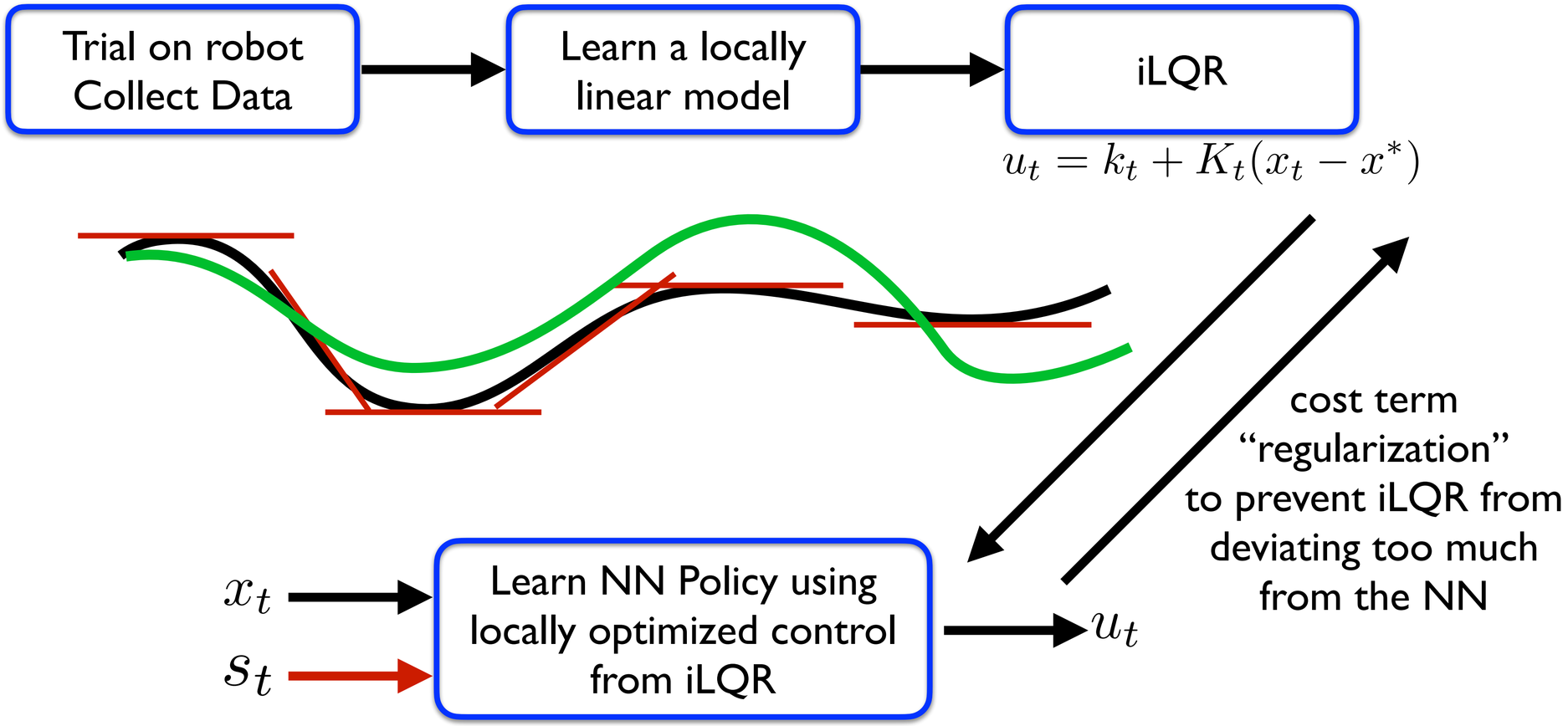
Model-based reinforcement learning

[Levine et al. 2015]

Guided Policy Search

End-to-end training of visuomotor policies

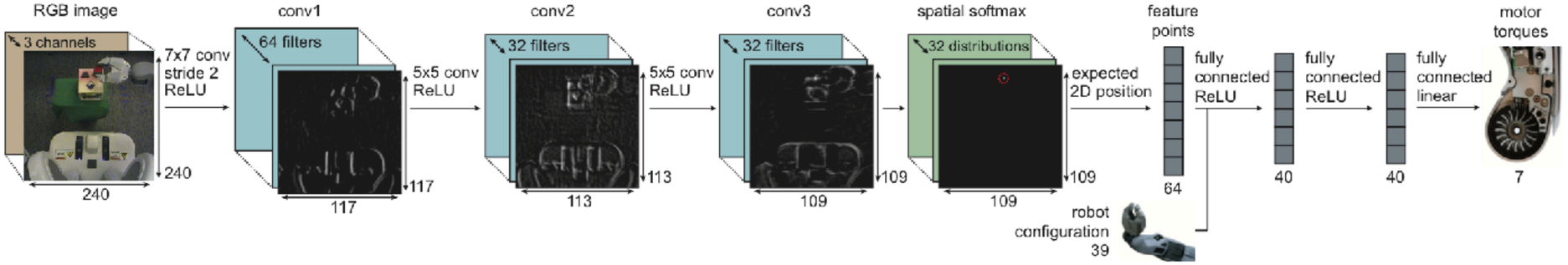
[Levine et al. 2016]



other sensors not part of
the model (e.g. vision)

End-to-end training of visuomotor policies

[Levine et al. 2016]



iLQR with state of the robot / task
(object location + robot positions)

Neural network input: visual information + robot position

Train network so its output is the same
as the output of the iLQR

End-to-end training of visuomotor policies

[Levine et al. 2016]

Bootstrapping RL with demonstrations

Apprenticeship learning

[Abbeel, 2010]

Apprenticeship learning

[Abbeel, 2010]

Apprenticeship learning

[Abbeel, 2010]