MPCRL with `leap-c`
– Differentiable MPC Layers for PyTorch (and JAX) –

Jasper Hoffmann

University of Freiburg

Predict, Control, Learn: Combining Model Predictive Control and Reinforcement Learning
Tutorial Session at ECC 2026, Reykjavík, July 2026

universität freiburg

NTNU | Norwegian University of Science and Technology

University of Zurich[UZH]

ROBOTICS & PERCEPTION GROUP

Tutorial Roadmap

	Talk	*Presenter*	*Duration*
✓	Markov Decision Processes – a unifying perspective on MPC and RL	Jasper Hoffmann	20min
✓	Synthesis of Model Predictive Control and Reinforcement Learning	Rudolf Reiter	30min
✓	Reinforcement Learning over MPC: Why does it work?	Dirk Reinhardt	30min
✓	Differentiable NMPC with acados	Katrin Baumgärtner	20min
➤	**MPCRL with `leap-c`**	**Jasper Hoffmann**	**20min**

You are now in the final talk of the tutorial session.

Tutorial webpage for slides and more!

What is `leap-c`?

Why `leap-c`?

Differentiable `acados`

- `acados` solves structured nonlinear OCPs fast
- solution and adjoint sensitivities are available
- exact sensitivity solver can be separated from nominal solver

$\Rightarrow$

Learning workflows need more

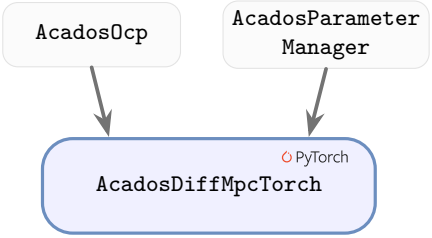
- tensor inputs and automatic differentiation
- convenient batching, warmstarting and initialization of solvers
- named parameters that receive gradients

$$(x_0, \texttt{params}) \mapsto (u_0^\star, X^\star, U^\star, V^{\mathrm{MPC}})$$

Turn solver sensitivities into learning infrastructure.

Outline

Building a `leap-c` Layer

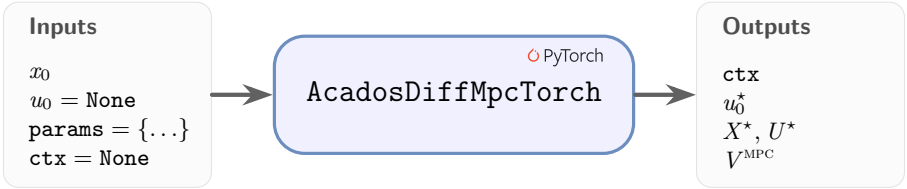


- `ocp`: MPC problem specification
- `manager`: named runtime parameters, e.g. `xref`, `mass`
- `diff_mpc`: the differentiable layer

Python Code 🐍

```python
# define OCP
ocp = AcadosOcp()
ocp.model = model
ocp.dims.N = N
# ... cost, constraints, solver options

# define parameters
manager = AcadosParameterManager(N_horizon=N)
xref = manager.register_parameter(
    "xref", default=xref_default, differentiable=True
)
mass = manager.register_parameter(
    "mass", default=mass_default, differentiable=True
)

# build layer
diff_mpc = AcadosDiffMpcTorch(ocp, manager)
```

Forward Pass – Default MPC Solve

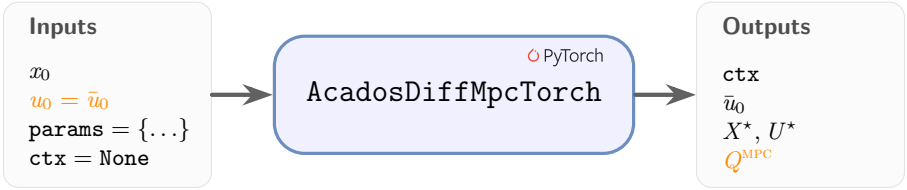


Python Code 🐍

```python
# xref and mass are torch tensors
params = {"xref": xref, "mass": mass}

ctx, u0, X, U, value = diff_mpc(
    x0, u0=None, params=params, ctx=None
)
```

- returns optimal first action $u_0^\star$ and trajectory
- `value` is $V^{\text{MPC}}(x_0)$
- `ctx` stores solver state internally

Forward Pass – Querying Q-values

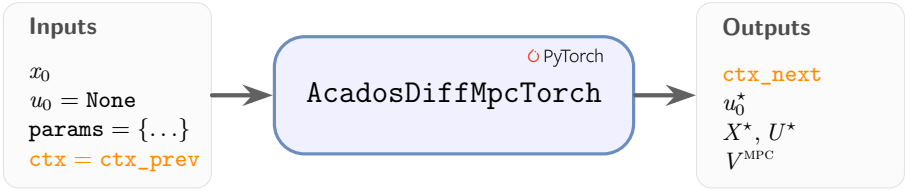


Python Code 🐍

```
# pass a candidate first action
ctx, u0, X, U, q_value = diff_mpc(
    x0, u0=u0_query, params=params, ctx=None
)
```

- `u0=None`: solve for $u_0^\star$, get V^{MPC}
- `u0=u0_query`: fix first action, get $Q^{\mathrm{MPC}}(x_0, u_0)$
- useful for critic training in MPC-RL

Forward Pass – Warmstarting with `ctx`

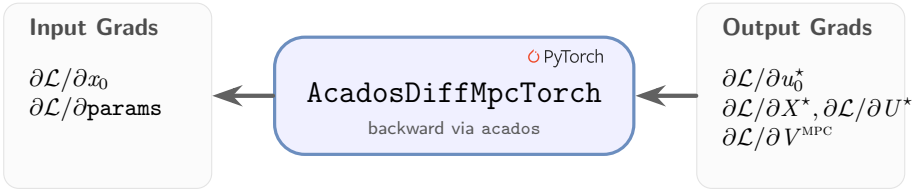


Python Code 🐍

```python
# first call
ctx, u0, X, U, value = diff_mpc(x0, params=params)

# reuse solution as warmstart
ctx, u0, X, U, value = diff_mpc(
    x0_next, params=params, ctx=ctx
)
```

- reuse previous solution as warmstart
- faster convergence in closed-loop rollouts
- same `ctx` caches sensitivity data

Backward Pass – Autograd Through MPC

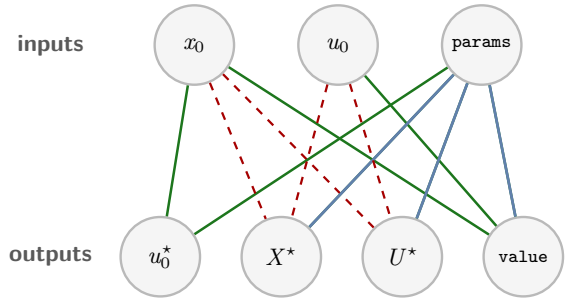


Python Code 🐍

```
xref.requires_grad_(True)
mass.requires_grad_(True)

_, _, _, _, value = diff_mpc(
    x0, params={"xref": xref, "mass": mass}
)

loss = (value - target_value).square().mean()
loss.backward()

xref.grad
mass.grad
```

- PyTorch sends upstream gradients from the loss
- acados adjoints compute vector-Jacobian products
- `xref.grad`, `mass.grad` are populated

Retrieving Sensitivities the PyTorch Way



Legend

- adjoint/VJP
- explicit Jacobian
- possible extension

Clean PyTorch interface: use `backward()` for scalar losses and `jacobian(...)` for selected sensitivities.

Python Code 🐍

```python
def policy(xref):
    params = {"xref": xref, "mass": mass}
    return diff_mpc(x0, params=params)[1]

# similarly query dV/dxref, dQ/du0, ...
du0_dxref = torch.autograd.functional.jacobian(policy, xref)
```

Outline

leap-c: General Overview

leap-c

Core differentiable MPC layer around acados.

- PyTorch today
- JAX backend soon
- solver-backed gradients

leap-c-lab

Editable examples, tutorials, and benchmark-style workflows.

- non-interactive notebooks on Hugging Face
- Docker images for local Marimo editing

Research Code Bases

Research repositories built on top of leap-c.

- leap-c-rl: collection of RL algorithms

Notebooks:

- Non-interactive notebooks: Hugging Face
- Interactive notebooks: Locally via Docker

Hugging Face demo

Try it: huggingface.co/spaces/leap-c/leap-c

Where We Are Testing `leap-c`

Real-world CartPole

MPC state-dependent tuning for swing-up/stabilization on hardware in about 6 minutes

Wave Energy

MPCRL showed promising initial results

HEMS

ongoing work on home energy management systems

some current and future application studies

Outline

Future Steps

`leap-c`:

- JAX integration implemented, tested in the next weeks

PyTorch

`leap-c-lab`:

- Extend examples to soccer robots, heatpumps etc.
- Provide a selection of problems for benchmarking

become a contributor for `leap-c`!

Contributors 12

Tutorial Roadmap

	Talk	*Presenter*	*Duration*
✓	Markov Decision Processes – a unifying perspective on MPC and RL	Jasper Hoffmann	20min
✓	Synthesis of Model Predictive Control and Reinforcement Learning	Rudolf Reiter	30min
✓	Reinforcement Learning over MPC: Why does it work?	Dirk Reinhardt	30min
✓	Differentiable NMPC with acados	Katrin Baumgärtner	20min
➤	**MPCRL with `leap-c`**	**Jasper Hoffmann**	**20min**

This concludes the tutorial.

Tutorial webpage for slides and more!

ECC26 TUTORIAL

Thank You! Any Questions?

Tutorial Webpage

Slides of this tutorial and links to more material.

tutorial webpage

School

Fall school on MPC and RL, Freiburg in Germany.

- 7–16 October 2026
- Lectures + exercises with `acados` and `leap-c`
- Participant projects

MPCRL26

Software

Tools for building learning predictive controllers.

- `acados`: fast NMPC solvers
- `leap-c`: differentiable MPC/RL workflows

`leap-c`

Backup: The ctx Object

What it stores

- forward solution iterate
- solver input, status, log
- cached sensitivity calculations

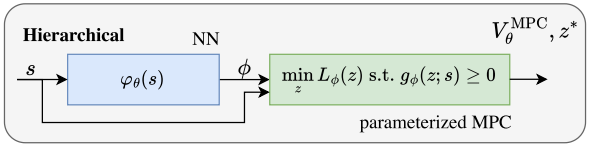
Why it matters

warmstarts, repeated sensitivity queries, batched training data

Python Code 🐍

```python
ctx, u0, X, U, value = diff_mpc(
    x0, params=params
)

ctx, u0, X, U, value = diff_mpc(
    x0_next, params=params, ctx=ctx
)

batch = collate_torch([
    {"ctx": ctx1, "x0": x01},
    {"ctx": ctx2, "x0": x02},
])
```

Backup: Hierarchical MPCRL with SAC



Hierarchical architecture from Talk II

Hierarchical MPCRL:

- MPC maps parameters to actions

$$\phi = \varphi_\theta(s), \qquad a = u_0^\star(s, \phi)$$

- neural network predicts state-dependent parameters

Use case:

- SAC tunes the neural-network parameters θ
- built on top of core `leap-c`

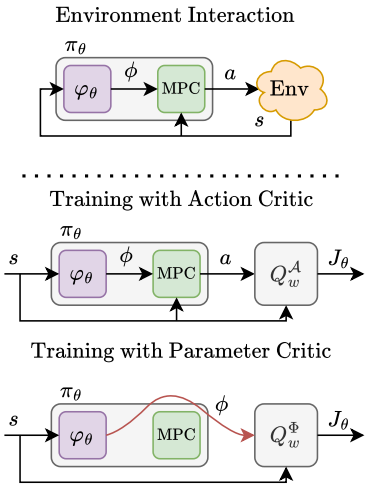
Backup: Parameter Feedback or Action Feedback?

Parameter critic: $Q^{\Phi}(s, \phi)$

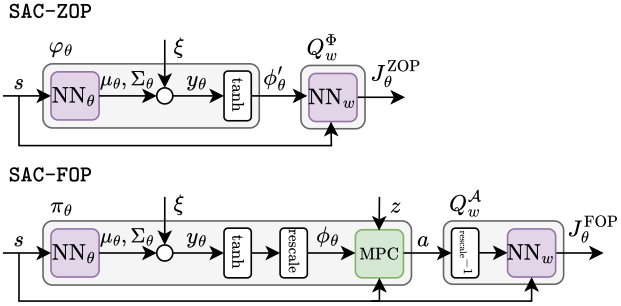
- gives feedback directly on MPC parameters
- avoids differentiating through MPC in actor update
- zero-order w.r.t. the MPC layer

Action critic: $Q^{\mathcal{A}}(s, a)$

- gives feedback on final action $a = u_0^\star(s, \phi)$
- actor update differentiates through MPC
- uses `leap-c` sensitivities



Backup: SAC-ZOP and SAC-FOP



SAC-ZOP (Zero-Order, Parameter Noise)

- uses parameter critic $Q_w^\Phi(s, \phi)$
- avoids differentiating through MPC during training
- fast and simple training loop

SAC-FOP (First-Order, Parameter Noise)

- uses action critic $Q_w^A(s, a)$
- differentiates through the MPC layer
- leverages first-order solution map information