NeuroMANCER Documentation

Aaron Tuor, Jan Drgona, Mia Skomski, Stefan Dernbach, James Koch, Zhao Chen,
Christian Møldrup Legaard, Draguna Vrabie
PNNL

December 2022

1 Introduction

NeuroMANCER (Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations) NeuroMANCER is a Pytorch-based framework for solving parametric constrained optimization problems.
NeuroMANCER code repository: `https://github.com/pnnl/neuromancer`

2 NeuroMANCER Requirements

List of Python dependencies: is stored in yml file: `https://github.com/pnnl/neuromancer/blob/master/env.yml`

2.1 Functional requirements

Generally, functional requirements are expressed in the form "system must do <requirement>".

User Interface: NeuroMANCER as a Scientific Machine Learning framework should provide a user-friendly interface for formulating and instantiating differentiable programming problems for:

- parametric constrained optimization
- physics-informed dynamical systems modeling
- parametric constrained optimal control

The NeuroMANCER framework should construct differentiable programs based on following input-output specifications:

- Input: high-level syntax and algebraic symbolic language for defining the objectives, constraints, and trainable components of the differentiable programming problem.
- Output: Instantiated differentiable programming problem represented by Pytorch nn.Module class.

Differentiable Programming Solvers: NeuroMANCER should come with a set of solvers and meta-heuristics for hyperparameter optimization providing an automated solution for complex differentiable programming problems.

Hard constraints guarantees: NeuroMANCER should provide a set of model architectures and solution methods that can guarantee the satisfaction of hard constraints with user-defined precision.

HPC support: NeuroMANCER should provide a set of templates for dispatch, training, and analysis of the differentiable programming problems using HPC machines such as GPU clusters.

2.2 Non-Functional requirements

Non-functional requirements take the form "system shall be <requirement>."

User experience: NeuroMANCER should be user-friendly and intuitive with emphasis on users from various engineering domains such as mechanical, electrical, chemical, and control engineering domains. NeuroMANCER should be a tool for easy prototyping and execution of developed programs. NeuroMANCER should be well documented with README, user manual, code tutorials, and docstrings compiled by pydot.

Architecture: The framework should be modular and provide a set of modeling abstractions and templates for constructing the problems mentioned above. Each class should come with standardized Type defined input-output specifications.

Interaction requirements:

- Interoperability with Pytorch ecosystem. Since NeuroMANCER is built on top of Pytorch, its APIs shall be designed in a way to allow easy integration of third-party model architectures implemented as Pytorch nn.Modules into NeuroMANCER's computational graphs.
- Interoperability with constrained optimization frameworks such as CVXpy, Pyomo, or CasADi is a plus. This could include: 1) using constrained optimization solvers as safety filters in online deployment of models trained in Neuromancer, 2) extraction of computational graphs or constrained optimizaiton problem formulations into CVXpy, Pyomo, or CasADi.
- NeuroMANCER should also come with a base class abstraction that will allow for easy implementation of new solvers.

Development and Maintanance requirements: NeuroMANCER shall be easily extensible with new model architectures, solvers, and features. NeuroMANCER development shall be safe with protected branches, reviewed merge requests, and automated pytest executions.

Open-source: NeuroMANCER shall be a free open-source repository.

Reliability: NeuroMANCER shall be reliable and robust. NeuroMANCER installation needs to be tested on all supported operating systems. All open-source examples and tutorials need to be bug-free and tuned, and verified on all operating systems. Open-sourced model architectures and solvers need to provide robust convergence across datasets and hyperparameter scenarios.

2.3 Neuromancer Use Case Diagram

The Neuromancer use case diagram is shown in Fig. 1

List of Neuromancer use cases

- Execute and modify tutorial and example scripts.
- Create new tutorials and example scripts.
- Formulate constrained optimization problems in high level symbolic language
- Construct differentiable programs of parametric optimization problems in a form of symbolic computational graphs
- Visualize computational graphs of differentiable programs
- Construct AggregateLoss class.

Figure 1: Neuromancer Use case diagram.

- Construct Trainer class.
- Solve differentiable parametric programming programs (constrained machine learning, constrained optimization, system identification, and control) with sampling-based automatic differentiation
- Evaluate and Visualize the performance of the obtained parametric solution on a given test case (e.g., prediction accuracy of system identification task, or closed-loop control performance of trained control policy)
- Construct DataLoader with dataset.
- Impement new Dataset Class.
- Impement new Callback Class.
- Modify Trainer class.
- Dispatch distribution of experiments with specified hyperparameters (e.g., training models on GPU cluster with hyperparameter search)
- Implement new component architectures (e.g., custom neural ordinary differential equation architectures)
- Implement new solvers (e.g., Augmented Lagrangian method, or Interior point algorithm)
- Expand or modify core Neuromancer library (e.g., modify forward pass method of the Problem class)
- Implement new visualization capability.

List of Neuromancer actors

- Core framework developer
- Researcher (methods and applications)
- External contributor
- End user
- Computing platform (Laptop CPU, GPU, HPC cluster)

3 Neuromancer Architecture and API

3.1 Neuromancer Architecture

UML Class diagram of Neuromancer's architecture: `https://github.com/pnnl/neuromancer/blob/master/figs/class_diagram.png`

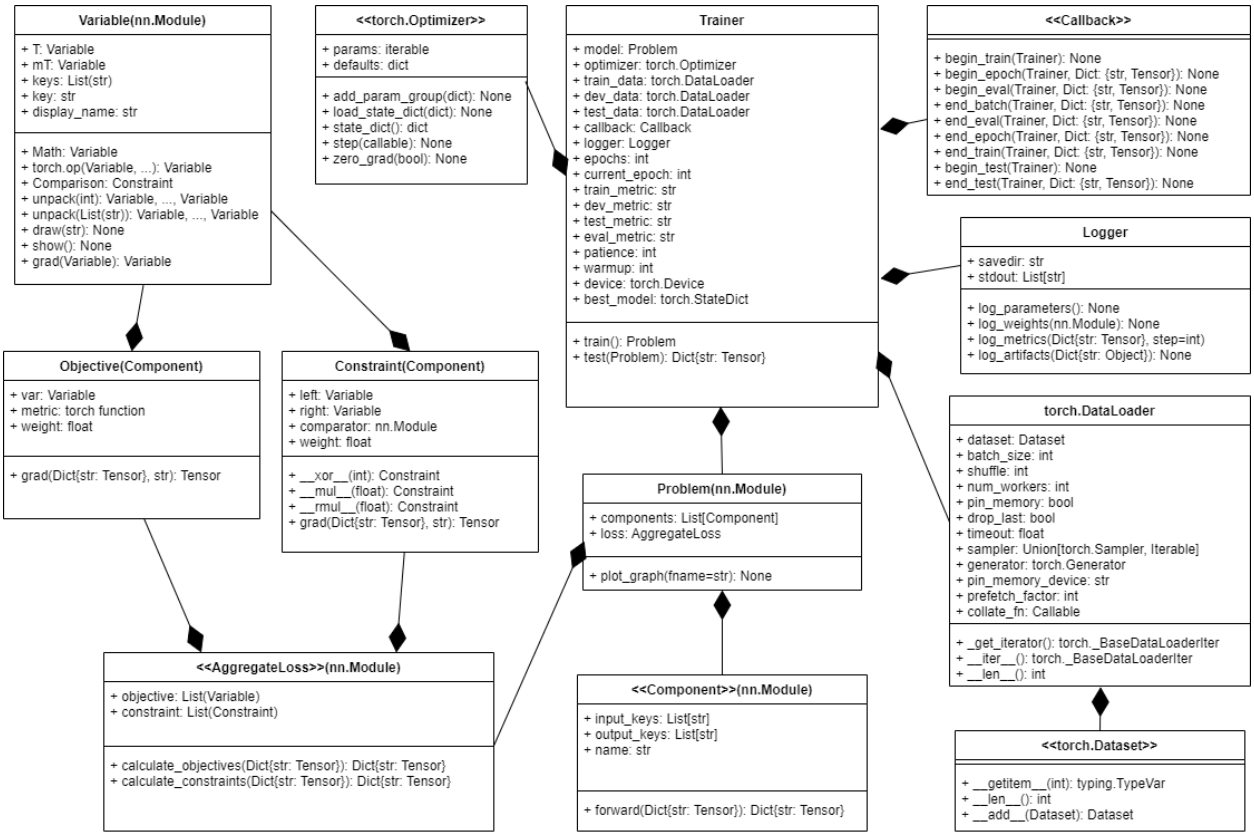


Figure 2: UML Class diagram of Neuromancer.

3.2 Neuromancer API documentation

API specifications of Neuromancer classes and functions can be found in: `https://pnnl.github.io/neuromancer/`

Trainer Class encapsulating boilerplate PyTorch training code. Training procedure is somewhat extensible through methods in Callback objects associated with training and evaluation waypoints. Trainer is instantiated with given Problem class and Pytorch Dataloader classes storing Neuromancer Datasets.
`https://pnnl.github.io/neuromancer/trainer.html`
Callback abstract classe for versatile behavior in the Trainer object at specified checkpoints. Allows the user to customize training, evaluation, and testing phases of the optimization algorithm.
`https://pnnl.github.io/neuromancer/callbacks.html`
Logger class for saving arguments, metrics, and artifacts (images, video) into specified directory. Also allows to control the verbosity of print statements during training.
`https://pnnl.github.io/neuromancer/loggers.html`
Dataset class compatible with neuromancer Trainer based on parent Pytorch Dataset class. Implements static, sequence, and graph structured datasets.
`https://pnnl.github.io/neuromancer/dataset.html`
DataLoader class from Pytorch combines a dataset and a sampler, and provides an iterable over the given dataset.
`https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader`
Problem class is similar in spirit to a nn.Sequential module. However, by concatenating input and output dictionaries for each component class we can represent arbitrary directed acyclic computation graphs. In addition the Problem class takes care of calculating loss functions via given instantiated weighted multi-objective AggregateLoss class which calculate objective and constraints terms from aggregated input and set of outputs from the Component modules. Problem class represents complete differentiable constrained optimization problem with scalar valued training metric suitable for gradient-based optimization via back-propagation algorithm.
`https://pnnl.github.io/neuromancer/problem.html`
AggregateLoss abstract class for calculating constraints, objectives, and aggregate loss values suitable for automatic differentiation via backpropagation algorithm. Implements different loss aggregation methods such as: Penalty Method, Barrier Method, or Augmented Lagrangian methodm.
`https://pnnl.github.io/neuromancer/loss.html`
Variable class is an abstraction that allows for the definition of constraints and objectives with some nice syntactic sugar. When a Variable object is called given a dictionary a pytorch tensor is returned, and when a Variable object is subjected to a comparison operator a Constraint is returned. Mathematical operators return Variables which will instantiate and perform the sequence of mathematical operations. Supported infix operators (variable $*$ variable, variable $*$ numeric): $+, -, @, *, <, <=, >, >=, ==, \wedge$
`https://pnnl.github.io/neuromancer/constraint.html#constraint.Variable`
Component abstract class allows to wrap arbitrary nn.Modules in Pytorch into symbolic representation of the computational graph. Component is mapping input keys onto output keys representing symbolic variables of the computational graph whose forward pass is defined by Pytorch nn.Module.
`https://pnnl.github.io/neuromancer/component.html`
Constraint is a Component class constructed by a composition of Variable objects using comparative infix operators, $'<', '>', '==', '<=', '>='$ and $'*'$ to weight loss component and $'\wedge'$ to determine l-norm of constraint violation in determining loss. A Constraint has the intuitive syntax for defining constraints of optimization problems via Variable objects.
`https://pnnl.github.io/neuromancer/constraint.html#constraint.Constraint`
Objective is a Component class constructed via neuromancer Variable object and given metric with forward pass that evaluates metric as torch function on Variable values. Objective allows to create Loss function terms directly from instantiated Variables.
`https://pnnl.github.io/neuromancer/_modules/constraint.html#Objective`

4 Neuromancer Methods and Algorithms

This section documents differentiable programming methods and algorithms for solution of: 1) parametric constrained optimization problems, 2) physics-constrained system identification problems, and 3) parametric optimal control problems.

4.1 Differentiable Parametric Programming

Constrained optimization problems where the solution x depends on the varying problem parameters ξ are called parametric programming problems. Differentiable Parametric Programming (DPP) is a set of methods that use automatic differentiation (AD) to compute sensitivities of constrained optimization problems w.r.t. to the problem parameters for obtaining parametric solutions of the problem.

Differentiable Parametric Programming (DPP) Concept Recent years have seen a rich literature of deep learning (DL) models for solving the constrained optimization problems on real-world tasks such as power grid, traffic, or wireless system optimization. Earlier attempts simply adopt imitation learning (i.e., supervised learning) to train function approximators via a minimization of the prediction error using labeled data of pre-computed solutions using iterative solvers (i.e. IPOPT). Unfortunately, these models can hardly perform well on unseen data as the outputs are not trained to satisfy physical constraints, leading to infeasible solutions.

To address the feasibility issues, existing methods have been imposing constraints on the output space of deep learning models for a subsequent differentiation using AD tools. These differentiable programming-based methods, also called end-to-end learning, directly consider the original objectives and constraints in the DL training process without the need of expert labeled data. Fig. 3 conceptually demonstrated the difference between supervised imitation learning and unsupervised Differentiable Parametric Programming (DPP) which solution is obtained by differentiating the objectives and constraints of the parametric optimization problem.

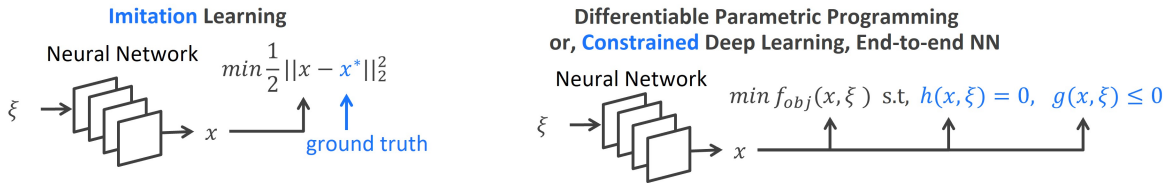


Figure 3: Imitation learning VS end-to-end learning using Differentiable Parametric Programming.

Differentiable Parametric Programming (DPP) Problem A generic formulation of the DPP is given in the form of a parametric constrained optimization problem:

$$\min_{\Theta} \mathcal{L}_{\text{obj}} = \min_{\Theta} \frac{1}{m} \sum_{i=1}^{m} \mathbf{f}(\mathbf{x}^i, \boldsymbol{\xi}^i) \tag{1a}$$

$$\text{s.t. } \mathbf{g}(\mathbf{x}^i, \boldsymbol{\xi}^i) \leq 0, \ \mathbf{h}(\mathbf{x}^i, \boldsymbol{\xi}^i) = 0, \tag{1b}$$

$$\mathbf{x}^i = \boldsymbol{\pi}_{\Theta}(\boldsymbol{\xi}^i), \ \boldsymbol{\xi}^i \in \Xi, \ \forall i \in \mathbb{N}_1^m \tag{1c}$$

Where Ξ represents the sampled dataset, and $\boldsymbol{\xi}^i$ represents i-th batch of the sampled problem data. The vector $\mathbf{x}^i$ represents optimized variables that minimize the loss function while satisfying a set of inequality and equality constraints ((1b)). The map $\pi_{\Theta}(\boldsymbol{\xi}^i)$ is given by a deep neural network parametrized by Θ and represents the parametric solution of the DPP problem.

Differentiable Loss functions There are several ways in which we can enforce the constraints satisfaction while learning the solution $\pi_{\Theta}(\boldsymbol{\xi}^i)$ of the differentiable constrained optimization problem ((1)). The simplest approach is to penalize the constraints violations by augmenting the loss function $\mathcal{L}_{\text{obj}}$ ((1a)) with the penalty functions given as:

$$\mathcal{L}_{\text{con}} = \frac{1}{m} \sum_{i=1}^{m} \left(Q_g ||\texttt{ReLU}(\mathbf{g}(\mathbf{x}^i, \boldsymbol{\xi}^i))||_l + Q_h ||\mathbf{h}(\mathbf{x}^i, \boldsymbol{\xi}^i)||_l \right) \tag{2}$$

Where l denotes the norm type and Q_g, Q_h being the corresponding weight factors. The overall loss then becomes $\mathcal{L}_{\text{penalty}} = \mathcal{L}_{\text{obj}} + \mathcal{L}_{\text{con}}$.

Other approaches include barrier functions, or Augmented Lagrangian type methods.

DPP Problem Solution A main advantage of having a differentiable objective function and constraints in the DPP problem formulation (1) is that it allows us to use automatic differentiation to directly compute the gradients of the parametric solution map $\pi_\Theta(\boldsymbol{\xi}^i)$. In particular, by representing the problem (1) as a computational graph and leveraging the chain rule, we can directly compute the gradients of the loss function $\mathcal{L}_{\text{penalty}}$ w.r.t. the solution map weights Θ as follows:

$$\nabla_\Theta \mathcal{L}_{\text{penalty}} = \frac{\partial \mathcal{L}_{\text{obj}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \Theta} + \frac{\partial \mathcal{L}_{\text{con}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \Theta} = \frac{\partial \mathcal{L}_{\text{obj}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \Theta} + \frac{\partial \mathcal{L}_{\text{con}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \Theta} \tag{3}$$

Where $\frac{\partial \mathbf{x}}{\partial \Theta}$ represent partial derivatives of the neural network solution map w.r.t. its weights that are typically being computed in deep learning applications via backpropagation. The advantage of having gradients (3) is that it allows us to use scalable stochastic gradient optimization algorithms such as AdamW [1] to solve the corresponding DPP problem (1) by direct offline optimization of the neural network. In practice, we can compute the gradient of the DPP problem by using automatic differentiation frameworks such as Pytorch [2].

DPP Optimization Algorithm The DPP solution algorithm is summarized in Algorithm 1.

Algorithm 1 Differentiable Parametric Programming Algorithm.

1: **input** training dataset Ξ of sampled problem parameters.
2: **input** differentiable solution map architecture $\pi_\Theta(\boldsymbol{\xi})$ parametrized by Θ.
3: **input** differentiable constrained optimization objective $\mathbf{f}(\mathbf{x}, \boldsymbol{\xi})$ and constraints $\mathbf{g}(\mathbf{x}, \boldsymbol{\xi}) \leq 0,\ \mathbf{h}(\mathbf{x}, \boldsymbol{\xi}) = 0$
4: **input** DPP loss function aggregator $\mathcal{L}_{\text{penalty}}$
5: **input** optimizer $\mathbb{O}$
6: **differentiate** DPP loss $\mathcal{L}_{\text{penalty}}$ to obtain the parameter gradients $\nabla_\Theta \mathcal{L}_{\text{penalty}}$ of the solution map $\pi_\Theta(\boldsymbol{\xi})$
7: **learn** solution map $\pi_\Theta(\boldsymbol{\xi}^i)$ parametrized by Θ via optimizer $\mathbb{O}$ using gradient $\nabla_\Theta \mathcal{L}_{\text{penalty}}$
8: **return** trained parametric solution $\pi_\Theta(\boldsymbol{\xi})$

DPP Syntax example The following code illustrates the implementation of Differentiable Parametric Programming in Neuromancer.

```
# Tutorial example for Differentiable Parametric Programming  in Neuromancer

import torch
import neuromancer as nm
import slim
import numpy as np

    """
    # # #  Dataset
    """
    #  randomly sampled parameters theta generating superset of:
    #  theta_samples.min() <= theta <= theta_samples.max()
    np.random.seed(args.data_seed)
    nsim = 20000  # number of datapoints: increase sample density for more robust results
    samples = {"a": np.random.uniform(low=0.2, high=1.5, size=(nsim, 1)),
               "p": np.random.uniform(low=0.5, high=2.0, size=(nsim, 1))}
    data, dims = nm.get_static_dataloaders(samples)
    train_data, dev_data, test_data = data

    """
    # # #  mpNLP primal solution map architecture
    """
    func = nm.blocks.MLP(insize=2, outsize=2,
                    bias=True,
                    linear_map=slim.maps['linear'],
                    nonlin=activations['relu'],
                    hsizes=[args.nx_hidden] * args.n_layers)
    sol_map = nm.Map(func,
            input_keys=["a", "p"],
```

```
            output_keys=["x"],
            name='primal_map')
    """
    # # #  mpNLP objective and constraints formulation in Neuromancer
    """
    # variables
    x = nm.variable("x")[:, [0]]
    y = nm.variable("x")[:, [1]]
    # sampled parameters
    p = nm.variable('p')
    a = nm.variable('a')

    # objective function
    f = (1-x)**2 + a*(y-x**2)**2
    obj = f.minimize(weight=args.Q, name='obj')

    # constraints
    con_1 = (x >= y)
    con_2 = ((p/2)**2 <= x**2+y**2)
    con_3 = (x**2+y**2 <= p**2)

    # constrained optimization problem construction
    objectives = [obj]
    constraints = [args.Q_con*con_1, args.Q_con*con_2, args.Q_con*con_3]
    components = [sol_map]

    if args.proj_grad:  # use projected gradient update
        project_keys = ["x"]
        projection = nm.GradientProjection(constraints, input_keys=project_keys,
                                           num_steps=5, name='proj')
        components.append(projection)

    # create constrained optimization loss
    loss = nm.get_loss(objectives, constraints, train_data, args)
    # construct constrained optimization problem
    problem = nm.Problem(components, loss, grad_inference=args.proj_grad)
    # plot computational graph
    problem.plot_graph()

    """
    # # # Metrics and Logger
    """
    args.savedir = 'test_mpNLP_Rosebnrock'
    args.verbosity = 1
    metrics = ["train_loss", "train_obj", "train_mu_scaled_penalty_loss", "
    train_con_lagrangian",
               "train_mu", "train_c1", "train_c2", "train_c3"]
    logger = nm.BasicLogger(args=args, savedir=args.savedir, verbosity=args.verbosity,
    stdout=metrics)
    logger.args.system = 'mpmpNLP_Rosebnrock'

    """
    # # #  mpQP problem solution in Neuromancer
    """
    optimizer = torch.optim.AdamW(problem.parameters(), lr=args.lr)

    # define trainer
    trainer = nm.Trainer(
        problem,
        train_data,
        dev_data,
        test_data,
        optimizer,
        logger=logger,
        epochs=args.epochs,
        train_metric="train_loss",
        dev_metric="dev_loss",
        test_metric="test_loss",
```

```
        eval_metric="dev_loss",
        patience=args.patience,
        warmup=args.warmup,
        device=device,
    )

    # Train mpLP solution map
    best_model = trainer.train()
    best_outputs = trainer.test(best_model)
    # load best model dict
    problem.load_state_dict(best_model)
```

Listing 1: Example Differentiable Parametric Programming implementation in Neuromancer syntax.

DPP codebase List of open-source DPP examples in Neuromancer: `https://github.com/pnnl/neuromancer/tree/master/examples/parametric_programming`

List of Neuromancer classes required to build and solve DPP problems:

- dataset - classes for instantiating Pytorch dataloaders with training evaluation and testing samples `https://github.com/pnnl/neuromancer/blob/master/neuromancer/dataset.py`
- constraints - classes for defining constraints and custom physics-informed loss function terms `https://github.com/pnnl/neuromancer/blob/master/neuromancer/constraint.py`
- loss - class aggregating all instantiated constraints and loss terms in a scalar-valued function suitable for backpropagation-based training `https://github.com/pnnl/neuromancer/blob/master/neuromancer/loss.py`
- solvers - implementation of iterative solvers for hard constraints such as gradient projection method `https://github.com/pnnl/neuromancer/blob/master/neuromancer/solvers.py`
- problem - class agrregating trainable components (policies, dynamics, estimators) with loss functions in a differentiable computational graph representing the underlying constrained optimization problem `https://github.com/pnnl/neuromancer/blob/master/neuromancer/problem.py`
- trainer - class for optimizing the problem `https://github.com/pnnl/neuromancer/blob/master/neuromancer/trainer.py`

4.2 Differentiable Physics-constrained System Identification

Differentiable System Identification Models Concept Differentiable models such as Neural ordinary differential equations (NODEs) or neural state space models (NSSMs) represent a class of black box models that can incorporate prior physical knowledge into their architectures and loss functions. Examples include structural assumption on the computational graph inspired by domain application, or structure of the weight matrices of NSSM models as shown in 4, or networked NODE architecture illustrated in Fig. 5. Differentiability of NODEs and NSSMs allows us to leverage gradient-based optimization algorithms for learning the unknown parameters of these structured digital twin models from observational data of the real system.

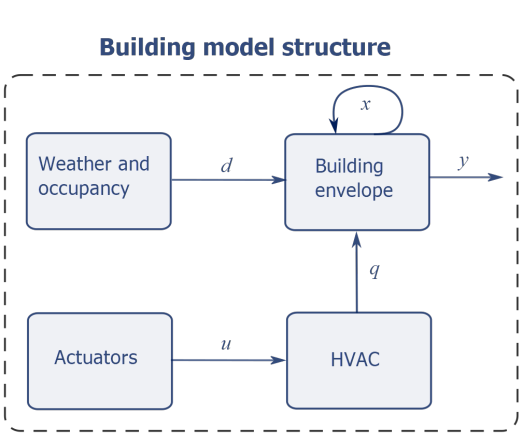
System Identification Problem Consider the non-autonomous partially observable nonlinear dynamical system

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)), \tag{4a}$$

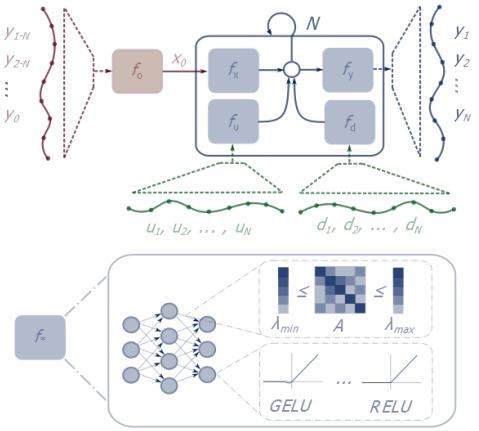
$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t)), \tag{4b}$$

$$\mathbf{x}(t_0) = \mathbf{x}_0, \quad \mathbf{u}(t_0) = \mathbf{u}_0 \tag{4c}$$

where $\mathbf{y}(t) \in \mathbb{R}^{n_y}$ are measured outputs, and $\mathbf{u}(t) \in \mathbb{R}^{n_u}$ represents controllable inputs or measured disturbances affecting the system dynamics.



(a) Structure of physics-based building thermal model.



(b) Structured recurrent neural dynamics model.

Figure 4: Generic structure of physics-inspired neural state space model (NSSM) architecture for building thermal dynnamics. Weights of individual neural blocks f_* are parametrized by linear maps with constrained eigenvalues, while component outputs are subject to penalty constraints parametrized by common activation functions.

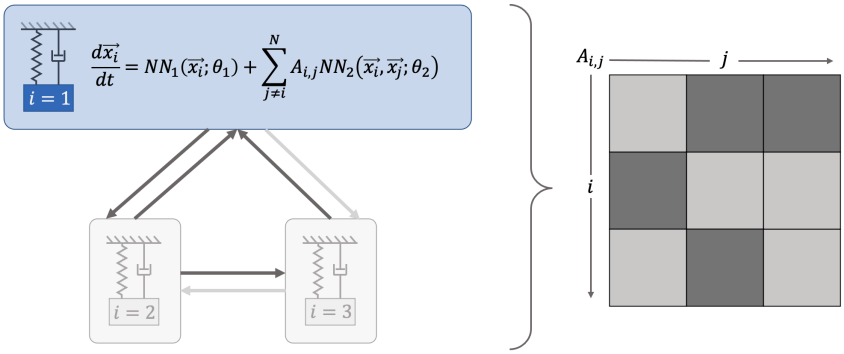


Figure 5: A small network of unknown nodes interact according to an adjacency matrix A. In this pictoral example, the nodes are assumed to be second-order oscillators with a directed graph structure, $A_{i,j}$. NN_1 approximates the intrinsic nodal physics and NN_2 approximates the coupling physics.

We assume access to a limited set of system measurements in the form of tuples, each of which corresponds to the input-output pairs along sampled trajectories with temporal gap Δ. That is, we form a dataset

$$S = \{(\mathbf{u}_t^{(i)}, \mathbf{y}_t^{(i)}), (\mathbf{u}_{t+\Delta}^{(i)}, \mathbf{y}_{t+\Delta}^{(i)}), \ldots, (\mathbf{u}_{t+N\Delta}^{(i)}, \mathbf{y}_{t+N\Delta}^{(i)})\}, \tag{5}$$

where $i = 1, 2, \ldots, n$ represents up to n different batches of input-output trajectories with N-step time horizon length.

The primary objective of the physics-constrained system identification is to construct structured digital twin models and learn their unknown parameters from the provided observation data to provide accurate and robust long-term prediction capabilities.

System identification models Our recent development work in Neuromancer has given us the capability to learn dynamical systems of the form:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t)) \tag{6}$$

or

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), t) \tag{7}$$

or

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{8}$$

where $\mathbf{x}(t)$ is the time-varying state of the considered system, $\mathbf{u}(t)$ are system control inputs, and $\mathbf{f}$ is the state transition dynamics. This modeling strategy can be thought of as an equivalent method to *Neural Ordinary Differential Equations* [3], whereby an ODE of the above forms is fit to data with a universal function approximator (e.g. deep neural network) acting as the state transition dynamics. To train an appropriate RHS, Chen et al. utilize a continuous form of the adjoint equation; itself solved with an ODESolver. Instead, we choose to utilize the autodifferentiation properties of PyTorch to build differentiable canonical ODE integrators (e.g. as in Raissi et al. [4]).

We wish to test the capability of this methodology in a variety of situations and configurations. Of particular interest is the predictive capability of this class of methods compared with Neural State Space Models and other traditional "black-box" modeling techniques.

Before moving on, it is important to note that there are two dominant neural ODE packages freely available. The first is *DiffEqFlux.jl* developed and maintained by SciML within the Julia ecosystem. The second is *torchdyn* which lives within the PyTorch ecosystem. Both packages are well-documented and have become established in application-based research literature.

System Identification Model Architectures with Hard Constraints In the following we will enumerate several baseline black-box model architectures that can be modified for construction of physics-constrained digital twins.

The most generic black *neural state space model (NSSM)* [5, 6, 7] is given by:

$$\mathbf{x}_{t+\Delta_t} = f(\mathbf{x}_t, \mathbf{u}_t) \tag{9a}$$

$$\hat{\mathbf{y}}_{t+\Delta_t} = g(\mathbf{x}_{t+1}) \tag{9b}$$

where f is a neural network which models the interaction between state and input/disturbance vectors, and g is output map parametrized by a neural network. For these models the sampling time Δ_t is a fixed interval usually determined by the sampling rate exhibited in the training data set.

Neural Ordinary Differential Equations (NODEs) [3] are a recently proposed method for modeling continuous time dynamics using neural networks. NODEs overcome the limitation of fixed time steps by learning the right hand side of an ODE with $\frac{dx}{dt}$ modeled as a neural network f. In general this is formulated as:

$$\mathbf{x}_{t+\Delta_t} = \mathbf{x}_t + \int_t^{t+\Delta_t} f(\mathbf{x}_t, \mathbf{u}_t, \tau) d\tau \tag{10}$$

In practice, using an Euler integration scheme, and f as a neural network, we have the following NODE timestepper model:

$$\mathbf{x}_{t+\Delta_t} = \mathbf{x}_t + \Delta_t * f(\mathbf{x}_t, \mathbf{u}_t, t) \tag{11}$$

These NODE timestepper models have the flexibility to train on unevenly sampled datasets, and conversely predict at arbitrary time intervals into the future.

Since partial knowledge about a system of interest is usually captured in the form of differential equations or their interacting structure, the NSSM and NODE modeling paradigm presents opportunity for incorporating prior knowledge by modeling only the unknown factors in the dynamics with neural networks.

In real-world engineering systems such as building thermal dynamics, the coupling of state variables is sparse and uniquely defined by the topology of modeled building. To address this issue, we can reformulate the generic NODE model (11) into a *networked NODE model* given as follows:

$$\mathbf{x}^i_{t+\Delta_t} = \mathbf{x}^i_t + \Delta_t * \sum_j^n a_{i,j} f^{i,j}(\mathbf{x}^i_t, \mathbf{x}^j_t) \tag{12}$$

Here $a_{i,j}$ represents a dynamical coupling term as an element of adjacency matrix $\mathbf{A}$ that uniquely defines sparse coupling of state variables with pair-wise interactions between connected states $f^{i,j}(x_t^i, x_t^j)$. This formulation now allows us to embed prior knowledge about the building topology via the adjacency matrix $\mathbf{A}$ and heat transfer physics via the dynamical coupling terms $f^{i,j}(x_t^i, x_t^j)$.

The model network NODE model (12) boils down to classical resistance capacitance (RC) network model [8] in the following case:

$$f^{i,j}(\mathbf{x}_t^i, \mathbf{x}_t^j) = \frac{1}{C_i R_{i,j}}(\mathbf{x}_t^j - \mathbf{x}_t^i) \tag{13}$$

with $\frac{1}{C_i R_{i,j}}$ representing the heat transfer coefficient defining the interaction dynamics between node i and j. Now, depending on the type of the state variable (air, wall, or supply temperatures) and known heat transfer equations (conduction, convection, advection, radiation) between the nodes one can define physics-based priors by leveraging the resistance and capacitance formulas summarized in Table 1. Here rho, c_p, V,

Table 1: Overview of the resistance and capacitance formulas.

notation	meaning	formula
C	generic thermal capacity	$C = \rho c_p V$
C_a	air heat capacity	$C = \rho^{\mathrm{a}} c_p^{\mathrm{a}} V$
C_m	mass heat capacity	$C = \rho^{\mathrm{m}} c_p^{\mathrm{m}} V$
R_{adv}	advective heat resistance	$R_{\mathrm{adv}} = \frac{1}{m c_p}$
R_{cond}	conductive heat resistance	$R_{\mathrm{cond}} = \frac{L}{kA}$
R_{conv}	convective heat resistance	$R_{\mathrm{conv}} = \frac{1}{h_c A}$
R_{rad}	radiative heat resistance	$R_{\mathrm{rad}} = \frac{1}{h_r A}$

A, m, h, and k stand for density, specific heat capacity, volume, area, mass flow, heat transfer coefficient and heat conductivity, respectively.

System Identification Learning Objective with Soft Constraints The primary learning objective is to minimize the mean squared error, $\mathcal{L}_y$, between predicted values and the ground truth measurements for the N-step prediction horizon:

$$\mathcal{L}_y = \frac{1}{n \cdot N \cdot n_y} \sum_{i=1}^{n} \sum_{t=1}^{N} \sum_{j=1}^{n_y} (\hat{\mathbf{y}}_{t,j}^{(i)} - \mathbf{y}_{t,j}^{(i)})^2 \tag{14}$$

where n_y is the dimension of the observations $\mathbf{y}_t$ and predictions $\hat{\mathbf{y}}$, and n is the number of N-step long training trajectories.

The system identification objective (14) can be augmented with various kind of physics-informed soft constraints. In the following we enumerate a few examples.

First, we apply inequality constraints on output predictions during training in order to promote the boundedness and convergence of our dynamical models. We define output lower and upper bounds $\underline{\mathbf{y}}$ and $\overline{\mathbf{y}}$, respectively, and derive corresponding slack variables $\mathbf{s}^{\underline{\mathbf{y}}}$ and $\mathbf{s}^{\overline{\mathbf{y}}}$ to include in the objective function as an additional term $\mathcal{L}_y^{\mathrm{con}}$:

$$\mathbf{s}^{\underline{\mathbf{y}}} = \max(0, -\hat{\mathbf{y}} + \underline{\mathbf{y}}) \tag{15a}$$

$$\mathbf{s}^{\overline{\mathbf{y}}} = \max(0, \hat{\mathbf{y}} - \overline{\mathbf{y}}) \tag{15b}$$

$$\mathcal{L}_y^{\mathrm{con}} = \frac{1}{n_y} \sum_{i=1}^{n_y} (\mathbf{s}_i^{\underline{\mathbf{y}}} + \mathbf{s}_i^{\overline{\mathbf{y}}}) \tag{15c}$$

Bounding system trajectories is a straightforward way of including prior knowledge by delineating a physically meaningful phase space of the learned dynamics.

To promote continuous trajectories of our dynamics models, we optionally apply a state smoothing loss which minimizes the mean squared error between successive predicted states:

$$\mathcal{L}_{dx} = \frac{1}{(N-1)n_x} \sum_{t=1}^{N-1} \sum_{i=1}^{n_x} (\mathbf{x}_t^{(i)} - \mathbf{x}_{t+1}^{(i)})^2 \tag{16}$$

We include constraints penalties as additional terms to the optimization objective 14, and further define coefficients, Q_* as hyperparameters to scale each term in the multi-objective loss function:

$$\mathcal{L} = Q_y \mathcal{L}_y + Q_{dx} \mathcal{L}_{dx} + Q_y^{\text{con}} \mathcal{L}_y^{\text{con}} \tag{17}$$

Differentiable System Identification Algorithm The physics-constrained system identification algorithm with differentiable digital twin models is summarized in Algorithm 2.

Algorithm 2 Physics-constrained system ID algorithm with differentiable models.

1: **input** training dataset S (5) of sampled input output trajectories.
2: **input** differentiable digital twin model architecture M, e.g. (9), (11), (12), parametrized by θ.
3: **input** System ID loss $\mathcal{L}$ (17)
4: **input** optimizer $\mathbb{O}$
5: **differentiate** System ID loss $\mathcal{L}$ to obtain the parameter gradients $\nabla_\theta \mathcal{L}$ of the system models M
6: **learn** system dynamics model M parametrized by θ via optimizer $\mathbb{O}$ using gradient $\nabla_\theta \mathcal{L}$
7: **return** trained system dynamics model M

Syntax and Usage Two Neuromancer dynamics classes handle continuous time dynamics: `ODEAuto` and `ODENonAuto`. As their names suggest, these classes handle the scenarios corresponding to Equations 6-7. Their usage is detailed below:

Autonomous ODEs: Autonomous ODEs are those that do not explicitly depend on time; as such, the dynamics are functions of state variables alone. A fully-specified neural ODE of this type requires a RHS function (either a neural network or other tensor-tensor mapping. The use of the `ODEAuto` class is as follows:

```
# Instantiate the ODE RHS as MLP:
fx = blocks.MLP(nx, nx, linear_map=slim.maps['linear'],
                nonlin=activations['leakyrelu'],
                hsizes=[10, 10])

# Instansiate the integrator, handing it the RHS "fx":
fxRK4 = integrators.RK4(fx, h=ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)

# Creating the dynamics model:
NODE = dynamics.ODEAuto(fxRK4, fy, name='dynamics', input_key_map={"x0": "x0_estimator"})
```

Note that the transition dynamics `fx` is a square mapping ($nx \to nx$) of states as expected.

Non-Autonomous ODEs: Non-autonomous ODEs depend on time, external inputs, or both time and inputs. The syntax for these systems changes as continuos-time representations of time and any external inputs must be available and provided to the integrator. This is handled with the construction and passing of interpolants.

Time as input: An example non-autonomous system with explicit dependence on time is the *Forced Duffing Oscillator*, given by the ODEs:

$$\frac{dx_1}{dt} = x_2 \tag{18}$$

$$\frac{dx_2}{dt} = -\delta x_2 - \alpha x_1 - \beta x_1^3 + \gamma \cos(\omega t) \tag{19}$$

Supposing that the model is known except for one or more of the parameters, one can build a consistent tensor-tensor mapping to pass to an integrator and `ODENonAuto` class. First, the ODE RHS is defined:

```
class DuffingParam(ODESystem):
    def __init__(self, insize=3, outsize=2):
        """
        :param insize:
        :param outsize:
        """
        super().__init__(insize=insize, outsize=outsize)
        self.alpha = nn.Parameter(torch.tensor([1.0]), requires_grad=False)
        self.beta = nn.Parameter(torch.tensor([5.0]), requires_grad=False)
        self.delta = nn.Parameter(torch.tensor([0.02]), requires_grad=False)
        self.gamma = nn.Parameter(torch.tensor([8.0]), requires_grad=False)
        self.omega = nn.Parameter(torch.tensor([0.5]), requires_grad=True)

    def ode_equations(self, x):
        # states
        x0 = x[:, [0]]  # (# batches,1)
        x1 = x[:, [1]]
        t =  x[:, [2]]
        # equations
        dx0dt = x1
        dx1dt = -self.delta*x1 - self.alpha*x0 - self.beta*x0**3 +
                self.gamma*torch.cos(self.omega*t)
        return torch.cat([dx0dt, dx1dt], dim=-1)
```

Note that in this definition, only the paramter ω is tunable; thus, this is a 1-parameter training task. Additionally, note the dimensionality: expected is a state dimension of three, with the third dimension corresponding to time. The specification of the Neural ODE begins with defining a continuous representation of time:

```
t = torch.from_numpy(t)
interp_u = LinInterp_Offline(t, t)
```

The rest of the setup is identical to the autonomous case with the exception of the dynamics class:

```
# Instantiate the ODE RHS:
duffing_sys = ode.DuffingParam()

# Instansiate the integrator, handing it the RHS "duffing_sys":
fxRK4 = integrators.RK4(duffing_sys, interp_u=interp_u, h=ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)

# Creating the dynamics model:
dynamics_model = dynamics.ODENonAuto(fxRK4, fy,
    input_key_map={"x0": "x0_estimator", "Time": "Timef", 'Yf': 'Yf'},
    name='dynamics', online_flag=False)
```

Other external inputs: Control signals are dealt with in the same manner as time: they must first be represented in a continuous form via an interpolant. Specification of these interpolants is as follows:

```
t = torch.from_numpy(t) # from numpy dataset
u = raw['U'].astype(np.float32) # getting control 'u' from data dictionary
u = np.append(u,u[-1,:]).reshape(-1,2)
ut = torch.from_numpy(u)
interp_u = LinInterp_Offline(t, ut)
```

The neural ODE is specified in the same way as the forced Duffing system:

```
# Get the dimension of extra inputs
nu = dims['U'][1]

# Construct black-box RHS mapping from nx+nu to nx
black_box_ode = blocks.MLP(insize=nx+nu, outsize=nx, hsizes=[30, 30],
                           linear_map=slim.maps['linear'],
                           nonlin=activations['gelu'])

# Hand it over to the integrator with the interpolant:
fx_int = integrators.RK4(black_box_ode, interp_u=interp_u, h=modelSystem.ts)

# Identity output mapping:
fy = slim.maps['identity'](nx, nx)

# Creating the dynamics model:
dynamics_model = dynamics.ODENonAuto(fx_int, fy, extra_inputs=['Uf'],
                input_key_map={"x0": "x0_estimator", "Time": "Timef", 'Yf': 'Yf'},
                name='dynamics', online_flag=False)
```

System Identification Neuromancer example The following code illustrates the implementation of System Identification in Neuromancer.

```
# Tutorial example for System Identification in Neuromancer

import torch
import neuromancer as nm
import slim
import psl


# Dataset
system = psl.systems['Brusselator1D']
modelSystem = system()
raw = modelSystem.simulate(ts=0.05)
psl.plot.pltOL(Y=raw['Y'])
psl.plot.pltPhase(X=raw['Y'])

nsteps = 1
nstep_data, loop_data, dims = nm.get_sequence_dataloaders(raw, nsteps, moving_horizon=False)
train_data, dev_data, test_data = nstep_data
train_loop, dev_loop, test_loop = loop_data

#  State estimator
nx = 2
estim =  nm.estimators.FullyObservable(
    {**train_data.dataset.dims, "x0": (nx,)},
    linear_map=slim.maps['identity'],
    input_keys=["Yp"],
)

```

```
# Dynamics model
brussels =  nm.ode.BrusselatorParam()
fxRK4 =  nm.integrators.RK4(brussels, h=0.05)
fy = slim.maps['identity'](nx, nx)
dynamics_model =  nm.dynamics.ODEAuto(fxRK4, fy, name='dynamics',
                                      input_key_map={"x0": f"x0_{estim.name}"})

#  Variables
yhat = variable(f"Y_pred_{dynamics_model.name}")
y = variable("Yf")
x0 = variable(f"x0_{estim.name}")
xhat = variable(f"X_pred_{dynamics_model.name}")

#  Objective terms
yFD = (y[:, 1:, :] - y[:, :-1, :])
yhatFD = (yhat[:, 1:, :] - yhat[:, :-1, :])
fd_loss = 2.0*((yFD == yhatFD)^2)
reference_loss = ((yhat == y)^2)

#  Optimization problem
objectives = [reference_loss, fd_loss]
constraints = []
components = [estim, dynamics_model]
loss = nm.PenaltyLoss(objectives, constraints)
problem =  nm.Problem(components, loss)
problem.plot_graph()

# Trainer
optimizer = torch.optim.Adam(problem.parameters(), lr=0.1)

logger =  nm.BasicLogger(args=None, savedir='test', verbosity=1, stdout="nstep_dev_"+
    reference_loss.output_keys[0])

simulator =  nm.OpenLoopSimulator(
    problem, train_loop, dev_loop, test_loop, eval_sim=True)

visualizer =  nm.VisualizerOpen(dynamics_model, 1, 'test', training_visuals=False,
    trace_movie=False)

callback =  nm.SysIDCallback(simulator, visualizer)

trainer = nm.Trainer(
    problem,
    train_data,
    dev_data,
    test_data,
    optimizer,
    callback=callback,
    epochs=100,
    eval_metric="nstep_dev_"+reference_loss.output_keys[0],
    train_metric="nstep_train_loss",
    dev_metric="nstep_dev_loss",
    test_metric="nstep_test_loss",
    logger=logger,
)
best_model = trainer.train()
best_outputs = trainer.test(best_model)
```

Listing 2: Example System Identification implementation in Neuromancer syntax.

Differentiable System Identification codebase List of open-source system ID examples in Neuromancer: `https://github.com/pnnl/neuromancer/tree/master/examples/system_identification`

List of Neuromancer classes required to build constrained systen identification models for dynamical systems:

- dataset - classes for instantiating Pytorch dataloaders with training evaluation and testing samples

https://github.com/pnnl/neuromancer/blob/master/neuromancer/dataset.py

- dynamics - classes parametrizing system models to be differentiated: https://github.com/pnnl/neuromancer/blob/master/neuromancer/dynamics.py
- estimators - classes parametrizing estimation of the initial conditions for dynamics models: https://github.com/pnnl/neuromancer/blob/master/neuromancer/estimators.py
- constraints - classes for defining constraints and custom physics-informed loss function terms https://github.com/pnnl/neuromancer/blob/master/neuromancer/constraint.py
- loss - class aggregating all instantiated constraints and loss terms in a scalar-valued function suitable for backpropagation-based training https://github.com/pnnl/neuromancer/blob/master/neuromancer/loss.py
- problem - class agrregating trainable components (policies, dynamics, estimators) with loss functions in a differentiable computational graph representing the underlying constrained optimization problem https://github.com/pnnl/neuromancer/blob/master/neuromancer/problem.py
- trainer - class for optimizing the problem https://github.com/pnnl/neuromancer/blob/master/neuromancer/trainer.py

4.3 Differentiable Predictive Control

Differentiable predictive control (DPC) method allows us to learn control policy parameters directly by backpropagating model predictive control (MPC) objective function and constraints through the learned digital twin model.

DPC Concept The conceptual methodology shown in Fig. 6 consists of two main steps. In the first step, we perform system identification by learning the unknown parameters of differentiable digital twins described in previous sections. In the second step, we close the loop by combining the digital twin models with control policy, parametrized by neural networks, obtaining a differentiable closed-loop dynamics model. This closed-loop model now allow us to use automatic differentiation (AD) to solve the parametric optimal control problem by computing the sensitivities of objective functions and constraints to changing problem parameters such as initial conditions, boundary conditions, and parametric control tasks such as time-varying reference tracking.

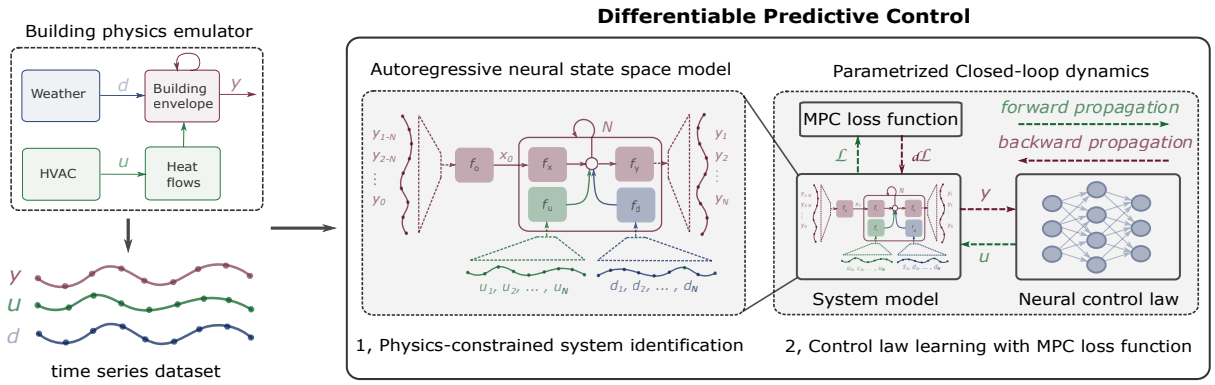


Figure 6: Conceptual overview of Differentiable Predictive Control (DPC) methodology. Step 1: physics-constrained system identification of a digital twin model. Step 2: Learning neural control policy by differentiating of the MPC objective and constraints through the system model.

DPC Problem Formulation Formally we can formulate the DPC problem as a following parametric optimal control problem:

$$\min_{\mathbf{W}} \frac{1}{mN} \sum_{i=1}^{m} \sum_{k=0}^{N-1} \left(\ell_{\texttt{MPC}}(\mathbf{x}_k^i, \mathbf{u}_k^i, \mathbf{r}_k^i) + p_x(h(\mathbf{x}_k^i, \mathbf{p_h}_k^i)) + p_u(g(\mathbf{u}_k^i, \mathbf{p_g}_k^i)) \right) \tag{20a}$$

$$\text{s.t. } \mathbf{x}_{k+1}^i = \mathbf{f}(\mathbf{x}_k^i, \mathbf{u}_k^i), \ k \in \mathbb{N}_0^{N-1} \tag{20b}$$

$$\mathbf{u}_k^i = \pi_{\mathbf{W}}(\mathbf{x}_k^i, \boldsymbol{\xi}_k^i) \tag{20c}$$

$$\mathbf{x}_0^i \in \mathbb{X} \subset \mathbb{R}^{n_x} \tag{20d}$$

$$\boldsymbol{\xi}_k^i = \{\mathbf{r}_k^i, \mathbf{p_h}_k^i, \mathbf{p_g}_k^i\} \in \Xi \subset \mathbb{R}^{n_\xi} \tag{20e}$$

The DPC loss function is composed of the parametric MPC objective $\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k) : \mathbb{R}^{n_x+n_u+n_r} \to \mathbb{R}$, and penalties of parametric constraints $p_x(h(\mathbf{x}_k, \mathbf{p_h}_k) : \mathbb{R}^{n_h+n_{p_h}} \to \mathbb{R}$, and $p_u(g(\mathbf{u}_k, \mathbf{p_g}_k)) : \mathbb{R}^{n_g+n_{p_g}} \to \mathbb{R}$. The MPC objective $\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k)$ is a differentiable function representing the control performance metric such as the following reference tracking with control action minimization in the quadratic form:

$$\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k) = ||\mathbf{x}_k - \mathbf{r}_k||_{Q_r}^2 + ||\mathbf{u}_k||_{Q_u}^2 \tag{21}$$

with reference states $\mathbf{r}_k$, and $\|\mathbf{a}\|_Q^2 = \mathbf{a}^T Q \mathbf{a}$ the weighted squared 2-norm. The control parameters $\boldsymbol{\xi}$ are sampled from the synthetically generated training dataset Ξ, where m represents the total number of parametric scenario samples, and i denotes the index of the sample.

DPC Problem Solution A main advantage of having a differentiable closed-loop dynamics model, control objective function, and constraints in the DPC problem formulation (20) is that it allows us to use automatic differentiation (backpropagation through time [9]) to directly compute the policy gradient. In particular, by representing the problem (20) as a computational graph and leveraging the chain rule, we can directly compute the gradients of the loss function $\mathcal{L}_{\texttt{DPC}}$ w.r.t. the policy parameters $\mathbf{W}$ as follows:

$$\begin{aligned} \nabla_{\mathbf{W}} \mathcal{L}_{\texttt{DPC}} = \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{W}} + \frac{\partial p_x(h(\mathbf{x}, \mathbf{p_h}))}{\partial \mathbf{W}} + \frac{\partial p_u(g(\mathbf{u}, \mathbf{p_g}))}{\partial \mathbf{W}} = \\ \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \\ \frac{\partial p_x(h(\mathbf{x}, \mathbf{p_h}))}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \frac{\partial p_u(g(\mathbf{u}, \mathbf{p_g}))}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} \end{aligned} \tag{22}$$

Where $\frac{\partial \mathbf{u}}{\partial \mathbf{W}}$ represent partial derivatives of the neural policy outputs w.r.t. its weights that are typically being computed in deep learning applications via backpropagation. The advantage of having gradients (22) is that it allows us to use scalable stochastic gradient optimization algorithms such as AdamW [1] to solve the corresponding parametric optimal control problem (20) by direct offline optimization of the neural control policy. In practice, we can compute the gradient of the DPC problem by using automatic differentiation frameworks such as Pytorch [2].

DPC Policy Optimization Algorithm The DPC policy optimization algorithm is summarized in Algorithm 3. The differentiable system dynamics model is required to instantiate the computational graph of the DPC problem The policy gradients $\nabla_{\mathbf{W}} \mathcal{L}_{\texttt{DPC}}$ are obtained by differentiating the DPC loss function $\mathcal{L}_{\texttt{DPC}}$ over the distribution of initial state conditions and problem parameters sampled from the given training datasets $\mathbb{X}$ and Ξ, respectively. The computed policy gradients now allow us to perform direct policy optimization via a gradient-based optimizer $\mathbb{O}$. Thus the presented procedure introduces a generic approach for data-driven solution of model-based parametric optimal control problem (20) with constrained neural control policies.

From a reinforcement learning (RL) perspective, the DPC loss $\mathcal{L}_{\texttt{DPC}}$ can be seen as a reward function, with $\nabla_{\mathbf{W}} \mathcal{L}_{\texttt{DPC}}$ representing a deterministic policy gradient. The main difference compared with actor-critic RL algorithms is that in DPC the reward function is fully parametrized by a closed-loop system dynamics model, control objective, and constraints penalties. The model-based approach avoids approximation errors in reward functions making DPC more sample efficient than model-free RL algorithms.

Algorithm 3 DPC policy optimization.

input training datasets of sampled initial conditions $\mathbb{X}$ and problem parameters Ξ
input differentiable digital twin model
input DPC loss $\mathcal{L}_{\texttt{DPC}}$
input optimizer $\mathbb{O}$
differentiate DPC loss $\mathcal{L}_{\texttt{DPC}}$ to obtain the policy gradient $\nabla_{\mathbf{W}}\mathcal{L}_{\texttt{DPC}}$
learn policy $\pi_{\mathbf{W}}$ via optimizer $\mathbb{O}$ using gradient $\nabla_{\mathbf{W}}\mathcal{L}_{\texttt{DPC}}$
return optimized policy $\pi_{\mathbf{W}}$

DPC Neuromancer Syntax example The following code illustrates the implementation of Differentiable Predictive Control in Neuromancer.

```
# Tutorial example for Differentiable Predictive Control

import torch
import neuromancer as nm
import slim
import numpy as np

    """
    # # #  Dataset
    """
    #  randomly sampled input output trajectories for training
    sequences = {
        "Y_max": xmax*np.ones([nsim, nx]),
        "Y_min": xmin*np.ones([nsim, nx]),
        "U_max": umax*np.ones([nsim, nu]),
        "U_min": umin*np.ones([nsim, nu]),
        "Y": 3*np.random.randn(nsim, nx),
    }
    nstep_data, loop_data, dims = nm.get_sequence_dataloaders(sequences, args.nsteps)
    train_data, dev_data, test_data = nstep_data
    train_loop, dev_loop, test_loop = loop_data

  """
    # # #  System model and Control policy
    """
    # Fully observable estimator as identity map: x0 = Yp[-1]
    # x_0 = Yp
    # Yp = [y_-N, ..., y_0]
    estimator = nm.estimators.FullyObservable({**dims, "x0": (nx,)},
                                        nsteps=args.nsteps,  # future window Nf
                                           window_size=1,  # past window Np <= Nf
                                           input_keys=["Yp"],
                                           name='est')
    # full state feedback control policy
    # Uf = p(x_0)
    # Uf = [u_0, ..., u_N]
    activation = nm.activations['relu']
    linmap = slim.maps['linear']
    block = nm.blocks.MLP
    policy = nm.policies.MLPPolicy(
        {f'x0_{estimator.name}': (nx,), **dims},
        nsteps=args.nsteps,
        bias=args.bias,
        linear_map=linmap,
        nonlin=activation,
        hsizes=[args.nx_hidden] * args.n_layers,
        input_keys=[f'x0_{estimator.name}'],
        name='pol',
    )

    # A, B, C linear maps
    fu = slim.maps['linear'](nu, nx)
```

```
    fx = slim.maps['linear'](nx, nx)
    fy = slim.maps['linear'](nx, ny)
    # LTI SSM
    # x_k+1 = Ax_k + Bu_k
    # y_k+1 = Cx_k+1
    dynamics_model = nm.dynamics.BlockSSM(fx, fy, fu=fu, name='mod',
                                          input_key_map={'x0': f'x0_{estimator.name}',
                                                         'Uf': f'U_pred_{policy.name}'})

    # model matrices values
    A = torch.tensor([[1.2, 1.0],
                      [0.0, 1.0]])
    B = torch.tensor([[1.0],
                      [0.5]])
    C = torch.tensor([[1.0, 0.0],
                      [0.0, 1.0]])
    dynamics_model.fx.linear.weight = torch.nn.Parameter(A)
    dynamics_model.fu.linear.weight = torch.nn.Parameter(B)
    dynamics_model.fy.linear.weight = torch.nn.Parameter(C)
    # fix model parameters
    dynamics_model.requires_grad_(False)


    """
    # # #  DPC objectives and constraints
    """
    u = nm.variable(f"U_pred_{policy.name}")
    y = nm.variable(f"Y_pred_{dynamics_model.name}")
    # constraints bounds variables
    umin = nm.variable("U_minf")
    umax = nm.variable("U_maxf")
    ymin = nm.variable("Y_minf")
    ymax = nm.variable("Y_maxf")

    # objectives
    action_loss = args.Qu * ((u == 0.) ^ 2)  # control penalty
    regulation_loss = args.Qx * ((y == 0.) ^ 2)  # target posistion

    # constraints
    state_lower_bound_penalty = args.Q_con_x*(y > ymin)
    state_upper_bound_penalty = args.Q_con_x*(y < ymax)
    inputs_lower_bound_penalty = args.Q_con_u*(u > umin)
    inputs_upper_bound_penalty = args.Q_con_u*(u < umax)

    objectives = [regulation_loss, action_loss]
    constraints = [
        state_lower_bound_penalty,
        state_upper_bound_penalty,
        inputs_lower_bound_penalty,
        inputs_upper_bound_penalty,
    ]

    """
    # # #  DPC problem = objectives + constraints + trainable components
    """
    # data (y_k) -> estimator (x_k) -> policy (u_k) -> dynamics (x_k+1, y_k+1)
    components = [estimator, policy, dynamics_model]
    # create constrained optimization loss
    loss = nm.get_loss(objectives, constraints)
    # construct constrained optimization problem
    problem = nm.Problem(components, loss)
    # plot computational graph
    problem.plot_graph()

    """
    # # #  DPC trainer
    """

```

```
    optimizer = torch.optim.AdamW(problem.parameters(), lr=args.lr)

    trainer = nm.Trainer(
        problem,
        train_data,
        dev_data,
        test_data,
        optimizer,
        epochs=args.epochs,
        train_metric="nstep_train_loss",
        dev_metric="nstep_dev_loss",
        test_metric="nstep_test_loss",
        eval_metric='nstep_dev_loss',
    )
    # Train control policy
    best_model = trainer.train()
    best_outputs = trainer.test(best_model)
```

Listing 3: Example DPC implementation in Neuromancer syntax.

DPC codebase List of published DPC code examples:
`https://github.com/pnnl/neuromancer/tree/DPC`

List of Neuromancer classes required to build DPC:

- dataset - classes for instantiating Pytorch dataloaders with training evaluation and testing samples `https://github.com/pnnl/neuromancer/blob/master/neuromancer/dataset.py`
- policies - classes parametrizing control feedback laws: `https://github.com/pnnl/neuromancer/blob/master/neuromancer/policies.py`
- dynamics - classes parametrizing system models to be differentiated: `https://github.com/pnnl/neuromancer/blob/master/neuromancer/dynamics.py`
- estimators - classes parametrizing estimation of the initial conditions for dynamics models: `https://github.com/pnnl/neuromancer/blob/master/neuromancer/estimators.py`
- constraints - classes for defining constraints and custom physics-informed loss function terms `https://github.com/pnnl/neuromancer/blob/master/neuromancer/constraint.py`
- loss - class aggregating all instantiated constraints and loss terms in a scalar-valued function suitable for backpropagation-based training `https://github.com/pnnl/neuromancer/blob/master/neuromancer/loss.py`
- problem - class agrregating trainable components (policies, dynamics, estimators) with loss functions in a differentiable computational graph representing the underlying constrained optimization problem `https://github.com/pnnl/neuromancer/blob/master/neuromancer/problem.py`
- trainer - class for optimizing the problem `https://github.com/pnnl/neuromancer/blob/master/neuromancer/trainer.py`

5 Neuromancer user documentation

README file: `https://github.com/pnnl/neuromancer#readme`
References: List of academic publications documenting technical details of the developed modeling methods and learning algorithms are provided in the **README**.

5.1 Examples

Neuromancer examples include: user interface tutorials, system identification, parametric constrained optimization, and differentiable predictive control for model-based control policy learning.
Set of open-source examples: `https://github.com/pnnl/neuromancer/tree/master/examples`

5.2 Installation instructions

Step-by step installation procedure is provided in the **README**.

For Ubuntu users the simplest way to install NeuroMANCER dependencies is via the env.yml file:

```
$ conda env create -f env.yml
$ conda activate neuromancer
(neuromancer) $
```

Install dependencies using Conda:

```
$ conda create -n neuromancer python=3.10.4
$ conda activate neuromancer
(neuromancer) $ conda config --add channels conda-forge
(neuromancer) $ conda install pytorch cudatoolkit=10.2 -c pytorch
(neuromancer) $ conda install scipy numpy matplotlib scikit-learn pandas dill mlflow pydot=1.4.2
(neuromancer) $ conda install pyts numba networkx plum-dispatch
(neuromancer) $ conda install -c anaconda pytest hypothesis
```

Install Neuromancer ecosystem:

```
(neuromancer) user@machine:~$ cd ../psl
(neuromancer) user@machine:~$ python setup.py develop
(neuromancer) user@machine:~$ cd ../slim
(neuromancer) user@machine:~$ python setup.py develop
(neuromancer) user@machine:~$ cd ../neuromancer
(neuromancer) user@machine:~$ python setup.py develop
```

To install torch-scatter you will need to know the pytorch and cuda versions on your conda environment:

```
$ (neuromancer) python -c "import torch; print(torch.__version__)"
1.12.1
$ (neuromancer) python -c "import torch; print(torch.version.cuda)"
10.2
(neuromancer) $ pip install torch-scatter -f https://data.pyg.org/whl/torch-1.12.1+cu102.html
```

The parametric programming examples have additional package dependencies for benchmarking against traditional constrained optimization solvers, e.g., cvxpy

```
(neuromancer) user@machine:~$ conda install cvxpy cvxopt seaborn
(neuromancer) user@machine:~$ pip install casadi
```

References

[1] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[3] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[4] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[5] Aaron Tuor, Jan Drgona, and Draguna Vrabie. Constrained neural ordinary differential equations with stability guarantees. *arXiv preprint arXiv:2004.10883*, 2020.

[6] Elliott Skomski, Soumya Vasisht, Colby Wight, Aaron Tuor, Ján Drgoňa, and Draguna Vrabie. Constrained block nonlinear neural dynamical models. In *2021 American Control Conference (ACC)*, pages 3993–4000. IEEE, 2021.

[7] Ján Drgoňa, Aaron R Tuor, Vikas Chandan, and Draguna L Vrabie. Physics-constrained deep learning of multi-zone building thermal dynamics. *Energy and Buildings*, 243:110992, 2021.

[8] Ercan Atam and Lieve Helsen. Control-oriented thermal modeling of multizone buildings: Methods and issues: Intelligent control of a building system. *IEEE Control Systems Magazine*, 36(3):86–111, 2016.

[9] GV Puskorius and LA Feldkamp. Truncated backpropagation through time and Kalman filter training for neurocontrol. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 4, pages 2488–2493. IEEE, 1994.