Deep implicit layers and applications

MVA master's internship

Matthieu Blanke

Supervised by Marc Lelarge

September 2021

Abstract

Implicit differentiation allows one to compute gradients in a challenging setting where the output may not be explicitly defined in terms of the input. Recent work led to the definition of deep implicit layers [**?**] which allow to integrate implicit operations into an automatic differentiation framework, paving the way for new gradient-based optimization approaches. We propose applications of implicit layers to a variety of new problems including neural differential equations, neural optimal control, system identification and experimental design. We test these methods with experiments and we compare our results with analytical solutions and already-existing methods.

Contents

1 Introduction

Deep learning is at the core of artificial intelligence and gave rise to extremely powerful technologies in a large number of fields such as self-driving cars [**?**], natural language processing [**?**] or board game programming [**?**]. The adjective "deep" refers to the structure of the underlying mathematical entity : deep learning is based on artificial neural networks, which are optimizable functions made of a stack of elementary blocks called layers. The ability to differentiate through a chain of layers with the well-known backpropagation algorithm [**?**] is the crucial motivation for these architectures, as it allows neural nets to be used in a gradient-based optimization framework. The great expressiveness of neural nets [**?**] combined with powerful gradient descent algorithms [**?**] make deep learning a very effective tool for a large number of tasks.

One of the central notions of modern deep learning is hence that of layers. Neural networks are built by stacking a chain of layers together, allowing for complex architectures and expressive models. These layers include fully connected layers, convolutional layers, transformers, etc [**?**][**?**]. Traditionally, the layers are made of well known functions such as linear maps, followed by nonlinear maps such as tanh, the sigmoid, or ReLU activation [**?**]. Since the corresponding derivatives are also well known, the neural nets can be differentiated through by gradient backpropagation.

In classical layers, the output is obtained by directly applying a usual function to an input. Implicit layers differ from this since they are not defined in terms of simple functions of the input but rather in terms of implicit, possibly complex conditions that the input and the output must satisfy. For example, the output can be defined as the value of some fixed point equation, or as the solution of an ordinary differential equation.

Still, implicit layers are designed to fit into an automatic differentiation function and hence satisfy differentiation properties. The central point of implicit layers is to establish differentiation formulae, so that the layer can be integrated in a gradient propagation framework just like "explicit" layers [**?**]. Deep implicit layers hence fit into the more general paradigm of differentiable programming which is the combination of differentiable algorithmic modules, ensuring an end-to-end differentiable model suitable for gradient-based optimization [**?**].

In this work, we apply deep implicit layers of different types to a variety of tasks. We will first focus on neural differential equations introduced by Chen *et al.* [**?**], which define an implicit neural layer. Since most physical systems can be described by differential equations, the ability to model them with a neural network allows us to cast differential systems into optimization programs. Then, we focus on system identification and experimental design, where we are able to use automatic differentiation through a model to find the optimal inputs. Our contributions are summarized as follows.

Implicit layers and automatic differentiation In section **??** we first define implicit layers and set the theoretical framework for automatic differentiation, and then introduce the main implicit layers used in the following sections together with the corresponding backward gradient formulae.

Learning dynamics from physical principles In section **??** we apply neural differential equations to an elementary physics problem and recover a theoretical uniqueness result by numerical optimization.

Neural optimal control In section **??** we demonstrate a connection between neural differential equations and optimal control, and use it to numerically solve two classical optimal control problems.

System identification In section **??** we leverage automatic differentiation of singular value decomposition to provide a numerical solver for an optimal control problem used in system identification.

Experimental design In section **??** we introduce a novel approach to the Bayesian experimental design problem based on automatic differentiation and illustrate it on a toy model.

2 Implicit layers and automatic differentiation

Automatic differentiation (AD) refers to a set of techniques allowing to evaluate the derivative of an operation, a function or a computer program automatically. The derivatives of elementary operations such as the sum, the product, trigonometric functions and other usual functions are well-known and implemented. As we will see, a program made of a sequence of such operations may be differentiated through by making use of the chain rule of differentiation and by computing the derivatives of these functions recursively.

One of the main motivations of AD is gradient-based optimization. Assume we want to minimize a scalar, differentiable function L of some input $x \in \mathbb{R}^d$:

$$\min_{x \in \mathbb{R}^d} \quad L(x). \tag{2.1}$$

Then, gradient-based algorithms proceed essentially by iterations of the form

$$x_{t+1} = x_t - \eta \nabla L(x), \qquad \eta > 0. \tag{2.2}$$

Functions endowed with AD can hence have their gradient computed automatically, making gradient descent very efficient for the minimization program (**??**).

Contribution In this section we give a theoretical framework for implicit layers inspired from [**?**]. We recall the definition of some implicit layers that we will use in the following sections together with their adjoint formulae.

2.1 The chain rule

Differentiable programming consists in stacking up automatically differentiable functions and in propagating the derivatives along this chain of operations. The chain rule expresses the derivative of one of the functions in terms of the derivatives of the next one.

Assume we want to compte the gradient of some scalar quantity $L \in \mathbb{R}$ with respect to $x \in \mathbb{R}^m$, and that L is a function of some intermediate differentiable quantity $y(x) \in \mathbb{R}^n$:

$$L(x) = L\big(y(x)\big). \tag{2.3}$$

Formally, using the two notations $L(x)$ and $L(y)$ simultaneously is incorrect as they refer to two different mathematical functions. However, we will use this notation shorthand for the sake of simplicity.

The chain rule of differentiation expresses the derivatives of L with respect to x in terms of the derivatives with respect to y as follows :

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j}. \tag{2.4}$$

Defining the Jacobians $\partial L/\partial x$, $\partial L/\partial y$ of $L(x)$ and $L(y)$ and the Jacobian $\partial y/\partial x$ of $y(x)$, equation (**??**) admits the following matrix notation

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}. \tag{2.5}$$

Transposing (**??**) we obtain the gradient chain rule :

$$\nabla_x L = \frac{\partial y}{\partial x}^{\top} \nabla_y L, \tag{2.6}$$

The previous equations connect the gradients of L with respect to both the input and the output of the function $y(x)$. This means that if one is able to compute the Jacobian of y at x, and the gradient

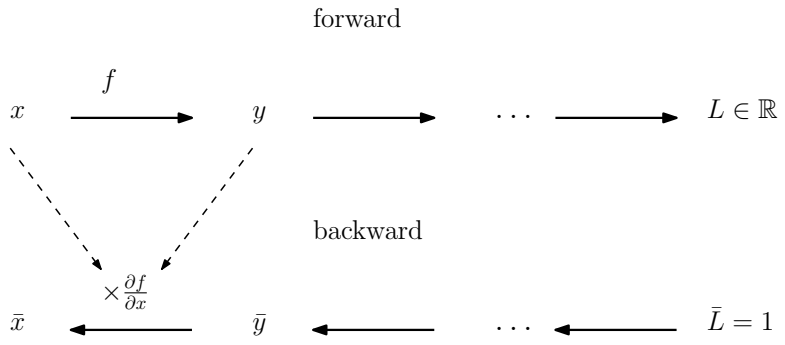


Figure 1: Reverse mode accumulation for AD. Once the forward pass is peformed, the backward pass can use some of the computed variables to backpropagate the adjoint.

of L with respect to one of the quantities x or y then one can compute the other gradient by a simple operation. In particular, there are two ways to read equation (**??**). First, one can fix a scalar input x_i and run the computation forward, then repeat for each input. This is known as **forward-mode accumulation**. Alternatively, **reverse-mode accumulation** fixes one scalar output and runs the computational graph backwards. The latter technique is more advantageous when the dimension of the output is smaller than that of the input, which is clearly the case in deep learning where the loss function is scalar and the number of parameters is large. We will therefore focus on reverse-mode accumulation in the following.

Reverse-mode accumulation works as follows [**?**]. In a feedforward computational graph, the **forward pass** consists in computing and storing the outputs $x, y(x), \dots, L$ recursively. After that, the **backward pass**, often refered to as backpropagation, computes the gradients $\nabla_L L = 1, \dots, \nabla_y L, \nabla_x L$ backwards, using $\nabla_y L$ to compute $\nabla_x L$. These two steps result in the the computation of the value of the function L and that of its gradient with respect to the inputs respectively. Reverse-mode accumulation is illustrated in figure **??**.

2.2 The notion of layer

We call a **layer** a map $y(x)$ for which one can implement a forward pass computing $y(x)$ as a function of x and a backward pass computing $\nabla_x L$ as a function of $\nabla_y L$, of $y(x)$ and possibily of additional variables stored during the forward pass. Hence, a layer is defined as a function that is compatible with reverse-mode accumulation and that can be integrated in an AD framework.

This definition extends that of artificial neural net layers. In the latter case, $x \mapsto y(x)$ is often made of a linear map and an activation function, for which the derivatives are well known. If $y(x)$ is a more complex function however, it may be challenging to directly compute the derivative of y with respect to x. Yet, interestingly, one can often still derive the Jacobian or at least a relation between the gradients $\nabla_x L$ and $\nabla_y L$, and hence perform backpropagation.

Throughout this work, we will adopt the following standard notation

$$\bar{x} := \frac{\partial L}{\partial x} = \nabla_x L^\top \tag{2.7}$$

for the **adjoint** of x in the computation of L. This notation stresses the fact that the fixed scalar output L does not matter in the definition of the layer since the rules of forward and backward propagation work independently of the final output. The chain rule may be expressed in terms of adjoints in the following way

$$\bar{x} = \bar{y} \frac{\partial y}{\partial x}. \tag{2.8}$$

Another important point in the definition of layer is the fact that the forward and backward computations are separated, the only condition being that the value of the output $y(x)$ may be required for

the backward pass. This has the fundamental consequence that the computational burden of calculating $y(x)$ does not intervene in the backward computation. Typically, $y(x)$ might be hard to compute and involve an iterative algorithm, but computing $\bar{x}$ might be a expressed as a simple function of $\bar{y}$ in an adjoint equation. In this case, deriving an adjoint equation keeps one from storing varibales and backpropagating through numerous and possibly non-differentiable steps as one would do naively.

2.3 A one-dimensional example

A common layer in neural networks is the fullly connected layer, *i.e.* a linear map with an additive bias, which is often combined with a nonlinear function like tanh. For the sake of simplicity, we restrict ourselves to one-dimensional variables. Assume the following relation in $\mathbb{R}$:

$$y = \tanh(wx + b), \tag{2.9}$$

where the weight w and the bias b are fixed. Then the forward pass is simply achieved by multiplying by w, adding b and applying the function tanh. The backward pass multiplies the adjoint of y by the Jacobian of $y(x)$, which is straightforward to compute

$$y'(x) = w \operatorname{sech}^2(wx + b). \tag{2.10}$$

Since hyperbolic functions are already implemented or easy to implement in most scientific programming languages, both of these passes are straightforward.

Let us now consider an implicit variation of this layer. Following [**?**], we take for example a fixed point layer which underlies so-called deep equilibrium models [**?**]. Assume y is obtained from x by solving the following fixed point equation

$$\tanh(wy + x) = y. \tag{2.11}$$

Unlike equation (**??**), (**??**) is implicit in y in the sense that it does not express y as a usual function of x. Performing the forward pass is hence a mathematical problem per se and may involve complicated numerical methods. Regardless, one can still differentiate through relation (**??**) with respect to x, and obtain

$$y'(x) = w \operatorname{sech}^2(wy + x)y'(x) + \operatorname{sech}^2(wy + x), \tag{2.12}$$

yielding

$$y'(x) = \frac{\operatorname{sech}^2(wy + x)}{1 - w \operatorname{sech}^2(wy + x)}. \tag{2.13}$$

Although the relation defining y is implicit, we can still recover an explicit formula for the Jacobian (**??**) by differentiation. Then, the adjoint of x with respect to some scalar function L can be obtained by the chain rule

$$\bar{x} = \bar{y}y'(x). \tag{2.14}$$

This example highlights the fact that although the output of the layer is defined implicitly, we can still differentiate this implicit equation and recover the Jacobian of the function $y(x)$. We will see in next section a more general result regarding implicit equations layers.

The fixed point layer also illustrates the possibility of separating the forward pass and the backward pass. Indeed, solving the fixed point equation often involves an iterative algorithm such as Newton method [**?**]. However, we can here readily compute the backward pass by simple algebraic operations, regardless of the forward pass computation. In particular, we are free from backpropagating through all the steps of the Newton solver, which would be costly in terms of memory.

2.4 Implicit equation layers

The previous result may be generalized to any implicit relation satistfying the implicit function theorem (see [**?**] for example). Assume that the function $x \in \mathbb{R}^n \mapsto y(x) \in \mathbb{R}^n$ we want to differentiate through is defined by some equation in $\mathbb{R}^n$:

$$\phi\big(x, y(x)\big) = 0. \tag{2.15}$$

In order to backpropagate the gradient from y to x, we want to compute the Jacobian $\partial y/\partial x$. If the assumptions of the theorem are satisfied, we can differente (**??**) with respect to x :

$$\frac{\partial \phi}{\partial x} + \frac{\partial \phi}{\partial y}\frac{\partial y}{\partial x} = 0, \tag{2.16}$$

meaning that

$$\frac{\partial y}{\partial x} = -\left(\frac{\partial \phi}{\partial y}\right)^{-1}\frac{\partial \phi}{\partial x}. \tag{2.17}$$

Note that this corresponds to equation (**??**) in the previous example. Then, by the backpropagation equation (**??**)

$$\bar{x} = -\bar{y}\left(\frac{\partial \phi}{\partial y}\right)^{-1}\frac{\partial \phi}{\partial x}. \tag{2.18}$$

It is important to note that the computation of the right-hand side of this equation does not require to compute and store the Jacobian $\partial\phi/\partial y$ and its inverse, it only requires solving a linear system and then performing a matrix multiplication. There are however two choices for this matter : **(a)** solving n linear systems of size $n \times n$

$$\frac{\partial \phi}{\partial y}\lambda = \frac{\partial \phi}{\partial x} \tag{2.19}$$

then multiplying by the vector $\bar{y}$ of size n or **(b)** solving one linear system of size $n \times n$

$$\frac{\partial \phi}{\partial y}^{\top}\lambda = \bar{y}^{\top} \tag{2.20}$$

and multiplying by the $n \times n$ matrix $\partial\phi/\partial x$. Since L is scalar and x and y are possibly high-dimensional $n \gg 1$, option **(b)** is much more efficient.

Equation (**??**) is refered to as the adjoint equation for equation (**??**). It is a general statement that solving the adjoint problem is of similar complexity as that of solving the forward equation [**?**], as we will see in the next two sections.

2.5 Adjoint of singular value decomposition

Let us now tackle a concrete example of implicit function : singular value decomposition. The singular value decomposition (SVD) of a matrix $A \in \mathbb{R}^{n\times d}$ is defined as

$$A = USV^{\top} \tag{2.21}$$

with

$$U \in \mathbb{R}^{n\times n}, \quad S \in \mathbb{R}^{n\times d} \text{ diagonal}, \quad V \in \mathbb{R}^{d\times d} \tag{2.22}$$

and

$$U^{\top}U = I_n, \qquad V^{\top}V = I_d. \tag{2.23}$$

The matrix A is decomposed as the product of two unitary matrices with a diagonal matrix whose diagonal entries are called singular values of A and are related to the principal components of its columns. This decomposition is extremely common in signal processing and machine learning and we will see an application example in section **??**. Therefore, being able to differentiate through it is a very useful property.

In computer programs, the SVD of a matrix is usually computed by the Householder method which consists in bidiagonalizing the matrix through a series of multiplications [**?**]. Applying AD to these operations is conceivable, but very inconvenient. Instead, it is possible to derive a closed-form adjoint formula for SVD [**?**][**?**].

Let us differentiate (**??**)

$$\mathrm{d}A = \mathrm{d}USV^\top + U\mathrm{d}SV^\top + US\mathrm{d}V^\top, \tag{2.24}$$

and the orthogonality equations (**??**):

$$\mathrm{d}U^\top U + U^\top \mathrm{d}U = 0, \qquad \mathrm{d}V^\top V + V^\top \mathrm{d}V = 0. \tag{2.25}$$

The latter equations impliy that matrices $\mathrm{d}U^\top U$ and $\mathrm{d}V^\top V$ are both anti-symmetric , and in particular have zero diagonals. Now pre-multiplying (**??**) by $U^\top$ and post-multiplying by V gives

$$U^\top \mathrm{d}AV = U^\top \mathrm{d}US + \mathrm{d}S + S\mathrm{d}V^\top V. \tag{2.26}$$

Taking the major diagonal yields

$$\mathrm{d}S = I_{n,d} \circ (U^\top \mathrm{d}AV), \tag{2.27}$$

where $\circ$ denotes the Hadamard product. Equation (**??**) means that the differential of S in A is the linear map $\ell : M \mapsto I_{n,d} \circ (U^\top MV)$. Let $\bar{A}$ and $\bar{S}$ denote the adjoints of A and S respectively. From (**??**), ℓ^* the transpose of ℓ satisfies

$$\bar{A} = \ell^*(\bar{S}). \tag{2.28}$$

By definition, for any matrix M,

$$\mathrm{tr}\big(\ell^*(\bar{S})M^\top\big) = \mathrm{tr}\big(\bar{S}^\top \ell(M)\big). \tag{2.29}$$

From (**??**),

$$\begin{aligned}
\mathrm{tr}\big(\bar{S}^\top \ell(M^\top)\big) &= \mathrm{tr}\big(\bar{S}^\top (I_{n,d} \circ (U^\top MV))\big) && (2.30)\\
&= \mathrm{tr}\big(\bar{S}^\top U^\top MV\big) \qquad \text{because the trace operates on diagonal entries} && (2.31)\\
&= \mathrm{tr}\big(U\bar{S}V^\top M^\top\big). && (2.32)
\end{aligned}$$

It follows that $\ell^*(\bar{S}) = V\bar{S}U^\top$ and hence

$$\bar{A} = V\bar{S}U^\top. \tag{2.33}$$

This formula is already implemented in several AD frameworks such as Torch [**?**].

Singular value decomposition is an example of implicit layers. Indeed, equations (**??**) and (**??**) connect the input A and the output S implicitly, without specifying an explicit way of computing S as a function of A since the orthogonal matrices U and V are not explicitly defined. Starting from these implicit conditions, we differentiated the equations and managed to derive a formula for the differential $\mathrm{d}S$, and hence for the adjoint $\bar{A}$. This result does not depend on the way of computing the actual SVD, which is a great advantage. In particular, it is crucially memory-saving as it keeps us from applying AD through all the steps of the forward computation of SVD. Note also that the adjoint equation we obtained is of the same type as the forward equation (**??**), and therefore does not bring additional computational complexity.

2.6 Differential equations

Let us see how differential equations can define an implicit layer and how they can be differentiated through. An ordinary differential equation (ODE) is an equation of the form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f\big(x(t), t, \theta\big), \qquad t \in [0, T] \tag{2.34}$$

where $T > 0$ is a time horizon, $\mathrm{d}/\mathrm{d}t$ denotes the time derivative, $x : \mathbb{R} \to \mathbb{R}^d$ is the trajectory, and the flow of the ODE f is a $\mathbb{R}^d$-valued function of the current state $x(t)$, of the time t and of some parameter vector $\theta \in \mathbb{R}^p$.

In the following, we will restrict ourselves to functions f that do not depend on t explicitly, in the sense that the differential equation takes the following simpler form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x, \theta). \tag{2.35}$$

This type of ODE is called an autonomous system as the evolution of the trajectory depends only on the current state $x(t)$. We will assume that the ODEs satisfy the assumption of the Cauchy-Lipschitz theorem [**?**] so that their solutions are always unique.

2.6.1 ODEs as implicit layers

Let $a \in \mathbb{R}^d$ be an input value and x_θ be the solution of the following Cauchy problem

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= f(x, \theta) \\ x(0) &= a. \end{aligned} \tag{2.36}$$

We define an ODE layer $y(a, \theta)$ of two inputs a and θ as a function of the form

$$y(a, \theta) = x_\theta(T). \tag{2.37}$$

As we defined it, an ODE layer evaluated at (a, θ) is obtained by integrating a differential system starting at a and whose flow f_θ is parametrized by θ. Note that this definition is implicit as the output depends on the output through the resolution of an ODE. Integrating a Cauchy-Lipschitz problem is a task of a certain complexity itself, and the ODE layer (**??**) is defined irrespective of this integration step. As we will see, we can derive an adjoint equation for this layer with respect to both of its inputs, making ODE layers fully implementable in an AD pipeline. This section is based on the paper of Chen *et al.* that introduced neural ODEs [**?**].

2.6.2 The adjoint ODE

As before, assume some scalar quantity L depends on the output y of the layer. We first want to determine how to backpropagate the gradient from the output to the input. The trajectory $\big(x(t)\big)$ of the ODE can be seen as a continuum of feedforward infinitesimal layers, where the states $x(s)$ at time $s > t$ are deterministic functions of $x(t)$. Specifically, between depths t and $t + h$, the dynamics transforms $x(t)$ into

$$x(t + h) = x(t) + hf\big(x(t), \theta\big). \tag{2.38}$$

Figure **??** gives a representation of this discretization viewpoint where the trajectory is an infinite sequence of inputs transformations with infinitely small layers of size $\mathrm{d}t = h$.

Applying the chain rule (**??**) between the input and the output of one infinitesimal layer, we obtain

$$\begin{aligned} \bar{x}(t) &= \bar{x}(t + h)\frac{\partial x(t + h)}{\partial x(t)} \\ &= \bar{x}(t + h)\left(I + h\frac{\partial f}{\partial x}\big(x(t), \theta\big)\right). \end{aligned} \tag{2.39}$$

Letting $h \to 0$, the adjoint state $\bar{x}$ satisfies the following differential equation

$$\frac{\mathrm{d}\bar{x}}{\mathrm{d}t} = -\bar{x}\,\frac{\partial f}{\partial x}\big(x(t), \theta\big). \tag{2.40}$$

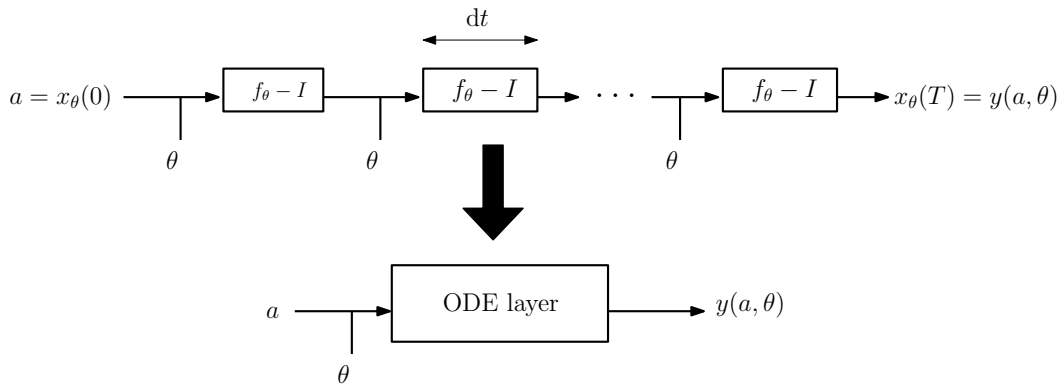


Figure 2: An ODE layer seen as the continuous limit of a series of residual cells.

The adjoint differential equation (**??**) is the continuous limit of the chain rule and the gradient backpropagation for a infinite series of infinitesimal weight-sharing layers.

This ODE allows one to compute the adjoint of the initial state once the forward pass is performed, *i.e.* once $x(t)$ is integrated. Indeed, integrating the adjoint ODE gives $\bar{y}$ as a function of $\bar{z}$:

$$\bar{y} - \bar{a} = \text{solve}\big((\mathbf{??})\big), \tag{2.41}$$

hence propagating the gradient through the ODE with the boundary condition being the adjoint of the output $\bar{y}$. Solving this ODE backwards is the continuous analog of performing backpropagation.

2.6.3 Adjoint of the parameter

The structure of ODE layers allowed us to compute the adjoint of the input as a function of the adjoint of the output. As we will see, we can also leverage the mathematics of differential equations to derive the gradient with respect to the parameter θ.

The scalar function L we are computing the gradient of depends on θ through $y(a, \theta) = x_\theta(T)$, and can more generally been seen as a function of any of the points $x_\theta(t),\ t \in [0, T]$ of the trajectory since the adjoint of the different $x(t)$ are connected by the adjoint ODE (**??**). Therefore, we are interested in the dependence of $x_\theta(t)$ with respect to θ, which is encoded in the sensitivity $\partial x_\theta(t)/\partial\theta$.

The evolution of the state x during an infinitesimal time h is given by (**??**). Let us differentiate with respect to θ :

$$\frac{\partial x}{\partial \theta}(t+h) = \frac{\partial x}{\partial \theta}(t) + h\frac{\partial f}{\partial x}\big(x(t), \theta\big)\frac{\partial x}{\partial \theta}(t) + h\frac{\partial f}{\partial \theta}\big(x(t), \theta\big). \tag{2.42}$$

In the limit $h \to 0$, we obtain the following ODE

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial x}{\partial \theta} = \frac{\partial f}{\partial x}\big(x(t), \theta\big)\frac{\partial x}{\partial \theta}(t) + \frac{\partial f}{\partial \theta}\big(x(t), \theta\big). \tag{2.43}$$

Now, the derivative of L with respect to θ is [1]

$$\bar{\theta} := \frac{\partial L}{\partial \theta} = \bar{x}(T)\frac{\partial x(T)}{\partial \theta} \tag{2.44}$$

Since the sensitivity at $t = 0$ is zero and the dynamics of the sensitivity is known (**??**), we can compute (**??**) by propagating this evolution from 0 to T :

$$\bar{\theta} = \int_0^T \frac{\mathrm{d}}{\mathrm{d}t}\left(\bar{x}(t)\frac{\partial x(t)}{\partial \theta}\right)\mathrm{d}t. \tag{2.45}$$

[1] note that here it would be wrong to write $\bar{\theta} = \bar{x}(t)\partial x(t)\partial\theta$ for $t \neq T$ because a variation of θ yields a variation of the $x(s),\ s > t$.

Differentiating (**??**) with respect to time (or depth) t, we compute :

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\bar{x}(t)\frac{\partial x(t)}{\partial\theta} &= \frac{\mathrm{d}\bar{x}}{\mathrm{d}t}\frac{\partial x(t)}{\partial\theta} + \bar{x}(t)\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial x(t)}{\partial\theta} \\ &= -\bar{x}\,\frac{\partial f}{\partial x}\big(x(t),\theta\big)\frac{\partial x(t)}{\partial\theta} + \bar{x}(t)\left(\frac{\partial f}{\partial x}\big(x(t),\theta\big)\frac{\partial x}{\partial\theta}(t) + \frac{\partial f}{\partial\theta}\big(x(t),\theta\big)\right) \\ &= \bar{x}(t)\frac{\partial f}{\partial\theta}\big(x(t),\theta\big).\end{aligned} \tag{2.46}$$

where we replaced the time derivatives of $\bar{x}(t)$ and $\partial x/\partial\theta$ with equations (**??**) and (**??**). Integrating this quantity allows one to compute the adjoint of θ :

$$\bar{\theta} = \int_0^T \bar{x}(t)\frac{\partial f}{\partial\theta}\big(x(t),\theta\big)\mathrm{d}t, \tag{2.47}$$

provided we can compute the derivative $\partial f/\partial\theta$. This can be achieved automatically when f itself is an AD layer for example. Here again, the adjoint equations (**??**) and (**??**) are integral equations of the same type and complexity as the forward equation (**??**).

When the loss L depends explicitly on points of the trajectory other that $x(T)$, it is possible to derive a more general relation similarly [**?**].

2.6.4 Neural differential equations

Given an ODE with parametric flow f_θ, we derived integral formulas for the adjoints of the input and the parameter. The integrands are functions of the derivatives $\partial f/\partial x$ and $\partial f/\partial\theta$. Even though these derivatives cannot be evaluated analytically, the integration can still be carried out numerically provided f has an AD implementation. In particular this is the case when f is a neural network. Then, the layer defined by the ODE is called a neural ODE [**?**].

Interestingly, a neural ODE can be thought of as the continuous limit of a residual neural net. A residual cell learns residual functions of the activations: $x_{t+1} = x_t + f_\theta(x_t)$ [**?**], as represented on figure **??**. A residual net (or ResNet) is build by stacking residual cells. Assuming that the weights are constant along the depth of the network t and taking the continuous limit, the activations are solutions of the following autonomous system [**?**]:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f_\theta(x). \tag{2.48}$$

Following this analogy, the values of the activations (x_t) are trajectories of the flow f_θ along depth t, hence forming an ODE layer as defined in (**??**).

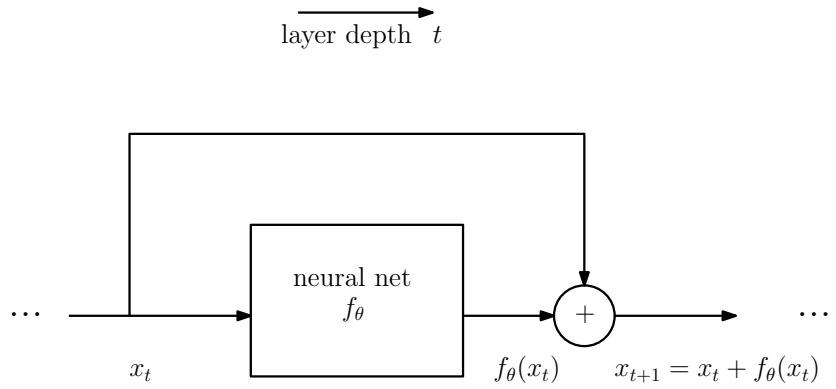


Figure 3: ResNet architecture. The layer f_θ learns the residual of the activations.

3 Learning dynamics from physical principles

Here we propose an application of neural ODEs to a problem of thoeretical physics. Let $T > 0$ be a time period. Which potential energy functions U yield T-isochronism for the one-dimensional physical system $\ddot{x} = -U'(x)$? Under mild assumptions, one can check that the only symmetric potentials satisfying this property are harmonic : $U(x) \propto x^2$.

Contribution In this section we propose to recover this uniqueness result experimentally by training a neural differential equation [**?**] to satisfy global T-periodicity. We also propose a theoretical derivation which simplifies that we found in the litterature [**?**].

3.1 The problem of isochronism

Isochronism is the property for a physical system of having a period that is independent of the motion's amplitude. Such a property allows one to measure time reliably, be it by the means of a pendulum or of a quartz crystal [**?**]. Consider a one-dimensional physical system

$$\ddot{x} = -U'(x), \tag{3.1}$$

where we chose a unitary mass without loss of generality. It is well known that the quadratic potential $U(x) = \frac{1}{2}\omega x^2$ yields a harmonic, isochronic motion of period $T = 2\pi/\omega$. We are interested in the following converse question : given a fixed time $T > 0$, are there other such symmetric functions U ? In mathematical terms, for which symmetric functions U do the dynamics (**??**) yield T-periodicity $X(T) = X(0)$ for **all the trajectories** $X = (x, \dot{x})^\top$, *i.e.* for all initial conditions $X(0)$? It turns out that under mild assumptions, the harmonic potential is the only solution [**?**] :

$$U(x) \propto x^2. \tag{3.2}$$

We apply neural ODEs to address this problem experimentally. We also present a mathematical proof of the result.

3.2 Learning the flow

As stated before, neural ODEs allow us to train the flow of an ODE by imposing a loss function on its trajectory. We hence parametrize the dynamics (**??**) in the following way

$$\ddot{x} = f_\theta(x), \tag{3.3}$$

where the neural network $f_\theta(x)$ approximates the force field $f(x) = -U'(x)$. Our objective is to yield T-periodic trajectories, so we want to optimize

$$\min_\theta \ \|X_\theta(T) - X_\theta(0)\|^2 . \tag{3.4}$$

We give a schematic view of the training pipeline in figure **??**.

Experimental setup In our experiment, we trained our network to the objective (**??**) on a dataset of $N = 100$ random points of the phase portrait $X_i \sim \mathcal{N}(0, 1)$, $1 \leq i \leq N$ and with $T = 2\pi$. We take for f_θ a 2-layer fully connected architecture with width 16 and tanh nonlinearity. We used the Python package `torchdyn` [**?**] which is built on `torchdiffeq` [**?**]. Our implementation is available at `https://github.com/MB-29/neural-dynamics`.

Results Our results are summarized in figure **??**. The circular trajectories centered on the origin in the phase space show that the potential converged to $U(x) = \frac{1}{2}x^2$. Hence, the phase portrait is that of a harmonic oscillator with angular frequency $\omega = 1$, which does equal $2\pi/T$. We further check that the flow converges to the restoring force $f(x) = -x$. Note that we didn't impose any f_θ to be an odd function, or any symmetry assumption. In particular, translating the center of the phase portrait

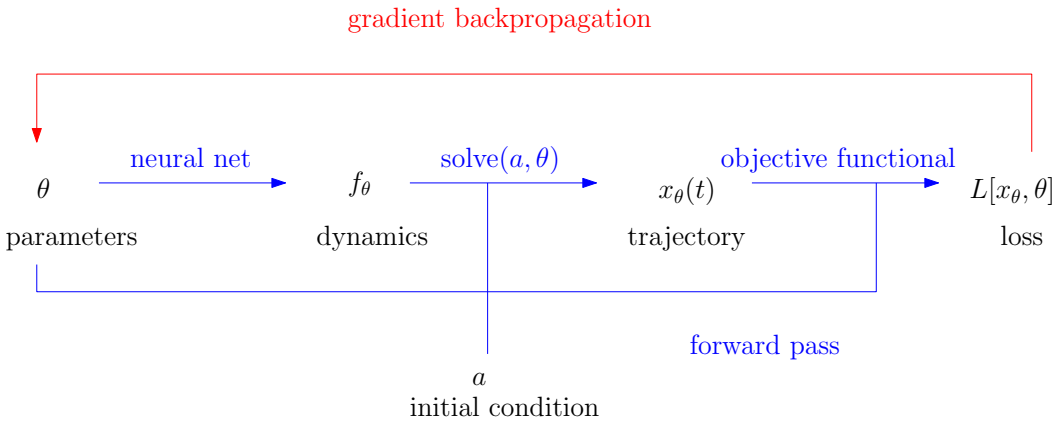


Figure 4: The pipeline for the training of a neural ODE representing the flow of some dynamics.

horizontally would yield the same loss and thus corresponds to an equally valid solution. This makes perfect sense as without any symmetry assumption, harmonic trajectories centered on any point of the real axis satisfy isochronism.

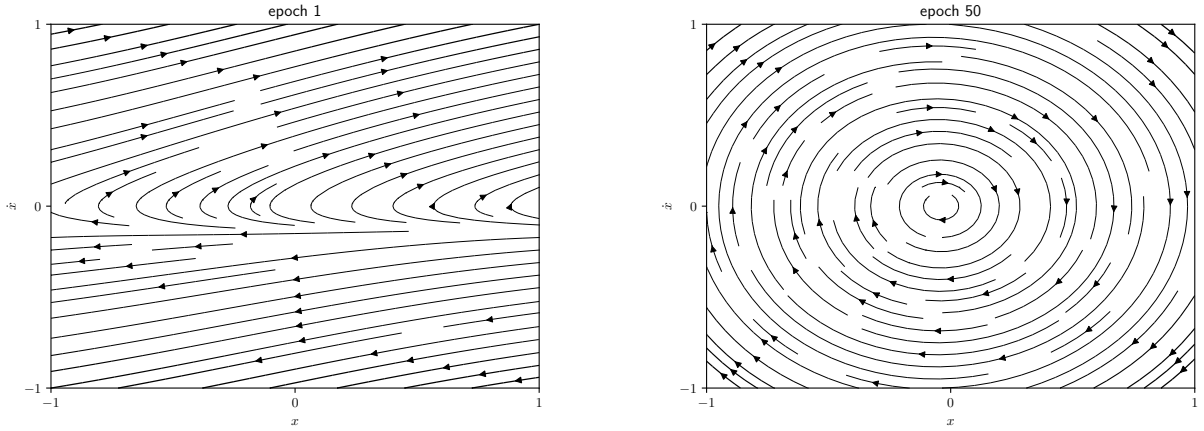


Figure 5: Evolution of the phase portrait after 50 training epochs.

3.3 Theoretical derivation

We prove here that harmonic potentials (**??**) are the only symmetric potentials yielding isochronism. Our computations are inspired by, yet different from that of [**?**], as ours focuses on symmetric potentials.

Let $U(x)$ denote the potential energy. We assume that U is a symmetric function, and is increasing on $\mathbb{R}_+$ (and hence decreasing on $\mathbb{R}_-$). It hence admits a global minimum at 0. The conservative dynamics (**??**) yields trajectories with constant energy

$$E = \frac{1}{2}\dot{x}^2 + U(x). \tag{3.5}$$

Hence, the momentum $\dot{x}$ can be expressed as a function of the position x :

$$\dot{x} = \pm\sqrt{2}\sqrt{E - U(x)}. \tag{3.6}$$

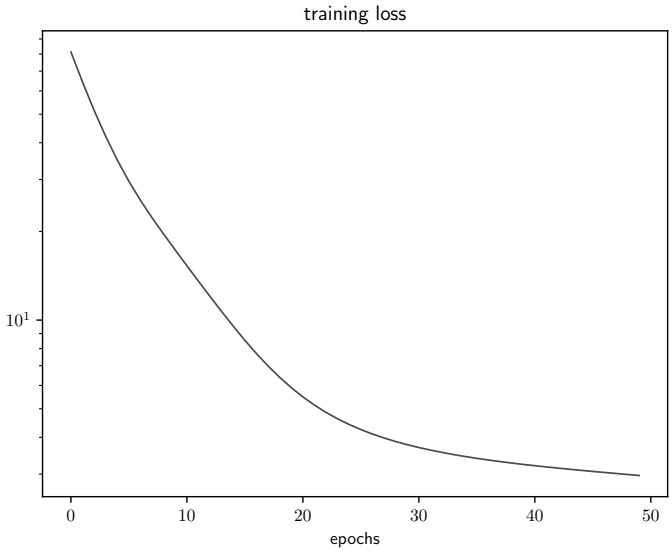


Figure 6: Loss curve for the training of objective (**??**).

Now assume that all trajectories following the dynamics (**??**) are T-periodic. The two extreme points of the trajectory are symmetric with restpect to 0 because the potential is symmetric. The turning points are $x = \pm\, y$, where the amplitude

$$y = y(E) > 0 \tag{3.7}$$

solves

$$\dot{x} = 0 \qquad \text{at} \qquad x = \pm\, y(E), \tag{3.8}$$

or, equivalently,

$$U\big(\pm\, y(E)\big) = E. \tag{3.9}$$

Since the travel time from 0 to y is equal to that from y to 0, the quarter of the period equals

$$T/4 = \int_0^{y(E)} \mathrm{d}t \tag{3.10}$$

$$= \frac{\sqrt{2}}{2} \int_0^{y(E)} \frac{\mathrm{d}x}{\sqrt{E - U(x)}}. \tag{3.11}$$

Changing variable to $u(x) = U(x)/E$ yields

$$T/4 = \frac{\sqrt{2}}{2} \int_0^1 x'(uE)\sqrt{uE}\frac{\mathrm{d}u}{\sqrt{u(1-u)}}, \tag{3.12}$$

where $x(U)$ is the inverse function of $U(x)$. Let $g(U) = x'(U)\sqrt{U}$. Since there is a one-to-one correspondance (**??**) between the oscillator amplitude y and the energy of the system E, isochronism means that the integral

$$\int_0^1 g(uE)\frac{\mathrm{d}u}{\sqrt{u(1-u)}} \tag{3.13}$$

is constant with respect to $E \in \mathbb{R}_+^*$, which implies that g is constant. Indeed, if $v := \inf\{u, g(u) \neq g(0)\} < +\infty$, taking $E = v + \varepsilon$ with small $\varepsilon > 0$ and $E \to 0$ yield different values for the integral (**??**). Therefore $x'(U) \propto 1/\sqrt{U}$ and

$$U(x) \propto x^2. \tag{3.14}$$

3.4 Discussion

This section showcases a novel application of neural ODEs, where we trained a neural network to be the flow of a differential system by imposing a loss on its trajectories. We recovered a theoretical result from mechanics experimentally, for which we also provided an analytical proof. Our experimental result perfectly matches the theoretical one although we didn't impose any symmetry constraint in our experiment, meaning that the harmonic oscillator is only a local minimum of the loss (**??**) and that the neural net could have converged to other phase portraits.

Remarkably, our the training objective of this problem is completely implicit. We defined a loss over the trajectories of a second order ODE, which are implicitly defined with respect to the underlying neural layer. It would be interesting to conduct this kind of approach in other fields, replacing time invariance with other types of symmetry.

4 Neural optimal control

As seen in section **??**, automatic differentiation and adjoint methods allow us to optimize a loss through a differential equation. This relates quite naturally to the field of optimal control theory, where one seeks to choose the best function so as to optimize a loss over the trajectory of a differential system. Furthermore, seeing the neural ODEs through the prism of control theory may lead to efficient model-based deep reinforcement learning.

Contribution In this section, we highlight the connection between neural ODEs and neural optimal control. We propose experiments applying neural control to a navigation problem and to the cartpole problem (inspired from [**?**]).

4.1 Optimal control framework

A controlled dynamical system is a system whose state is driven by dynamics of the form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f\big(x(t), u(t)\big), \tag{4.1}$$

where f is a known function and $u(t)$ is the control of the operator. The operator seeks to find the best possible function u with the aim of optimizing some loss function L of the trajectory x and the control u. Formally, this consists in solving the following minimization problem

$$\begin{aligned} &\min_u L[x, u] \\ \text{such that} \quad &\frac{\mathrm{d}x}{\mathrm{d}t} = f(x, u). \end{aligned} \tag{4.2}$$

In a control problem as formulated in (**??**), the objective functional depends on the control function u implicitly, since the trajectory x is obtained by integrating a u-dependent flow. In some cases, the optimal solution can be derived analytically using tools like Pontryagin's maximum principle and the Hamilton-Jacobi-Bellman equation [**?**].

We adopt a differential programming point of view. Take for u a neural network, or any AD-compatible function, parametrized by some weights θ. We assume here that u is a function of the current state :

$$u(t) = u_\theta\big(x(t)\big). \tag{4.3}$$

Then setting $g_\theta(x) = f\big(x, u_\theta(x)\big)$ the optimization problem (**??**) can be expressed in terms of θ :

$$\begin{aligned} &\min_\theta L[x_\theta, \theta] \\ \text{such that} \quad &\frac{\mathrm{d}x_\theta}{\mathrm{d}t} = g_\theta(x). \end{aligned} \tag{4.4}$$

This equation has the form of (**??**), meaning that a neural controlled dynamics problem may be trained like a neural ODE. In particular, we can perform gradient descent on the weights θ to find the best possible control parameters for (**??**).

This approach differs with the established theories for solving optimal control like Pontryagin's maximum principle and the Hamilton-Jacobi-Bellman equation. Although both of these methods rely on an adjoint differential equation to compute the gradient of the loss with respect to the control, they provide analytical equations to determine the optimal control. In contrast, neural optimal control aims at obtaining an approximate solution of the problem iteratively by the means of gradient descent.

It is also interesting to draw a parallel between neural optimal control and deep reinforcement learning. Reinforcement learning (RL) casts a stochastic optimal control problem into Markov Decision Process (MDP), where an agent evolves in a state space and interacts with an environment by taking actions, with the aim of maximizing a reward function [**?**]. In a continuous setting where the state

of the MDP is analog to the system state $x(t)$, the states are governed by the dynamics (**??**) and hence by the action taken by the agent according to a neural-network policy, which is analog to the control u. The reward is analog to minus the loss, and optimizing the policy amounts to finding the optimal control. Following this analogy, we will see in section **??** that neural optimal control allows us to tackle reinforcement learning problems in a continuous-time, continuous-state and continuous-action setting. More specifically, this analogy relates to model-based reinforcement learning, where the agent has (partial) knownledge of its environment. We refer to [**?**] for a survey on model-based RL.

In this section, we apply neural ODEs to two well-known optimal control problems.

4.2 Zermelo's navigation problem

Zermelo's navigation problem is a classic optimal control problem where the objective is to control a ship to to reach a certain point in space in a windy environment with some optimality criterion [**?**]. Although the goal is usually to reach the objective with time optimality (*i.e.* in minimum time) [**?**], we will study an example with a spatial criterion.

Consider a ship whose goal is to cross a windy river with minimum deviation. At time $t = 0$, the ship is located at the origin $(x, y) = (0, 0)$. Its goal is to reach the opposite bank $y = \ell$ with minimal deviation $x(T)$. In absence of wind, it has constant speed v. While corssing the river, the wind $w(y)$ pushes it towards the right, the magnitude being a known function of y such that $w(y) > v \ \forall y$. The pilot chooses the angle u of the ship with respect to the shore. We give a representation of the system in figure **??**. The ship obeys the following 2-dimensional dynamics :

$$\begin{aligned} \dot{x} &= v \cos u + w(y) \\ \dot{y} &= v \sin u \end{aligned} \tag{4.5}$$

and the control problem takes the form

$$\begin{aligned} \min_{u} \quad & x(T) \\ \text{such that} \quad & (\mathbf{??}). \end{aligned} \tag{4.6}$$

This optimal control problem can be solved by Pontryagin's maximum principle [**?**]. The solution can be expressed as a function of the current position (see [**?**] page 76 for example):

$$\cos u(t) = -\frac{v}{w\big(y(t)\big)}. \tag{4.7}$$

We propose to recover this result experimentally by parametrizing the control by a neural network $u_\theta(t)$, and by training it to optimize (**??**). Neural ODEs allow us to implement the dynamics (**??**) with the neural control and to backpropagate through the trajectory X_θ.

Experimental setup In our experiment, we choose for u_θ a 2-layer fully connected net with width 16 and tanh nonlinearity, followed by a sigmoid layer and an affine function to ensure $0 \leq u_\theta \leq \pi$. We train this net with the Adam optimizer [**?**] that we initialized with a learning rate of 0.05. We set $\ell = 2$, $v = 1$ and use a Gaussian wind profile $w(y) = w_{\min} + (w_{\max} - w_{\min}) \exp\left[-(y - \ell/2)^2/(\ell/10)^2\right]$, with $w_{\min} = v + 0.1$ and $w_{\max} = 2v$. We choose a large time horizon $T > 0$ to ensure that the shore is reached in the first iterations, and set the wind to 0 beyond $y = \ell$. For this task, we use Julia programming language with the library `DiffEqFlux.jl` which implements neural ODEs [**?**].

Results The training loss $L(\theta) = x_\theta(T)$ minus the optimal loss given by integrating the optimal trajectory is plotted on figure **??**. Figure **??** shows the neural trajectory after training, as well as the optimal trajectory and the naive trajectory obtained with $u(t) = \pi/2$. The obtained trajectory converged to the optimum, proving that the neural ODE learned the optimal control. We can further check that the optimal control has been learned by plotting the control values as shown on figure **??**. We note however that the loss curve is somewhat irregular and lacks smoothness, suggesting that training could be unstable at a low number of iterations.

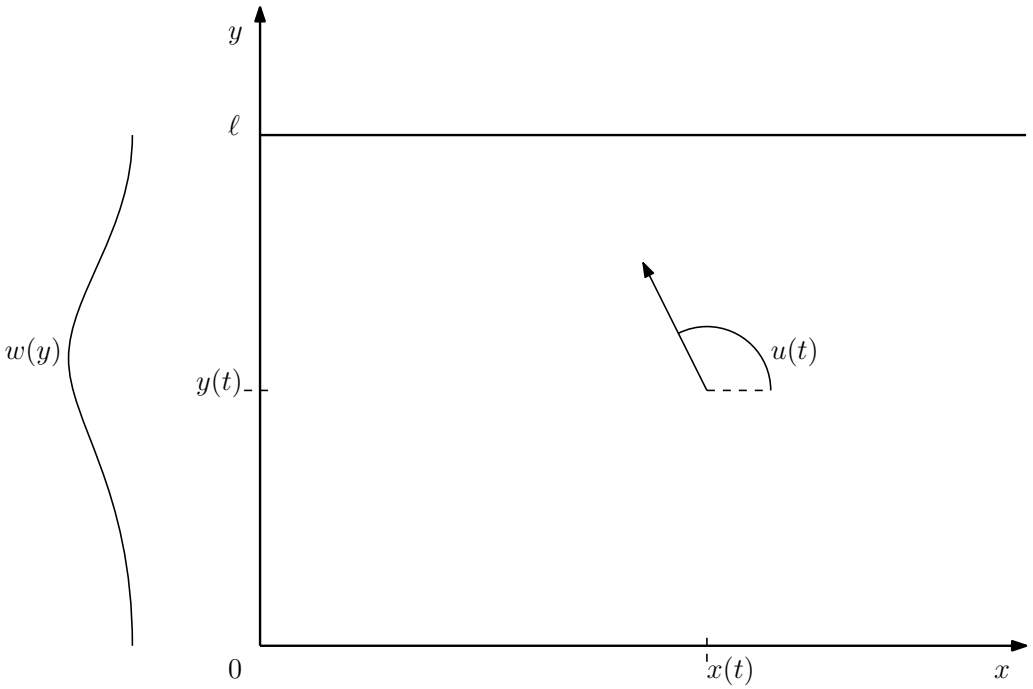


Figure 7: Minimum deviation problem.

4.3 Cartpole

Cartpole, also known as inverted pendulum, is a well-known problem in the field of reinforcement learning. An agent can move the pivot point of a pendulum left and right and his goal is to drive and keep the pendulum in its upright unstable equilibrium. This problem can naturally be described in the framework of optimal control theory. Cartpole is also a reference environment to test RL algorithms. It is available in the OpenAI toolkit [**?**] in a simplified form, where the pole starts upright and the space of actions is reduced to two elements : "push left" and and "push right". This environment was described by Barto, Sutton, and Anderson in 1983 [**?**].

We are interested in solving the carptole problem with its full dynamics, which we can derive with classical mechanics. Parametrizing the system as in figure **??**, the Lagrangian of the system is

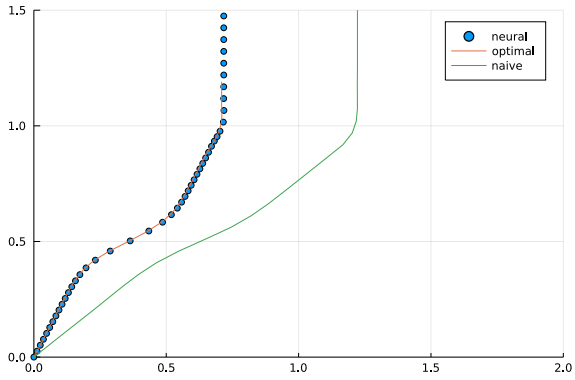
$$\begin{aligned} L(x,\dot{x},\phi,\dot{\phi}) =& \frac{1}{2}M\dot{x}^2 \\ &+\frac{1}{2}m\left(\frac{\mathrm{d}}{\mathrm{d}t}(x+L\sin\phi)\right)^2 \\ &+\frac{1}{2}m\left(\frac{\mathrm{d}}{\mathrm{d}t}\cos\phi\right)^2 \\ &-mgL\cos\phi. \end{aligned} \tag{4.8}$$

The control of the operator is materialized by a horizontal force of magnitude f. The Euler-Lagrange equations yield the following non linear evolution equations

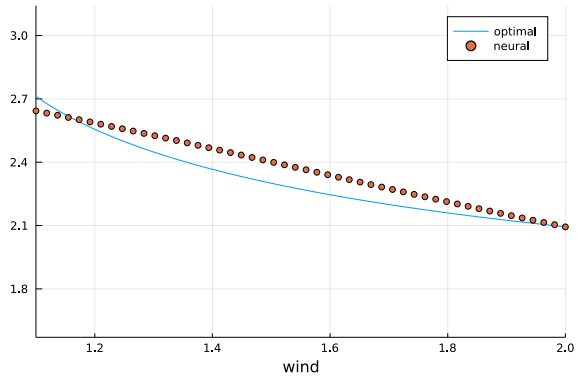
$$\begin{aligned} \ddot{x} =& \frac{f+m\sin\phi(L\dot{\phi}^2-g\cos\phi)}{M+m\sin^2\phi} \\ \ddot{\phi} =& \frac{-f\cos\phi-mL\dot{\phi}^2\sin\phi\cos\phi+(M+m)g\sin\phi}{L(M+m\sin^2\phi)} \end{aligned} \tag{4.9}$$

Our approach is inspired by [**?**] We implement dynamics (**??**) and we model f by a neural network $f_\theta(x,\dot{x},\phi,\dot{\phi})$. We train f_θ to optimize the following objective

$$\min_\theta \phi^2(T)+\dot{\phi}^2(T). \tag{4.10}$$



(a) Learned trajectory.



(b) Values of the control $u_\theta(w)$ in an experiment where u is chosen to be a function of $w(y)$. The neural control learned an approximation of the optimal control $\cos u(w) = -v/w$ (equation (**??**)).

Figure 8: Results after training the neural control.

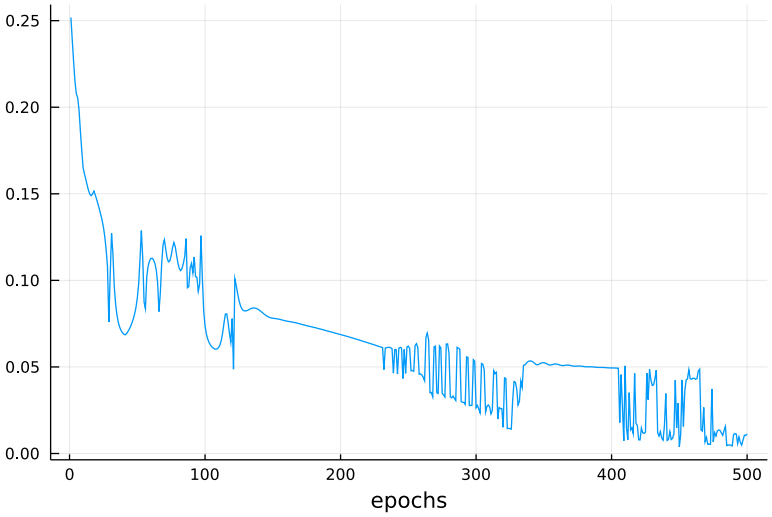


Figure 9: Training loss for objective (**??**).

Interestingly, the linear regime $|\phi| \ll 1$ is solvable analytically. Indeed, the system fits into the linear-quadratic regulator (LQR) formalism where Riccati's equation applies [**?**] and gives the optimal control. We refer to [**?**] for a formal mathematical description.

Experimental setup In our experiments, we initialized the pole at its stable equilibrium position $\phi = \pi$. We set the time horizon to $T = 2$ and the physical quantities equal to $m = 1$, $M = 1$, $L = 1$, $g = 9.8$. For the neural control we took a 2-layer fully connected net with width 16 and tanh nonlinearity. We trained it with the Adam optimizer [**?**] that we initialized with a learning rate of 0.05. Here again, we used Julia library `DiffEqFlux.jl`. We also experimented in the linear regime with LQR parameters $Q = \mathrm{diag}(10^{-4}, 0, 100, 0)$, and $R = 10^{-2}$.

Results The loss over the training epochs is represented on figure **??**. On figure **??**, we show the states of the system at the final time T at different stages of the training. As we can see, the pole ends upright after 100 episodes. Then next episodes reduce the final angular velocity. The neural net is hence successfully learning the optimal control after a few hundreds of epochs. On figure **??**, we plot the neural control obtained after 16 epochs and the corresponding time evolution of the angle ϕ together with the exact solution obtained with Riccati's equation. As we can see, the two curves are

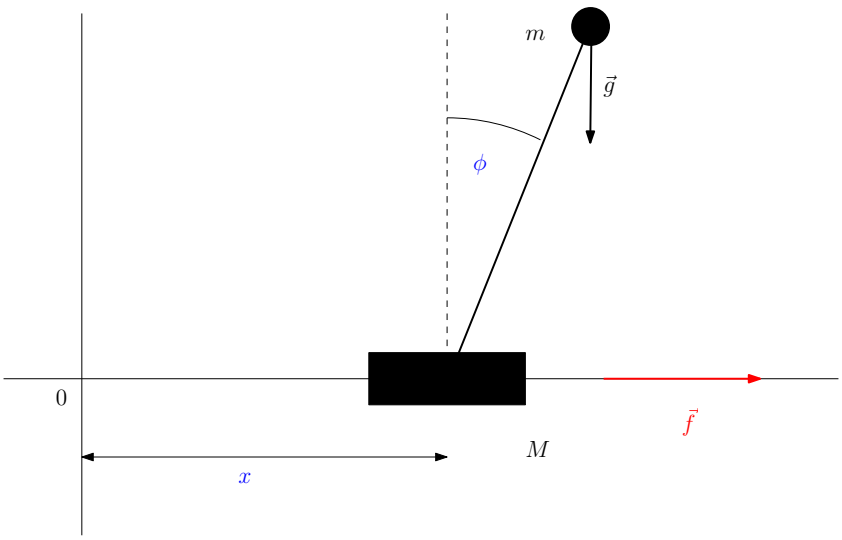
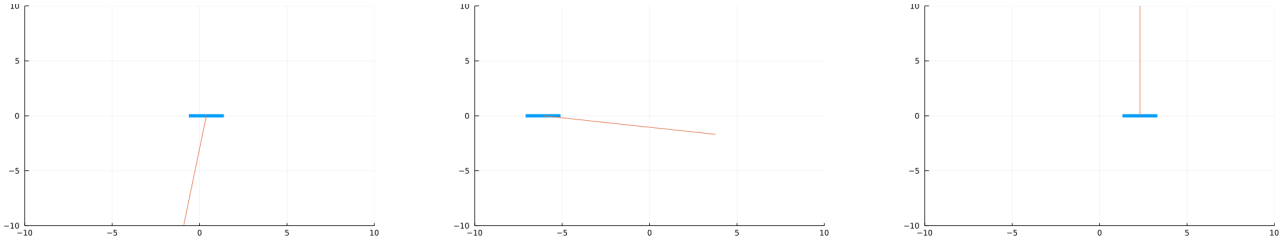


Figure 10: The cartpole system

close, meaning that the neural net successfully learned the linear Riccati mapping $U = -KX$ [?].



(a) Episiode 1 (b) Episiode 10 (c) Episiode 100

Figure 11: Last frame of the system evolution at different training episodes.

4.4 Discussion

Neural ODEs allow one to tackle continuous-time optimal control problems using gradient descent by modelling the control with a neural net and by propagating the gradient in a neural ODE. Our results show that this approach is successful at modelling and solving some classical problems. Since most physical systems can be described by ODEs, it could generalize gradient-based optimization and model-based reinforcement learning to a very large spectrum of applications.

Although it is suggested that this novel ODE-based technique could allow for faster training [?], a thorough benchmark with traditional RL techniques is yet to be carried out. It would be interesting to compare the performances of both approaches on similar control problems. In [?] for example, Biferale *et al.* tackle a 2D navigation problem with an actor-critic algorithm by discretizing the action space of navigation angles. Using neural control allows one to use continuous actions and might hence prove more succesful at learning the optimal navigation. Likewise, prior work on the cartpole focused on the simplified, linearized RL environment with binary inputs. Recently, Yıldız *et al.* [?] used neural ODEs combined with an actor-critic algorithm to tackle RL problems in a continuous-time fashion. In this respect, they desined a new RL framework with compatible environments. Their results show that neural control can learn complicated continuous-time controls, proving the method to be robust and efficient. This work could pave the way for a novel, continuous-time RL framework. In particular, an interesting research direction is that of learning the environment dynamics, which can be done by inferring ODE parameters.

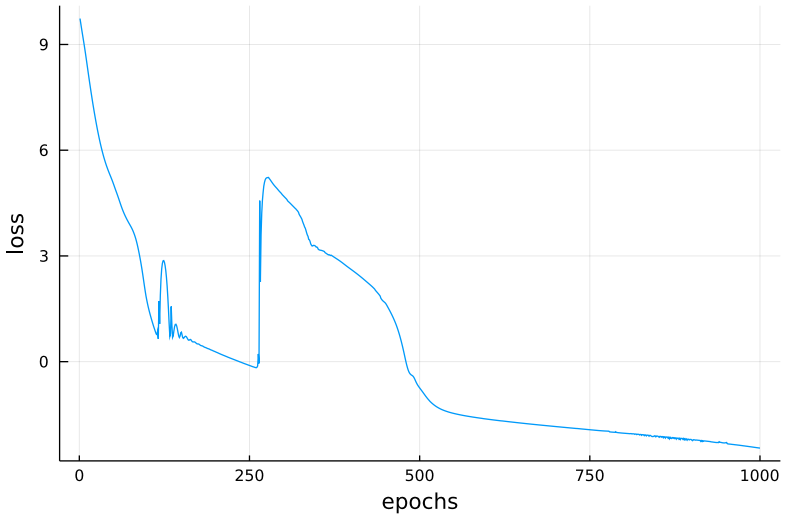


Figure 12: Training loss for objective (**??**).

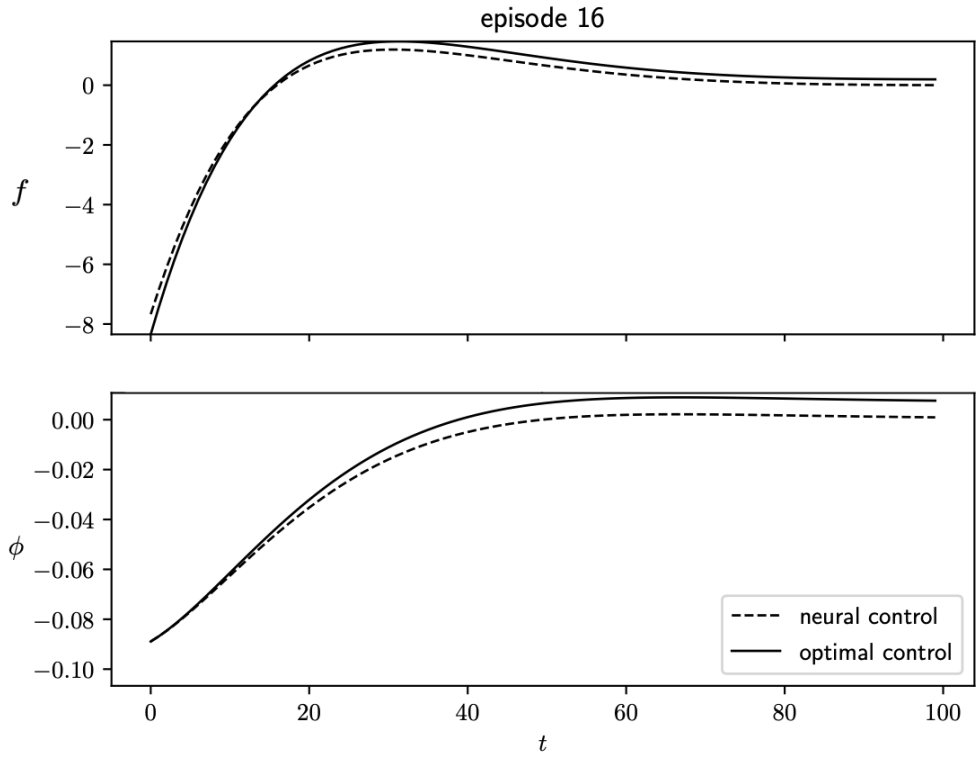


Figure 13: Neural control after 16 epochs of training versus the analytical solution for the LQR cartpole.

5 System identification

System identification is a problem of great importance in many fields such as econometrics, reinforcement learning, robotics or mechanical engineering [?][?][?]. The task consists in estimating the dynamics of an unknown system by sampling trajectories from it. We focus here on linear systems : let $A \in \mathbb{R}^{d\times d}$ and $B \in \mathbb{R}^{d\times m}$ be two matrices. Consider the linear, controlled dynamical system

$$\begin{aligned} x_{t+1} &= Ax_t + Bu_t + w_t, \qquad 0 \leq t \leq T-1, \\ x_0 &= 0 \end{aligned} \tag{5.1}$$

where $x_t \in \mathbb{R}^d$ is the state, $w_t \sim \mathcal{N}(0, \sigma^2)$ is normally distributed noise and the control $u_t \in \mathbb{R}^m$ is an input variable chosen by the operator with some energy constraint

$$\frac{1}{T}\sum_{t=1}^{T} \|u_t\|^2 \leq \gamma^2. \tag{5.2}$$

We further assume the condition $\rho(A) \leq 1$ on the spectral radius of A which ensures that the system is stable and controllable [?].

The goal of system identification is to choose the best input sequence (u_t) to estimate the unkown matrix A. Note that it may be that B is unknown as well and we want to estimate both A and B. In this case the results for estimating A can be extended [?]. In this work, we will assume that B is known.

There has been substantial work in the field of system identification. Recent papers have provided sharp bounds for the estimation of A in the passive case, where $u_t = 0$ and the input is reduced to noise [?]. Recently, significant attention has been given to the question of the best input for system estimation, be it for linear [?] or nonlinear systems [?]. This recent line of work has focused on providing optimal theoretical rates for this task, but little attention has been paid to efficient algorithm implementations.

The fundamental difficulty in this control problem lies in that the variables x_t that produce our estimate of A depend on the unkown matrix A itself. As a consequence, one can not directly optimize the trajectory with respect to the control in an exact way. What one can do however is to iteratively update an estimate $\hat{A}$ of A, and optimize over the trajectories as if this estimate was the ground truth. This approach is refered to as adaptive experimental design and is described and analyzed by Wagenmaker and Jamieson [?]. We will review it in section **??**.

The algorithm of [?] relies on solving a complex optimal control problem, for which no efficient numerical method is provided. We propose a novel optimization technique based on automatic differentiation to address this issue.

Regardless of the trajectory optimization, one estimates A by least squares regression over the observed trajectories of the system (**??**), which is proved to be nearly optimal for this task [?]. Therefore, we start by reviewing the main concepts of Ordinay Least Squares in the next section.

Contribution In this section, we propose a novel approach to solve a planning control problem based on AD and provide experiments illustrating the effectiveness of our method. We also recall mathematical concepts of least square regression and the adaptive algorithm of [?] which implements planning.

5.1 Ordinary Least Squares

Least squares is a standard approach in regression analysis for noisy observations. Here we briefly recall its mathematical foundations and then adopt an active learning point of view.

5.1.1 Mathematical framework

Consider noisy observations of some linear relation

$$Y = X\beta + \eta, \qquad X, Y \in \mathbb{R}^{n\times d}, \qquad \eta \sim \mathcal{N}(0, \sigma^2 I). \tag{5.3}$$

The unknown here is the matrix $\beta \in \mathbb{R}^{d\times d}$ and the goal is to estimate it from the noisy observations Y. The least squares estimator of β is the solution of

$$\min_{\alpha} \ \|Y - X\alpha\|^2 \tag{5.4}$$

and is equal to

$$\hat{\beta}(X) = (X^\top X)^{-1} X^\top Y \quad \in \mathbb{R}^{d\times d}, \tag{5.5}$$

yielding the residual

$$\hat{\beta}(X) - \beta = (X^\top X)^{-1} X^\top \eta. \tag{5.6}$$

Hence for some realization of the noise η, the error in OLS is

$$\left\|\hat{\beta}(X) - \beta\right\|^2 = \left\|(X^\top X)^{-1} X^\top \eta\right\|^2. \tag{5.7}$$

We define the loss as the expectation of (**??**) :

$$L(X) = \mathbb{E}\left[\left\|\hat{\beta}(X) - \beta\right\|^2\right]. \tag{5.8}$$

Let $s_1, \dots, s_d \geq 0$ be the singular values of X, and let

$$S = \begin{pmatrix} \mathrm{diag}(s_1, \dots, s_d) \\ 0 \end{pmatrix}. \tag{5.9}$$

Then the singular value decomposition of X is

$$X = USV^\top, \qquad U \in O_n(\mathbb{R}), \quad V \in O_d(\mathbb{R}). \tag{5.10}$$

The loss (**??**) may thereby be expressed as a function of the singular values $0 \leq s_1 \leq \cdots \leq s_d$ of X [**?**] :

$$L = \sum_{i=1}^{d} \frac{1}{{s_i}^2}. \tag{5.11}$$

5.1.2 Active learning for least squares

The control problem induced by equation (**??**) poses the question of determining the best trajectory (x_t) for an optimal estimation of A. We are first interested in the following question : what is is the best choice of $x_1, \dots, x_n$ to estimate A at a fixed variance budget, *i.e.* such that the total variance is bounded by some constant :

$$\frac{1}{n}\sum_{k=1}^{n} {x_k}^\top x_k \leq r^2 \qquad ? \tag{5.12}$$

Following the previous expression of the estimation error (**??**), we consider the optimization problem

$$\begin{aligned} \min_{X} \quad & \sum_{i=1}^{n} \frac{1}{{s_i}^2(X)} \\ \text{such that} \quad & \frac{1}{n}\mathrm{tr}(XX^\top) \leq r^2. \end{aligned} \tag{5.13}$$

Alternatively, another problem considered in the system identification framework is [**?**]

$$\begin{aligned} \max_{X} \quad & {s_1}^2(X) \\ \text{such that} \quad & \frac{1}{n}\mathrm{tr}(XX^\top) \leq r^2. \end{aligned} \tag{5.14}$$

The objective functions and the constraints of problems (**??**) and (**??**) depend only on the singular values of X. For example, (**??**) is reduced to

$$\begin{aligned} \max_{0\leq s_1\leq\cdots\leq s_d} \quad & s_1 \\ \text{such that} \quad & \frac{1}{n}\sum_{i=1}^{d} {s_i}^2 \leq r^2. \end{aligned} \tag{5.15}$$

Then, all solutions of (**??**) are obtained my multiplying $S = \begin{pmatrix} \mathrm{diag}(s_1,\ldots,s_d) \\ 0 \end{pmatrix}$ by orthogonal matrices on the right and on the left. The optimum is reached at

$${s_1}^2 = \cdots = {s_d}^2 = r^2/d. \tag{5.16}$$

This choice also minimizes (**??**) by Jensen's inequality. This result tells us that in the ordinary least squares framework, the best estimation is achieved when the inputs are chosen such that the point cloud is isotropic (the variance according to all directions should be the same) and with maximum variance.

5.2 Optimal control for system identification

Now that we recalled a few notions about OLS, let us get back to our dynamics problem. As we mentioned, we base our estimate of A on observed trajectories (x_t) by performing OLS :

$$\hat{A} \in \underset{\beta\in\mathbb{R}^{d\times d}}{\mathrm{argmin}} \sum_{t=1}^{T-1} \|x_{t+1} - Bu_t - \beta x_t\|^2 . \tag{5.17}$$

We do not choose the points x_t directly, but we choose the control values (u_t) which determine (x_t) through (**??**). The question is : how to choose (u_t) ? Formally, we wish to have $\hat{A}$ as close to A as possible :

$$\min_u \left\|\hat{A} - A\right\|^2 . \tag{5.18}$$

This objective is not explicitly defined since A is not known and $\hat{A}$ depends on (x_t) through (**??**) and hence on the unknown A. However, we know from the theory of least squares that the relation $Y = AX$ is best estimated when X has large singular values, in the sense of equations (**??**) and (**??**). Let

$$X = \sum_{t=1}^{T} x_t {x_t}^\top \tag{5.19}$$

be the **design matrix** of the system. Expression (**??**) is not valid in our case because of the dependence between the x_t introduced by the dynamics. Yet, the analysis of [**?**] shows that the estimation error may be bounded by a function of the key quantity $s_{\min}(X)$. From there, we can formulate an objective in terms of u and X and hence define an acutal optimal control problem for u :

$$\begin{aligned} \max_{u_1,\ldots,u_T} \quad & s_{\min}\left(\sum_{t=1}^{T} x_t {x_t}^\top\right) \\ \text{such that} \quad & (\textbf{??}) \quad \text{and} \quad \frac{1}{T}\sum_{t=1}^{T} {u_t}^\top u_t \leq \gamma^2. \end{aligned} \tag{5.20}$$

The hope behind this definition is that maximizing this surrogate objective for the least squares error in our particularly entangled case will make our trajectory approximately optimal for the estimation of A. Note however that although we casted our objective into an optimal control problem, it is still not completely explicit because the dynamics (**??**) is unknown.

5.3 Adaptive experimental design

As we pointed out, problem (**??**) is still ill-posed since the dynamics yielding the first constraint is unknown, and is actually precisely what we want to find. A natural way to face this difficulty is to proceed iteratively, by alternating between *planning* and *estimation*.

First, given an estimate $\hat{A}$ of A, one optimizes a surrogate of (**??**) where the true dynamics is replaced by the estimated one :

$$x_{t+1} = \hat{A}x_t + Bu_t + w_t, \qquad 1 \leq t \leq T. \tag{5.21}$$

This phase is called planning, and can be formulated as

$$\begin{aligned} &\max_{u_1,\dots,u_T} \; s_{\min}\left(\sum_{t=1}^{T} x_t {x_t}^\top\right) \\ &\text{such that} \qquad (\mathbf{??}) \qquad \text{and} \qquad \frac{1}{T}\sum_{t=1}^{T} {u_t}^\top u_t \leq \gamma^2. \end{aligned} \tag{5.22}$$

Second, one plays the planned optimal control and observes the resulting trajectory. The observations are then used to update one's estimate $\hat{A}$ by least squares regression (**??**).

Following this principle, Wagenmaker *et al.*[**?**] propose an asymptotically optimal algorithm consisting in two phases. This approach proceeds in epochs of geometric length and is summarized in algorithm **??**.

Algorithm 1: Adaptive active system identification

Input: n_{epochs}, energy γ^2
Result: control u_t

initialize u_t as white noise with the energy constraint;
observe a first trajectory x_t and append it to X ;
for $1 \leq i \leq n_{\text{epochs}}$ **do**
  estimation : $\hat{A} = \text{OLS}(X)$;
  planning : $u_t = \text{Planning}(\hat{A}, T)$;
  play the dynamics with u_t, observe x_t ;
  append x_t to X;
  $T = 3 * T$;
end

Since planning consists in optimizing a version of (**??**) where the true dynamics A is replaced with the current estimate $\hat{A}$, the approximation error of $\hat{A}$ creates a bias. The optimization problem being an approximation of the desired one, the obtained solution can only be an approximation of the optimal control. Yet, the theoretical analysis [**?**] proves that this approach does converge with asymptotically optimal rate. These theoretical bounds are out of the scope of our work.

The main bottleneck of this approach lies in solving (**??**) efficiently. The authors of [**?**] mention that they proceed by an alternating minimization technique but do not elaborate on their approach. In the following, we propose a new method for planning based on automatic differentiation.

5.4 Differentiable planning

There are two major difficulties in optimization problem (**??**). First, we are optimizing a functional over a trajectory whose states are mutually dependent through the control. Second, this functional is an implicit function based on a nonlinear algebra operation, namely singular value decommposition. As it turns out, both of these can be tackled by the means of automatic differentiation.

Integrating the trajectory (**??**) is a sequence of T elementary operations which can be differentiated through. The noise can be considered as an external quantity and does not intervene in the backpropagation step — backpropagating with fixed noise is related to the so-called "reparametrization trick" as introduced in [**?**]. It is also conceivable to extend our approach to the continuous limit with neural ODEs with the use of the adjoint method for backpropagation as described in section **??**, which allows for memory savings.

As for SVD, we showed in section **??** that this operation is AD-compatible and we derived a closed formula for its adjoint. Therefore, the whole minimization program can be solved by automatic differentiation and gradient descent. Our approach is illustrated in figure **??**.

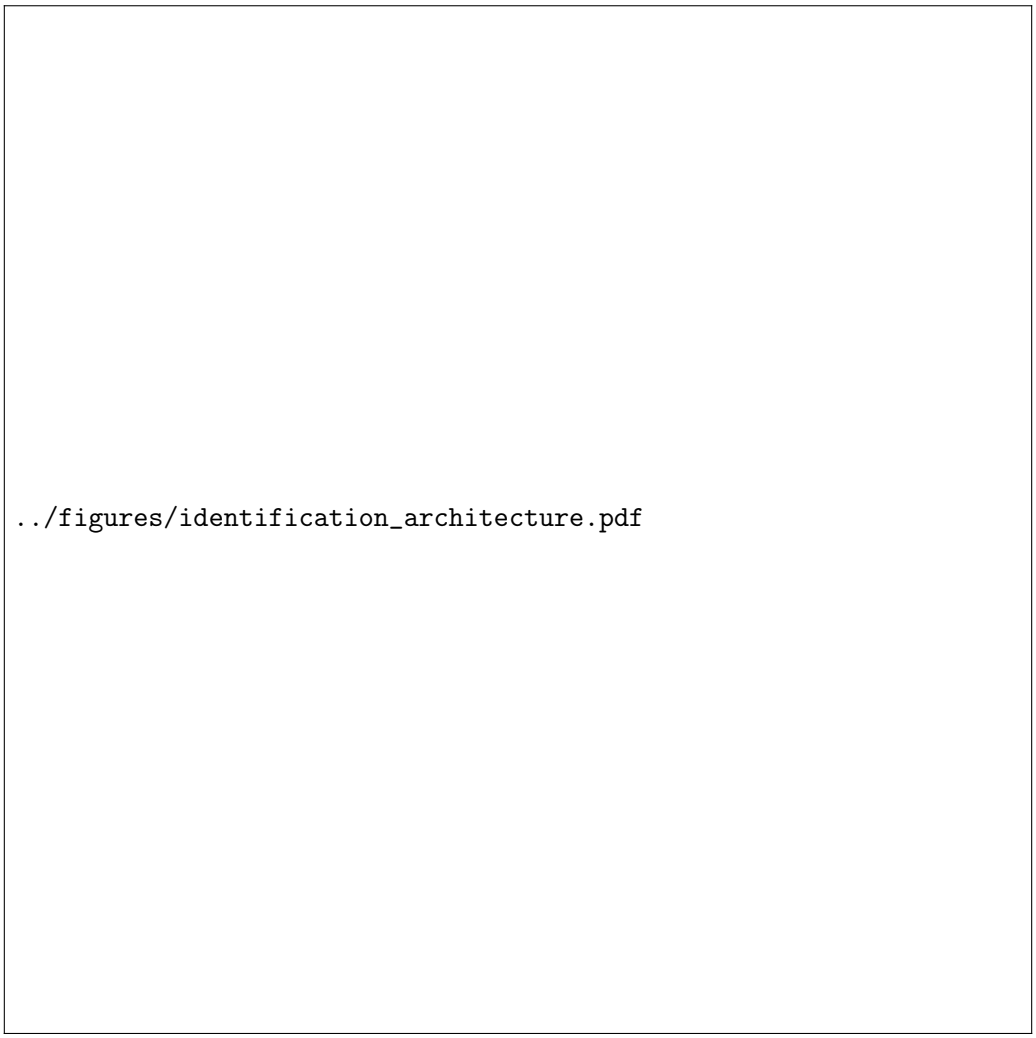


Figure 14: The training architecture for differentiable planning.

We change the total energy constraint $\sum {u_t}^\top u_t \leq T\gamma^2$ by ${u_t}^\top u_t \leq \gamma^2 \ \forall t$. This allows us to readily implement the energy constraint by normalizing the output of the control.

5.5 Experimental results

We implement this adaptive identification algorithm in the automatic differentiation framework and compare it to an oracle and to the baseline of [**?**]n which consists in normally distributed controls of mean energy γ^2 : $u_t \sim \mathcal{N}(0, \gamma^2 I/d)$. The oracle has access to the true dynamics matrix A and plans according to (**??**), with the same optimization method as the active learning agent.

Experimental setup We experiment our method on different types of unknown dynamics matrices A. We restrict ourselves to $B = I$. We choose $\sigma = 0.1$, $\gamma = 1$. We choose the control to be a neural

net function of both the current state and the time : $u_t = u_\theta(x_t, t)$. We parametrize it as a 2-layer fully connected net with width 16 and tanh nonlinearity. The estimation error is compared to that of the Gaussian baseline averaged over 1000 realizations. We implement our experiments with Pytorch [**?**] which implements differentiable SVD. Our code is available at `https://github.com/MB-29/differentiable-SysID`.

Results Our results are presented on figure **??**. The active learning algorithm outperforms the baseline and its performance matches that of the oracle algorithm, like in the work of Wagenmaker *et al.*. We noted that the planning phase can get slow, due to the exponential times of the algorithm. We could accelerate it using GPU parallelization.

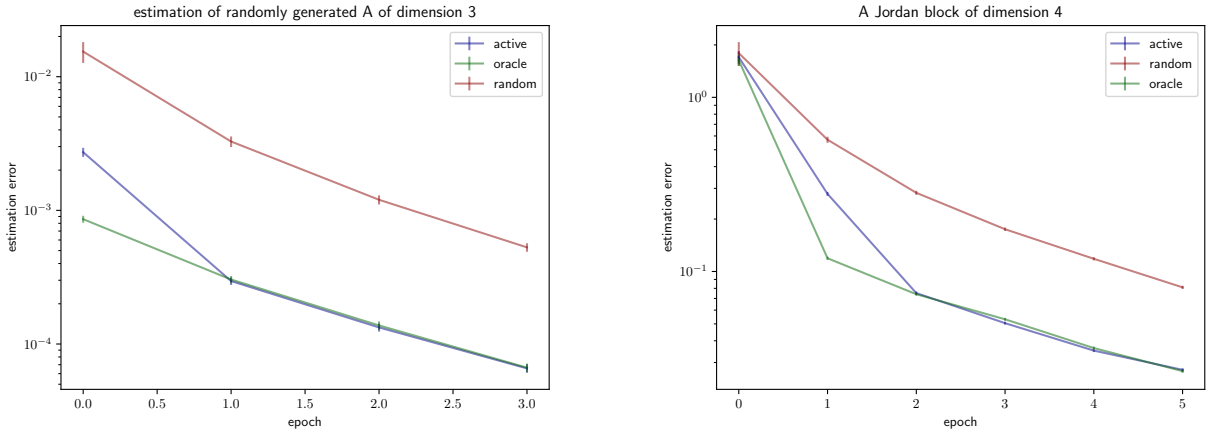


Figure 15: Estimation error for different types of A.

5.6 Discussion

We propsed a gradient-descent-based numerical approach for optimal design planning. Our results prove its efficiency on different types of matrices. Together with an adaptive algorithm, it can hence be used to solve the system identification control problem.

Our approach offers the advantage to be based on deep learning, thereby allowing one to parallelize the training on a GPU and guaranteeing the control to be arbitrarily expressive. Moreover, the control can be defined as a function of both the time and the current state, which is arguably a key for greater efficiency to mitigate transient effects [**?**][**?**]. A point we did struggle on is that of training the net which can be slow due to the large number of operations we differentiate through, and the SVD operation sometimes yields singular matrix exceptions. We can face this issue by carefully choosing the learning rate of the optimizer, by regularizing the matrices [**?**] and possibly by taking a continuous-time perspective so that we can apply ODE adjoint equations.

6 Experimental design

In all scientific problems, experiments are a fundamental step as they are the only possible way to collect data and thereby to confront a theory or a model to reality. However, it is often resource consuming to perform real-world experiments, be it computer simulations (e.g. for fluid dynamics [**?**] or particle physics [**?**]) or in real life. Therefore, being able to choose the best conditions (in a sense yet to define) that guarantee an experiment to be as efficient as possible is of a paramount importance.

Optimal experiment design seeks experiments that maximize some criterion over their expected output. The objectives underlying these criteria include parameter infererence, prediction and model discrimination [**?**]. In essence, experimental design answers the question "where to collect data in order to gain maximum information and minimum cost ?" [**?**].

Most recent publications in the field of Bayesian Optimal Design have focused on developping algorithms to estimate and optimize untractable crtieria for optimal design, including nested Monte-Carlo simulations, Polynomial Chaos surrogates and variational approximations. [**?**][**?**][**?**][**?**][**?**].

In our approach instead, we use a linear approximation and consider a simpler, tractable utility criterion. We then leverage automatic differentiation to compute gradients of our model with respect to the parameters and the inputs, allowing for gradient-based search of the optimal design. It has already be noted that although exact inference is intractable in implicit models, computing the gradients through the latent variable brings precious information, see for example [**?**]. We take a similar path and try to "mine gold" from the simulator to optimize experimental design.

Contribution In this section, we propose a new approach of local Bayesian optimal design based on automatic differentiation. We provide experimental results on a toy model. We also recall the mathematical framework for experimental design.

6.1 Theoretical framework

Consider the parametrized modelling of some experiment :

$$\Phi(x, \theta), \qquad x \in \mathrm{R}^d, \quad \theta \in \mathrm{R}^m. \tag{6.1}$$

Assume the experimenter is able to choose a design x and to measure some noisy realization of the model under the optimal parameter θ_* :

$$y = \Phi(x, \theta_*) + \varepsilon \tag{6.2}$$

where the noise $\varepsilon \sim \mathcal{N}(0, \sigma)$ is normally distributed. The goal is to infer the parameter θ from the observations y_i at minimal cost.

6.1.1 Bayesian framework

Finding an efficient design can be formulated as an optimization problem of some citerion $U(x)$ over the posterior distribution $p(\theta|x)$:

$$\max_x \quad U(x). \tag{6.3}$$

Here we care only about the choice of one single design x. We will describe the case of multiple designs in the framework of sequential optimal design in section **??**.

A common criterion is that of mutual information : the optimal design must maximize the Kullback-Leibler divergence from the posterior to the prior. Since y is unknown, the maximization is performed on the expected value of the model [**?**] :

$$U(x) = \mathbb{E}_{y|x}\left[\mathrm{KL}\big(p(\theta|y_i, x), p(\theta)\big)\right]. \tag{6.4}$$

From an information-theoretic viewpoint, the motivation of (**??**) is that the larger the divergence between the two measures the smaller the entropy under the posterior ditribution, and so the best the estimate of θ. In fact, one can show that the entropy difference of the two measures yields in expectation the same criterion as the divergence [**?**] :

$$U(x) = \mathbb{E}_{y|x}\left[S\big[p(\theta)\big] - S\big[p(\theta|y_i, x)\big]\right], \tag{6.5}$$

where the entropy is defined as

$$S[p] = -\int p \log p. \tag{6.6}$$

Typically, the criterion (**??**) is intractable and evaluating it requires complicated computational artifacts such as nested Monte Carlo sampling [**?**]. In the next section, we present an approach that avoids those technical difficulties by deriving a linear approximation of the model.

6.1.2 Linear approximation and local optimality

Least squares regression as presented in section **??** corresponds to a doubly linear case where Φ is linear in both x and θ. As we showed, the optimal design to perform inference of the regression parameter corresponds to isotropic inputs with maximum variance (equation (**??**). From there, it is natural to study the case where the model is still linear in the parameter θ, but without any assumption on the dependence on x. Assume that we have a good guess $\hat{\theta}$ of the true parameter. A key quantity we will consider is the x-dependent **model sensitivity** at our guess $\hat{\theta}$:

$$g(x) = \frac{\partial \Phi}{\partial \theta}(x, \hat{\theta}). \tag{6.7}$$

The sensitivity encodes the variability of the model with respect to the parameter. It is indeed natural to choose inputs in areas where this variability is large so that one can discriminate between different values of the parameter. This statement is illustrated on a simple example in section **??** .

In a region close to our estimate $\hat{\theta}$, we may expand Φ to the first order in $\theta - \hat{\theta}$:

$$\Phi(x, \theta) = \Phi(x, \hat{\theta}) + \big\langle \theta - \hat{\theta}, g(x) \big\rangle + \mathcal{O}\big((\theta - \hat{\theta})^2\big). \tag{6.8}$$

We assumed the observation noise to be additive and Gaussiain, so the likelihood of an observation y is

$$p(y|\theta) = \exp\left(-\frac{1}{2\sigma^2}\big(y - \Phi(x, \theta)\big)^2\right). \tag{6.9}$$

Assume a Gaussian prior

$$\theta \sim \mathcal{N}(\hat{\theta}_0, C_0) \tag{6.10}$$

on the parameter. Then, the linear approximation (**??**) yields a Gaussian distribution for y with covariance matrix $gg^\top$ [2]. It follows that the posterior on θ is normally distributed with covariance.

$$C = C_0 - \frac{1}{\sigma^2 + g^\top C_0 g} C_0 g g^\top C_0 \tag{6.11}$$

The entropy of this posterior equals

$$S = \frac{1}{2}\big(1 + \ln(2\pi) + \ln\det C\big). \tag{6.12}$$

Note that this quantity is actually independent of the actual observation y. This comes from the fact that a Gaussian conjugate prior makes the posterior covariance independent of the observation. We may then compute the expected information gain as defined by (**??**) :

[2] This also holds for non-Gaussian prior on θ provided the distribution can be expanded quadratically around the most probable value [**?**].

$$U(x) = \frac{1}{2} \log \left(1 + \frac{1}{\sigma^2} g(x)^\top C g(x) \right). \tag{6.13}$$

Optimizing this creterion yields a so-called local optimal design due to the linear approximation (**??**) [**?**]. A more detailed derivation can be found in [**?**]. Note that the mutual information criterion for a Gaussian posterior is equivalent to the log-D optimality criterion in Bayesian optimal design theory

$$U(x) = -\mathbb{E}_{y|x} \left[\log \det \operatorname{cov}(\theta) \right], \tag{6.14}$$

which measures the volume of the uncertainty ellipsoid in experimental design theory [**?**].

6.2 Differentiable minimum entropy

We can make use of automatic differentiation to compute the key quantity $g(x) = \partial\Phi/\partial\theta$ even for complex, implicit models. Indeed, the gradient can be accumulated along the computational flow of the model using automatic differentiation, as explained in section **??**. Differential equations are a case in point : we showed in section **??** a formula to compute the gradients with respect to parameters through an ODE model.

There has been previous research on optimal design focusing on the linear approximation (**??**) with normally distributed errors. In the fields of optimal design for chemical engineering and pharmacokinetics for example, this methodology is used to choose the best time to measure the state of a dynamical system in order to infer parameters [**?**]. In this respect, one solves the adjoint differential equation to compute the sensitivity of the ODE model. This is refered to as the "direct method" [**?**]. The strength of our approach is that it can be applied to any model or simulation provided with an automatic differentiation implementation.

Let us now explain how to find an optimal design by using reverse-mode AD. We recall that our aim is to maximize a quadratic function of the sensitivity $U(x) = g(x)^\top C g(x)$, with $g(x) = \partial\Phi/\partial\theta(x, \hat{\theta})$, at fixed $\theta = \hat{\theta}$. We optimize this function by gradient ascent using AD. For this matter, we need to specify the forward and the backward passes for the function $g(x)$.

For the forward pass, we first compute the model output $\Phi(x, \theta)$. Since we assumed or model to be endowed with AD with respect to x and θ, we may then compute the adjoint of θ in the evaluation of Φ evaluated at $\theta = \hat{\theta}$, yielding $g(x)$. We record the operations giving g as a function of x in the computation of this adjoint. This requires that the backward pass of $\theta \mapsto \Phi(x, \theta)$ is itself automatically differentiablew with respect to x.

For the reverse pass, we evaluate the loss $U(x)$ as a function of $g(x)$. With the operations from the forward pass being tracked, we can backpropagate through $x \mapsto g(x)$ and hence compute the gradient $\nabla U(x)$.

6.3 Sequential design

In many applications, decisions are made sequentially and one collects a sequence of observations. For each observation, one can choose the design x based on what one has observed until then. In a Bayesian framework, a natural way of proceeding is to start from a prior distribution on the model parameter and to update the posterior after each observation, so that the design for the next experiment is optimized with respect to the current posterior and hence to the current knowledge of the parameter. This method is called bayesian sequential experimental design and we formalize it in algorithm **??**.

In our framework where the model might be a complicated, implicit function, this approach poses the question of inferring the parameters of the model. Performing Bayesian inference on probabilistic models with intractable density — which are refered to as implicit models — is a complex problem and a fully-fledge field of research. There exists many approaches to this problem, often refered to as likelihood-free inference [**?**][**?**].

Likelihood for implicit models is not in the scope of this work. We perform inference on the linearized model (**??**) for which the output variable is an affine function of θ, using standard Gaussian inference

formulae. Assuming a Gaussian prior for θ, the posterior distribution is also Gaussian. Assume we have already performed n experiences and are to choose the next design x_{n+1}, which yields an observation y_{n+1}. Applying the formulas for a Gaussian conjugate prior with mean $\hat{\theta}_n$ and covariance C_n, the posterior is parametrized by

$$C_{n+1} = C_n - \frac{1}{\sigma^2 + g^\top C_n g} C_n g g^\top C_n \tag{6.15a}$$

$$\hat{\theta}_{n+1} = \theta_n + \frac{1}{\sigma^2}\big(y_{n+1} - \Phi(x_{n+1}, \hat{\theta}_{n+1})\big) C_n g. \tag{6.15b}$$

We can hence iterate parameter inference and design optimization and hope that our estimates converge to the true parameter θ_*. As long as the first estimate of θ is close enough to θ_* for the linear approximation to be valid, each observation is expected to bring information and hence to make $\hat{\theta}_{n+1}$ at least as close to the true value as $\hat{\theta}_n$.

Algorithm 2: Sequential bayesian experimental design

Input: noise variance σ^2, prior mean $\hat{\theta}_0$ and covariance C_0, number of samples N
Result: posterior mean $\hat{\theta}_N$ and covariance C_N

for $0 \leq n \leq N-1$ **do**
  Find the optimal design x_n according to the current belief $(\hat{\theta}_n, C_n)$;
  Run an experiment with design x_n, observe y_n ;
  Update the posterior parameters $(\hat{\theta}_{n+1}, C_{n+1})$ according to (**??**);
end

6.4 Application

Here we apply our differential entropy minimization method on a simple example. The encouraging results we obtain suggest this method could prove efficient for more complex problems.

6.4.1 Exponential decay

First consider the following fairly simple model :

$$\Phi(x, \theta) = \mathrm{e}^{-\theta x}, \qquad x \geq 0,\ \theta > 0. \tag{6.16}$$

Assume one can observe noisy realizations according to (**??**), with a prior knowledge centered in $\hat{\theta} = 1$. Where (at which x) should one measure y to best refine one's knowledge of the parameter θ ? Although the model is explicitly defined in terms of well-known functions, deriving the posterior distribution for θ can be hard to carry out analytically, and the same goes for the computation of criterion (**??**).

Regardless of the criterion, the simple form of equation (**??**) lets us make a few observations. For small x, any parameter θ will yield values of y close to 1. For large values of x, the model will essentially yield $y = 0$. These two extreme inputs make outputs independent of the parameter, and are hence unlikely to be relevant for our inference task. At intermediate x values however, the output varies more or less with θ. This variation is described by the sensitivity (**??**). Computing the derivative at $\hat{\theta}$, we obtain

$$g(x) = -x\mathrm{e}^{-\hat{\theta}x}. \tag{6.17}$$

Intuitively, we should choose x where the model varies the most with respect to θ so that we can best use an observation y to discriminate the true value of the parameter. This amounts to optimize

$$\max_x \quad g^2(x), \tag{6.18}$$

which is reached at $x = 1/\hat{\theta}$. This is illustrated in figure **??** : $x = 1/\hat{\theta}$ is indeed the value for which there is a maximum sensitivity to θ assuming the true parameter is $\hat{\theta}$.

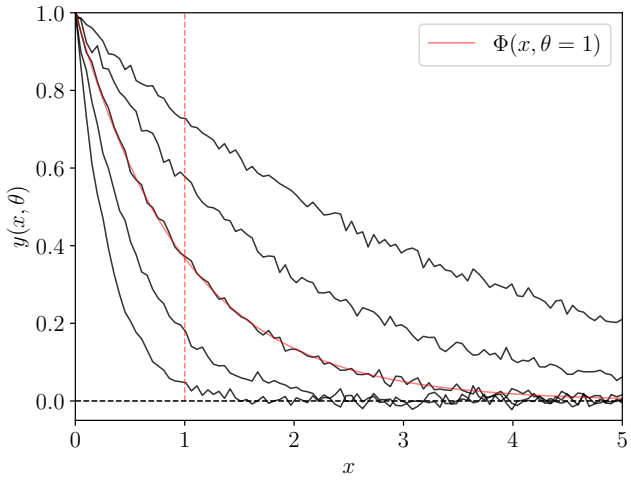


Figure 16: Noisy realizations of model (**??**) are plotted in black. The red line plot is the model output for $\theta = 1$; the red dashed line is the abscissa $x = 1/\hat{\theta}$.

This is precisely the sense of the result derived in section **??** in the one-dimensional case. Knowing how much the output varies with respect to the parameter is a precious information about where one should observe the ouput. The ability to calculate this quantity, be it analytically or by the means of automatic differentiation, allows one to compute an approximate criterion (**??**) which describes the information gain of the first order linearized model (**??**).

In the case of a multivariate parameter, the prior covariance matrix encodes the first order interdependence of the components which are known a priori and determines the coefficients of the quadratic loss.

6.4.2 Differential equation viewpoint

It is interesting to note that model (**??**) can also be seen as a more complicated model involving a differential equation. Consider the following Cauchy problem :

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= -\theta u \\ u(0) &= 1. \end{aligned} \tag{6.19}$$

Let $u(t, \theta)$ be a solution of (**??**) for some parameter θ. Then an equivalent formulation (up to renaming the variables) for (**??**) is

$$\Phi(t, \theta) = u(t, \theta). \tag{6.20}$$

This allows us to check that the sensitivity computed by automatic differentiation is equal to the exact sensitivity we derived (**??**).

6.4.3 Design of observation times

In many fields, the model we are interested in is described by a differential system of the form

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= f(u, \theta) \\ u(0) &= u_0, \end{aligned} \tag{6.21}$$

where the parameters of interest are those of the flow $f(u, \theta)$. In these models, the experimenter chooses the time of the measurements $x = t$ [**?**][**?**], and the observable quantity is some scalar function of the state u :

$$\Phi(t,\theta) = u_0 + \int_0^t f\big(u(s),\theta\big)\mathrm{d}s. \tag{6.22}$$

In our approach, the key quantity is the model sensitivty

$$g(t) = \nabla_\theta \Phi(t,\hat{\theta}), \tag{6.23}$$

which we needs to be a differentiable function of t. Although differentiable solvers ensure automatic differentiation with respect to the parameters and the input of the ODE, they do not implement differentiation with respect to the integration time t, which prevents us from automatically obtaining $\mathrm{d}g/\mathrm{d}t$. We here present a computational trick for the derivative $\mathrm{d}g/\mathrm{d}t$.

Since $g(t) = \partial y/\partial\theta(t,\theta_*)^\top$, by Schwarz's theorem

$$\begin{aligned}\frac{\mathrm{d}g}{\mathrm{d}t} &= \frac{\partial^2\Phi}{\partial t\partial\theta}\\ &= \frac{\partial^2\Phi}{\partial\theta\partial t}\\ &= \frac{\partial}{\partial\theta} f\big(u(t,\theta),\theta\big).\end{aligned} \tag{6.24}$$

This symmetry of the partial derivatives allows us to implement a differentiable sensitivity function as follows. The forward pass is performed by the means of internal differentiation as explained in section **??**. For the backward pass, we simply differentiate through the computation of $f\big(u(t,\theta),\theta\big)$ with respect to θ.

6.5 Experimental results

We test our AD-based method for experimental design on the toy model (**??**). We perform sequential bayesian experimental design as described by algorithm **??**. At each step, the design is obtained by differentiable minimum entropy **??** and the parameter posterior is updated by the Gaussian formulae (**??**).

Experimental setup We implement differential model (**??**) in the `torchdyn` framework [**?**], which allows automatic differentiation through the ODE. We optimize loss (**??**) and compute the sensitivity using the trick introduced in section **??**. We take $\theta_* = 1.7$ and choose a prior of mean $\hat{\theta}_0 = 2.0$ and covariance $C_0 = 1$. We minimize the entropy criterion with the Adam optimizer [**?**] using a learning rate of 0.1.

Results The results of this experiment are presented on figure **??**. We plot the posterior parameters, the design sequence and the error $|\hat{\theta}_n - \theta_*|$ versus the iterations. The posterior mean converges to θ_* and the posterior variance tends to 0, proving that our estimations converge with no bias and hence that the approximate Gaussian inference updates are valid. We also note that the design obtained by automatic differentiation of the sensitivity perfectly matches the analytical optimal design $t = 1/\hat{\theta}$. This means that automatic differentiation successfully computes the model sensitivity and its gradient with respect to the design, hence allowing gradient descent on (**??**).

6.6 Discussion

We introduced a novel tehnique for optimal design of experiments based on automatic differentiation. We started by deriving a linear approximation of the model and the corresponding quadratic mutual information criterion [**?**]. We then leveraged automatic differentiation twice to both compute and differentiate the model sensitivity, hence allowing for a gradient-based search of the optimal design. We also proposed a sequential design algorithm where inference is based on the same linear approximation. We applied this inference method on a simple, yet implicit model and our results match theory.

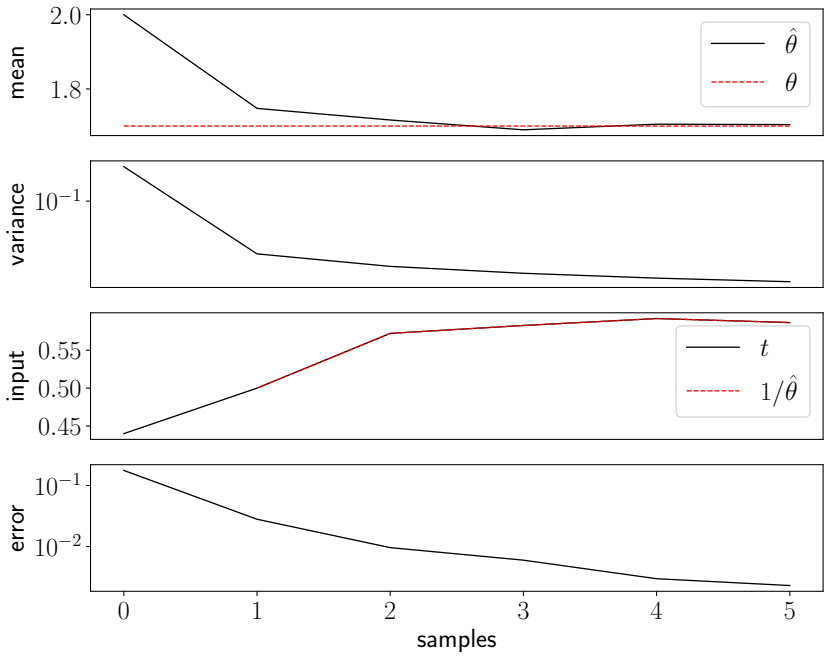


Figure 17: Sequential optimal design for model (**??**). The design is obtained by computing the model sensitivity by automatic differentiation and performing gradient descent.

This new approach is in contrast with the current research directions and could provide a new method for optimal design. Still, the sequential design paradigm poses the question of how to infer the parameters of implicit models. This is a very active field of research, and solutions based on automatic differentiation have recently been proposed [**?**]. It might be interesting to combine our method for optimal design with existing AD-based method for parameter inference.

7 Conclusion

In this work, we studied deep implicit layers and applied them to a variety of applications. We defined a theoretical framework for automatic differentiation and introduced some implicit layers with a derivation of their adjoint equations. We then proposed applications to a variety of problems. We used neural differential equations as a numerical optimization tool which allowed us to solve differential equation optimization problems by gradient descent. We then proposed automatic-differentiation-based methods for optimal design along with experimental evidence proving their efficiency.

Implicit layers provide an elegant way of posing a problem and integrating it into an differentiatiable programming framework. They allow us to use and differentiate through advanced mathematical techniques such as differential equations, matrix decompositions or thrird-party-variable derivatives. Our results demonstrate that implicit layers allow for numerical optimization on a variety of new tasks such as differential equations optimization, optimal control and experimental design. Besides, deep implicit models inherit the advantages of deep learning including powerful gradient descent optimizers, GPU parallelization and arbitrarily large expressiveness. Our experimental results match the already-existing ones and the analytical solutions with great precision.

There are many interesting research directions linked to our work. Although implicit layers prove to be very efficient for a variety of tasks, the way they learn remains poorly understood. The particular weight-sharing, continuous architecture of neural ODEs for example, makes them arguably harder to understand than classical neural nets. In our experiments, this materializes by somewhat irregular training losses. It would be interesting to compare the loss landscapes of both of these types of layers. More generally, implicit layers lack a solid theoretical background, as opposed to the more classical approaches we faced. We also highlighted the connection between reinforcement learning and neural control, which could bridge the gap between classical RL and continuous problems. This direction is studied in [**?**] and could be further investigated. Finally, our approach of differentiable experimental design exemplifies the general idea of studying complex, implicit models at the first order, an approach that can lead to surprisingly general outcomes [**?**].