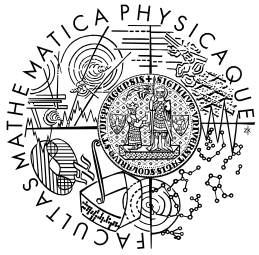

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Přemysl Kaska

Solving variational problems by machine learning

Mathematical Institute of Charles University

Supervisor of the bachelor thesis: doc. RNDr. Michal Pavelka, Ph.D.
Study programme: Mathematics
Study branch: Mathematical modeling

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, doc. RNDr. Michal Pavelka, Ph.D., for the introduction to this topic and for counseling. I would also like to thank my family for their never-ending support.

Title: Solving variational problems by machine learning

Author: Přemysl Kaska

Institute: Mathematical Institute of Charles University

Supervisor: doc. RNDr. Michal Pavelka, Ph.D., Mathematical Institute of Charles University

Abstract: The bachelor thesis aims to explore various basic approaches to the calculus of variations problems in physics using machine learning. Our goal is to review these methods on simple, specific problems and compare them to more traditional numerical methods. The thesis begins with an introduction to the calculus of variations in physics and to neural networks. Next, I choose basic, variational problems, including the brachistochrone, harmonic oscillator, and the Laplace and Poisson equation, and provide their analytical solutions. Finally, I introduce machine learning methods such as direct minimization of the action functional, minimizing the functional with neural networks and Neural ODEs, and apply them to our problems. The experiments are run in the programming language Julia, and the results are then discussed and compared with the analytical solution and other numerical methods.

Keywords: neural networks, calculus of variations, machine learning

Contents

Introduction

In today's age, machine learning has become extremely influential, with applications in various fields, including speech and image recognition, natural language processing, informatics, and natural sciences. Physics is no exception, and many new machine-learning techniques are being researched. Raissi et al. [2019], Greydanus et al. [2019], Chen et al. [2019]

Deep learning, in particular, is what we are interested in in this thesis. And there is no deep learning without neural networks. Neural networks are functions with many parameters that are tweaked in such a way that the network can approximate any function. To learn those parameters, we need to minimize a so-called loss function Goodfellow et al. [2016, Section 4.3].

In physics, there is a powerful principle called the Principle of Least Action Goldstein et al. [2002]. The action is a functional whose stationary points give the trajectory of a physical system. In some cases, which will be discussed, the trajectory is a minimum point, and therefore, this functional can be minimized and used as a cost function.

With that in mind, the goal of this work is to become acquainted with several basic deep-learning methods, particularly for solving variational problems. I will implement these methods, test them on some basic problems, and evaluate the results by comparing them to other numerical methods.

Structure of the thesis

The first chapter dives into the background of the calculus of variations and its application in physics and introduces tools to solve simple problems analytically. In the second chapter, deep learning with neural networks is introduced. The third chapter addresses simple boundary value problems, including the brachistochrone problem, the harmonic oscillator, and the Laplace equation. These problems are then solved analytically, and all can be viewed as minimization tasks. Finally, the fourth chapter introduces three methods for solving chosen problems:

1. Discretizing the action and minimizing it as a function of its inner nodes.
2. Discretizing the action and minimizing it as a function of neural network parameters for different boundary conditions.
3. Approximating the right-hand side of governing ODEs as a neural network function.

By applying these methods to our problems, we will discuss their advantages and disadvantages and compare the results with traditional numerical methods, such as numerically solving the Euler-Lagrange equations with finite difference schemes.

Related work

The first method is based on the article Nature's cost function Strang et al. [2023], and the third method has been introduced by Chen et al. [2019]. Similar work

also includes the Deep Ritz method E and Yu [2018] and Physics-Informed Neural Networks (PINNs) Raissi et al. [2019].

1. Introduction to Calculus of Variations

This chapter covers the basics of the calculus of variations and its application in physics, giving us the necessary framework to use in Chapter 3. Essentially, this is an excerpt from selected chapters in Czech books [Podolský, 2024] and [Černý and Pokorný, 2021], but it can be found in most introductory books on the topic, such as Goldstein et al. [2002], Morin [2008] where the reader can find more details. With that in mind, the proofs in this chapter are omitted.

Our goal is to work with infinite dimensional spaces, namely the space of functions on closed intervals, but first, we need to define a few key concepts.

1.1 Theory

The following definitions and theorems can all be seen with proofs in chapters 13.1 - 13.3 here [Černý and Pokorný, 2021].

Definition 1 (Functional)**.** *A mapping from linear normed space to* $\mathbb{R}$ *is called a functional.*

Definition 2 (Gâteaux Derivative)**.** *Let* X *be a linear normed space,* $F : X \to \mathbb{R}$ *be a functional, and* $a \in D_F$*. Let* $h \in X$ *and suppose there exists* $\delta > 0$ *such that* $\{a + th : |t| < \delta\} \subset D_F$*. Then we say that* F *has a Gâteaux derivative at* a *in the direction* h *if the following limit exists:*

$$\delta F(a; h) \equiv \lim_{t \to 0} \frac{F(a + th) - F(a)}{t} = \frac{d}{dt} F(a + th)\bigg|_{t=0}.$$

Definition 3 (Local Minimum)**.** *Let* X *be a normed linear space and* $F : X \to \mathbb{R}$ *be a functional. We say that* $a \in D_F$ *is a point of local minimum of the functional* F *if there exists* $\varepsilon > 0$ *such that* $F(a) \leq F(x)$ *for all* $x \in U_\varepsilon(a) \cap D_F$*.*

Remark. The local maximum is then defined analogously.

Definition 4 (Stationary Point)**.** *Let* X *be a linear normed space and* $F : X \to \mathbb{R}$ *be a functional. We say that* $a \in D_F$ *is a stationary point of the functional* F *if* $\delta F(a; h) = 0$ *for all* $h \in X$*.*

A variational problem, in the context of this text, is then meant to be a problem of finding this stationary point for a given functional. Since our goal is eventually to minimize functionals, the following theorem plays a key role. In the next section, we work with a special type of functional (given by integral), and its application will give us nice results.

Theorem 1 (Necessary Condition for Minimum (Maximum))**.** *Let* X *be a linear normed space. Let* $F : X \to \mathbb{R}$ *have a local minimum (maximum) in* $a \in X$*. and be defined on the neighbourhood of* a*. It then holds: If* $\delta F(a; h)$ *exists, then* $\delta F(a; h) = 0$*.*

The concepts defined above are similar to multivariable calculus. For example, the Gâteaux derivative is just the directional derivative in the context of finite-dimensional spaces. This thinking can give us some intuition and will be useful for the next chapters. In calculus, we needed a second derivative to classify extremals. We proceed with the same here, and it is especially important for us since the actual problems we focus on are problems where the stationary point is a minimum.

Definition 5 (Second Gâteaux Derivative)**.** *Let X be a linear normed space and $F : X \to \mathbb{R}$ be a functional. Let $a, h, k \in X$ and $\delta F(a; h)$ exist. If the following limit exists, it is called the second Gâteaux derivative in the direction h and k:*

$$\delta^2 F(a; h, k) \equiv \lim_{t \to 0} \frac{\delta F(a + tk; h) - \delta F(a; h)}{t}.$$

Theorem 2 (Sufficient Condition for Minimum)**.** *Let X be a linear normed space, $a \in X$ be a stationary point of the functional $F : X \to \mathbb{R}$, and let $h \in X$. If $\delta^2 F(a; h, h) \leq 0$ for all $h \in X$, then F has a local minimum in a.*

Lastly, we define the convexity of a functional, and with that, we are able to classify even a global minimum.

Definition 6 (Convex Functional)**.** *Let X be a linear normed space and $M \subset X$ be convex set. Functional $F : X \to \mathbb{R}$ is said to be convex if:*

$$F(\lambda x + (1 - \lambda)y) \leq \lambda F(x) + (1 - \lambda)F(y) \text{ for all } x, y \in M \text{ and } \lambda \in [0, 1].$$

Theorem 3 (Global Minimum for Convex Functional)**.** *Let X be a linear normed space and $F : X \to \mathbb{R}$ is a convex functional on the whole X. Then each stationary point is a point of global minimum of F on X.*

1.2 Functionals Represented by an Integral

Let us consider the following functional:

$$F(y) \equiv \int_a^b f(x, y(x), y'(x))\, dx \tag{1.1}$$

on a set $M := \{y \in C^1([a, b]) : y(a) = A, y(b) = B\}$.

Meaning we want to keep the endpoint fixed. This is of course not a linear set because for $y_1 \in M$, and $y_2 \in M, y_1 + y_2 \notin M$ for nontrivial A, B. This can be easily fixed by defining a line v between A and B and setting $y = u + v$, where $\Phi(u) \equiv F(u + v)$. We then get $u(a) = 0 = u(b)$ and therefore a linear set. Having defined the functional, let us examine its stationary points.

Euler-Lagrange Equation

Provided results give us a crucial theorem. The condition for a point y to be stationary of $F(y)$ is given by an ODE.

Theorem 4 (Euler-Lagrange Equation)**.** *Let $f \in C^2([a,b] \times \mathbb{R}^2)$ and $y(x) \in M$. Then a necessary condition for y(x) to be a stationary point of $F(y)$ (1.1) is that $y(x)$ solves the following Euler-Lagrange equation:*

$$\frac{d}{dx}\left(\frac{\partial f}{\partial y'}\right) - \frac{\partial f}{\partial y} = 0. \tag{1.2}$$

If the functional depends on more independent functions, e.g. $F(y_1(x), y_2(x))$, we get a system of ODEs, and if these functions depend on more variables, e.g. $F(y(x_1, x_2))$, we get a PDE. We will disclose it briefly in the next section. Finally, we state a theorem that helps us to classify the stationary points.

Definition 7 (Jacobi's Equation)**.** *Jacobi's equation is a differential equation*

$$-(Ph')' + Qh = 0,$$

where:

$$P(x) := \frac{\partial^2 f}{\partial z^2}(x, y_0(x), y_0'(x)) > 0$$

and:

$$Q(x) := \frac{\partial^2 f}{\partial y^2}(x, y_0(x), y_0'(x)) - \frac{d}{dx}\left(\frac{\partial^2 f}{\partial y \partial z}(x, y_0(x), y_0'(x))\right).$$

Point $x \in (a,b]$ is said to be conjugate to a, if a nontrivial solution satisfying $h(a) = h(x) = 0$ exists.

Theorem 5 (Jacobi's Theorem)**.** *Let $f \in C^3([a,b] \times \mathbb{R}^2)$, $y_0 \in M \cap C^2([a,b])$ be a stationary point of a functional F, $f_{zz}(x, y_0(x), y_0'(x)) > 0$ on $[a,b]$ and P and Q are defined as above. It then holds:*

If there is no point conjugate to a on $(a,b]$, then y_0 is a point of a local minimum of F.

1.3 Applications in Physics

In physics, this theory yields an elegant and versatile principle formulated in the following theorem.

Theorem 6 (Hamilton Variational Principle)**.** *The motion of the system in the time interval $t \in [t_1, t_2]$ takes such a path $\mathbf{q_0}(\mathbf{t})$, that $\mathbf{q_0}(\mathbf{t})$ is the stationary point of the action functional S defined as:*

$$S \equiv \int_{t_1}^{t_2} L(t, \mathbf{q}(\mathbf{t}), \mathbf{q}'(\mathbf{t}))\, dt, \tag{1.3}$$

where L i the Lagrange function defined $L = T - V$ (difference between kinetic and potential energy) and $\mathbf{q}, \mathbf{q}'$ are generalized coordinates and velocities. Therefore, $\mathbf{q_0}(\mathbf{t})$ must satisfy the system of Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \mathbf{q_0'}}\right) - \frac{\partial L}{\partial \mathbf{q_0}} = 0. \tag{1.4}$$

.

Lagrange equations are the equations of motion, and they are equivalent to Newton's equation [Goldstein et al., 2002, Chapter 1]. Having said that, the only thing we need for studying the dynamics of a system and its evolution is this Lagrange function. This is how we approach some of our problems in Chapter 3.

We end this chapter by mentioning what happens when we go from $q^j(t)$ to a dense setting. For example, a scalar field $\varphi(\mathbf{x}, t)$. Then, the Lagrangian takes this form:

$$\mathcal{L}(\varphi, \boldsymbol{\nabla}\varphi, \frac{\partial \varphi}{\partial t}, \mathbf{x}, t) \tag{1.5}$$

and we call it the Lagrangian density function. Furthermore, we get:

$$\mathcal{S}[\varphi] \equiv \int_{t_1}^{t_2} L\, dt = \int_{\Omega} \mathcal{L}(\varphi, \boldsymbol{\nabla}\varphi, \frac{\partial \varphi}{\partial t}, \mathbf{x}, t)\, \mathrm{d}^3\mathbf{x}\, \mathrm{d}t. \tag{1.6}$$

where:

$$L = \int_V \mathcal{L} \mathrm{d}^3\mathbf{x},$$

The Lagrange equation is then a partial differential equation:

$$\sum_{\mu=0}^{3} \frac{\partial}{\partial x^\mu} \left(\frac{\partial \mathcal{L}}{\partial_\mu \varphi} \right) - \frac{\partial \mathcal{L}}{\partial \varphi} = 0, \tag{1.7}$$

where:

$$\partial_\mu \varphi \equiv \frac{\partial \varphi}{\partial x^\mu}.$$

Remark. The equations (1.4) and (1.7) are boundary value problems and, therefore, in general, do not have a unique solution. In this thesis, however, we only look at problems with a unique solution.

2. Introduction to Deep Learning

Machine learning is a broad field of study that studies programs that can learn according to the experience they get. A Possible definition is given by Mitchell [1997]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Now, the task, measure, and performance can be a variety of things, but since we want to talk about deep learning, the following will be sufficient for us.

Deep learning is a part of machine learning based on neural networks. A neural network (our computer program) is the model that executes task T. An example is image recognition, a form of classification in which the Neural network assigns a category to a given input (image). The experience is then the process of predicting what is on the image. In our case, it will be a supervised experience. This means we know what is on the image, and we can compare it with the prediction using the measure P, which, in this case, is called a loss function.

Now, we let us define it more precisely, but sometimes, we still stick with the example of image recognition for more clarity. For a complete introduction, see Part I and II Goodfellow et al. [2016], and online lectures 1-3 Straka [2024], as they were the main resources for this chapter.

2.1 Neural Networks

A neural network, also called an artificial neural network, is a collection of neurons, just like the name suggests. These neurons influence each other, and their relationship may vary, giving us different types of neural networks.

A classic type of neural network, and the one we focus on, is a multilayered perception. This a network that consists of several fully connected neuron layers. The first layer is called an input layer, it is followed by a number of hidden layers and finished with an output layer. Figure (2.1) illustrates the fully connected neural network.

Remark. We denote x_i for input layer neurons, $h_i^{(k)}$ for the hidden (k for k-th hidden layer), and y_i the output layer neurons.

Remark. The name deep learning comes from the depth of the layers. This means that it is common for some tasks to use many hidden layers.

Each neuron is then influenced by all neurons from the previous layer in the following way. For the first hidden layer, for the next hidden layers and for the

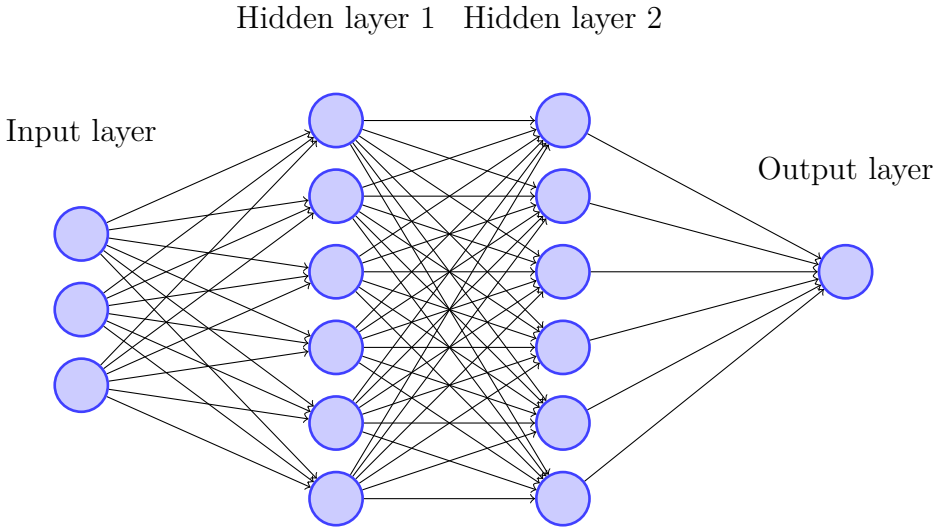

Figure 2.1: Neural network architecture illustration.

output layer, respectively:

$$\begin{aligned} h_i^{(1)} &= f\left(\sum_{j=1}^{N}(w_{ij}^{(1)}x_j) + b_i^{(1)}\right), \\ h_i^{(k)} &= f\left(\sum_{j=1}^{N}(w_{ij}^{(k)}h_j^{(k-1)}) + b_i^{(k)}\right), \\ y_i &= g\left(\sum_{j=1}^{N}(w_{ij}^{(n)}h_j^{(n-1)}) + b_i^{(n)}\right), \end{aligned} \tag{2.1}$$

where $w_{ij}^{(k)}$ are called weights that connect j-th element from the $(k-1)$-th layer to the i-th element from the k-th layer, $b_i^{(k)}$ is called a bias to neuron $h_i^{(k)}$ and f, g are non-linear activation functions.

For hidden layers activation function, common choices are $\tanh(x)$ or $ReLU(x) = \max(0, x)$, but theoretically, any non-polynomial function works (see next section (8)). The output layer activation depends on our task. Going back to our image recognition example we would use:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

for binary classification Wikipedia contributors [2024a], or:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for a distribution classification.

More compactly described with matrices:

$$\begin{aligned} \mathbf{h^{(1)}} &= f(\mathbf{W^{(1)}x} + \mathbf{b^{(1)}}), \\ \mathbf{h^{(k)}} &= f(\mathbf{W^{(k)}h^{(k-1)}} + \mathbf{b^{(k)}}), \\ \mathbf{y} &= g(\mathbf{W^{(n)}h^{(n-1)}} + \mathbf{b^{(n)}}), \end{aligned} \tag{2.2}$$

where by $\mathbf{y} = f(\mathbf{x})$, we mean elementwise application of f: $y_i = f(x_i)$. Furthermore, we say that the collection of all weights and biasses are parameters $\boldsymbol{\theta}$. The output layer is then only a function of the input one, and the parameters, with linear and non-linear mappings applied alternatively:

$$\mathbf{y} = \mathbf{NN}(\mathbf{x}, \boldsymbol{\theta}).$$

2.1.1 Universal Approximation Theorem

Universal approximation theorems explain why neural networks are as useful as they are: Their structure allows them to approximate any continuous function. It was believed that simple 2-layer perceptrons couldn't represent functions very well until a 1989 paper by Cybenko [1989], which proved the following.

Definition 8 (Sigmoidal)**.** *A function $\sigma(t)$ is called a sigmoidal function if*

$$\sigma(t) \to 1 \text{ as } t \to +\infty, \quad \text{and} \quad \sigma(t) \to 0 \text{ as } t \to -\infty.$$

Theorem 7 (Universal Approximation)**.** *Let I^n be $[0,1]^n$, let σ be any continuous sigmoidal function. Then finite sums of the form:*

$$G(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{w}_j \cdot \mathbf{x} + \theta_j)$$

are dense in $C(I^n)$. In other words, given any $f \in C(I^n)$ and $\epsilon > 0$, there is a function $G(x)$ of the above form, for which

$$|G(x) - f(x)| < \epsilon \quad \text{for all } x \in I^n.$$

Remark. In our notation, it would be:

$$G(\mathbf{x}) = W^{(2)} \cdot (f(\mathbf{W^{(1)}}\mathbf{x} + \mathbf{b^{(1)}})),$$

meaning it is sufficient to have only one hidden (arbitrary big) layer with a weight matrix $W^{(1)}$ and one output neuron with a weight matrix $W^{(2)}$ to approximate any scalar function.

It was then generalized into the following theorem Wikipedia contributors [2024b].

Theorem 8 (general universal approximation)**.** *Let $\sigma \in C(\mathbb{R}, \mathbb{R})$.*

Then σ is not polynomial if and only if for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m)$, $\varepsilon > 0$, there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that:

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$$

where $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$.

Remark. This gives us an approximation of any function on a compact with an arbitrary non-polynomial function. The proof can be found here: Pinkus [1999].

2.1.2 Training Neural Networks

Now that we know a neural network can be a good approximation, we need to know how it actually learns. Training can then be thought of as repeatedly predicting an output and measuring how well the task was done. When training, we work with training data. A set of inputs and their corresponding outputs. As already mentioned, we use a so-called lost function to measure the task's performance. The most common Loss function is the mean square error:

$$L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - f_i(\mathbf{x};\boldsymbol{\theta})\right)^2. \tag{2.3}$$

Here, $\mathbf{x}$ are the training inputs $\mathbf{f}(\mathbf{x};\boldsymbol{\theta})$ is the model output (output layer), and $\mathbf{y}$ are desired (training) outputs.

In our image recognition case, now consider assigning a category to a black-and-white picture (for example, what number is there). The input would be a vector $\mathbf{x}$, where each element x_i gives value to a pixel depending on its shade, and the output would be a probability distribution $\mathbf{y}$, where y_i is the probability of the i-th category.

This is where the supervised learning comes from because during training, we know what there is on the picture so that the output will be just a canonical basis vector, and now we can compare it with our model prediction $\mathbf{f}(\mathbf{x};\boldsymbol{\theta})$ by the loss function (2.3).

The goal is then to minimize this lost function with respect to the neural network's parameters since we want the prediction to be as good as possible. There are many ways of minimizing functions, but in the context of neural networks methods based on gradient descent are the most widely used.

Remark. In training, the loss function itself is not always the function to be minimized; for this, we denote an error function:

$$E(\boldsymbol{\theta}) = \mathbb{E}_{(x,y)\sim p_{\text{data}}} L(\mathbf{f}(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}),$$

where we take the mean from the gradient. Standard (batch) gradient descent is then thought of as minimizing this error function for all training data. Stochastic descent is when we choose one random sample at each iteration, effectively minimizing the loss function itself.

2.1.3 Gradient Descent-based Methods for Optimization

Our goal is to minimize the error function:

$$E(\boldsymbol{\theta})$$

with respect to $\boldsymbol{\theta}$.

Since the gradient vector is in the direction of the function's steepest growth, it is natural to go in the opposite direction. Regular gradient descent is then defined using the following iterative method:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\nabla_\theta \mathsf{E}\left(\boldsymbol{\theta}_{t+1}|t\right),$$

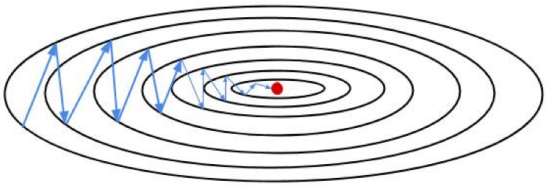

Figure 2.2: Zig-Zag effect for gradient descent.

where α is a constant called a learning rate. If the gradient is Lipschitz continuous and we choose sufficiently small α, then the gradient descent converges to a local minimum. However, it can be very slow as the Zig-Zag effect can occur (see the picture (2.2) taken from Hu [2019]). Also, once we reach a local minimum, it is stuck there if the learning rate is small.

Adding momentum helps with this, giving us a gradient descent with momentum:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t - \alpha \nabla_\theta E(\boldsymbol{\theta}|t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1},$$

where β is called momentum, and it symbolizes how much we remember the direction from the past steps.

Even better methods are with adaptive learning rates. They change the learning rate and momentum at each step and are the methods used in practice. Currently, one of the best ones is the Adam (adaptive momentum) optimizer introduced by Kingma and Ba [2017].

2.1.4 Backpropagation

Now that I have introduced numerical minimization algorithms based on gradient descent. We will end this chapter by showing how to actually compute the gradient for a standard mean square error loss (2.3). Luckily, in practice, we don't have to do this since there exist many automatic differentiation tools we can use. For now, imagine a stochastic descent with the loss:

$$L(f(x;\boldsymbol{\theta}), y) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f_i(x;\boldsymbol{\theta})\right)^2.$$

The name backpropagation comes from the fact that we first compute derivatives with respect to the last layer weights and second to last layer neurons, and using the chain rule, we compute the derivatives at each layer going backwards:

$$\frac{\partial L}{\partial w_{i,j}^{(n)}}, \frac{\partial L}{\partial b_i^{(n)}}, \frac{\partial L}{\partial h_i^{(n-1)}}.$$

These are also computed using the chain rule. Here f_i in the loss function is the output neuron y_i from (2.1), and for the simplicity of notation, we denote:

$$z_i^{(k)} = \sum_{j=1}^{N} (w_{ij}^{(k)} h_j^{(k-1)}) + b_i^{(k)}.$$

And we get:

$$\begin{aligned}
\frac{\partial L}{\partial w_{i,j}^{(n)}} &= \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial z_i^{(n)}}\frac{\partial z_i^{(n)}}{\partial w_{i,j}^{(n)}}, \\
\frac{\partial L}{\partial b_i^{(n)}} &= \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial z_i^{(n)}}\frac{\partial z_i^{(n)}}{\partial b_i^{(n)}}, \\
\frac{\partial L}{\partial h_i^{(n-1)}} &= \sum_k \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial z_k^{(n)}}\frac{\partial z_k^{(n)}}{\partial h_i^{(n-1)}}.
\end{aligned} \tag{2.4}$$

The computation of the derivatives with respect to the last hidden layer neurons is especially important since now we can imagine it as the output layer, and from that, compute:

$$\frac{\partial L}{\partial w_{i,j}^{(n-1)}}, \frac{\partial L}{\partial b_i^{(n-1)}}, \frac{\partial L}{\partial h_i^{(n-2)}}$$

and so on, all the way to the first layer. Completing our backpropagation and giving us the gradient with respect to all parameters.

3. Analytical Solution of Classical Problems

In Chapter 1, I introduced a framework for finding a path of a physical system by solving Euler-Lagrange equations and showed how it corresponds to a stationary point of an action integral. Now, we use it to solve classical boundary value problems and obtain analytical solutions, which will then be the benchmark for further machine learning methods. Namely, I looked at the Brachistochrone Problem, Poisson and Laplace equations in a specific setting and a harmonic oscillator problem. Since in the next Chapter, we will be minimizing functionals; we need to discuss the cases when the action (or other functional) is indeed minimal. This constraint is a sizeable one, and it will limit our range of problems quite a bit. But there is still some variety left. Elements of this Chapter were again partly chosen from Podolský [2024].

3.1 Brachistochrone

Due to its historical significance, one of the introductory problems to the calculus of variations is always a brachistochrone problem. From Greek "bráchistos chrónos" meaning shortest time. It aims to make the fastest 2-point connection (slide) in a gravitational field.

Formulation and Solution

Let us find a smooth function $y(x)$, such that the ball sliding from point $A = y(a)$ on the curve $y(x)$ will reach the point $B = y(b)$ in the shortest possible time. Without the loss of generality and for more simplicity. let $A = 0, a = 0$.

The time it takes to slide from A to B is given by the following integral:

$$t = \int_0^b dt = \int_0^b \frac{dl}{v} = \int_0^b \frac{\sqrt{1 + y'^2}}{\sqrt{2gy}} dx, \tag{3.1}$$

where v is substituted from the conservation of energy law $1/2mv^2 - mgy = 0$.

Remark. I chose the y-axis to be positive in the downward direction, just so as not to deal with the negative sign.

Before getting the solution, one more theorem is helpful.

Theorem 9 (Beltrami Identity)**.** *If the function $f(y, y')$ defined* (1.1) *does not explicitly depend on x, and $y(x)$ solves The Euler-Lagrange equation* (1.2)*, then it holds:*

$$f - y' \frac{\partial f}{\partial y'} = C.$$

Let us at least sketch a proof.

Proof.

$$\frac{d}{dt}\left(f - y'\frac{\partial f}{\partial y'}\right) = \frac{\partial f}{\partial y}y' + \frac{\partial f}{\partial y'}y'' - y''\frac{\partial f}{\partial y'} - y'\frac{d}{dt}\left(\frac{\partial f}{\partial y'}\right) = y'\left(\frac{\partial f}{\partial y} - \frac{d}{dt}\left(\frac{\partial f}{\partial y'}\right)\right) = 0,$$

where the last part is obtained from the satisfaction of the Euler-Lagrange equation. □

Given this, for brachistochrone, we get:

$$f(y, y') = \frac{\sqrt{1 + y'^2}}{\sqrt{2gy}}$$

and:

$$\frac{\partial f}{\partial y'} = \frac{y'}{\sqrt{2gy}\sqrt{1 + y'^2}},$$

and therefore, the equation we need is in the form:

$$C = \frac{\sqrt{1 + y'^2}}{\sqrt{2gy}} - \frac{y'^2}{\sqrt{2gy}\sqrt{1 + y'^2}}$$

$$C = \frac{1}{\sqrt{2gy}\sqrt{1 + (y')^2}}$$

$$y(1 + y'^2) = C. \tag{3.2}$$

This equation can then be solved by the separation of parameters. However, the solution would not be pleasant. Instead, it is known that the solution is given by a parametric function:

$$\begin{aligned} x &= \frac{1}{2}C(\phi - \sin\phi), \\ y &= \frac{1}{2}C(1 - \cos\phi), \\ \phi &\in (0, T), T < \pi. \end{aligned} \tag{3.3}$$

We can show this by looking at (3.2) in a differential form:

$$y(dx^2 + dy^2) = Cdx^2$$

$$\frac{1}{2}C(1 - \cos\phi)[(\frac{1}{2}C(1 - \cos\phi)\, d\phi)^2 + (\frac{1}{2}C(\sin\phi)\, d\phi)^2] = C(\frac{1}{2}C(1 - \cos\phi)\, d\phi)^2$$

$$\frac{1}{2}[(1 - \cos\phi)^2 + (\sin\phi)^2] = 1 - \cos\phi$$

$$\frac{1}{2}(2 - 2\cos\phi) = 1 - \cos\phi.$$

This holds, proving that our parametric function is a solution and by fixing the parameters C and T, we can find the solution for the boundary values.

The curve (3.3) is called a cycloid, and it solves our problem, but we only showed that it is a stationary point of our functional. Nevertheless, it can be proven that this solution is a unique stationary point, and it is a minimum Coleman [2012]. This is what our intuition would tell us: we could just tweak the slide in such a way that the ball would slide faster, and that is, by the way, going to be the first approach in the next Chapter.

3.2 Laplace and Poisson Equation

Traditionally Laplace equation is understood as:

$$\Delta u = 0, \tag{3.4}$$

where Δ is a differential operator defined:

$$\Delta u = \sum_{k=1}^{N} \frac{\partial^2 u}{\partial x_k^2},$$

and by Poisson equation:

$$-\Delta u = f. \tag{3.5}$$

Before trying to solve these equations in a specific setting, let us look at what they have in common with the calculus of variation and minimization; we take two different views. These are not meant to be rigorous and are shown to connect the problem to minimization; therefore, some key concepts are just referenced and not properly defined. The first one is from a view of Field theory and, in particular, the electrostatic field. The Lagrangian density (3.6) has a form of:

$$\mathcal{L}(\mathbf{x}) = \frac{\epsilon_0}{2}(\nabla V(\mathbf{x}))^2 - \rho(\mathbf{x})V(\mathbf{x}), \tag{3.6}$$

where ϵ_0 is a permittivity of a free space constant $V(\mathbf{x})$ is an electrostatic potential and $\rho(\mathbf{x})$ is a charge density.

Therefore from (1.7), we obtain:

$$\begin{aligned} 0 &= \rho(\mathbf{x}) + \epsilon_0 \nabla \cdot \nabla V(\mathbf{x}) \\ -\frac{\rho(\mathbf{x}, t)}{\epsilon_0} &= \Delta V(\mathbf{x}). \end{aligned}$$

Meaning that the potential is the stationary value of the integral

$$\int_\Omega \mathcal{L}\, \mathrm{d}^3\mathbf{x}.$$

If the charge density is considered to be zero, it can be shown, under additional conditions, that the functional is convex and by (3) the potential is a local minimum.

The other view is through a weak solution of the equation and the Ritz method.

Starting from the equation (3.5):

$$-\Delta u = f$$

on Ω open bounded. We multiply both sides by a specific test function $v \in V$ that vanishes on the boundary (usually taken from a Sobolev space [Bulíček et al., 2018, Chapter 2]) and integrate both sides over Ω:

$$\int_\Omega (\Delta u) v \, dx = \int_\Omega -f v \, dx,$$

and by using the integration by parts theorem, we get:

$$\int_\Omega (\Delta u)v\,dx = -\int_\Omega \nabla u \cdot \nabla v\,dx + \int_{\partial\Omega} (\nabla u \cdot n)v\,ds$$
$$\int_\Omega (\Delta u)v\,dx = -\int_\Omega \nabla u \cdot \nabla v\,dx.$$

The second term vanishes due to the functions v belonging to a specific (Sobolev space). Taking all this together, we get an integral equation called a weak formulation of the equation:

$$\int_\Omega \nabla u \cdot \nabla v\,dx = \int_\Omega fv\,dx \quad \forall v \in V, \tag{3.7}$$

or standardly noted:

$$a(u,v) = L(v), \quad \forall v \in V, \tag{3.8}$$

where $a(u,v)$ is a bilinear function and $L(v)$ a linear function.

The weak solution is then defined as such a function u that solves (3.7) for all test functions $v \in V$. The equivalent form to a weak solution will give us the next lemma.

Lemma 10. *function $u \in V$ is a local minimum of the integral*

$$\phi[u] = \frac{1}{2}\int_\Omega |\nabla u|^2\,dx - \int_\Omega fu\,dx, \tag{3.9}$$

only if u is a weak solution of the equation (3.7).

Remark. The lemma is taken from Bulíček et al. [2018] page 14, where the proof can be seen.

With this thinking, let us Finally solve the equation. We start with one dimension and proceed with two dimensions. For the simplicity of the solution, we take quite a specific form. However, this will not lose us any generality if the boundary has the same shape.

Consider the following equation:

$$\begin{aligned} u''(x) &= -C \quad x \in [0,b] \\ u(0) &= A, u(b) = B, \end{aligned} \tag{3.10}$$

where C is a constant. I chose a constant right-hand side because the equation is then the same as an equation for a free fall in a homogenous field (only this time it's in space and not time); this was the first toy I studied and the first problem mentioned in Strang et al. [2023].

In 1 dimension, it is just an ODE, and the general solution is a parabola:

$$u(x) = -\frac{g}{2}x^2 + kx + q.$$

By substituting the boundary condition, we get:

$$u(x) = -\frac{g}{2}x^2 + \frac{B + (g/2)b^2 - A}{b}x + A. \tag{3.11}$$

In 2 dimensions, we work on a square, and the equation is the following:

$$\begin{aligned} \Delta u(x,y) &= 0 \\ x,y \in \Omega, \quad \Omega &= [0,1]^2 \\ u(x,0) = f(x), \quad u(x,1) &= u(0,y) = u(1,y) = 0. \end{aligned} \tag{3.12}$$

Remark. This problem, where only one side is nonzero, is, in fact, sufficient to solve the equation for all boundaries. More on that can be read in the section 12.3 Trench [2013]

Let us use a Fourier separation of variables. For that, we consider the solution in the form: $u(x, y) = X(x)Y(y)$ and by applying the Laplace operator we get:

$$X''(x)Y(y) + X(x)Y''(y) = 0$$
$$\frac{X''(x)}{X(x)} = -\frac{Y''(y)}{Y(y)}.$$

The only way this can happen is when:

$$-\frac{X''(x)}{X(x)} = \frac{Y''(y)}{Y(y)} = \lambda^2.$$

We get 2 ODEs and start with the one for x since we know that $X(0) = X(1) = 0$:

$$\frac{-X''}{X} = \lambda^2$$
$$X'' + \lambda^2 X = 0$$

and with the boundary, we get:

$$X(x) = C\sin(\lambda x), \quad \lambda = k\pi, k \in \mathbb{Z}.$$

For y, we have $Y(1) = 0$, therefore:

$$\frac{Y''}{Y} = \lambda^2$$
$$Y'' - \lambda^2 Y = 0$$
$$Y(y) = C_2 e^{\lambda y} + C_3 e^{-\lambda y}$$

and by substituting $\lambda^2 = \frac{k^2}{\pi^2}$ and taking $Y(1) = 0$ we have:

$$Y(y) = C_2 e^{k\pi y} + C_3 e^{-k\pi y}$$
$$0 = C_2 e^{k\pi} + C_3 e^{-k\pi}$$
$$C_3 = -C_2 e^{2k\pi}$$
$$Y(y) = C_2 e^{k\pi y} - C_2 e^{2k\pi} e^{-k\pi y}.$$

For both X and Y, the solution holds for all λs, and therefore, after combining all the constants, the final solution is:

$$u(x, y) = \sum_{k=1}^{\infty} B\sin(k\pi x)(e^{k\pi y} - e^{2k\pi} e^{-k\pi y}) \tag{3.13}$$
$$B(1 - e^{2k\pi}) = 2\int_0^1 f(x)\sin(kx)\,dx,$$

where the last equation works in such a way that the coefficient's $B(1 - e^{2k\pi})$ are Fourier coefficients of the boundary condition.

Furthermore if we choose $f(x) = sin(x)$ we get:

$$u(x, y) = \sin(\pi x)\frac{e^{\pi y} - e^{\pi(2-y)}}{1 - e^{2\pi}}. \tag{3.14}$$

3.3 Harmonic Oscillator

The last problem in this section is going to be a harmonic oscillator. Since this is one of the most important problems in physics, it deserves some of our attention. Also it was chosen to show the behaviour of the methods in Chapter 4 for non-minimal stationary points (4.1). The Lagrangian for the oscillator has the following form:

$$L(x, x') = T - V = \frac{1}{2}mx'^2 - \frac{1}{2}kx^2.$$

Therefore, the equation will be the known equation:

$$\begin{aligned} mx(t)'' + kx(t) &= 0 \\ x''(t) + \frac{k}{m}x(t) &= 0. \end{aligned} \tag{3.15}$$

For simplicity lets choose $x(0) = A, x(b) = B$, and denote $\omega^2 = k/m$ the solution is:

$$\begin{aligned} x(t) &= C\sin(\omega t) \\ C &= \frac{B}{\sin(\omega b)}. \end{aligned}$$

Remark. if A is nonzero, the general solution is the following:

$$y(x) = A\cos(\omega x) - A\cot(b\omega)\sin(\omega x) + B\frac{\sin(\omega x)}{\sin(b\omega)}.$$

Now let us explore whether this solution is a minimum of the action functional

$$S = \int_0^b \frac{1}{2}mx'^2 - \frac{1}{2}kx^2 dt.$$

We use the Jacobi theory (5). Jacobi equation (7) has a form of

$$\begin{aligned} P(t) &= \frac{\partial^2}{\partial x'^2}\left(\frac{1}{2}mx'^2 - \frac{1}{2}kx^2\right) = m \\ Q(t) &= \left(\frac{\partial^2}{\partial x^2} - \frac{d}{dx}\frac{\partial^2}{\partial y \partial z}\right)\left(\frac{1}{2}mx'^2 - \frac{1}{2}kx^2\right) = -k \\ -(Ph')' + Qh &= -mh'' - kh = 0 \\ h(t) &= C\sin(\omega t), \end{aligned}$$

which gives the same equation as the original one. For function $x(t) = C\sin(\omega t)$ the point $\frac{k\pi}{\omega}$ is a point conjugate to 0. Therefore, we can conclude the following: If

$$t < \frac{k\pi}{\omega}, \tag{3.16}$$

then the function

$$x(t) = C\sin(\omega t) \tag{3.17}$$

is a point of a local minimum of the action. Otherwise, if

$$t > \frac{k\pi}{\omega}, \tag{3.18}$$

the function is not the minimum, and for equality, we don't know anything.

4. Machine Learning Methods for Solving Variational Problems

Finally, we get to the main part of the work. I explored Variational problems, discussed in Chapter 3 (3), numerically and with the help of Neural Networks. The experiments were realized in the programming language Julia The Julia Language [2023], with the help of packages such as Flux and SciML, and the code for the experiments is available in a Github Repository. Especially SciML is a great environment for scientific machine learning, with many tutorials and great documentation available at SciML [2023b] and served as a basis for our code.

This chapter is then divided into the introduction of the methods (4.1) and the general results (4.2), with the specific implementation of each method to each problem available at the appendix (A).

4.1 Methods

4.1.1 Directly Minimizing the Functional

The first method was just a numerical minimization of a discretized functional. I based it on a paper Strang et al. [2023]. The main reason I looked at this method is that it served as a basis for numerical minimization and was directly generalised in the subsequent method (4.1.2).

Let us show how the method works on the functional (1.1). For the problem with fixed endpoints, we then have:

$$F(y) \equiv \int_a^b f(x, y(x), y'(x))\, dx$$

Now let us discretize it considering the same time distance $dx \to \Delta x$

$$F := \sum_{i=0}^{N} f(y_i, {y'}_i, x_i)\Delta x, \quad \text{where} \quad y_i = y(x_i),\ y'_i := \frac{y(x_{i+1}) - y(x_i)}{\Delta x}. \tag{4.1}$$

By this, we get instead of $F(y_i)$ and we minimize F with respect to these inner points $\{2, 3, \ldots, N-1\}$, with the endpoint fixed, with gradient descent or better methods (Adam, etc.).

Equivalence with Finite Difference Method

Let us show that, at least in 1 dimension, discretizing the action function and then finding its minimum is equivalent to solving the Euler-Lagrange equation with the finite difference method. For our examples, let's already assume that the stationary point is a minimum. Therefore, we have the equivalent conditions.

$$\mathbf{y_0} \text{ is a minimum of } F = \sum_{i=0}^{N} f(y_i, y'_i, x_i)\Delta x$$
$$\text{if } \nabla_y F|\mathbf{y_0} = 0$$

and with the derivative being approximated by the forward difference:

$$y'_i := \frac{y(x_{i+1}) - y(x_i)}{\Delta x}, \tag{4.2}$$

We get:

$$0 = \frac{\partial f}{\partial y_i} = \Delta x \left(\frac{\partial f}{\partial y_i} + \frac{\partial f}{\partial y'_i} \frac{-1}{\Delta x} + \frac{\partial f}{\partial y'_{i-1}} \frac{1}{\Delta x} \right) = \Delta x \left(\frac{\partial f}{\partial y_i} - \frac{1}{\Delta x} \left(\frac{\partial f}{\partial y'_i} - \frac{\partial L}{\partial y'_{i-1}} \right) \right)$$

and dividing by Δx, we get the finite difference scheme contributors [2023] of the Euler-Lagrange equation:

$$\left(\frac{\partial f}{\partial y} \right)_i - \frac{1}{\Delta x} \left(\left(\frac{\partial f}{\partial y'} \right)_i - \left(\frac{\partial f}{\partial y'} \right)_{i-1} \right) = 0$$

Remark. Different discretizations would lead to different schemes.

This scheme is an example of an implicit scheme; solving it then requires solving a system of equations. For linear equations such as (3.4), (3.15), we get a system of linear equations; for that, we know that the conjugate gradient method Hestenes and Stiefel [1952] should work better than the regular gradient descent. Alongside the analytical solution, the finite difference method will be a good benchmark for our methods.

Remark. A Nonlinear system of equations would arise from a nonlinear differential equation (boundary value problem), and in both cases, we do not have a unique solution. In such cases, better numerical minimization methods are not necessarily worth it because they can escape local minima, which can also be solutions. This is not our case, however, and it is one of the reasons that we can use the Adam optimizer, which works best for all problems.

4.1.2 Minimizing the Functional with Neural Network

Next up, I utilized the universal function approximation, the neural network. Just like in regular training, we use different training data. Here, we use different boundary conditions for our problem. The boundary points will be on the network's input layer, and the output will be the same inner nodes as the previous method. Again, we consider equidistant nodes. If we denote N_Ω, the inner point nodes and $N_{\partial\Omega}$, the boundary. The neural network will be a mapping:

$$NN : N_{\partial\Omega} \to N_\Omega, \tag{4.3}$$

with the loss function being again a discretized functional with finite differences for derivatives (4.1), specifically for the functional (1.1):

$$\begin{gathered} \text{Loss}(\boldsymbol{\theta}) := \sum_{i=0}^{N} f(y_i(\boldsymbol{\theta}), y'_i(\boldsymbol{\theta}), x_i)\Delta x = \sum_{i=0}^{N} f(y_i(\boldsymbol{\theta}))\Delta x \\ \text{with} \quad y_0 = A, \quad y_N = B, \end{gathered} \tag{4.4}$$

where we take $A \in [A1, A2]$ and $B \in [B1, B2]$ and the ranges are up to our choosing and also $A, B \in N_{\partial\Omega}$.

Since the inner nodes are a neural network output, the loss function (4.4) is then the function of the neural network parameters. We will then train it by minimizing with stochastic gradient descent, meaning we change the boundary A and B at each iteration of the minimization.

Remark. Note that the loss function can take many forms. It can be a standard least square loss compared with the experimental data, or it can be the error of the governing equations (this is called PINN). These loss functions would be even more general because they don't require the stationary points to be minimal. But since we focus on variational problems, we will just stick with the action as a lost function.

The big advantage of this method is that it essentially solves the equation for different boundaries at the same time, contrary to the first method, where if we want a different boundary, the minimization would have to be done from the beginning. Of course, this will take many more iterations since we are training on multiple possible boundary conditions.

At the same time, lower precision can be expected, and so we aim to achieve at least a similar precision to the method before.

4.1.3 Neural Ordinary Differential Equations

Last but not least, we show the method called Neural ordinary differential equations proposed by Chen et al. [2019]. In its simplest form, it works in the following way. We try to approximate an ordinary differential equation (or a system):

$$\frac{dy}{dx} = f(x),$$

but at first, we don't know the right-hand side. This can be the case for non-trivial systems in different fields of study. We would like to learn it, though, and it can be done. The function $\boldsymbol{f}(x)$ will be given as a Neural network:

$$f(x) = NN(x, \boldsymbol{\theta}).$$

And the loss will be:

$$\begin{aligned} \text{Loss}\left(y(x, \boldsymbol{\theta})\right) &= \text{Loss}\left(y(a) + \int_a^b NN(y(x), x, \boldsymbol{\theta})\, dx\right) \\ &= \text{Loss}\left(\text{ODE Solve}(y(x_0), NN, \boldsymbol{\theta}, a, b)\right). \end{aligned} \tag{4.5}$$

This means the Loss function will depend on the numerical solution of the ODE, with the right-hand side being a Neural network depending on parameters $\boldsymbol{\theta}$. This loss is often given as a mean square difference from experimental data (A neat tutorial with this approach is available here: SciML [2023a]). But in our case, we once more discretize $\boldsymbol{y}(x) \to y_i$ and we take the some loss function:

$$\text{Loss}\left(y_i(\boldsymbol{\theta})\right) = \sum_{i=0}^{N} f(y_i(\boldsymbol{\theta}), y_i'(\boldsymbol{\theta}), x_i)\Delta x, \quad \text{with} \quad y_0 = A, \quad y_N = B. \tag{4.6}$$

For the numerical solution of the ODE, this method has been proposed for initial value problems, so we use the Tsitouras 5/4 Runge-Kutta method Tsitouras [2011]

since it is a default method in the Julia package DifferentialEquatins.jl. This means, however, that we have an overdetermined equation since we need to know the initial state and also the final position. This is still less data than the usual loss (2.3) from data, but gives us a restriction to a system.

An interesting question might be, what if the numerical solution was obtained by a boundary value problem such as some explicit finite difference method schemes. This goes beyond this work and can be an option for further research.

The advantage of this approach is its generality. In our case, it might seem useless to learn an equation like this if we already know it, but by learning from the data, this approach might be useful in other fields.

On the other hand, for our purpose, the precision of the solution is not going to be the best.

4.2 Experiments

Taking all this together, I applied the above-mentioned methods to the chosen problems in Chapter 3. For all methods, I tried different optimization algorithms, different discretization step, and for the methods with neural networks, I experimented with their architecture. These results for concrete problems, as well as thorough formulations of each method for each problem, are available in the appendix (A). Here follow the general findings.

Results

With all these methods, I have successfully obtained solutions to the problems. Here are some general takeaways.

Discretization

Generally, the smaller the discretization step, the more iterations we need to make to get good accuracy, except for the brachistochrone, where too big of a step causes the first step to explode due to the singularity. The error at the beginning of the middle curve can be seen in figure (4.5). Therefore, For the brachistochrone problem, the equidistant nodes are not the best option.

Optimization

For optimization methods, I first tried regular gradient descent and momentum but later used the Adam optimizer for all problems since it worked the best. The optimization was started from the inner points being zero for direct minimization, and the parameters of the neural network parameters were, by default, initialized by a Xavier scheme introduced by Glorot and Bengio [2010].

Neural network structure and hyperparameters

The other parameters, such as learning rates, activation functions, and neural network parameters, were also chosen on performance and by no means were the best possible. The networks generally had only a few layers, but for NN minimization, they had a bigger output layer and, therefore, had more parameters;

for the activation function, I mostly used $\tanh(x)$. Concretely, these choices can be seen for 1D Poisson's equation (A.1), 2D Laplace equation (A.2), oscillator (A.3), and brachistochrone (A.4).

Accuracy

When it comes to accuracy, these methods are not as good as a classic finite difference solution as can be seen in the same tables for comparing the methods here: (A.1), (A.2), (A.3), (A.4). In particular, the neural network methods can seem like using a cannon to kill a fly, so the accuracy, for such simple problems, is not their best side.

Oscilator and stationary points

Due to the restricting condition for a minimum for a harmonic oscillator (3.16), if we try a larger time domain, the solution is no longer a minimum. Indeed, when we try choosing times $t > \frac{k\pi}{\omega}$, the minimization fails, as can be seen in the figures (4.1), (4.2). This example shows the "Unconstrained Energy effect" mentioned

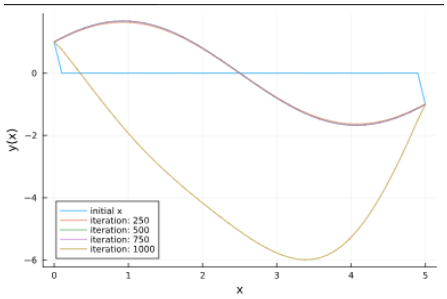

Figure 4.1: Oscilator minimal action solution.

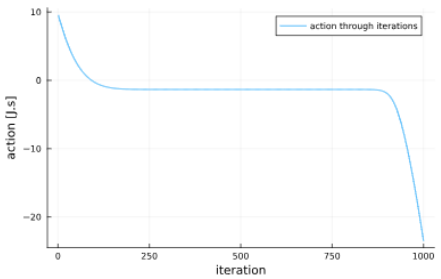

Figure 4.2: Action during the interaction process for oscilator.

in Strang et al. [2023] and is caused by the action being stationary. At first, it looks like it will converge, but in some directions, it can later escape the solution and, since we have uniqueness, never find a valid solution.

Methods

Now, let's give some key takeaways to each method.

Direct minimization

The minimization of the discretized functional as a function of inner nodes was the key to the work since the other methods were based on it. For all cases, it worked; however, it didn't reach the precision and the efficiency of the finite difference method. The picture (4.3) shows the minimization of brachistochrone for 300 inner nodes and (4.4) the Laplace equation solution with a grid of 20x20 inner nodes.

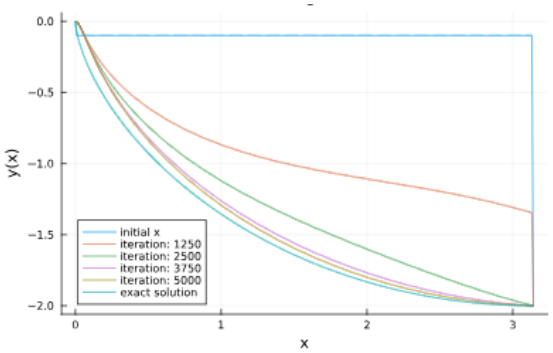

Figure 4.3: Brachistochrone solution through iterations.

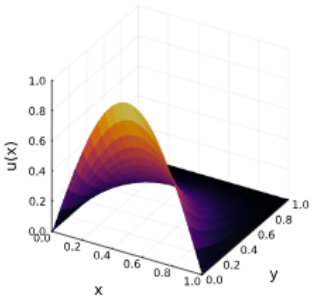

Figure 4.4: Laplace eq. minimization solution with $\sin(\pi x)$ boundary.

Neural Network minimization

For Neural network minimization, I have trained a mapping that maps certain boundary values of the problem to the inner points.

$$NN : N_{\partial\Omega} \to N_{\omega},$$

where $N_{\partial\Omega}$ are boundary points chosen onlt from a specific domain and N_{ω} are the inner points. At first, I tried it for a single boundary, essentially repeating the previous method, except this time, the inner nodes are outputs of a neural network, and later, I tried making the boundary domain larger.

The accuracy of the mapping heavily depends on the domain; generally, the larger the domain, the more iterations we need, see: (A.5), (A.6). With even more iterations, I can expect to get better accuracy, but my Julia code was not optimized for performance, so it would take a considerable time to learn. The mapping also only works on the trained domain. Unfortunately, it doesn't extrapolate the mapping outside it.

Figure (4.6) shows an output of the trained network for the left and right boundary for the oscillator, Figure (4.5) shows the output of the network for the right brachistochrone bound and a figures (4.7), (4.8) show trained network for Laplace equation with 2 boundary sides being a periodic function composed by sine functions, This, however, took way too many iterations, and for further use, should be re-implemented. Details are again in the appendix.

Neural ODEs

Lastly, for Neural Ordinary differential equations. I transformed the second-order ODEs into a system and learned the dependence of the second derivative on the first and the function itself.

$$u' = v$$
$$v' = u'' = NN(u, v, \boldsymbol{\theta}),$$

for 1D poisson equation and oscillator, this worked well, since the equations are simple (A.3), (A.9). And for brachistochrone, made use of Beltrami identity (9) and learned $x' = NN(x, \boldsymbol{\theta})$, on a certain domain by minimizing the functional. Generally, the accuracy of this method was the lowest, again visible on the tables

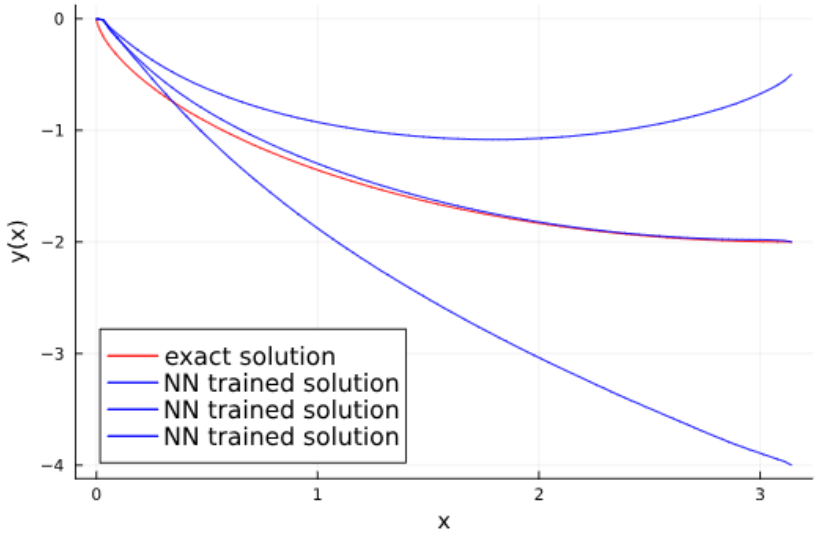

Figure 4.5: Trained brachistochrone problem for different right endpoints.

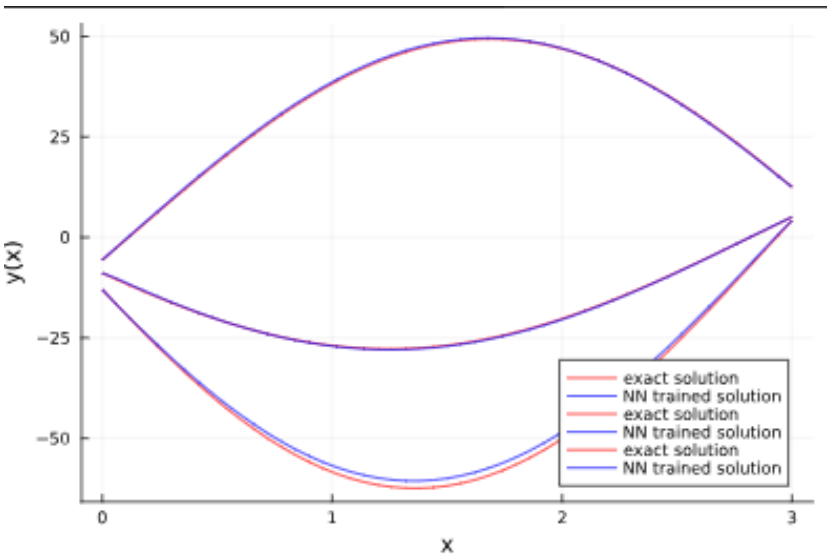

Figure 4.6: Trained oscillator solution for 3 random boundaries.

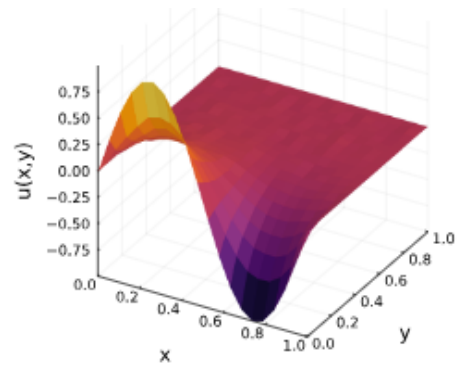

Figure 4.7: Output for trained Laplace equation network for $\sin(2\pi x)$ boundary.

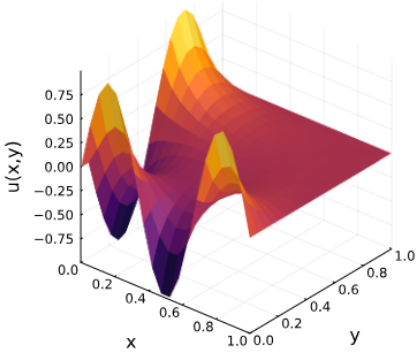

Figure 4.8: Output for trained Laplace equation network for random left and down periodic boundary.

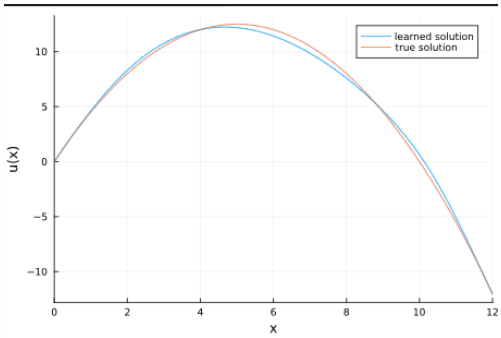

Figure 4.9: NeuralODE solution to 1D Poison equation.

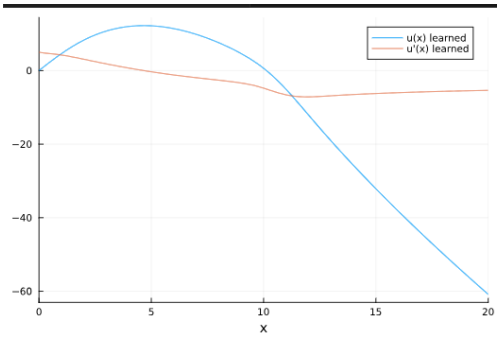

Figure 4.10: Neural 1D Poisson eq. outside the learning domain.

in the appendix, since loss (4.5) is by far the most complicated. But by the naked eye, the solution of this method resembles the actual solution see figure (4.9). The same fact applies to the network not working outside the domain applied here, as illustrated in the figure(4.10).

Conclusion

Machine learning is at the forefront of human invention and has been used for a variety of practical applications. In recent years, many researchers have studied scientific machine learning, and many new techniques have been developed. My work focused on testing some machine learning methods to solve variational problems.

In Chapter One, I introduced the basics of Calculus of variations, mainly discussed functionals represented by integral (1.1) and gave the conditions for finding stationary points (4) and how to classify them (5). Also I mentioned the action functional and formulated Hamiltion variational principle (6).

Before applying this knowledge to concrete problems, I first introduced the basics of machine learning in Chapter 2. The focus was mainly on Neural Networks (2.1). I showed their structure and stated how they behave as the "Universal approximators" (7). Then, the learning of the network was discussed, with optimization methods based on Gradient descent (2.1.3) and how the gradients are computed with backpropagation (2.1.4).

Chapter 3 followed on the calculus of variation, where I chose 3 problems that can solved by minimization and solved them analytically with the concepts from Chapter 1. Namely the brachistochrone problem (3.1), The Laplace and Poisson equation (3.2), and the harmonic oscillator (3.3) were chosen. Then, I discussed that the solutions are, indeed, the minima of their respective functionals.

Finally, in Chapter 4, I discussed 3 methods based on numerical minimization of a functional: Directly minimizing the discretized functional (4.1.1), Training a neural network on minimizing the functional for different boundaries (4.1.2) and Neural Ordinary differential equations (4.1.3). These methods were then successfully tested on our problems (A), showing that the action (or a functional in general) can be used as a loss function. Moreover, in the case of Neural network minimization, the neural network can be trained to map boundary nodes to inner nodes, which solve the problem (A.5), (A.6). This should be a primary goal to explore more thoroughly in the next work, specifically scaling up the models and seeing how they behave.

Bibliography

Miroslav Bulíček, Robert Černý, Oldřich John, Josef Málek, Milan Pokorný, Mirko Rokyta, and Jana Stará. *Úvod do moderní teorie parciálních diferenciálních rovnic*. May 2018. URL https://www.karlin.mff.cuni.cz/~mbul8060/moderni_teorie.pdf. PDF version.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019. URL https://arxiv.org/abs/1806.07366.

Rodney Coleman. A detailed analysis of the brachistochrone problem, 2012. URL https://arxiv.org/abs/1001.2181.

Wikipedia contributors. Finite difference method – wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/wiki/Finite_difference_method. Accessed: 2023-10-17.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989. URL https://link.springer.com/article/10.1007/BF02551274.

Weinan E and Bing Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, pages 249–256, 2010. URL http://www.jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf.

Herbert Goldstein, Charles P. Poole, and John L. Safko. *Classical Mechanics*. Addison-Wesley, 3rd edition, 2002. ISBN 978-0201653612.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL https://www.deeplearningbook.org/.

S. Greydanus, M. Dzamba, and J. Yosinski. Hamiltonian Neural Networks. In H Wallach, H Larochelle, A Beygelzimer, F d Alché-Buc, E Fox, and R Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/26cd8ecadce0d4efd6cc8a8725cbd1f8-Paper.pdf.

Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952. doi: 10.6028/jres.049.044. URL https://doi.org/10.6028/jres.049.044.

Jiawei Hu. Gentle introduction to gradient descent with momentum, rmsprop, and adam, 2019. Accessed: 2023-10-17.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. URL https://www.cs.cmu.edu/~tom/mlbook.html.

David Morin. *Introduction to Classical Mechanics: With Problems and Solutions*. Cambridge University Press, 2008.

Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999. doi: 10.1017/S0962492900002919.

Jiří Podolský. Hamiltonův variační princip. In *Teoretická mechanika ve třech knihách*, chapter 4, pages 74–87. Karolinum, 2024. ISBN 978-80-246-5746-2.

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

SciML. Diffeqflux.jl: Neural ordinary differential equations example, 2023a. URL https://docs.sciml.ai/DiffEqFlux/stable/examples/neural_ode/. Accessed: 2023-07-17.

SciML. Sciml documentation, 2023b. URL https://docs.sciml.ai/Overview/stable/. Accessed: 2023-10-17.

Milan Straka. NPFL114: Introduction to Machine Learning (Summer 2023). https://ufal.mff.cuni.cz/courses/npfl114/2223-summer, 2024. Accessed: 2024-07-18.

Tim Strang, Isabella Caruso, and Sam Greydanus. Nature's cost function: Simulating physics by minimizing the action, 2023. URL https://arxiv.org/abs/2303.02115.

The Julia Language. Julia language documentation, 2023. URL https://docs.julialang.org/en/v1/. Accessed: 2023-07-17.

William F. Trench. *Elementary Differential Equations with Boundary Value Problems*. Trinity University, 2013. URL https://math.libretexts.org/Bookshelves/Differential_Equations/Elementary_Differential_Equations_with_Boundary_Value_Problems_(Trench)/12%3A_Fourier_Solutions_of_Partial_Differential_Equations/12.03%3A_Laplace's_Equation_in_Rectangular_Coordinates. PDF version.

Ch. Tsitouras. Runge–kutta pairs of order 5(4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, 62(2): 770–775, 2011. doi: 10.1016/j.camwa.2011.06.002. URL https://doi.org/10.1016/j.camwa.2011.06.002.

Wikipedia contributors. Binary classification. https://en.wikipedia.org/wiki/Binary_classification, 2024a. Accessed: 2024-07-18.

Wikipedia contributors. Universal approximation theorem. https://en.wikipedia.org/wiki/Universal_approximation_theorem, 2024b. Accessed: 2024-07-18.

Robert Černý and Milan Pokorný. Klasický variační počet. In *Základy matematické analýzy pro studenty fyziky 2*, chapter 13, pages 227–283. MatfyzPress, 2021. ISBN 978-80-7378-454-6.

List of Figures

List of Tables

List of Abbreviations

ODE - Ordinary Differential Equation
PDE - Partial Differential Equation
NN - Neural Network
NODE - Neural Ordinary Differential Equation
ML - Machine Learning
SciML - Scientific Machine Learning
Adam - Adaptive Momentum

A. Attachments

A.1 Code

The code is available in a Github Repository.
URL:`https://github.com/enaipi/ML-variations`

A.2 Problems results

Here, I formulate the methods for each problem and give tables with the results; these results should justify the claims made in Chapter 4, Section Experiment (4.2). I only chose such boundary conditions that the analytical solution is as simple as possible, but of course, the numerical methods do not need this simplification.

Laplace and Poisson equation

1D

Consider equation:

$$\begin{aligned} u''(x) &= -1 \quad x \in [0, 12] \\ u(0) &= 0, u(12) = -12. \end{aligned} \tag{A.1}$$

This is the easiest problem in our work, so let's be more thorough with this one and shorten the description for the next ones. The solution to this problem is a parabola given by (3.11) as:

$$u_0(x) = \frac{1}{2}x(10 - x). \tag{A.2}$$

Methods are formulated in the following way.

- **Finite difference method**

  $$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = 1, \quad u_0 = 0, u_N = 0, h = \frac{12}{N}.$$

  This system of linear equations and was solved with conjugate gradient method in Matlab.

- **Direct minimization of the functional**

  $$\text{Loss}(u_i) = \sum_{i=0}^{N} \left(\frac{1}{2} \left(\frac{u_{i+1} - u_i}{\Delta x} \right)^2 - u_i \right) \Delta x,$$

  $$\text{where } u_i = u(x_i),\ \Delta x = \frac{12}{N},\ \text{and } u_0 = u_N = 0.$$

  And we minimize with respect to u_i, with the initial inner points being 0.

- **Neural Network minimization**

$$\text{Loss}(\theta) = \sum_{i=0}^{N} \left(\frac{1}{2} \left(\frac{u_{i+1}(\theta) - u_i(\theta)}{\Delta x} \right)^2 - u_i(\theta) \right) \Delta x,$$

where $u_i(\theta)$ is a neural network output, $\Delta x = \dfrac{12}{N}$, and $u_0, u_N \in [C_1, C_2]$, taken differently at each iteration. And we minimize with respect to (θ). This way we train a neural network:

$$NN : \{u_1, u_N\} \to \{u_i\}_{i=1}^{N-1}.$$

- **Neural ODEs** We try to learn the system:

$$\begin{aligned} u' &= v \\ v' = u'' &= NN(u, v, \theta), \end{aligned} \tag{A.3}$$

where $NN(u, v, \theta)$ should be -1 by minimizing a loss function:

$$\text{Loss}(\theta) = \sum_{i=0}^{N} \left(\frac{1}{2} \left(\frac{u_{i+1}(\theta) - u_i(\theta)}{\Delta x} \right)^2 - u_i(\theta) \right) \Delta x,$$

where $u(x_i)(\theta)$ is given by a numerical solution of the (A.3), with chosen initial conditions $u(0) = 0, v(0) = 5$, and everything else is same as before.

Remark. For the neural ODE, the loss can be just as well:

$$\text{Loss}(\theta) = \sum_{i=0}^{N} \left(\frac{1}{2} {v_i}^2 - u_i(\theta) \right) \Delta x,$$

but we stick with the previous for all the problems, since we can already use the loss function from the other methods.

- **error** Since all methods give discrete solutions, the error will be measured in the Euclidean norm.

$$\|\mathbf{u} - \mathbf{u_0}\| = \left(\sum_{i=0}^{N} (u_i - u_{0,i})^2 \right)^{\frac{1}{2}}, \tag{A.4}$$

u_0 being a solution (A.2).

And for neural ODEs, we compute the error as

$$\frac{1}{K} \sum_{j=1}^{K} \|\mathbf{u}_j - \mathbf{u}_{0,j}\|, \tag{A.5}$$

whereby $\mathbf{u}_j, \mathbf{u}_{0,j}$ we mean a different solution to a different boundary condition, and the constant K can be arbitrary. These errors will be the same for all problems, so we only mention them here.

Table (A.1) shows the results for $N = 50$ and Adam optimizer. First, for Neural network minimization, we keep the boundary fixed to have an easier comparison.

Method	LR	Iter.	Act.	# params	Error
CG (FDM)	-	50	-	-	2.48e-13
Minimization	0.3	3000	-	-	7.7e-3
NN Minimization-fixed boundary	0.01	3000	tanh	1 713	6.6e-3
Neural ODEs	5e-3	1000	tanh	201	1.34

CG: Conjugate gradient method; LR: learning rate; Iter.: Iterations; Act.: Activation function; # params: Number of parameters and Error is $\|\mathbf{u} - \mathbf{u_0}\|$ (A.4)

Table A.1: Comparison of the errors for different methods of solving 1D Poisson equation.

2D

Consider equation:

$$\begin{gathered} \Delta u(x,y) = 0 \\ x, y \in \Omega, \quad \Omega = [0,1]^2 \\ u(x,0) = sin(\pi x), \quad u(x,1) = u(0,y) = u(1,y) = 0. \end{gathered} \tag{A.6}$$

The solution is by (3.14)

$$u_0(x,y) = \sin(\pi x)\frac{e^{\pi y} - e^{\pi(2-y)}}{1 - e^{2\pi}}.$$

For the methods, we have:

- **Finite difference method**

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = 0, \quad u_{i,j} = u(x_i, y_j) \text{ and}$$

$$u_{i,0} = \sin(x_i), \quad u_{i,1} = u_{0,j} = u_{1,j} = 0.$$

- **Direct minimization of a functional**

$$\text{Loss}(u_{i,j}) = \sum_{j=0}^{N}\sum_{i=0}^{N} \frac{1}{2}\left(\left(\frac{u_{i+1,j} - u_{i,j}}{\Delta x}\right)^2 + \left(\frac{u_{i,j+1} - u_{i,j}}{\Delta y}\right)^2\right)\Delta x \Delta y,$$

  where $\Delta x = \Delta y = \frac{1}{N}$, and the boundary being the same as above.

- **Neural Network minimization**

$$\text{Loss}(\theta) = \sum_{j=0}^{N}\sum_{i=0}^{N} \left(\left(\frac{u_{i+1,j}(\theta) - u_{i,j}(\theta)}{\Delta x}\right)^2 + \left(\frac{u_{i,j+1}(\theta) - u_{i,j}(\theta)}{\Delta y}\right)^2\right)\frac{\Delta x \Delta y}{2}$$

  except now $u_{i,j}$ are neural network outputs. We minimize this function and, at each interaction, change the boundary in a form:

$$u_{i,0} = A_k \sum_{k=1}^{T} \sin(k\pi \mathbf{x}), \quad u_{0,j} = B_k \sum_{k=1}^{T} \sin(k\pi \mathbf{y}), \tag{A.7}$$

Similarly, this can be done for the other 2 edges, but I only made it for one edge and 2 edges, see: (A.6). By this, we train a network:

$$NN : N_{\partial\Omega} \to N_{\Omega},$$

where $N_{\partial\Omega}, N_{\Omega}$ is the notation from here (4.3).

- **Neural ODEs**

  Since this is not an ODE, I skipped the neuralODE method, but it would be interesting to explore some neuralPDEs, where the numerical solution would be a result of an explicit finite difference scheme; this can be a part of future work.

Again, (A.2) compares the methods, this time for N = 20x20.

Method	LR	Iter.	Act.	# params	Error
CG (FDM)	-	50	-	-	6.9e-3
Minimization	0.05	500	-	-	7.2e-3
NN Minimization-fixed boundary	10e-3	1000	tanh	15 561	7.2e-3

CG: Conjugate gradient method; LR: learning rate; Iter.: Iterations; Act.: Activation function; # params: Number of parameters and Error is $\|\mathbf{u} - \mathbf{u_0}\|$ (A.4).

Table A.2: Comparison of the errors for different methods of solving 2D Laplace equation.

Harmonic oscilator

Consider equation:

$$\begin{aligned} x''(t) + x(t) = 0, \quad t \in [0, 3] \\ x(0) = 0, x(3) = \sin(3), \end{aligned} \tag{A.8}$$

to get $sin(t)$ as a solution. Our methods are then formulated here.

- **Finite difference method**

$$\frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2} + x_n = 0,$$

$$x_n = x(t_n), x_0 = 0, x(3) = \sin(3), \Delta t = \frac{3}{N}.$$

  This is no longer a semi-positively definite matrix, so I just used a default "\" solver in Matlab.

- **Direct minimization of the functional**

$$\text{Loss}(x_n) = \sum_{n=0}^{N} \left(\frac{1}{2} \left(\frac{x_{n+1} - x_n}{\Delta t} \right)^2 - \frac{1}{2} {x_n}^2 \right) \Delta t.$$

- **Neural Network minimization**

$$\text{Loss}(\theta) = \sum_{n=0}^{N} \left(\frac{1}{2} \left(\frac{x_{n+1}(\theta) - x_n(\theta)}{\Delta t} \right)^2 - \frac{1}{2} {x_n}^2(\theta) \right) \Delta t,$$

with $x_n(\theta)$ being a neural network output, $x_0, x_N \in [C_1, C_2]$ randomly chosen during optimization, training a neural network:

$$NN : \{x_0, x_N\} \to \{x_n\}_{n=1}^{N-1}.$$

- **Neural ODEs** We are learning the system:

$$\begin{aligned} x' &= z \\ z' = x'' &= NN(u, v, \theta), \end{aligned} \tag{A.9}$$

where $NN(u, v, \theta)$ should go to $-x$ by minimizing a loss function:

$$\text{Loss}(\theta) = \sum_{n=0}^{N} \left(\frac{1}{2} \left(\frac{x_{n+1}(\theta) - x_n(\theta)}{\Delta t} \right)^2 - \frac{1}{2} {x_n}^2(\theta) \right) \Delta t,$$

where $u(x_i)(\theta)$ is given by a numerical solution of the (A.9), with initial conditions $x_0 = 0, z_0 = 1$, chosen so it gives a solution that we want (A.8). Comparison of the methods is with N = 50 inner points is then in a table (A.3).

Method	LR	Iter.	Act.	# params	Error
Matlab "\" FDM	-	-	-	-	0.16
Minimization	0.01	1000	-	-	0.15
NN Minimization-fixed boundary	1e-3	3000	tanh	1 713	0.04
Neural ODEs	1e-3	500	tanh	81	1.51

FDM: Finite difference method; LR: learning rate; Iter.: Iterations; Act.: Activation function; # params: Number of parameters and Error is $\|\mathbf{u} - \mathbf{u_0}\|$ (A.4)

Table A.3: Comparison of the errors for different methods of solving harmonic oscilator.

Brachistochrone

Consider equation:

$$y(x)(1 + y'(x)^2) = 2, \quad x \in [0, \pi] \tag{A.10}$$

$$y(0) = 0, y(\pi) = 2 \tag{A.11}$$

with positive y being a downwards direction again. The solution is then by (3.3).

$$\begin{aligned} x &= (\phi - \sin\phi), \\ y &= (1 - \cos\phi). \end{aligned} \tag{A.12}$$

The methods are formulated here.

- **Direct minimization of the functional**

$$\text{Loss}(y_i) = \sum_{i=0}^{N} \frac{\sqrt{1 + (y_i')^2}}{\sqrt{2gy_i}} \Delta x,$$

$$y_0 = \epsilon, y_n = 2, \Delta x = \frac{\pi}{N}, y_i' = \frac{y_{i+1} - y_i}{\Delta x}.$$

Remark. $y_0 = \epsilon$ is chosen for all methods because of the square root. This is surely not the best workaround, but since we work with equidistant nodes, it is the most straightforward one.

- **Neural Network minimization**

$$\text{Loss}(\theta) = \sum_{i=0}^{N} \frac{\sqrt{1 + (y_i'(\theta))^2}}{\sqrt{2gy_i(\theta)}} \Delta x, \quad \Delta x = \frac{\pi}{N}.$$

This time, changing only the second boundary $y_N > 0 \in [C_1, C_2]$, we also set the output of the network to $-|x|$ to ensure its negative; without it, I didn't make it work.

- **Neural ODEs**

And finally, for brachistochrone, we use the Beltrami identity (9) and learn.

$$y' = NN(y), \tag{A.13}$$

where we want to get:

$$NN(y) = \sqrt{\frac{2}{y} - 1}$$

by minimizing:

$$\text{Loss}(\theta) = \sum_{i=0}^{N} \frac{\sqrt{1 + (y_i'(\theta))^2}}{\sqrt{2gy_i(\theta)}} \Delta x, \quad \Delta x = \frac{\pi}{N}.$$

where y_i is again a numerical solution of (A.13). I minimized this loss using a build-in function NeuralODE() from a SciML framework SciML [2023b]. Partly because of this, I got better results than for the other problems where I used my implementation, see (A.5).

Remark. Example for finite difference method can be

$$y_{i+1} = y_i + \Delta x \sqrt{\frac{2}{y_i} - 1},$$

where $\Delta x = \frac{\pi}{N}$ and $y_i = y(x_i), y_0 = \epsilon, y_N = 2$.

However, I didn't include it in the comparison since it has to be quite modified for it to work; but it nicely illustrates the problem with the first step, which will be really big due to the singularity. Showing that the equidistant nodes are not the best choice for brachistochrone.

For brachistochrone, I chose a denser grid N = 300, $\epsilon = 10^{-4}$ and computed the error from interpolated curve (A.12) with linear splines at gridpoints by (A.4). The comparison of the error for different methods is shown (A.4).

Method	LR	Iter.	Act.	# params	Error
Minimization	0.003	5000	-	-	1.09
NN Minimization-fixed boundary	5e-4	10 000	tanh	9 930	1.02
Neural ODEs	5e-3	4 000	tanh	1 153	1.03

LR: learning rate; Iter.: Iterations; Act.: Activation function; # params: Number of parameters and Error is $\|\mathbf{u} - \mathbf{u_0}\|$ (A.4)

Table A.4: Comparison of the error for different methods of solving brachistochrone problem.

Neural network minimization analysis

Lastly, let's analyze how the NN minimization behaves when choosing a boundary from different domains. First, I trained the network for the 1-dimensional problems and used the number of nodes $N = 50$ for Poisson eq. and oscillator with e neural network with $\tanh(x)$ with 2769 parameters. The error was then computed by (A.5) with $K = 10$ random examples by using (3.11), (3.3) general solutions.

For the brachistochrone problem, I used $N = 100$ same $\tanh(x)$ activation and 4419 parameters. Here I computed the error only for the 1 boundary $y_N = -2$ that I had the analytical solution (A.2) for. These findings are (for different training domains described (4.4)) shown in the table (A.5), as well as in the figures (4.5), (4.6).

Problem	boundary nodes $[C_1, C_2]$	Iter.	LR	Error
Poisson 1D (A.1)	$u_0, u_n \in [-10, 10]$	10 000	5e-4	2.24
	$u_0, u_n \in [-25, 25]$	40 000	5e-4	4.12
	$u_0, u_n \in [-100, 100]$	250 000	1e-4	4
Harmonic oscillator (A.8)	$x_0, x_n \in [0, 10]$	50 000	5e-4	5.4
	$x_0, x_n \in [0, 30]$	100 000	5e-4	4.5
	$x_0, x_n \in [-10, 10]$	200 000	5e-4	3.5
Brachistochrone (A.10)	$y_n \in [-4, eps]$	100 000	1e-4	0.91
	$y_n \in [-10, eps]$	200 000	1e-4	0.64

bdd: Boundary condition domain (4.4)*; Iter.: Iterations; LR: learning rate; and Error is* $\overline{\|\mathbf{u} - \mathbf{u_0}\|}$ (A.5)

Table A.5: Neural network minimization on different domains error comparison

For 2D Laplace equation (A.2) I chose a grid N = 20x20, model also with $tanh(x)$ activation function and 17 348. And trained 2 models: First, I trained it only on 1 side having a periodical boundary (A.7) up to $T = 5$, and then 2 sides of the square have a periodical boundary, but only up to $T = 2$. Both times I have successfully trained the network to map the boundary to the solution, but in the second case had 20x more interaction.

Laplace eq. in 2D (A.2)	boundary nodes domain	LR	Iter	Error
Model 1	$u_{i,0} = A_k \sum_{k=1}^{5} \sin(k\pi \mathbf{x})$	1e-4	5000	1.00
Model 2	$u_{i,0} = A_k \sum_{k=1}^{3} \sin(k\pi \mathbf{x})$ $u_{0,j} = B_k \sum_{k=1}^{3} \sin(k\pi \mathbf{x})$	1e-4	100 000	0.16

bdd: Boundary condition domain (A.7)*; Iter.: Iterations; LR: learning rate; and Error is* $\overline{\|\mathbf{u} - \mathbf{u_0}\|}$ (A.5) $A_k, B_k \in \mathcal{N}(0,1)$ *are random coefficients*

Table A.6: Neural network minimization error for Laplace Equation, trained for 2 different boundary domain models.