

Institute for Parallel and Distributed Systems

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Bachelor Thesis

# **A Framework for Distributed Training of Physics-Informed Neural Networks Using JAX**

Johannes Frederic Braun

**Course of Study:** Informatik

**Examiner:** Prof. Dr. rer. nat. Dirk Pflüger

**Supervisor:** Raphael Leiteritz

**Commenced:** April 5, 2021

**Completed:** Oktober 5, 2021



## Abstract

The intention of this thesis is to evaluate the high-performance machine learning framework JAX.

In the course of this work, a physics-informed neural network that solves the Burgers' equation is implemented. This problem is chosen, as it is a well known and researched numerical problem and thus allows for great comparability.

Here, a basic version of the physics-informed neural network with Flax is first created, which is an ecosystem for JAX that allows to implement neural networks. This version was then first improved with the tools offered via JAX. Afterwards, a SPMD version of this physics-informed neural network is also implemented, where multiple graphics processor units are utilized in the training. Additionally, the physics-informed neural network is extended to predict the parameters of the partial differential equation that describes the Burgers' equation. This was done by the physics-informed neural network, while still learning to estimate the Burgers' equation.

For the optimized basic physics-informed neural network and the physics-informed neural network that also estimates the parameters of the partial differential equation promising results with JAX were achieved.

The outcome of the SPMD physics-informed neural network was dissatisfactory, as it did not yield any improvements compared to the basic version. Although, this might stem from the small amount of data points used for each iteration and further points discussed in this paper.

Additionally, a caveat must be voiced, as it often becomes apparent that the documentations of JAX and Flax are a work in progress. Because of this, a lot of crucial features have to be found out by trial and error, while working with these frameworks.

Yet still, JAX and hence also Flax are considered a compelling framework to implement high-performance neural networks. Especially because of its potent Autograd and straightforward XLA just in time compilation. Through these components a performant physics-informed neural network can be quickly setup as shown in this thesis. Here, Autograd aids in creating the necessary gradients for the physical loss. Whereas the XLA just in time compilation yields drastic improvements to the run time of the training performed on the physics-informed neural network. These features then lead to previously mentioned promising results for the basic physics-informed neural network and the physics-informed neural network that also estimates the parameters of the partial differential equation.



# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
<b>2</b>	<b>Background</b>	<b>17</b>
2.1	Deep Learning and Neural Networks . . . . .	17
2.2	Automatic Differentiation . . . . .	19
2.3	Burgers' Equation . . . . .	22
2.4	JAX: Autograd and Accelerated Linear Algebra . . . . .	23
2.5	Flax: A Neural Network Ecosystem for JAX . . . . .	26
<b>3</b>	<b>Implementation</b>	<b>29</b>
3.1	Physics-Informed Neural Network with GPU parallelism . . . . .	33
3.2	Physics-Informed Neural Network with Parameter Learning . . . . .	34
3.3	Further changes . . . . .	35
<b>4</b>	<b>Results</b>	<b>37</b>
4.1	Setup . . . . .	37
4.2	Single GPU Physics-Informed Neural Network . . . . .	37
4.3	Multi GPU Physics-Informed Neural Network . . . . .	40
4.4	Partial Differential Equation learning Physics-Informed Neural Network	41
<b>5</b>	<b>Conclusion</b>	<b>47</b>
	<b>Bibliography</b>	<b>51</b>



# List of Figures

1.1	Outlines of the reference and regressed aneurysm projected on perpendicular surfaces for concentration, velocity magnitude and pressure . . . . .	15
2.1	Example feedforward network, with two hidden layers, a single output and four input nodes . . . . .	17
2.2	Example Computation of a node in the hidden layer . . . . .	18
2.3	Forward mode example . . . . .	20
2.4	Forward mode example with function $f(x) = \sin(x^2)$ . . . . .	20
2.5	Reverse mode example . . . . .	21
2.6	Example Burgers' equation, with the colour scale representing $u(x,t)$ . . .	23
4.1	MSE loss of the basic single GPU PINN, with the benchmark as red line . . .	38
4.2	MSE loss of the improved single GPU PINN, with the benchmark as red line . . .	39
4.3	Burgers' equation plotted with the output of the single GPU PINN . . . . .	40
4.4	Loss curve of the MSE for the PINN that learned the PDE parameters, with the benchmark as red line . . . . .	42
4.5	Loss curve of the first PDE variable, with the benchmark as red line . . . . .	43
4.6	Loss curve of the second PDE variable, with the benchmark as red line . . . . .	44
4.7	Burgers' equation generated via the output of the PINN, which also learned the PDE parameters . . . . .	45





# List of Listings

2.1	vmap example . . . . .	25
2.2	pmap example . . . . .	25
2.3	linnen neural network example . . . . .	26
2.4	Flax optimizer example . . . . .	27
3.1	PINN imports and NN . . . . .	30
3.2	PINN MSE . . . . .	30
3.3	PINN u- and f-function . . . . .	31
3.4	PINN helper functions . . . . .	32
3.5	PINN main . . . . .	33
3.6	PINN GPU MSE Version 1 . . . . .	34
3.7	PINN PDE NN . . . . .	35
3.8	PINN PDE f-function . . . . .	36
3.9	PINN MSE with data shuffling . . . . .	36



# Acronyms

- API** application programming interface. 23
- CPU** central processing unit. 24
- GPU** graphics processing unit. 14
- IPVS** Institute for Parallel and Distributed Systems. 37
- JIT** just in time. 14
- MSE** mean squared error. 18
- NN** neural networks. 13
- PDE** partial differential equation. 13
- PINN** physics-informed neuronal network. 13
- SPMD** single-program multiple-data. 14
- TPU** tensor processing unit. 23
- XLA** accelerated linear algebra. 14



# 1 Introduction

Machine learning is a field of computer science that implements algorithms, which optimize themselves through data or experience. Today these algorithms are employed in a variety of applications such as biomedicine, image- and speech recognition, to name some prolific fields.

A subcategory of machine learning are the neural networks (NN). These NNs were first created by McCulloch and Pitts in 1943, inspired by the neurons of the human brain [RN20]. These artificial neurons work by weighting and summing inputs, adding a bias and using an activation function. A NN then is an ordered collection of these neurons, which calculates a certain result based on internal figures of merit. These figures of merit are realized by weighting correctness of the output and the nodes participation in this result. Even though it is yet to be fully understood why NNs achieve such good results [RN20], their success in a plethora of complex modern applications speaks for itself. These achievements in fields such as biomedicine, image recognition and many more were in part aided by rapid advancements in computational power and memory availability.

Yet it remains a problem that NNs need extensive amounts of data to be trained successfully. Therefore, the quality and quantity of the available data is often a critical factor for the accuracy of a NNs output. Adequate training data is often hard to find or a labour-intensive process, when being created. It also can rarely be automatised. Hence data quality and quantity are often the limiting factor in machine learning, as it is needed to create sufficiently expressive NNs.

A promising approach to overcome these limiting factors is given by physics-informed neuronal network (PINN), which are a class of surrogates to approximate physical processes leveraging available data and domain knowledge [RYK20]. By employing penalty terms, they can restrict the solution space to a region, where physically sound predictions can be made. This enables them to capture dynamics of complex differential equation systems, while also having lower prerequisites in terms of the required training data. However, if data is available in abundance, they can learn and reconstruct the underlying partial differential equation (PDE) as well. These promising results of PINNs [RPK18] are also aided by leveraging recent developments in automatic differentiation, which allows for easy differentiation of NNs with respect to their inputs and construct PINNs, by exploiting structured prior information [RPK18].

An example of a practical application in the field of bio-medics is the collection of data for a flow in an 3D intracranial aneurysm and the reconstruction of patient specific parameters from it, such as velocity and pressure fields. This could then be used to determine the

wall shear stress of said aneurysm, which can be crucial in prognosis of vascular diseases. Such methods allow for reliable estimations, without further costs and thus simplify the state-of-the-art approach [RYK20].

For creating the PINN, JAX [BFH+18] and Flax [HLO+20] will be used. JAX is a framework that provides automatic differentiation and accelerated linear algebra (XLA) [BFH+18] for machine learning research. Whereas Flax is a high-performance neural network library and ecosystem for JAX. JAX and Flax are developed in close collaboration and are therefore made to be used in conjunction.

Both frameworks and an exemplary problem, in the form of the Burgers' equation, will then be used to first implement a PINN test case by setting up a feed-forward neural network.

Here, a standard optimization approach for NNs is used, a physics-informed loss functions is defined and utility functions are built for representing the problem geometry. Thereafter, the PINN is going to be expanded to allow for learning of PDE parameters defined in the loss function of the same problem. Throughout this process, the training is optimized by using features given by JAX, such as auto-vectorization, just in time (JIT) compilation and its advanced automatic differentiation features for higher-order derivatives. Afterwards, the single-program multiple-data (SPMD), provided by JAX, are used to evaluate, how well the training scales on multiple graphics processing unit (GPU) on the same machine.

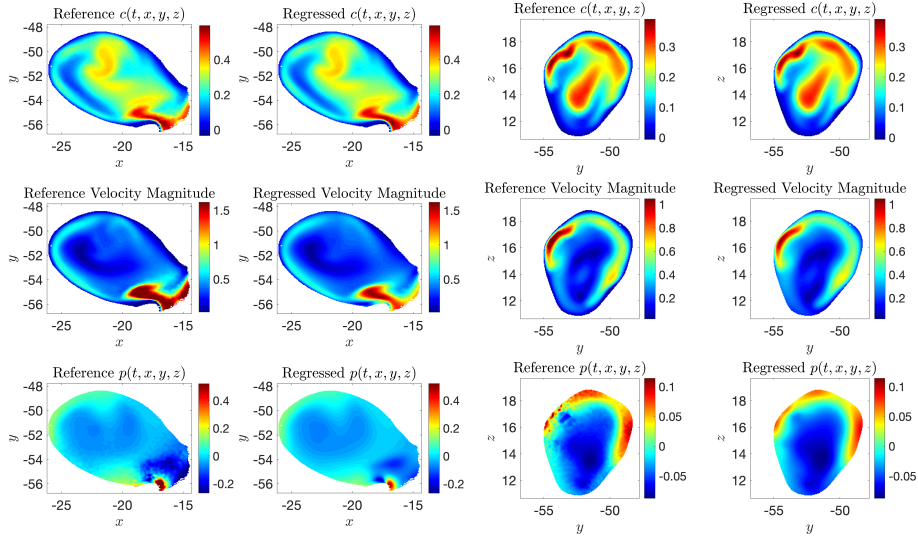
This paper will be started by first looking at the relevant background, which will begin with a quick overview of NNs and automatic differentiation. Afterwards the Burgers' equation will be presented in section 2.3, which will be the toy problem used in this paper for the PINN in this paper. The following section will be about JAX and all the core features that will be used. The final part of the Background chapter will be about the Flax framework, which is used in conjunction with JAX to build the PINN. The subsequent chapter will introduce and illustrate the PINNs in detail, which were created with JAX and Flax. Based on these implementations, the results achieved will be looked at, when running them on a computational cluster and finally evaluate and compare them to benchmarks found in the literature.

---

## Related Work

There are several publications that prove the concept of PINNs with a plethora of theoretical applications, such as the work presented in [RPK18], or real world utilization, as can be found in the publication [RYK20]. Therefore, the ambition is to expand upon these foundations, by implementing them with JAX and Flax.

As the PINNs, later introduced in this work, already get compared to those found in the appendix of the paper [RYK20] in chapter 4, it will not be discussed here. Hence, the work will be put in context, by presenting a practical application of a PINN that also solves a fluid mechanic problem found in paper [RYK20]. This also serves to indicate the potential of PINNs, when applying PINNs to real-life problems.



**Figure 1.1:** Outlines of the reference and regressed aneurysm projected on perpendicular surfaces for concentration, velocity magnitude and pressure

A problem the publication [RYK20] approached is a patient-specific intracranial aneurysm that is located at the level of the eye and beneath the brain. Here, the paper generated the data, by using patient-specific parameters of the intracranial aneurysm, the patterns of the blood flow at the entry and a homogeneous concentration. This data is then used to train a PINN, which can be used to predict the wall shear stress of the aneurysm sac. The authors then conclude, by introducing the possibility of the results to support medical professional in making an assessment.

Yet, this approach is not limited to biomedicine, but may be applied to any field that is dependent on fluid dynamics. Therefore, this framework can be expanded to the areas electromagnetics, aerodynamics, nonreactive flow mixing and more. This is also true for the problem presented in this work, as it in like manner solves fluid dynamics, it can be applied to similar fields. Even though the problem geometry is significantly less complex, when compared to the publication [RYK20].



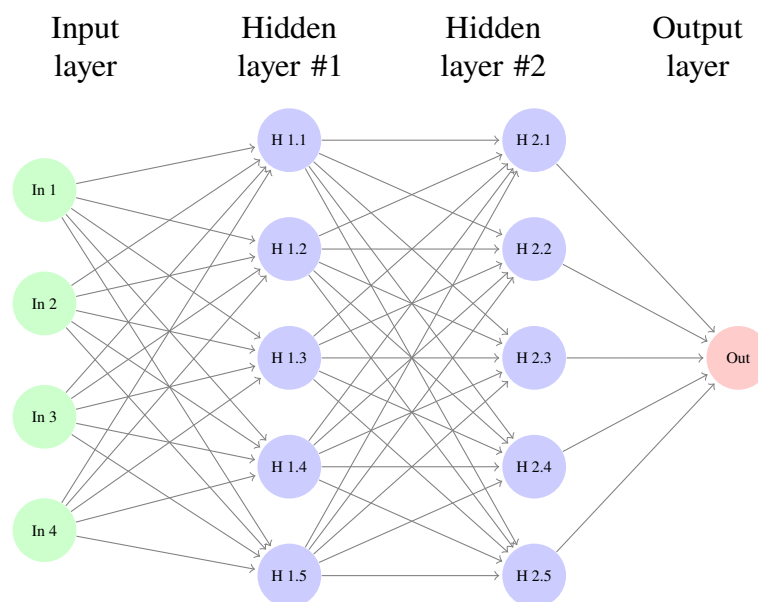


## 2 Background

### 2.1 Deep Learning and Neural Networks

Even though, deep learning and NNs are by no means new concepts, they are as prevalent and important as ever. In this context the term "deep" refers to the amount of employed layers and the therein resulting computational depth from input to output. The core concepts of NNs are complex algebraic circuits with adjustable coupling strength and many layers, which provide a cutting-edge approach to solve many modern problems, such as visual recognition, image synthesis and speech recognition [RN20]. Although, it is not yet fully understood why NNs achieve such promising results, their success in the previously mentioned fields is self-evident.

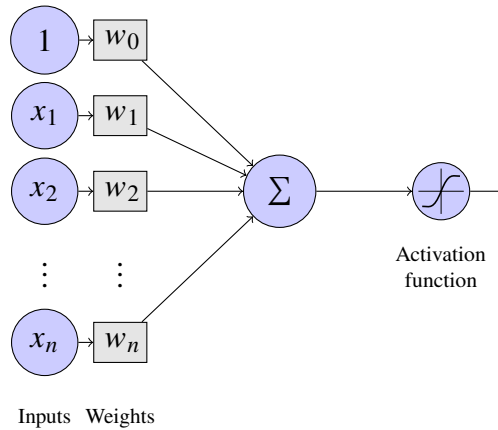
As implied by the name, a feedforward network is only connected in one direction, which makes it a directed acyclic graph with explicit input and output points, as can be seen in Fig. 2.1.



**Figure 2.1:** Example feedforward network, with two hidden layers, a single output- and four input nodes

The network outlined in Fig. 2.1 network is still rather shallow and short, yet it suffices to demonstrate the basic idea. It can be thought of as a dataflow- or computational graph, where each node corresponds to a fundamental calculation. A deeper insight into such a calculation is given in Fig. 2.2. Each of the presented calculations consists of inputs  $x_1$  to  $x_n$  from their preceding nodes and an input  $x_0$ , which is set to 1. This constant input  $x_0$  is called the bias. The weights  $w_0$  to  $w_n$  and the associated inputs are then multiplied and the results are summed up. These weights are then later fine-tuned to attain an output from the NN, which better approximates the desired result.

The next component of the computation found in Fig. 2.2 is the activation function. This function is used to map the result of the summation to a distinct value. Here, the activation function is consciously chosen by the developer, to fit the underlying geometry of the problem, the NN is designed to solve. However, it is crucial that the chosen activation function is nonlinear, as any composition of linear functions can only represent a linear function. This nonlinear component in the fundamental computation allows a sufficiently expressive NN to approximate any given function. This is proven by the *universal approximation theorem* [RN20], which states that a network of just two layers, with the first nonlinear and the second linear, can represent any continuous function to an arbitrary rate of certainty. In the Fig. 2.2 the hyperbolic tangent function is chosen.



**Figure 2.2:** Example Computation of a node in the hidden layer

The complete computation of a node, with  $y$  being the result, is then:

$$(2.1) \quad y = \tanh\left(\sum_{i=0}^n x_i * w_i\right) \quad \text{with } x_0 = 1$$

To quantify the accuracy a NN can achieve in representing the desired results, a loss function is created, fitting the problem. The objective of the loss function is to compare the output of a NN with the expected data and assign a value to describe the degree of deviation. In a numerical setting, this could be a simple mean squared error (MSE), yet it may be harder to quantify the achieved accuracy in image recognition. The loss function is a differentiating factor between PINNS and NNs. NNs compare their outputs only to

training data and calculate their loss accordingly. PINNs additionally implement a physical function, which is employed to restrict the solution space to a region of physically sound predictions. Therefore, less data is needed to obtain a certain degree of accuracy for a PINN.

Another important aspect in the development of NNs is the training. Hereby, the weights of the NN are optimized to better fit the desired result, by employing the previously introduced loss function. Current NNs and PINNs use optimization functions based on stochastic gradient descent. In this approach the gradient of the loss function is calculated with respect to the current parameters of the NN. This is then used to determine the gradient direction, in which the loss is minimized. Afterwards, the gradient direction is multiplied with the learning rate, which determines the degree of change that is now applied to the parameters, which define the calculations of the NN. Hence, the purpose of the training is to minimize the loss achieved with the parameters of the NN.

## 2.2 Automatic Differentiation

Automatic- or computational differentiation is a third alternative to numerical- or symbolic differentiation. It is not prone to rounding and approximation errors, like numerical solutions, or hard to compute, such as symbolic differentiation [Hof15]. Automatic differentiation is particularly convenient and fitting for functions realised in computer code.

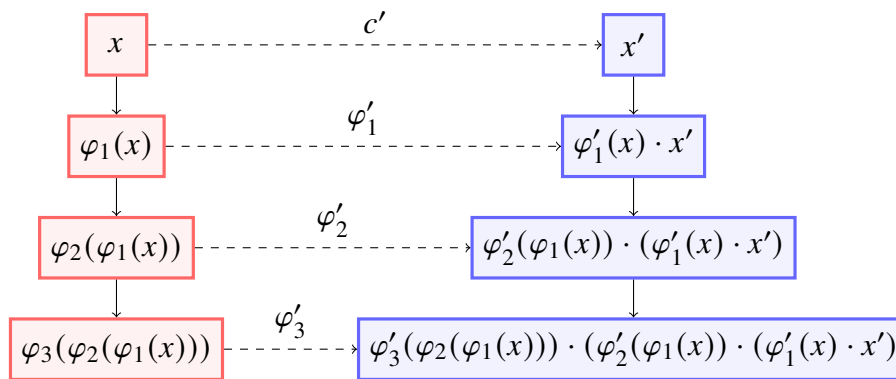
The fundamental concept of automatic differentiation is that elementary derivative laws from calculus, such as the chain rule, can be put into effect in a numerical setting [De10]. Through this auto. diff. can calculate derivatives with machine precision [Hof15]. These fast and accurate results then substantiate the current interest in the subject, as they provide an easy solution to a common problem. Yet automatic differentiation also has downsides, as the resulting derivative function can get quite long and therefore cannot be read by a human as easily.

The two main modes of automatic differentiation will be discussed, which are the forward- and the reverse mode. Here, only the conceptual idea of these modes will be explained and no implementation will be presented. For further information [Hof15] and [BPR15] can be consulted, which give a broad overview of different implementations.

### 2.2.1 Forward Mode

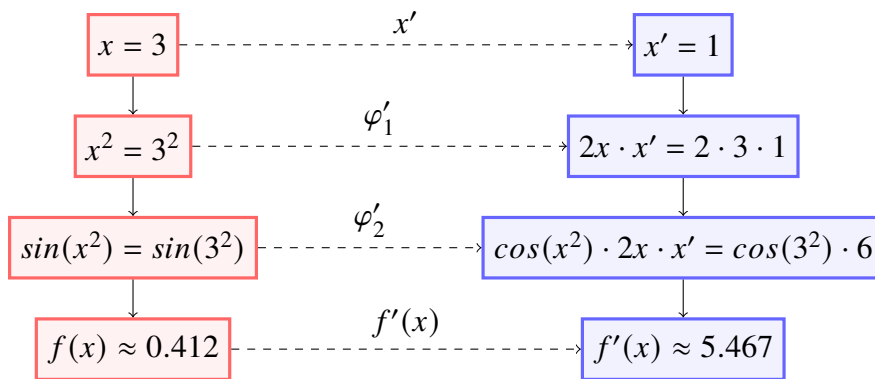
In the forward mode the function is computed from the inside out. For this, the independent variable of the function is first defined. The concept behind the forward mode will be shown first in theory in Fig. 2.3 and then with a real function and a fixed independent variable.

Let  $x, x' \in \mathbb{R}$  be real numbers with  $x$  being the fixed variable of the function and  $x' = \frac{df(x)}{dx}$  the derivative of  $x$ . Let further  $f, \varphi_1, \varphi_2, \varphi_3 : \mathbb{R} \rightarrow \mathbb{R}$  be differentiable functions with  $f = \varphi_1 \circ \varphi_2 \circ \varphi_3$ . This function  $f$  is then used in Fig. 2.3, where the left side is just the function  $f$  as defined previously. The right sides are the derivatives of the left side, where  $x'$  is the gradient of  $x$  and every subsequent derivative is computed with the chain rule from the gradient of the left function and the prior derivative. Thus, it ends up with the derivative  $f' = \varphi_3'(\varphi_2(\varphi_1(x))) \cdot (\varphi_2'(\varphi_1(x)) \cdot (\varphi_1'(x) \cdot x'))$  of the function  $f$  just by consecutively applying basic derivative laws, which in this case was the chain rule. Finally it can be said that the forward mode is best used for functions  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  with  $n > m$ , meaning that the operation has fewer inputs than outputs.



**Figure 2.3:** Forward mode example

As the concept of forward automatic differentiation has been introduced, it will be applied to a function and real numbers. For this the function  $f(x) = \sin(x^2)$  will be used and the fixed variable  $x = 3$  with its derivative  $x' = 1$ . As with the prior Fig. 2.3 the left side of Fig. 2.4 is the function and the right side the derivatives.

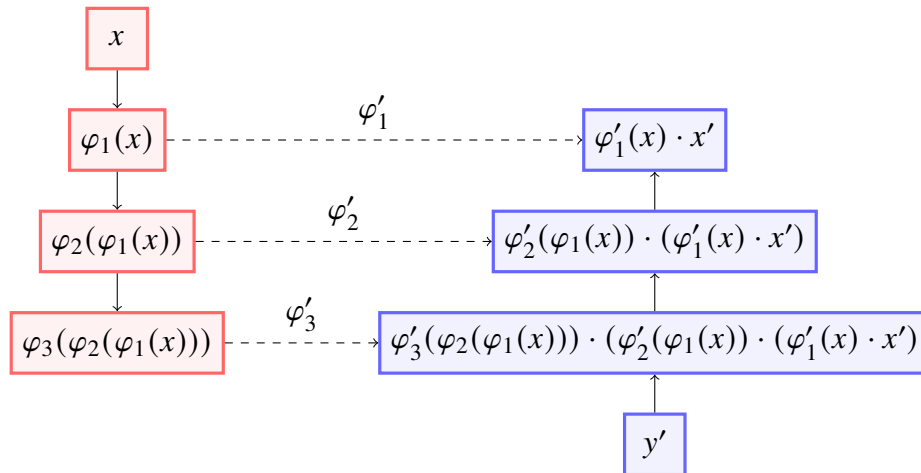


**Figure 2.4:** Forward mode example with function  $f(x) = \sin(x^2)$

### 2.2.2 Reverse Mode

In the reverse mode the function is computed from the outside to the inside. This difference to the forward mode can easily be seen, by comparing Fig. 2.3 and Fig. 2.5. For this, the variable to be differentiated is set to a fixed value and every part-expression is calculated subsequently. Due to the sequential computation of this every step of the calculation must be stored, which can lead to a higher memory consumption, compared to the forward mode. There are solutions to some degree to this problem, which this work will not go into detail. Further reference is discussed in [Hof15].

Let again  $x, y' \in \mathbb{R}$  be real numbers with  $x, y'$  being the fixed variable of the function. Let further  $f, \varphi_1, \varphi_2, \varphi_3 : \mathbb{R} \rightarrow \mathbb{R}$  be differentiable functions with  $f = \varphi_1 \circ \varphi_2 \circ \varphi_3$ . The function  $f$  is then used in Fig. 2.5, where the left side is just the function  $f$  as defined previously. The right sides here are also the derivatives of the left side, yet here the derivatives are created by replacing the outer function with through the chain rule.



**Figure 2.5:** Reverse mode example

It is obvious that reverse and forward mode are conceptually similar, yet their computational paths are exactly the inverse of each other. This variation in computation also explains the difference in performance for forward- and backward mode already mentioned. The reasoning for this is that a function with one output and  $m$  inputs would only need one sweep down with reverse mode, yet  $m$  with the forward mode. This is also true the other way around. So, for a function with one input and  $m$  outputs would need one sweep with the forward and  $n$  sweeps with the backward mode. Hence, reverse mode is superior for functions  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  with  $n < m$ .

## 2.3 Burgers' Equation

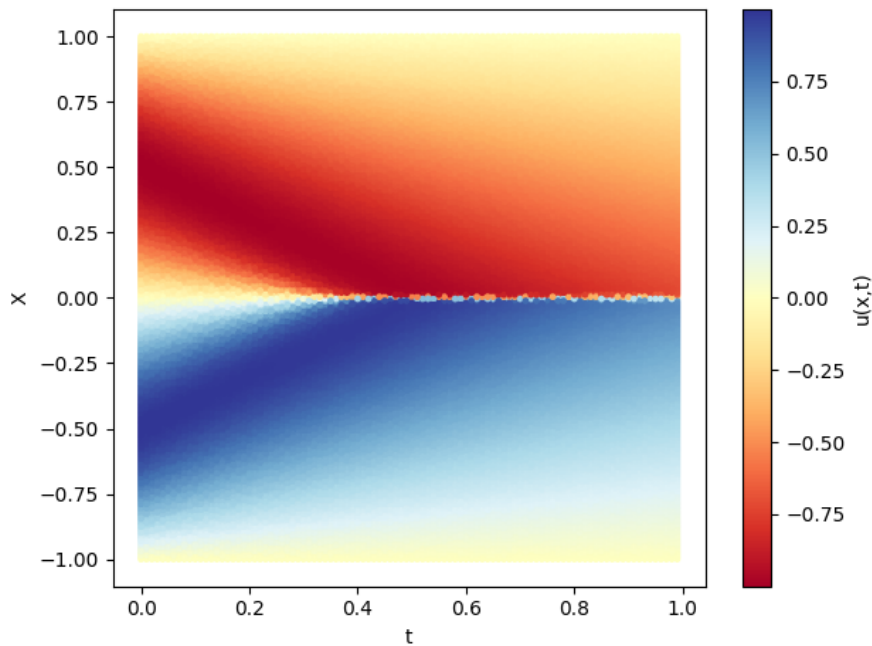
The Burgers'- or Bateman-Burgers' equation is a PDE first proposed by Harry Bateman and then later investigated by Johannes Martinus Burgers [BP72]. It is used in various fields of applied mathematics and fluid dynamics [BP72]. The general- or viscous Burgers' equation describes a field  $u(x, t)$  with a given diffusion coefficient  $\nu$ . Here,  $x$  describes the spacial-,  $t$  the temporal coordinate and  $u(x, t)$  the speed of a fluid at a spatial- and temporal coordinate.

$$(2.2) \quad \frac{\delta u}{\delta t} + u \frac{\delta u}{\delta x} = \nu \frac{\delta^2 u}{\delta x^2}$$

When the diffusion is not taken into account (i.e.  $\nu = 0$ ) it is called the inviscid Burgers' equation.

$$(2.3) \quad \frac{\delta u}{\delta t} + u \frac{\delta u}{\delta x} = 0$$

This work will use a one space dimensional version of the inviscid Burgers' equation, alongside with Dirichlet boundary conditions [SSE20], as an exemplary toy problem to be learned by the PINN. Hence, the diffusion coefficient  $\nu$  in the loss function of the PINN in this thesis will be disregarded. The Burgers' equation is chosen, as it is a widely known, researched and often used as a test case for numerical problems. This physical phenomenon poses a notoriously hard function to approximate numerically, because of its steep gradient. The data set for the training process is chosen from the paper [RPK18]. Here a Latin Hypercube Sampling [OS02] strategy was utilized to generate the dataset of the Burgers' equation. An example of this particular Burgers' equation with 25600 datapoints can be seen in Fig. 2.6. In this figure the variables  $x$  and  $t$  of the Burgers' equation are mapped on the axes  $y$  and  $x$  respectively. The coloured axis on the right side of Fig. 2.6 represents the result of the Burgers' equation  $u(x, t)$ .



**Figure 2.6:** Example Burgers' equation, with the colour scale representing  $u(x, t)$

## 2.4 JAX: Autograd and Accelerated Linear Algebra

JAX is a new framework for high-performance machine learning development that combines Autograd [MDA15] and XLA [FJL18].

The implementation of Autograd in JAX allows to automatically differentiate native Python and NumPy functions. It can differentiate through most, otherwise complex structures, and take derivatives of its own derivatives. The Autograd engine in JAX also supports reverse-mode differentiation, as well as forward-mode differentiation and the two can be composed arbitrarily to any order.

Another novelty in JAX is that it uses XLA to compile and run the NumPy programs on GPU and tensor processing unit (TPU). For this JAX uses the one-function application programming interface (API) accelerators *jit*, *pmap* and *vmap*. The first accelerator *jit* allows for deliberate JIT compilation, which is then compiled into XLA-optimized kernels, without employing another library or substituting Python with a different programming language. Similarly, *pmap* allows to use SPMD parallel programming and program multiple GPUs or TPUs with little effort. The last accelerator is *vmap*, which automatically vectorizes a function along an axis of an input and thus speeds up and simplifies computation over vectors.

All these methods can be used arbitrarily in conjunction to easily compose sophisticated functions. These methods then also show good performance, as their blocks are compiled to XLA automatically [BFH+18]. Hence, JAX allows to solve complex numerical problems, without having to worry about performance or extensive prior knowledge.

Yet, a caveat with JAX is that it is relatively new and therefore still has some bugs. Although, this is less likely, when using JAX in known cases, where bugs should have been found already.

### 2.4.1 Autograd in JAX

A fundamental principle of Machine Learning is defining a model with a sophisticated scalar function and optimizing from this model [**autogradEffortlessGradientsInNumpys**]. The gradient of these functions is often crucial, yet writing them can be an annoyance that slows down the productivity [FJL18]. Therefore, JAX provides an updated version of Autograd that can automatically differentiate native Python and NumPy functions. It can take derivatives through loops, branches, recursion and derivatives of derivatives, which usually poses a problem to such functions. Reverse-mode differentiation is provided via *grad*, along with forward-mode differentiation, which can be used together in any indiscriminate succession [BFH+18].

### 2.4.2 XLA just in time compilation

Just in time- or JIT compilation is performed to achieve the advantages of static compilation and interpretation.

For this, JAX additionally uses XLA to optimize the performance of the compiled code. XLA is a compiler from TensorFlow that enables code optimization, without alteration to the source code.

Statically compiled code is faster, especially if the compiled code is then able to run directly on the target hardware. The time it takes for compilation can vary, as the optimization and analysis can be time consuming. This is the crucial pitfall that JIT compilation can face. As it is compiled at run time, it should never cause pauses to the normal flow of the program, meaning the complete program would come to a halt and let the compilation catch up. Hence, JIT compilation can lead to a time loss, when it causes unmanageable pauses to the computation. Yet, it can also benefit from information, which is exclusively available at run-time, such as the control flow of the program or the input parameters. Through this it could achieve improvements, which would not be feasible at compile-time [Ayc03].

The code written in Python with JAX runs on the GPU/ TPU or the central processing unit (CPU), if neither is available. This computation can be sped up, if there is a sequence of operations, by using the *jit* function or the *jit* decorator in JAX. This then automatically end-to-end compiles the wrapped or decorated function through XLA [BFH+18].



### 2.4.3 Auto-Vectorization

*vmap* is the function in JAX to automatically map vectors along an axis. This provides a method to easily vectorize computation. *vmap* pushes down the loop to the function primitives to improve performance and handles batch dimensions for the user. For simple use-cases all of this is no problem, but may become a nuisance for larger problems.

---

**Listing 2.1** vmap example

---

```
1 from jax import vmap
2 predictions = jax.vmap(squared_error_f)(x_batched_f, t_batched_f)
3
```

---

The listing 2.1 shows an actual use of the *vmap* function in the implemented PINN. Here, the function *squared\_error\_f* is vector mapped with the data from *x\_batched\_f* and *t\_batched\_f* over the common axis of the batches. In this case it would be no problem to do it all manually, yet it would result in more code and is thus more prone to errors.

As with all other JAX functions, *vmap* can also be used arbitrarily with other JAX functions.

### 2.4.4 Single-Program Multiple-Data

*pmap* is used to create SPMD programs that are run on multiple GPUs/TPUs in parallel. Through this the code also gets XLA compiled, which gives similar results to *jit*. It is also important that functions with *pmap* inside them should not be additionally optimized with the *jit* function, as this would lead to sub optimal data management.

The example for *pmap* shown in listing 2.2 shows its very basic usage. It takes an array and returns an array of the same size. Herein, *pmap* creates multiple parallel processes with the data input. These processes are all the same and defined in the lambda function, which in this case just squares the input and returns it.

---

**Listing 2.2** pmap example

---

```
1 import jax.numpy as jnp
2
3 out = pmap(lambda x: x ** 2)(jnp.arange(8))
4 print(out)
5
6 [0, 1, 4, 9, 16, 25, 36, 49]
7
```

---

## 2.5 Flax: A Neural Network Ecosystem for JAX

The aim of Flax is to provide a tool to the JAX ecosystem, which allows to delve into the use-cases, where JAX is at its best. For this it provides a high-performance neural network library that is flexible by design. It provides a plethora of useful features to quickly create NNs in conjunction with JAX. Yet, for the sake of brevity this work will focus on *flax.linen* and *flax.optim*, which are also the most important to get started [HLO+20].

### 2.5.1 Linnen: Neural Networks in Flax

*linen* is consciously built similar to PyTorch [PGM+19] and TensorFlow [ABC+16] to allow for fast and easy understanding, while not trying to replicate them. It instead tries to be more maintainable, scalable and performant, by using the composable function-transformation approach seen in JAX [HLO+20].

---

**Listing 2.3** linen neural network example

---

```
1  from typing import Sequence
2
3  import jax
4  import jax.numpy as jnp
5  import flax.linen as nn
6
7  class MLP(nn.Module):
8      features: Sequence[int]
9
10     def __call__(self, x):
11         for feat in self.features[:-1]:
12             x = nn.relu(nn.Dense(feat)(x))
13         x = nn.Dense(self.features[-1])(x)
14         return x
15
16     model = MLP([12, 8, 4])
17     batch = jnp.ones((32, 10))
18     variables = model.init(jax.random.PRNGKey(0), batch)
19     output = model.apply(variables, batch)
20
```

---

In the example code listing 2.3 a NN example implemented in *linen* can be seen. The model is created, initialized and run one time in the lines 17, 19 and 20. From line 3 to 15 the entire NN is defined and it automatically knows what to do with the parameters given in line 17, as it is defined in line 8. The computation is done in line 11 and following, where a dense layout is chosen with rectified linear, also called *relu*, as activation function. With the last layer being the exception, since there is no activation function given. The input does not have to be specified, but is given through the initialisation in line 19. This example serves to show how flexible and maintainable the created NN is.

## 2.5.2 Optim: optimizers in Flax

Flax also provides an easy-to-use optimizer framework, which creates an optimizer object that then can be used in the training.

---

### Listing 2.4 Flax optimizer example

---

```
1  from flax import optim
2  optimizer_def = optim.Adam(learning_rate=0.1)
3  optimizer = optimizer_def.create(model)
4
```

---

This example in listing 2.4 shows an optimizer first being defined in line 2 and then created with a NN called "model". It is another example of the flexibility found in Flax.



## 3 Implementation

In this chapter, the implemented code is shown and explained. In the course of this work, several versions have been created to showcase the various ways to solve the Burgers' equation with JAX and Flax. Hence, the simplest way the PINN is implemented is presented first, where the complete code is shown and explained. Afterwards, the variations of said PINN are discussed, by outlining the altered parts of code.

The first part of the code in listing 3.1 starts with the imports. JAX, Sequence, JAX numpy called *jnp*, *linspace* and the optimizer are imported. JAX, Flax and their modules used in the code have already been explained in the chapter 2, therefore they will not be discussed again in detail.

The next Part of listing 3.1, going from line 8 to 20, contains the complete definition of the NN. This NN is setup with the variable *features*, which is an array containing the size of the computational layers and the output layer. With this array, a dense feedforward NN is created in line 12 and saved in the class. Afterwards, the *call* function can be used to run the NN with an input and its parameters, which gives the output of the NN. For this the *layers* of the NN defined by the *features* variable to iterate through the NN, apply the parameters and use a tangent hyperbolicus activation function.

Next comes the loss function in listing 3.2, which has an outer function that gets data points of the Burgers' equation as input. This data is split up into five batches of  $x$ ,  $t$ ,  $u$ , for the data loss, and  $x$ ,  $t$  for the physical loss. The data is chosen at random from the 25600 data points given by [RPK18], with the  $x$  and  $t$  data points for the data- and physical loss function being different. This wrapper function then calls the actual calculation, which takes the parameters of the NN as input and is also compiled with *jit* to increase the performance. Here, the complete loss is returned, which consists of the data loss calculated in the *mse\_u* and the physical loss calculated in the *mse\_f-function*. These two functions then call the functions *squared\_error\_u* and *squared\_error\_f* respectively to compute the following loss functions, which are both the mean squared error:

$$(3.1) \quad MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2 \quad MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2$$

These loss values then get summed up in line 14 and returned in line 16 as previously mentioned, which results in the formula for the combined mean square loss:

$$(3.2) \quad MSE = MSE_u + MSE_f$$

### 3 Implementation

---

#### Listing 3.1 PINN imports and NN

---

```
1 import jax
2 from jax import random, numpy as jnp
3 from flax import linen as nnhttps://www.overleaf.com/project/60a0ff09f494b81e65df96ae
4 from flax import optim
5 from typing import Sequence
6
7
8 class DeepFeedForwardNN(nn.Module):
9     features: Sequence[int]
10
11     def setup(self):
12         self.layers = [nn.Dense(feats) for feats in self.features]
13
14     def __call__(self, inputs):
15         x = inputs
16         for l_num, lyr in enumerate(self.layers):
17             x = jnp.tanh(lyr(x))
18             if l_num != len(self.layers) - 1:
19                 x = jnp.tanh(x)
20         return x
21
```

#### Listing 3.2 PINN MSE

---

```
1 def make_mse_func(x_batched, t_batched, u_batched, x_batched_f, t_batched_f):
2     def mse(param):
3         def squared_error_u(x, t, u):
4             pred = u_function(param, x, t)
5             return jnp.square(pred[0] - u[0])
6
7         def squared_error_f(x, t):
8             f = f_function(param, x, t)
9             return jnp.square(f)
10
11         mse_u = ((jnp.sum(jax.vmap(squared_error_u)
12                             (x_batched, t_batched, u_batched), axis=0)) / N_u)
13         mse_f = ((jnp.sum(jax.vmap(squared_error_f)(x_batched_f, t_batched_f), axis=0)) / N_f)
14         return mse_f + mse_u
15
16     return jax.jit(mse)
17
```

This nesting of functions seems unnecessary, yet it is crucial for calling them the way JAX and Flax intend them to. Therefore, these wrappers around functions are a reappearing pattern in JAX and Flax, which first require a function with constant data that is then later called again with the dynamic input.

---

**Listing 3.3** PINN u- and f-function

---

```
1 # u(t,x)
2 @jax.jit
3 def u_function(param, x, t):
4     u = model.apply(param, jnp.concatenate((x, t)))
5     return u
6
7
8 # f(t,x)
9 @jax.jit
10 def f_function(param, x, t):
11     u = u_function(param, x, t)
12     u_t = jax.jacfwd(grad_t_ufunction(param, x))(t)
13     u_x = jax.jacfwd(grad_x_ufunction(param, t))(x)
14     u_xx = jax.jacfwd(grad_grad_x_ufunction(param, t))(x)
15     f = u_t[0] + u[0] * u_x[0] - (0.01 / jnp.pi) * u_xx[0]
16     return f
17
```

The next part presented in Fig. 3.3 of the created PINN begins with the *u\_function*, which takes the PINN parameters, a *x*- and a *t* data point. These inputs are then used to calculate and return the PINN output, which is also JIT compiled through a *jit* decorator. After this follows the *f\_function* from lines 9 to 16, which takes the PINN parameters, a *x*- and a *t* data point as inputs just like the *u\_function*. This computes  $f(t_f^i, x_f^i)$  of the physical loss for the following function, which was taken from [RPK18]:

$$(3.3) \quad f := u_t + u \cdot u_x - (0.01/\pi) \cdot u_{xx}$$

Here, the *u* is just the output of the PINN and is calculated through the *u\_function*. Afterwards, the gradient of the *u\_function*  $u_x$  and  $u_t$  are calculated, by using the JAX gradient, which was explained in section 3.4. The forward mode called *jax.jacfwd* is chosen as the gradient method, as the resulting matrices are tall. These gradients again need wrappers, for the same reason as already mentioned at the *MSE* function in listing 3.2. The last component of the *f\_function* is  $u_{xx}$ , which again is a gradient, but this time of  $u_x$ . This second-degree gradient can be calculated the same way as the first degree derivative, which can be seen when comparing lines 1 to 6 and 15 to 20 in listing 3.4.

The last part of the PINN outlined in listing 3.5 is the main, where the PINN is initialized and the training is performed. It starts with the model being setup with the features already mentioned earlier in line 3. Afterwards, two keys are generated with the JAX pseudo random number generator, which will be used later. From line 6 to 8 the Adam optimizer [KB17] is defined with the learning rate and the learning rate decay, called weight decay in Flax. Then, the first pseudo random key is used to create an input, which is required to initialize the PINN and define its input dimensions. This is done in line 11, where the optimizer is created for the model. In line 11 the PINN model gets initialized inside

### 3 Implementation

---

#### Listing 3.4 PINN helper functions

---

```
1 def grad_x_ufunction(param, t):
2     def ufunc(x):
3         u = model.apply(param, jnp.concatenate((x, t)))[0]
4         return u
5
6     return jax.jit(ufunc)
7
8 def grad_grad_x_ufunction(param, t):
9     def grad_ufunc(x):
10        u_x = jax.jacfwd(grad_x_ufunction(param, t))(x)
11        return u_x[0]
12
13    return jax.jit(grad_ufunc)
14
15 def grad_t_ufunction(param, x):
16    def ufunc(t):
17        u = model.apply(param, jnp.concatenate((x, t)))[0]
18        return u
19
20    return jax.jit(ufunc)
21
```

the optimizer, with the second pseudo random key and the input generated in line 4. A peculiarity of JAX and Flax is shown, as the input of the PINN is not defined in its class, but rather when it is initialized.

Lastly, the training of the PINN, done from line 14 to 19 in Fig. 3.5, is presented. Here, a callable function with the nested MSE functions with *jax.value\_and\_grad* is created, by calling *make\_mse\_func* and giving it the Burgers' equation data as input. Afterwards, a loop is initiated, for the training process, which consist of 50001 iterations in this example. Here, the callable function created in line 14 is used to return the loss value and the gradient of the current PINN. For this it needs the parameters of the PINN, as the nested *mse* function in listing 3.2 is called. The parameters of the PINN are accessed via the optimizer with *optimizer.target*, as the PINN is initialized through the optimizer in line 11. This gradient is applied to the parameters of the PINN through the optimizer in line 19, which is the end of an iteration and starts the next.

Finally, the way the data for the Burgers' equation is loaded will not be discussed, as this was taken from [RPK18] and adjusted. Therefore, the interested reader is recommended to refer to the appendix of [RPK18] and inspect the code of this publication.



**Listing 3.5** PINN main

```

1 if __name__ == "__main__":
2     ##### Neural Network #####
3     model = DeepFeedForwardNN(features=[20, 20, 20, 20, 20, 20, 20, 20, 1])
4     key1, key2 = random.split(random.PRNGKey(10)) # random keys
5
6     learning_rate = 0.0001
7     weight_decay = 0.02
8     optimizer_def = optim.Adam(learning_rate=learning_rate, weight_decay=weight_decay)
9
10    nnInput = random.normal(key1, (2,)) # Dummy input for the NN to initialize
11    optimizer = optimizer_def.create(model.init(key2, nnInput)) # Initialization call
12
13    ##### NN Training #####
14    grad_fn = jax.value_and_grad(make_mse_func
15                                (x_train, t_train, u_train, x_train_f, t_train_f))
16    for i in range(50001):
17        # perform one gradient update
18        loss_val, gradient = grad_fn(optimizer.target)
19        optimizer = optimizer.apply_gradient(gradient)
20

```

## 3.1 Physics-Informed Neural Network with GPU parallelism

The objective of this section is to improve the performance of the PINNs training, by employing the SPMD solution *pmap* provided by JAX.

This version is shown in the listing 3.6, where the changes from the PINN in listing 3.2 are done from line 11 to 20. Here, the data input is split into the same amount of sub arrays as there are available GPUs, which is two for the utilized setup in this work. This is done in line 11 to 16, by splitting every data array given as input using the JAX numpy *split* function. Afterwards, *pmap* replicates the *lambda* function inside of it, for every sub array that is given as its input. This *lambda* function is still the same, as presented previously in the code 3.2 of chapter 3. Therefore, the loss is calculated in parallel, then summed up and lastly the summation is normalized. Hence, this improvement is best suited, when the loss is to be calculated for a large amount of data points.

Additionally, a version is evaluated, where *grad\_fn*, found in 3.5 at line 18 is parallelised. This is done by giving it an array filled with copies of the *optimizer.target*. Yet, this is not feasible, as *jax.value\_and\_grad* requires a very specific format.

### 3 Implementation

---

#### Listing 3.6 PINN GPU MSE Version 1

---

```
1 def make_mse_func(x_batched, t_batched, u_batched, x_batched_f, t_batched_f):
2     def mse(param):
3         def squared_error_u(x, t, u):
4             pred = u_function(param, x, t)
5             return jnp.square(pred[0] - u[0])
6
7         def squared_error_f(x, t):
8             f = f_function(param, x, t)
9             return jnp.square(f)
10
11         num_of_gpus = 8
12         x_batched_split = jnp.asarray(jnp.split(x_batched, num_of_gpus))
13         t_batched_split = jnp.asarray(jnp.split(t_batched, num_of_gpus))
14         u_batched_split = jnp.asarray(jnp.split(u_batched, num_of_gpus))
15         x_batched_f_split = jnp.asarray(jnp.split(x_batched_f, num_of_gpus))
16         t_batched_f_split = jnp.asarray(jnp.split(t_batched_f, num_of_gpus))
17
18         mse_u = (jnp.sum((jax.pmap(lambda x, t, u: jnp.sum(jax.vmap(squared_error_u)(x, t, u),
19             axis=0)))(x_batched_split, t_batched_split, u_batched_split)) / N_u)
20         mse_f = (jnp.sum((jax.pmap(lambda x, t: jnp.sum(jax.vmap(squared_error_f)(x, t),
21             axis=0)))(x_batched_f_split, t_batched_f_split)) / N_f)
22
23         return mse_f + mse_u
24
25     return mse
```

## 3.2 Physics-Informed Neural Network with Parameter Learning

In the next section, the adaptations, which are required for the prior PINN to achieve it to learn the parameters of the PDE given in the physical loss, are presented.

The first adaption can be found, when comparing the code in 3.7 to 3.1, where the lines 7 and 8 have been added. These lines add the variables *pde\_var1* and *pde\_var2* to the parameters of the PINN, so the gradient is applied to them by the Adam optimizer. For this, the parameter *pde\_var* is added as a sub-directory for the variables *pde\_var1* and *pde\_var2* to be stored in and also functions to initialise them.

The second adaption is visible in 3.8, where line 4, 5 and 6 are added and line 11 is altered. In the lines 4 and 5 the parameters of the PINN get *unfrozen*, which creates a mutable version of the parameters, as the parameters are normally an immutable dictionary and cannot be accessed otherwise. With this, it is possible, to get the variables *pde\_var1* and *pde\_var2*, by accessing the sub dictionary *pde\_var* created in 3.7, in which both variables

**Listing 3.7** PINN PDE NN

```

1 class DeepFeedForwardNN(nn.Module):
2     features: Sequence[int]
3
4     def setup(self):
5         self.layers = [nn.Dense(feats) for feats in self.features]
6
7         pde_var1 = self.variable('pde_var', 'pde_var1', lambda: jnp.ones((1,)))
8         pde_var2 = self.variable('pde_var', 'pde_var2', lambda: jnp.ones((1,)))
9
10    def __call__(self, inputs):
11        x = inputs
12        for l_num, lyr in enumerate(self.layers):
13            x = jnp.tanh(lyr(x))
14            if l_num != len(self.layers) - 1:
15                x = jnp.tanh(x)
16        return x
17

```

are stored. Hence, the variables  $pde\_var1$  and  $pde\_var2$  can be inserted into the  $f$ -function, which now calculates:

$$(3.4) \quad f := u_t + pde\_var1 \cdot u_x - pde\_var2 \cdot u_{xx}$$

Now, the physical loss of the PINN is dependent on  $pde\_var1$  and  $pde\_var2$  and the optimizer will change these variables to fit the data, as this is the point where the loss is optimal. Therefore, the PINN will learn the variables of the PDE while optimizing the loss, which is possible with an abundance of data. The target value for the variables, where the loss of the physical function is minimal, is 1 for  $pde\_var1$  and  $0.01/\pi$  for  $pde\_var2$ .

The training shown in listing 3.5 stays unaltered, as the new variables are also stored in the `optimizer.target`, where they get optimized alongside the parameters of the PINN.

### 3.3 Further changes

In this chapter, adaptations applied to all versions of the PINN are introduced and therefore these changes are omitted in the prior sections. This allows for better comparability with the paper [RPK18].

Here, the MSE function shown in Fig. 3.2 is modified, by adding some randomness to the data batches. This is done in the lines 3 to 9 in Fig. 3.9, where an array with the length of the shorter batch is created. This array is created with the function `np.arange()`, which creates evenly spaced values, with the distance 1 within the specified interval. Then, the array is randomized with the function `np.random.shuffle()`. Finally, the shuffled array is used to reorder the data batches in unison, which is done in the lines 5 to 9.

### 3 Implementation

---

#### Listing 3.8 PINN PDE f-function

---

```
1 # f(t,x)
2 @jax.jit
3 def f_function(param, x, t):
4     unforzen_params = unfreeze(param)
5     pde_var1 = unforzen_params['pde_var']['pde_var1']
6     pde_var2 = unforzen_params['pde_var']['pde_var2']
7     u = u_function(param, x, t)
8     u_t = jax.jacfwd(grad_t_ufunction(param, x))(t)
9     u_x = jax.jacfwd(grad_x_ufunction(param, t))(x)
10    u_xx = jax.jacfwd(grad_grad_x_ufunction(param, t))(x)
11    f = u_t[0] + pde_var1 * u[0] * u_x[0] - pde_var2 * u_xx[0]
12    return f[0]
13
```

---

#### Listing 3.9 PINN MSE with data shuffling

---

```
1 def make_mse_func(x_batched, t_batched, u_batched, x_batched_f, t_batched_f):
2     # randomize the batches
3     randomize1 = np.arange(len(x_batched))
4     np.random.shuffle(randomize1)
5     x_batched = x_batched[randomize1]
6     t_batched = t_batched[randomize1]
7     u_batched = u_batched[randomize1]
8     x_batched_f = x_batched_f[randomize1]
9     t_batched_f = t_batched_f[randomize1]
10
11    def mse(param):
12
13        # Define the squared loss for a single pair (x,t) and (u)
14        def squared_error_u(x, t, u):
15            pred = u_function(param, x, t)
16            return jnp.square(pred[0] - u[0])
17
18        def squared_error_f(x, t):
19            f = f_function(param, x, t)
20            return jnp.square(f)
21
22        mse_u = ((jnp.sum(jax.vmap(squared_error_u)(x_batched, t_batched, u_batched), axis=0))
23                / N_u)
24        mse_f = ((jnp.sum(jax.vmap(squared_error_f)(x_batched_f, t_batched_f), axis=0)) / N_f)
25        return mse_f + mse_u
26
27    return jax.jit(mse)
```

---

## 4 Results

In this chapter the results of the PINNs implemented in chapter 3 are presented. For this the machines, on which the results were achieved, are introduced and the results are compared to those found in related literature. Especially, the values from the paper [RPK18] and the scientific machine learning library DeepXDE [LMMK21] are used.

### 4.1 Setup

For testing and running the PINNs implemented in chapter 3 the servers provided by the Institute for Parallel and Distributed Systems (IPVS) Stuttgart are utilized. In total two different servers are used, one for the PINNs that runs on a single GPU and another one for the PINNs that have multiple GPUs employed.

For the first case the *pcsgs02* server is used, which has a Nvidia GeForce RTX 3080 GPU.

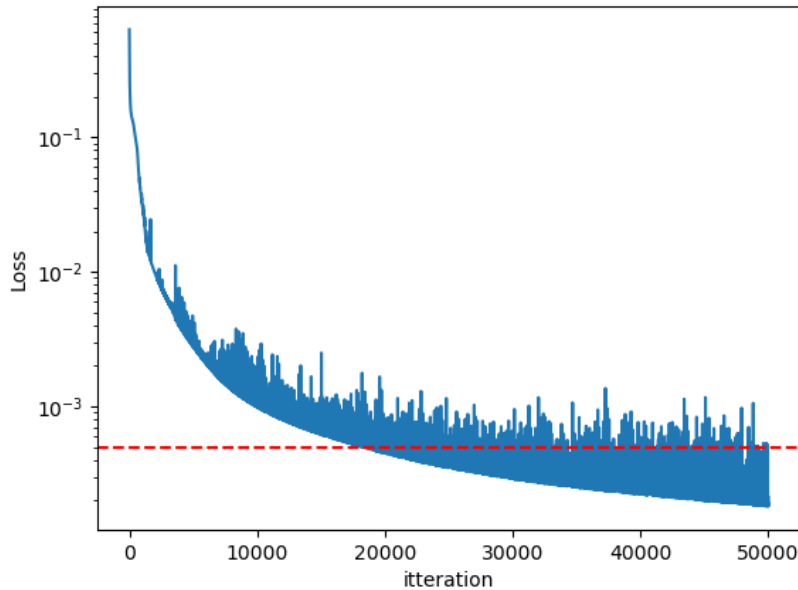
For the second the *argon-tesla-2* server is utilized, which has two Quadro GP100 GPUs.

### 4.2 Single GPU Physics-Informed Neural Network

In this section, the PINN is tested on the *pcsgs02* server, which runs on a single GPU and uses the physical function to restrict the solution space to a region, where physically sound predictions can be made. With this, the amount of data that is required to achieve a low loss for the PINN in this work can be reduced.

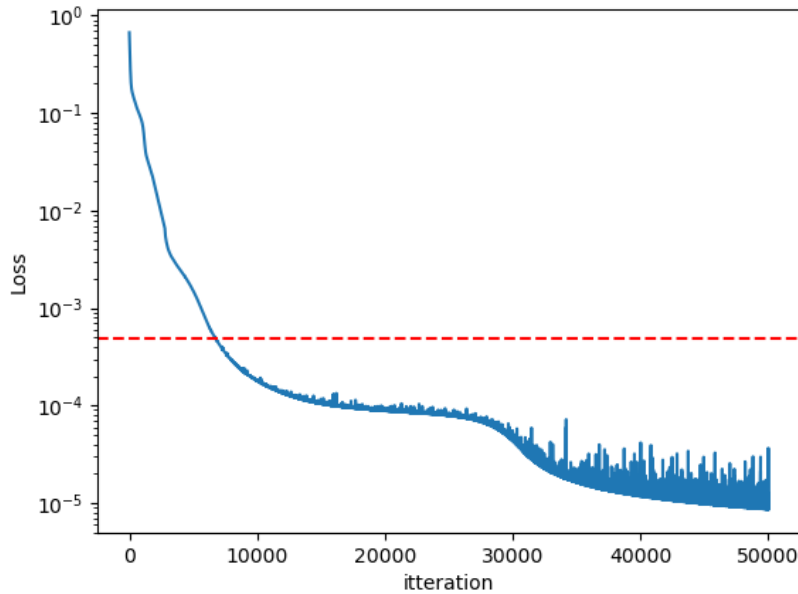
Hence, 200 random data points are used for the data loss function and 10000 for the physical loss function for the figures 4.1, 4.2 and 4.3. Figures 4.1 and 4.2 show the loss for this data set over the iterations, with the y-axis being logarithmic, as the loss is minimized towards zero. The red line in both figures is the loss taken as benchmark from the paper [RPK18], which was  $4.9E - 04$  for 200 data-, 10000 physical loss points and 50000 iterations. Here, the loss of Fig. 4.1 shows similar results as the benchmark at iteration 50.000 or even a bit better. Yet, the loss outlined in Fig. 4.2 is significantly smaller, with the precise loss value being  $6.93E - 06$  at the iteration 50000. Although, this precise value is just a snapshot and the loss in Fig. 4.2 experiences some fluctuation. This considerable improvement is achieved, by shuffling parts of the data, which is used to generate the mean squared error.

Here, the shuffling is done to the data array in such a way that the data belonging together also is shuffled in unison. Therefore, the data still describes the Burgers' equation, just not in the same order. Through this, the variance of the training data was increased. As the data now changes with every iteration and a lock-in effect on the training sequence is avoided.



**Figure 4.1:** MSE loss of the basic single GPU PINN, with the benchmark as red line

The time improvements, which could be achieved through JAX, have been tried. For this, a version of the algorithm, where all JIT compilations had been removed is tested. This version runs on the *ipvs02* server and takes 84923 seconds. The next version is outlined in chapter 3, where it only needed 3165 seconds, which is only 3.7% of the time of the prior version and more than 26 times faster. This can be improved even further, by wrapping the *grad\_fn* function, found in line 14 of listing 3.5, in *jax.jit()*. With this improvement, the PINN only takes 1686 seconds, which is 1.99% or 53.72% of the respective prior versions. Additionally, the very slow version of the PINN, where nothing is JIT compiled and the calculation of the gradients is not optimized, is used, to JIT compile the *grad\_fn* function, just as in the last example. Here, the training of the PINN with 50000 iterations takes 1715 seconds, which is just 1.7% more than the fully optimized version. This difference can be explained either through the difference in the calculation of the gradients or just pure randomness, as the sample size for the test was only five trials.



**Figure 4.2:** MSE loss of the improved single GPU PINN, with the benchmark as red line

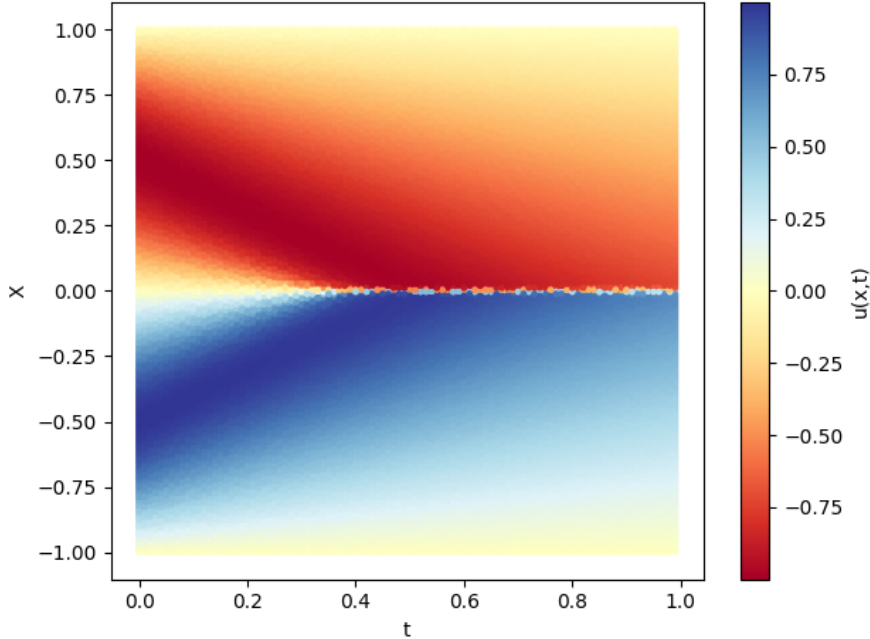
However, the result from the performed experiments are that JAX makes it very easy to drastically improve the performance and run time of code, as it is enough to JIT compile the `grad_fn` function. This `jit` function then optimizes the entire process of generating the loss and the gradient. The only part that is not compiled by `jit` function is the application of the gradient to the parameters of the PINN.

Here, it is also interesting to mention that the GPU usage varied throughout these versions. The least efficient version of the PINN has a GPU usage of 1 – 5% with large fluctuations. Whereas the fastest version has a GPU usage of 7 – 8%, with little to no fluctuation.

The work additionally compares the performance of the PINN to DeepXDE [LMMK21], which is a library for scientific machine learning that already has an example PINN implemented, which solves the Burgers' equation. Their model has a final loss of  $6.60E - 06$ , which is slightly better than the PINN of this paper. Although, the training of the DeepXDE Burgers' equation model only did 22566 iterations and took 169 seconds, which is about a fifth of the time the PINN in this paper needed at the same iteration step. Yet, it must be mentioned that this TensorFlow binary is optimized with oneAPI Deep Neural Network Library [NB21]. Additionally, it is expected that the team associated with the library is more experienced in developing scientific machine learning.

Finally, the results of the trained PINN, with the loss mentioned earlier, are outlined. For this the PINN calculates every of the 25600  $u(x, t)$  data points and plot it afterwards. The product of this can be seen in Fig. 4.3. The version of the Burgers' equation the PINN created in Fig. 4.3 can be compared with the one plotted from the data generated through

Latin Hypercube Sampling in Fig. 2.6. As expected, these look exactly the same to the human eye, as the MSE is only  $6.93E - 06$ . Especially, since the MSE of the loss is smaller than in the publication [RPK18], which already presented a precise prediction, compared to the real result.



**Figure 4.3:** Burgers' equation plotted with the output of the single GPU PINN

### 4.3 Multi GPU Physics-Informed Neural Network

This chapter introduces a PINN, which is modified, to use the *pmap* function provided by JAX, to employ SPMD parallelism. For this, it is run on the *argon-tesla-2* server which has two Quadro GP100 GPUs, as already mentioned in the section 4.1.

Additionally, the single GPU version of the PINN will be run on the same server to compare the results with one another. As the *argon-tesla-2* server has more GPUs than the *pcsgs02* server. Yet, each GPU has less computing power compared to the Nvidia GeForce RTX 3080. Therefore, it would not make sense to compare the outcome of the multi GPU PINN to the results of the single GPU PINN on the *pcsgs02* server. The graphs for the loss and the output of the multi GPU PINN will also be omitted, as these are the same as in section 4.2, since these PINNs produce the same result and only their optimization is altered.



For the multi GPU PINN version several versions are tested. The fastest run time can be achieved, when the *grad\_fn* function is also wrapped in *jax.jit*. Even though it can contribute to sub-par data movement, when wrapping a *pmap* function, it still leads to the best run time. This is most likely due to the fact that otherwise large parts of the code run without being XLA JIT compiled.

The run time of the single GPU PINN version on the *argon-tesla-2* server is 2235 seconds. This results in a slower run time than in section since the individual GPUs of the used server provide less computing power. Whereas the multi GPU PINN version only achieved 3363 second. Here, the run time of the single PINN version is significantly faster, as it only requires 66.46% of the time and is about 1.5 times faster. Additionally, the single GPU version of the PINN only used 8 – 10% of the GPU, whereas the multi GPU version took 10 – 16% per GPU.

This lacklustre result is due to the fact that it is against the design of JAX to wrap *pmap* in *jit*. Yet, the run time is even worse when the *grad\_fn* function is not optimized with the *jit* function.

As already mentioned, several versions have been implemented, where *pmap* is used in different ways or different parts of the program. However, these versions either perform worse or are not feasible, because of the restrictive inputs JAX and Flax require. For instance, one of the approaches not feasible evaluates the possibility of wrapping the *grad\_fn* function in *pmap* and run several iterations of the PINN in parallel. Yet, this is not viable, as already described in section 3.1, since it is not possible to use these functions in conjunction in the current version of JAX and Flax. Because of the distinct format of the *optimizer.target*, *pmap* is not able to map over an array with several *optimizer.target* instances.

Although, this also does not seem to be the intended use for *pmap*. Since the documentation implies that it is supposed to be used to implement functions that compute parts of the data in parallel. Therefore, the approach of wrapping the *grad\_fn* function in *pmap* might not be intended altogether.

Another factor is the small amount of data points used with every iteration. As the overhead of the data movement might exceed the improvements, achieved from splitting up the computation.

## 4.4 Partial Differential Equation learning Physics-Informed Neural Network

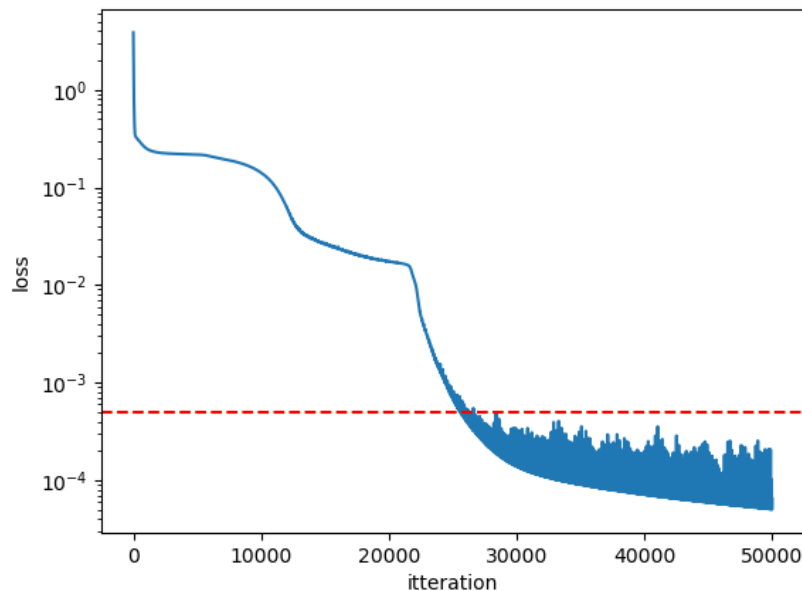
In this section the result that can be achieved with the PINN shown in section 3.2 are presented, which uses the abundance of available data, to also learn the parameters of the PDE.

Here, the MSE of the PINN, the error curves of the two parameters that were learned and of course the Burgers' equation created with the resulting PINN, to check the correctness of the implemented solution, will be examined.

To compare the results, the paper [RPK18] has been used again, as a benchmark. Hence, the same hyper parameters of 2000 random points for the data and the physical loss each, 50000 iterations of learning and 8 layers of 20 neurons are employed. However, the PINN in this paper was trained by using the Adam optimizer, whereas the L-BFGS [LN89] optimizer is mostly used in the literature. This has been done, as neither JAX nor FLAX provides L-BFGS optimizer [LN89].

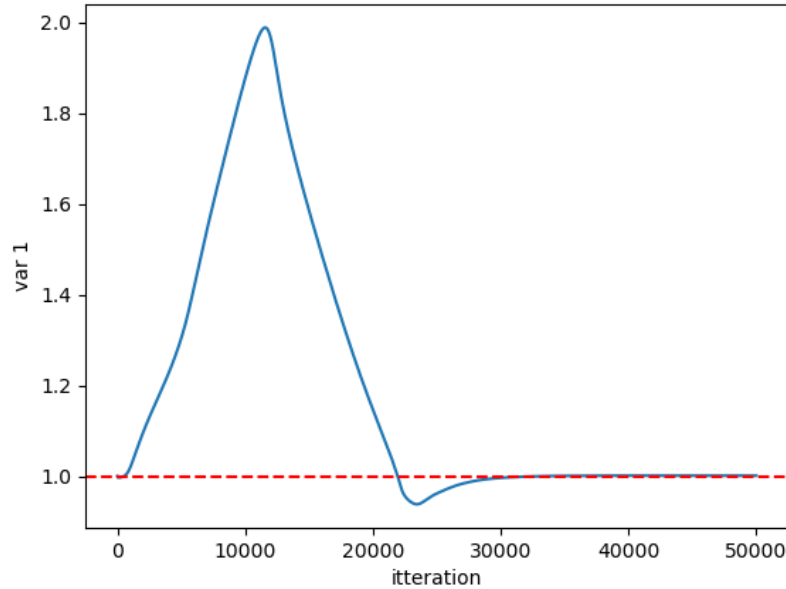
As a loss value was not provided for the PDE learning PINN in the paper [RPK18], but just a graph, the loss that was already used in the section 4.2 is utilized again. This value is once again shown by a red line in Fig. 4.4 and should be sufficient, as the prior PINN only optimized for the loss. Here, the graph outlines that the PINN in this paper still outperformed the benchmark. The exact value of the loss in Fig. 4.4 is  $6.48E - 5$ , yet including the fluctuations, the value is  $1E - 4$ .

It is also relevant to mention that the first variable did not optimize in the direction of the target value of 1 until about the 10000th iteration, as can be seen in Fig. 4.5. This is indicative of how the PINN discovers the parameters of the PDE, as the PINN first must learn to solve the Burgers' equation and then fit both variables to minimize the physical loss.



**Figure 4.4:** Loss curve of the MSE for the PINN that learned the PDE parameters, with the benchmark as red line

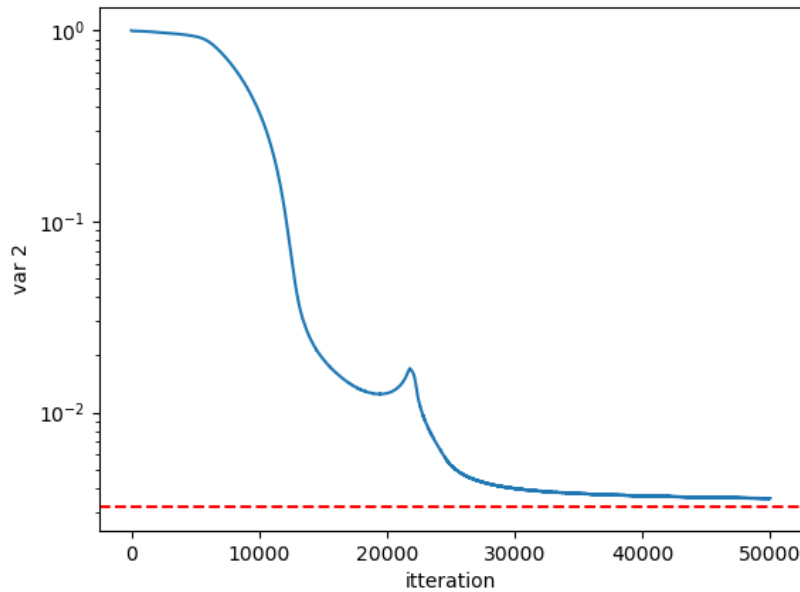
The first variable of the Burgers' equation the PINN learned, which can be seen in Fig. 4.5. Here, the final value achieved is 1.000915, which is an error of 0.092% compared to the actual value of 1.0, which is again shown by a red line. This is better than the result in the from the benchmark, which had an error of 0.141%.



**Figure 4.5:** Loss curve of the first PDE variable, with the benchmark as red line

The second variable of the introduced PDE can be seen in Fig. 4.6, where the red line shows the variables real value in the Burgers' equation of  $0.01/\pi$  or  $\approx 3.183E - 3$ . Here, the PINN implemented in this paper learns the value  $3.534E - 3$ , which has an error rate of 11.027%. Where PINN presented in this publication did better with the first variable, now it performs way worse, as the paper [RPK18] only had an error rate of 1.902%. This discrepancy can be explained by the random initialization of the PINN parameters or the optimizer, which was used, as this work employed the Adam optimizer and the benchmark used the L-BFGS optimizer. Yet, randomness is a rather unlikely factor to attribute the complete difference in the results. Hence, the optimizer is viewed as the more likely reason for these different results.

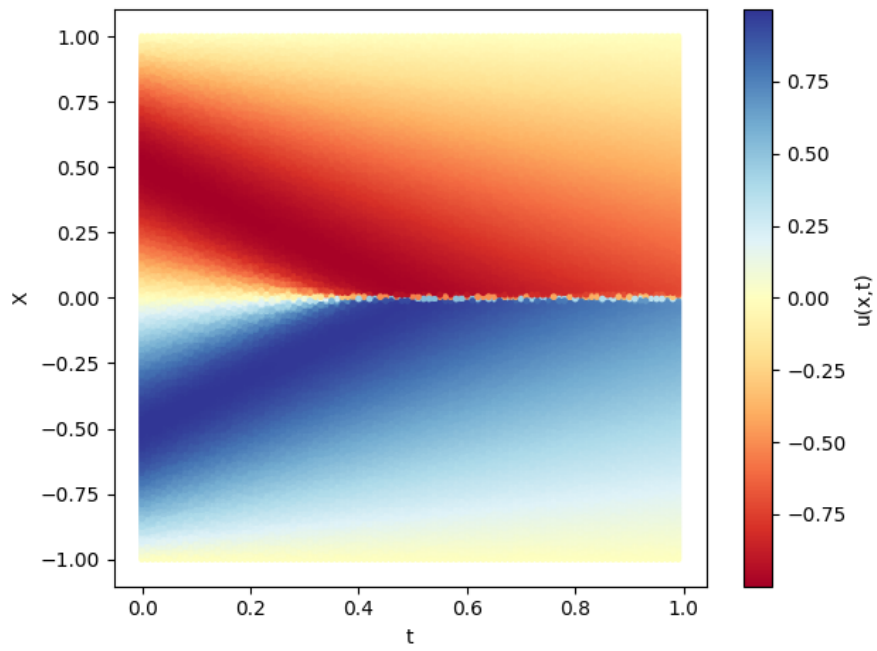
The optimization process of this PINN was also timed. Here, a time of 3315 seconds is achieved for the PINN, as it was shown in the section 4.2 and 1761 seconds, when the *grad\_fn* function is also JIT compiled. These results are similar to the results of the version without PDE learning in section 4.2 of 3165- and 1686 seconds. Although, 10200 points for the learning of the PINN in section 4.2 and 8000 for the PINN in this section are used. A possible reason for this similarity is that the data loss is more computationally complex and therefore the setups might be around the equilibrium of physical- and data



**Figure 4.6:** Loss curve of the second PDE variable, with the benchmark as red line

loss computation. Another factor, which is expected to increase the run time, is the PDE learning. Although, the extent to which it inflates the run time the PINN needs to learn its parameters is not known.

In the last part of this section, the output of the PINN is evaluated, to confirm the correctness of the implemented learning. This can be seen in Fig. 4.7, where the output of the PINN trained is plotted, which looks exactly like the Burgers' equation in 2.6. Here, the graph was again plotted so that the  $x$  values are on the left-, the  $t$  values on the bottom- and  $u(x, t)$  on the coloured axis. Even though, the loss that can be achieved with this PINN is higher, compared to section 4.2, the PINN still outputs a graph that is indistinguishable from the figures 4.3 and 2.6. This is to be expected, as the loss is still better than the loss in the benchmark [RPK18], with which it is still possible to generate a Burgers' equation that seems tantamount to the original.



**Figure 4.7:** Burgers' equation generated via the output of the PINN, which also learned the PDE parameters



## 5 Conclusion

JAX and Flax are potent frameworks that lend themselves to developing PINNs, as they provide a simple and efficient way to get the gradients needed for the physical loss function. This alleviates some of the work from the developer and allows for easy prototyping. Although, there can be problems with JAX, as the documentation is in some instances misleading and one has to use a trial and error approach. This work showed an example of this in Fig. 3.4 and 3.3, where the implementation of helper functions is required for the gradients.

JAX additionally provides a flexible and easy to use tool to improve the performance of functions via *jax.jit*. As shown in the results, this drastically improves the performance and the time it requires to train the PINN. Especially, since this improvement can be easily achieved, by JIT compiling the *grad\_fn*, as shown in chapter 4. This approach to optimization makes it especially convenient, as one does not have to determine the most efficient way to JIT compile their code, while implementing it. Therefore, efficiency can be treated like an afterthought that can just be added, after the NN and its training are setup.

The setup of the PINN and the training outlined in chapter 3, with its optimizer, is also straightforward with Flax. A class for the PINN can be realized that is as extensive or as plain, as required for any specific project. Here, one is free to implement every layer manually or build a more flexible version, which iterates over arrays that tell the PINN how to be setup up. Whereas the first one is certainly easier to understand and follow, the second version has less code and allows for quicker changes.

The implementation of the PDE learning is also straightforward to add to an existing PINN, as can be seen in section 3.2. Here, the variables can just be added to the dictionary, where the parameters of the PINN are stored. Afterwards, the optimizer calculates the gradient of these new parameters alongside the prior ones. Yet, here the documentation of Flax lacks critical information, as a lot of this had to be found out by testing during the course of this work. Since this is a prime example, where JAX and Flax would be utilized, it should be documented how to implement this.

Flax is also lacklustre in its optimizer package. Here, specifically for the purpose of this work, the L-BFGS optimizer was missing. Especially, since this is the optimizer of choice in DeepXDE [LMMK21] and for the PDE learning PINN in the publication [RPK18], has been used in the work as benchmark for the results. This comes to show that JAX and Flax are still in early development, which gets even more emphasised by

the *experimental* package found in JAX and the fact that distributed learning, on multiple nodes, is not yet possible with these packages, even though this should be a core feature for a high-performance machine learning framework.

Another inconvenience is that the *grad\_fn*- and the *pmap* function cannot be used in conjunction, as their data formats do not match. This impedes the performance of the PINN, as the whole computation is not run in parallel, but only parts. Especially, since the *grad\_fn* function should not be wrapped in *jax.jit*, as doing this leads to sub optimal data transport for the *pmap* function. Yet, this JIT compilation yields drastic time improvements for the training, as shown in the results found in section 4.2. These problems could be less significant, when the PINN uses vastly more data, for which the loss could be calculated in parallel, as was implemented in the work. However, the code outside of the loss calculation would still not be XLA optimized. This includes the calculation of the gradient, which should not be neglected, as can be seen in the results. Hence, it would be better if one could just wrap *grad\_fn* in a *pmap* function and run several iterations of the training at the same time. This would then XLA compile the training similarly to the *jax.jit* wrapper.

Summarised, JAX and Flax provide convenient functionalities for the developer, yet requires precise formats, which at times feel restricting. This strictness is obviously intended by the teams behind these frameworks, as it is exploited to enhance performance. Although, the issues caused by this strictness could be improved or even mitigated, by refining their documentations.

Considering all these points, the verdict is that JAX and Flax provide a suitable framework for developing high-performance PINNs with relative ease. Yet, there is clear room for improvements. Although, when the PINN and the underlying geometry of the problem is not that complex and the user can rely upon the documentation, these complications might not matter. In such cases JAX and Flax present a suitable environment to implement high-performance machine learning, as gradients and complex code optimization can be utilized effortlessly.

## Outlook

Some of the key points that will be mentioned in the outlook, as a possible next step, are already introduced in the conclusion.

For the first possible next step for future work, it would be interesting to try JAX and Flax with a specific problem, which would exhaust the computing power of the servers. This would allow to verify if the performance and time improvements achieved through JAX scale to larger PINNs. This is especially interesting, as it could be exceedingly cumbersome to optimize a more complex problem manually. Here, it would also be intriguing to implement a PINN with another framework and compete to achieve the same performance enhancements that JAX provides effortlessly.



---

As JAX and Flax are new and constantly evolving, it would be compelling to examine them at a later stage of their development. Here, the new state of these frameworks could be compared to the critique expressed in the conclusion in section 5. For this, the focus should be on the parallelisms, the distributed learning and their interoperability with the *grad\_fn* and *pmap* functions.

Lastly, it would be intriguing to investigate further new frameworks, which also aim to be used for high-performance machine learning. These could then match up against the results found in the chapter 4 and the publication [RPK18].



# Bibliography

- [ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283 (cit. on p. 26).
- [Ayc03] J. Aycock. “A Brief History of Just-in-Time”. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300. URL: <https://doi.org/10.1145/857076.857077> (cit. on p. 24).
- [BFH+18] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: <http://github.com/google/jax> (cit. on pp. 14, 24).
- [BP72] E. R. Benton, G. W. Platzman. “A table of solutions of the one-dimensional Burgers equation”. In: *Quarterly of Applied Mathematics* 30 (1972), pp. 195–212 (cit. on p. 22).
- [BPR15] A. G. Baydin, B. A. Pearlmutter, A. A. Radul. “Automatic differentiation in machine learning: a survey”. In: *CoRR* abs/1502.05767 (2015). arXiv: 1502.05767. URL: <http://arxiv.org/abs/1502.05767> (cit. on p. 19).
- [De10] R. Deidinger. “Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming”. In: *Society for Industrial and Applied Mathematics* 52.3 (2010), pp. 545–563. DOI: [10.1137/080743627](https://doi.org/10.1137/080743627) (cit. on p. 19).
- [FJL18] R. Frostig, M. J. Johnson, C. Leary. “Compiling machine learning programs via high-level tracing”. In: *Systems for Machine Learning* (2018) (cit. on pp. 23, 24).
- [HLO+20] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, M. van Zee. *Flax: A neural network library and ecosystem for JAX*. Version 0.3.3. 2020. URL: <http://github.com/google/flax> (cit. on pp. 14, 26).
- [Hof15] P. H. W. Hoffmann. “A Hitchhiker’s Guide to Automatic Differentiation”. In: *Numerical Algorithms* 72.3 (Oct. 2015), pp. 775–811. ISSN: 1572-9265. DOI: [10.1007/s11075-015-0067-6](https://doi.org/10.1007/s11075-015-0067-6). URL: <http://dx.doi.org/10.1007/s11075-015-0067-6> (cit. on pp. 19, 21).

- [KB17] D. P. Kingma, J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 31).
- [LMMK21] L. Lu, X. Meng, Z. Mao, G. E. Karniadakis. “DeepXDE: A deep learning library for solving differential equations”. In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067 (cit. on pp. 37, 39, 47).
- [LN89] D. C. Liu, J. Nocedal. “On the Limited Memory BFGS Method for Large Scale Optimization”. In: *MATHEMATICAL PROGRAMMING* 45 (1989), pp. 503–528 (cit. on p. 42).
- [MDA15] D. Maclaurin, D. Duvenaud, R. P. Adams. “Autograd: Effortless Gradients in Numpy”. In: *ICML 2015 AutoML Workshop*. 2015. URL: /bib/maclaurin/maclaurin-autograd/automl-short.pdf, <https://indico.lal.in2p3.fr/event/2914/session/1/contribution/6/3/material/paper/0.pdf>, <https://github.com/HIPS/autograd> (cit. on p. 23).
- [NB21] R. Nozal, J. L. Bosque. “Exploiting co-execution with oneAPI: heterogeneity from a modern perspective”. In: *CoRR* abs/2106.01726 (2021). arXiv: 2106.01726. URL: <https://arxiv.org/abs/2106.01726> (cit. on p. 39).
- [OS02] A. M. J. Olsson, G. E. Sandberg. “Latin Hypercube Sampling for Stochastic Finite Element Analysis”. In: *Journal of Engineering Mechanics* 128.1 (2002), pp. 121–125. DOI: 10.1061/(ASCE)0733-9399(2002)128:1(121) (cit. on p. 22).
- [PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems* 32. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf> (cit. on p. 26).
- [RN20] S. J. Russel, P. Norvig. *Artificial Intelligence: A Modern Approach*. 4. Pearson, 2020. ISBN: 0134610997 (cit. on pp. 13, 17, 18).
- [RPK18] M. Raissi, P. Perdikaris, G. Karniadakis. “Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations”. In: *Journal of Computational Physics* 378 (Nov. 2018). DOI: 10.1016/j.jcp.2018.10.045 (cit. on pp. 13, 15, 22, 29, 31, 32, 35, 37, 40, 42–44, 47, 49).
- [RYK20] M. Raissi, A. Yazdani, G. E. Karniadakis. “Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations”. In: *Science* 367.6481 (2020), pp. 1026–1030. ISSN: 0036-8075. DOI: 10.1126/science.aaw4741. eprint: <https://science.sciencemag.org/content/367/6481/1026.full.pdf>. URL: <https://science.sciencemag.org/content/367/6481/1026> (cit. on pp. 13–15).

[SSE20] O. Saldır, M. Sakar, F. Erdogan. “Numerical Solution of Fractional Order Burgers’ Equation with Dirichlet and Neumann Boundary Conditions by Reproducing Kernel Method”. In: *Fractal and Fractional* 4 (June 2020), p. 27. DOI: [10.3390/fractalfract4020027](https://doi.org/10.3390/fractalfract4020027) (cit. on p. 22).

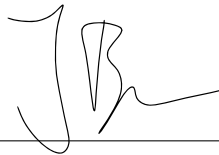
All links were last followed on October 3, 2021.



## **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Heimsheim, 5.10.2021,

A handwritten signature in black ink, consisting of stylized initials 'JB' followed by a horizontal line extending to the right.

---

place, date, signature