universität
wien

BACHELOR THESIS

Title of the Bachelor Thesis

Neural Networks for Differential Equations with focuse on Competitive Physics Informed Networks

submitted by

Josef Wagner

Aspired Academic Degree

Bachelor of Science (BSc.)

Vienna, February 2025

Studienkennzahl lt. Studienblatt:	120 31 612
Subject:	Mathematics
Supervisor:	Assoz. Prof. Dr. Philipp Christian Petersen , M.Sc.

Abstract

Can neural networks be applied to mathematical problems? While this might be a broad question, consider the following application to differential equations.
Even if the method does not restrict to partial differential equations, the prototypical case that will be discussed is the Schrödinger equation,

$i * \partial_t u + \partial_x^2 u + u|u|^2 = 0.$

If we want to compute this numerically, we can write

$\min_{u \in C^2(\mathbb{C})} \; i * \partial_t u + \partial_x^2 u + u|u|^2.$

Now we are replacing $C^2(\mathbb{C})$ by neural networks, requiring that every $C^2(\mathbb{C})$ function can be approximated by a neural network,

$\min_{u \in \textit{neural networks}} i * \partial_t u + \partial_x^2 u + u|u|^2.$

In this work, we build this method and formulate it in a general setting.
In the first chapter, we start with neural networks.
In chapter two we formulate and implement the above minimization problem.Further in the fourth chapter we reformulate the original equation in terms of a min max problem, and try to understand the consequences from a game theoretic perspective in chapter three.

Contents

1 Neural networks

1.1 Construction

In this chapter we follow Liquet 2024[chapter 4.1] [1] to build and introduce **feedforward neural networks**. We start by defining a argument dimension m > 0 and a image dimension k > 0.
Let $W \in \mathbb{R}^{k \times m}$ a matrix and $b \in \mathbb{R}^m$ a vector. We set

$$f(x) := W * x + b. \tag{1}$$

Let x_0 be given a vector in $\mathbb{R}^k$,
since $f(x_0) \in \mathbb{R}^m$ we can define a vector $z := f(x_0) \in \mathbb{R}^m$. Now we apply a function σ to z,

$$a := \sigma(z) = \sigma\left(W * x + b\right) \tag{2}$$

This is one step in (or layer) in a recursive definition.
To indicate we are in step i,we write $a^{[i]}, \sigma^{[i]}, z^{[i]}, W^{[i]}, b^{[i]}$, $0 < i < \infty$.

The next step $i + 1$ is defined similar. Except it can vary in dimension and the $\sigma^{[i+1]}, z^{[i+1]}, W^{[i+1]}, b^{[i+1]}$ are not equal to the ones in step i.
We have,

$$f^{i+1}(a^{[i]}) = W^{[i+1]} * a^{[i]} + b^{[i+1]} = W^{[i+1]} * \sigma^{[i+1]}(z^{[i]}) + b^{[i+1]} = W^{[i+1]} * \sigma^{[i]}\left(W^{[i]} * x + b^{[i]}\right) + b^{[i+1]}. \tag{3}$$

The resulting vector from step i, $a^{[i]}$ is of dimension m if σ is applied componentwise. Hence, the weight matrix $W^{[i+1]}$ has the constraint for the outer dimension m: $W^{[i+1]} \in \mathbb{R}^{j \times m}$ $j > 0$. This way we can build for a given number of steps $i \in 0, .., n$

a feedforward neural network.

For instance for $i = 3$,

$$x \rightarrow W^{[1]} * x + b^{[1]} \xrightarrow{\sigma^{[1]}} a^{[1]} \rightarrow W^{[2]} * a^{[1]} + b^{[2]} \xrightarrow{\sigma^{[2]}} a^{[2]} \rightarrow w^{[3]} * a^{[2]} \xrightarrow{\sigma^{[3]}} a^{[3]}$$

Where the lower case w in the last step indicates the w is a vector and scalar product $w * x$ yields a real number. Suppose all dimensions are three. We can write the matrix multiplication $W * x + b$ and $\sigma(a)$ componentwise.

$$x \rightarrow \begin{matrix} [w_1 * x + b_1] \\ [w_2 * x + b_1] \\ [w_3 * x + b_3] \end{matrix} \xrightarrow{\sigma} \begin{matrix} \sigma(w_1 * x + b_1) = a_1 \\ \sigma(w_2 * x + b_2) = a_2 \\ \sigma(w_3 * x + b_3) = a_3 \end{matrix}$$

The vector x can also be a matrix $X \in \mathbb{R}^{k \times k}$, then the vector b changes to the matrix $[b, .., b]$

At this point it's time to introduce terminology.
The tuple $(w_k * x + b_k \mid a_k)$ is a **neuron**.
The matrices W are weights and the vectors b **biases**. All weights and biases are **parameters**.
Changeable variables that are not parameters are **hyperparameters**.
The functions σ are **activations function**.

1.2 Parameter optimization

In this section we focus on parameter optimization, another task is finding the hyperparameters of the neural network.

We write $\theta = (W_1, W_2, .., b_1, b_2, ..)$ for the parameters of the neural network NN. In the regression framework we try to predict values of a function $g(x)$ at unknown values x based on known values $g(x_0), ..., g(x_i)$. If we proceed with a neural network $NN(x, \theta)$, we minimize the distance from the NN to g at corresponding points (or vectors) $x_1, .., x_j,\ j \leq i$. This reads Després (2020)[page 10][3]

$$\min_{\theta} \sum_{x_i} (NN(x_i, \theta) - g(x_i))^2. \tag{4}$$

Here we have measured distance with the **L2 loss**. If we take the mean we get

Definition 1.1 (Mean squared error)**.** [13] Let K be a number of samples $x_1, ..., x_K$,further let $\hat{y}_i$ the predicted value of sample i, then
$MSE := \frac{1}{K} \sum_{i=1}^{K} (g(x_i) - \hat{y}_i)^2 = \frac{1}{K} \sum_{i=1}^{K} (y_i - \hat{y}_i)^2$
is the mean-squared error.

Remark. [13]

- -The MSE is differentiable
- - if $(g(x_i) - \hat{y}_i) > 1$ then one sample can increase the MSE considerable.

Since are trying to minimize the distance or **loss function** L in terms of the parameters we compute the respective derivatives.
Once the derivatives of weights and biases are computed we adjust the current weights and biases.
We choose an **optimizer** which controls how this adjustment is done. A simple optimizer to understand the concept is **gradient descent**. For a given $\gamma \in \mathbb{R}$ the adjusted parameters are given by

$$b^{[i+1]} := b^{[i]} - \gamma \frac{\partial L}{\partial b^{[i]}}$$
$$W^{[i+1]} := W^{[i]} - \gamma \frac{\partial L}{\partial W^{[i]}},$$

equivalently

$$\theta^{[i+1]} := \theta^{[i]} - \gamma \frac{\partial L}{\partial \theta^{[i]}}. \tag{5}$$

The parameters at iteration zero are set manually.

1.3 Back-propagation

In this chapter we follow Liquet 2024 [1] for explaining the **back-propagation algorithm**. We address the computation of the partial derivatives $\frac{\partial L}{\partial \theta^{[i]}}$ numerically with the back-propagation algorithm. Back-propagation can be explained through an example. Let
$w_{k,j}^{[i]}$ the j,k entry of Weight Matrix $W^{[i]}$. Now we consider a neural network of the form

$$\begin{array}{rcl} z^{[1]} & = & W^{[1]}x + b^{[1]} \\ a^{[1]} & = & tanh(z^{[1]}) \\ z^{[2]} & = & W^{[2]}a^{[1]} + b^{[2]} \\ \hat{y} = a^{[2]} & = & tanh(z^{[2]}) \end{array} . \tag{6}$$

Further the **least square** loss function,

$$Ł := \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - a^{[2]})^2 \tag{7}$$

.

We compute the derivative with respect to the weights and biases.
Suppose we are trying to compute the derivative of $NN(x, \theta)$ with respect to $w_{1,1}^{[1]}$. Since we got a nested function we apply the chain rule,

$$\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w_{1,1}^{[1]}}. \tag{8}$$

$$\tag{9}$$

We notice 8 can also be written by summarizing every derivative except the last,

$$\frac{\partial L}{\partial w_{1,1}^{[1]}} = \frac{\partial L}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w_{1,1}^{[1]}}. \tag{10}$$

The backprop algorithm suggests computing the derivative of every $z^{[1]}$ in a backwards manner $z^{[i]}$ from $z^{[i+1]}$ by noticing,

$$\begin{aligned} \frac{\partial L}{\partial z^{[i]}} &= \frac{\partial L}{\partial a^{[i]}} \frac{\partial a^{[i]}}{\partial z^{[i]}} \\ &= \frac{\partial L}{\partial z^{[i+1]}} \frac{\partial z^{[i+1]}}{\partial a^{[i]}} \frac{\partial a^{[i]}}{\partial z^{[i]}} \end{aligned}$$

.

From the derivative $\frac{\partial L}{\partial z^{[i]}}$ we can compute the weights of the corresponding parameters $W^{[i]}, b^{[i]}$.

For instance for the network above 6. We define $g(x) := tanh'(x) = 1 - tanh(x)^2$. The derivative of the tanh. We calculate the derivates,

1.
$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = (a^{[2]} - y_0) * g(z^{[2]})$$
$$\frac{\partial L}{\partial z^{[1]}} = \frac{\partial L}{\partial z^{[2]}} \frac{\partial z^{[1+1]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} = \frac{\partial L}{\partial z^{[2]}} W^{[2]} * g(z[1])$$

2.
$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial z^{[2]}} a^{[1]T} \qquad \frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial z^{[2]}} \qquad \frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial z^{[1]}} x^T \qquad \frac{\partial L}{\partial b^{[1]}} = \frac{\partial L}{\partial z^{[1]}}.$$

2 PINN

2.1 Formulation

In this chapter we follow Raissi et al. (2019)[chapter 3][4]. We have given a differential equation of the form

$$\partial_t u(t,x) + A[u(t,x)] = 0. \qquad x \in \Omega \qquad t \in [0,T] \tag{11}$$

Where $\Omega \subset \mathbb{R}^n$,A is a (nonlinear) differential operator and u is the unknown solution function.
We define $f := \partial_t u(t,x) + A[u(t,x)]$.

Now we approximate $u(t,x)$ with a neural network $NN1((t,x),\theta)$. This defines also a 'physics informed' neural network $NN2((t,x),\theta) := A[NN1((t,x),\theta)] + \partial_t NN1((t,x),\theta)$ for f through differentiating $NN1((t,x),\theta)$. NN1 and NN2 have the same parameters

which can be learned with the shared loss function

$$MSE = MSE_u + MSE_f \tag{12}$$
$$MSE_u = \frac{1}{|R|} \sum_{i=1}^{|R|} |u(t^i, x^i) - y^i|^2 \tag{13}$$
$$MSE_f = \frac{1}{|\partial R|} \sum_{i=1}^{|\partial R|} |f(\hat{t}^i, \hat{x}^i)|^2. \tag{14}$$

Here we require data points $(t^i, x^i, y^i) \in R$ and $(\hat{t}^i, \hat{x}^i) \in \partial R$ for f. MSE_u for the boundary conditions of 2.1 and MSE_f for f.

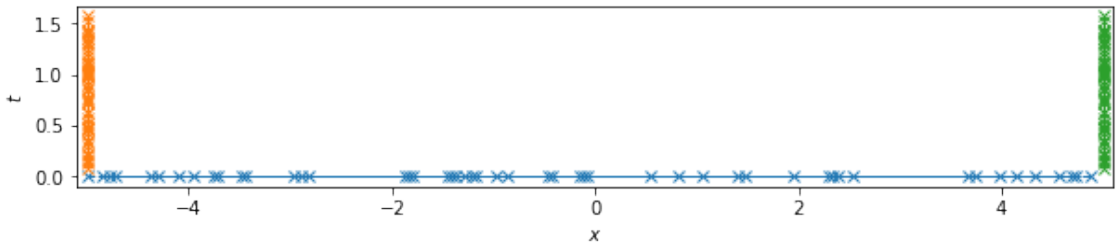

Figure 1: Training points on the boundary, the axes are rotateted

2.2 Continuous time PINN‘s - Shroedinger equation

In this section we apply the method to the **Shroedinger equation** following Raissi et al. (2019)[chapter 3.1,1][4] and the corresponding code[17] for this example.
The code presented is not complete and has been altered.
We have a Shroedinger equation of the form,

$$i * h_x + |h|^2 * h = 0 \tag{15}$$
$$h(0, x) = 2sech(x) \tag{16}$$
$$h(t, -5) = h(t, -5) \tag{17}$$
$$h_x(t, -5) = h_x(t, -5) \tag{18}$$

We start with a rectangle $\Omega = [-5, 5] \times [0, \pi/2]$ And 150 training points for boundary conditions **16,17,18**.

Additionally, 20 000 training points in the interior of Ω. Raissi has generated the training set with **Latin Hypercube Sampling strategy**. We start by defining a neural network with tanh activation function and output of dimension two corresponding to the real and the imaginary part of h.

In python, we can implement a neural network for instance with tensorflow keras [14].

In [1]:
```
nn = Sequential([
        Input(shape=(2,)),
        layers.Dense(100, activation='tanh'),
        layers.Dense(100, activation='tanh'),
        layers.Dense(100, activation='tanh'),
        layers.Dense(100, activation='tanh'),
        layers.Dense( 2)
       ])
```

Recall this is a nested construction.In practice one can consider building a function with arguments $([2, 100, 100, 100, 100, 2], tanh)$ instead.Since tanh has image [-1,1] we

leave out the last activation function,this is important for this equation.Since not otherwise mentioned the initial parameters are getting set to uniformly drawn samples from the normal distribution.
We have initialized the underlying neural network of h(t,x). We define helper function netuv where we compute the partial derivatives $\partial_x u, \partial_x v$ for 18.

We proceed with defining the corresponding network of $f = i * h_x + |h|^2 * h$. In python we can write

In [2]:
```
def h(x,t):
    uv =  nn(tf.concat([x,t],1))
    u = uv[:,0:1]
    v = uv[:,1:2]
    return u,v

def net_f_uv(x, t):
    with tf.GradientTape(persistent=True) as tape:
       tape.watch(x)
       tape.watch(t)

       u_x, v_x = netuv(x,t)

       u_t = tape.gradient(u, t)[0]
       u_2x = tape.gradient(u_x, x)[0]

       v_t = tape.gradient(v, t)[0]
       v_2x = tape.gradient(v_x, x)[0]

    f_u = u_t + 0.5*v_2x + (u**2 + v**2)*v
    f_v = v_t - 0.5*u_2x - (u**2 + v**2)*u

    return f_u, f_v
```

Here we write f_u and f_v for the real and imaginary part of f,
$f = -\partial_t v + 0.5\partial_x^2 u + (u^2 + v^2)u + i(\partial_t u + 0.5\partial_x^2 v + (u^2 + v^2)v)$. We define the loss function corresponding to the boundary conditions and f.

$$L_1 = \frac{1}{50}\sum_{i=1}^{50} |h(0, x^i) - y^i|^2 \tag{16}$$

$$L_2 = \frac{1}{100}\sum_{i=1}^{100} |h(t^i, -5) - h(t^i, 5)|^2 + |\partial_x h(t^i, -5) - \partial_x h(t^i, 5)|^2 \tag{17}$$

$$L_3 = \frac{1}{20000}\sum_{i=1}^{20000} |f(t_f^i, x_f^i)|^2 \tag{15}$$

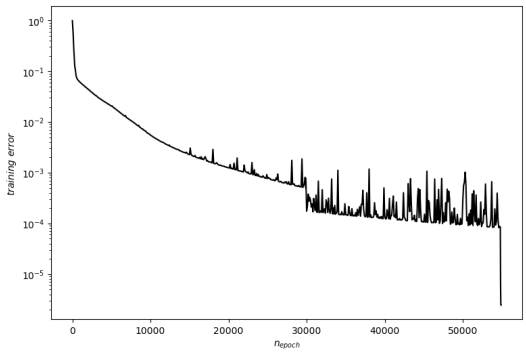

Figure 2: training loss: 55 000 Adam and 100 L-BFGS iterations

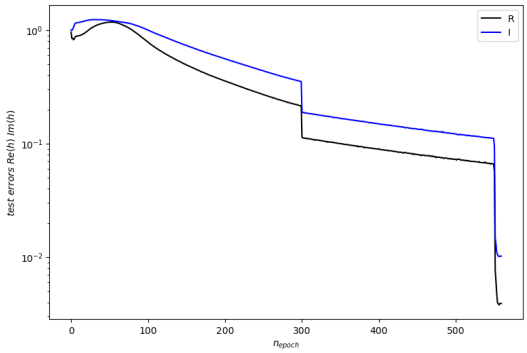

Figure 3: test loss: Real part and Imaginary part(blue)

And the shared loss function for the neural network accordingly $L = L_1 + L_2 + L_3$. At this point we can start to minimize L with respect to the shared parameters θ. A pseudocode would read:

In [3]:
```
for i in range(0,iterations):
        Loss = L_1(R) + L_2(boundary_R[0]) + L_3(boundary_R[1])
        gradients = compute_gradients(Loss,theta)
        optimizers.apply_gradients(gradients,theta)
```

Following the original implementation we train the network with **Adam** and then **L-BGFS**. Particular for 55 000 Adam and 100 L-BFGS iterations.The error we achieve measured in relative L_2 Norm against a numerically computed solution are,
$ErrorRe$: 0.00390891332227114
$ErrorIm$: 0.010224100025730693
.

Remark. Adam [9] is a momentum Optimizer and has changing parameters in the training process L-BGFS(limited memory BFGS method)[10] is a quasi Newton method

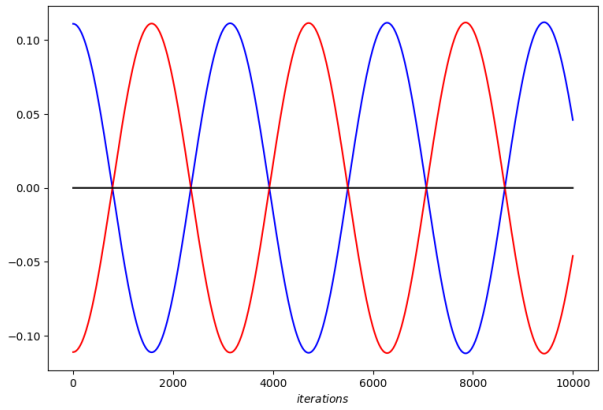

Figure 4: Oscillating losses for $L_1 = x * y = -L_2$

which utilizes Line search methods.
In practice L-BFGS has been seen to be omitted in PINN implementations. Also Levenberg-Marquardt (LM) algorithm has been remarked(Taylor (2022)). [15]

Road fork This is core algorithm and there are several ways to go from here addressing different issues of this algorithm. For instance Raissi claims that the number of necessary training points grows exponential with dimensions,leading to a bottleneck in higher dimensions.And goes on to propose **discrete time PINNs** (Raissi et al. (2019)[chapter 3.2][4]).
Wang utilizes more structure of the pde and proposes **Causal training for PINNs**Wang (2022)[16].
The path we will take in this work are **Competitive PINNs**(Zeng 2023).

3 Game theory for neural networks

3.1 Motivation

In chapter four we will consider a variant of PINNs with two networks with separate parameters. The connection is network 1 has the loss function L and the second network has loss function $-L$. An example is

$$L_1 := x * y \qquad\qquad L_2 := -L_1 \tag{19}$$

If we apply gradient descent to each component we see oscillating losses 3.1. This motivates us to consider game theory.

3.2 Background

In this chapter we follow Letcher(2019) chapter 2[13] For convenience we write $NN_i := Player_i$.

Definition 3.1 (Defintion 1 Letcher (2019))**.** [13] A differentiable game, is a set of NN's and corresponding continuously differentiable losses L_i, where $L_i : \mathbb{R}^k \to \mathbb{R}$ the parameters θ are $\theta = (\theta_1, ..., \theta_k) \in \mathbb{R}^n$ w_i corresponds to NN_i.

The **simultaneous gradient** is given by $\psi(\theta) := (\nabla_{\theta_1} L_1, ..., \nabla_{\theta_n} L_n)$.
The **Jacobian matrix** of a game can be written in terms of ψ:

$$J(w) = \nabla\psi(\theta)^T \tag{20}$$

or in matrix notation:
$J(w) = \begin{bmatrix} \nabla_{\theta_1}\nabla_{\theta_1} L_1 & \nabla_{\theta_1}\nabla_{\theta_2} L_1 \dots \nabla_{\theta_1}\nabla_{\theta_n} L_1 \\ \nabla_{\theta_2}\nabla_{\theta_1} L_2 & \nabla_{\theta_2}\nabla_{\theta_2} L_2 \dots \nabla_{\theta_2}\nabla_{\theta_n} L_2 \\ \vdots & \ddots \\ \nabla_{\theta_n}\nabla_{\theta_1} L_n & \nabla_{\theta_n}\nabla_{\theta_2} L_n \dots \nabla_{\theta_n}\nabla_{\theta_n} L_n \end{bmatrix}$. We can decompose the **Jacobian Matrix** in a symmetric and antisymmetric component due to the following lemma.

Lemma 3.2. *For a quadratic Matrix M we can write $M = R + A$, where R is **symmetric** and A is **antisymmetric**.*

Proof. Define
$R := \frac{1}{2}(M + M^T)$
$A := \frac{1}{2}(M - M^T)$,
then $R^T = \frac{1}{2}(M + M^T) = \frac{1}{2}(M + M^T)^T = -S^T$
and $A^T = \frac{1}{2}(M - M^T) = -\frac{1}{2}(M - M^T)^T = -A^T$.
Further we have J = R + A. □

Definition 3.3. (Hamiltonian and potential game's) Let J be the Jacobian, we say a game is **Hamiltonian** if $J = A$ (J is antisymmetric), and **potential** if $J = R$ (J is symmetric).

The general case is that $A \neq 0$ and $R \neq 0$.

Example. For the initial zero-sum problem $L_1 = x * y^T = -L_2$, we compute the Jacobian,
$J(w) = \begin{bmatrix} \partial_{\theta_1}\partial_{\theta_1} L_1 & \partial_{\theta_1}\partial_{\theta_2} L_1 \\ \partial_{\theta_2}\partial_{\theta_1} L_2 & \partial_{\theta_2}\partial_{\theta_2} L_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ since $J = -J^T$ the game is Hamiltonian function.
More generally for a zero-sum problem with Loss L we compute,
$J = \begin{bmatrix} \partial_1\partial_1 L & \partial_1\partial_2 L \\ \partial_2\partial_1 - L & \partial_2\partial_2 - L \end{bmatrix}$,
$R = \begin{bmatrix} \partial_1\partial_1 L & 0 \\ 0 & \partial_2\partial_2 - L \end{bmatrix}$ and $A = \begin{bmatrix} 0 & \partial_1\partial_2 L \\ \partial_2\partial_1 - L & 0 \end{bmatrix}$.

Even if Hamiltonian and zero-sum games have none empty intersection. Neither of them is a subset of another. We need a solution concept. There are two solution concepts

which do in general not coincide: Writing $\hat{w_i}$ for the parameters where the coordinate w_i is omitted.

1.

   Definition 3.4 (Defintion 3 Letcher(2019))**.** [13](local Nash equilibrium) A point w is a **local Nash equilibrium** if $\forall\ i\ \exists$ *a neighbourhood* U_i :
   $L_i(w_i, \hat{w_i}) \ \leq L_i(u_i, \hat{w_i})\ \forall\ u_i\ \in U_i$.
   (componentwise minimum)

2.

   Definition 3.5 (Defintion 4 Letcher (2019))**.** [13](Stable critical point) A critical point $\psi(w) = 0$ is **stable** if $J(w) \succeq 0$ ($J(w)$ is positive semidefinite).

If we find a sufficient condition for gradient descent to converge to a critical point we have a result for stability. We can consider the following result.

Lemma 3.6 (Proposition 12 Lee(2016))**.** *[8] Let x_k be the sequence generated by gradient descent. Suppose f is continuously differentiable with isolated critical points and compact sublevel sets then lim x_k exists and converges to a critical point.*

A function is **coercive** if

$$\lim_{\|w\|\to\infty} H(w) = \infty.$$

Coerciveness implies compact sublevel sets by Heine Borel. The next Theorem tells us that in this case Hamiltonian game's can be solved with gradient descent in the sense of stable critical points 3.5.

Satz 3.7 (Theorem 4)**.** *Let $H(w) := \frac{1}{2}\|\psi(w)\|_2^2$,$w_k$ the sequence of gradient descent for H, then*

(i) $\nabla H = A^T\psi(w)$

(ii) If the jacobian J is invertible and lim w_k exists and converges to a critical point, then H converges to a stable critical point.

Proof. Since we are in a Hamiltonian game,
$A^T\psi = (A + S)^T\psi = J^T\psi$.

We sketch the remaining calculation in the case n=2,$\theta = (\xi, \eta)$:

$$\begin{aligned}\frac{1}{2}\nabla_\theta((\nabla_\xi L_1)^2 + (\nabla_\nu L_2)^2) &= \frac{1}{2}\begin{bmatrix}2(\nabla_\xi\nabla_\xi L_1)(\nabla_\xi L_1) + 2(\nabla_\xi\nabla_\nu L_2)(\nabla_\nu L_2)\\ 2(\nabla_\nu\nabla_\nu L_2)(\nabla_\nu L_2) + 2(\nabla_\nu\nabla_\xi L_2)(\nabla_\xi L_1)\end{bmatrix}\\ &= \begin{bmatrix}(\nabla_\xi\nabla_\xi L_1) & (\nabla_\xi\nabla_\nu L_2)\\ (\nabla_\nu\nabla_\xi L_2) & (\nabla_\nu\nabla_\nu L_2)\end{bmatrix}\begin{bmatrix}(\nabla_\xi L_1)\\ (\nabla_\nu L_2)\end{bmatrix}.\end{aligned}$$

For (ii), the previous lemma 3.6 claims gradient descent converges to a critical point of ∇H. From (i) we have $\nabla H = J^T\psi(W)$, since J is invertible we conclude $\psi(w)$ is zero. Further $S = 0$ implies $u^T Ju = u^T(0 + A)u = 0$(A is antisymmetric) that is positive semidefinitness of J and hence the critical point is stable. □

Definition 3.8. We say $H(w) = \frac{1}{2}\|\psi(w)\|_2^2$ is the **Hamiltonian**.

Continuing with example $L_1 = xy$ the Hamiltonian function $\frac{1}{2}(x^2 + y^2)$ is coercive hence we find a stable critical point with gradient descent for H, namely (0,0), this happens to be a local Nash equilibrium.

3.3 Optimizers

In this subchapter we outline guiding ideas of three optimizers. The first two adjust the gradient and the third optimizer focuses on Nash equilibria.The corresponding algorithms can be found in the appendix 5.1.

3.3.1 Consensus

(Chapter 4.1 Mescheder 2017)[5]
From theorem 3.7 we know that gradient descent on the Hamiltonian function $\frac{1}{2}\|\psi(w)\|^2$ will convergence to a stable critical point in the coercive case. For the general Hamiltonian function one can consider a perturbed version of the vector field ψ. Consensus optimization [5] adjusts the vector field ψ to $w := \psi + \lambda * J^T\psi := \psi + \lambda * \nabla H$. And apply a optimizer to the components of w. Second perspective w is the associated gradient of:
$\hat{L_1}(\theta_1, \theta_2) = L_1(\theta_1, \theta_2) - \lambda H$
$\hat{L_2}(\theta_1, \theta_2) = L_2(\theta_1, \theta_2) - \lambda H$.

3.3.2 SGA

SGA builds on Consensus optimization. To generalize Consensus from Hamiltonian games to general games SGA replaces J with the antisymmetric component A. We define the **Symplectic Gradient Adjustment(SGA)** $\psi_\lambda := \psi + \lambda * A^T\psi$ Further we

can adjust the sign of λ with the following procedure to find stable critical points. We write $\theta(v, w)$ for the angle of (v,w).

Definition 3.9. Let $\psi_\lambda := v + \lambda w$ with $w, v \in \mathbb{R}^n$ the alignment is given by $align(\psi_{lambda}, w) = \frac{d}{d\lambda}(cos(\theta(\psi_\lambda, w)) \mid_{\lambda=0})$.

Lemma 3.10. *Let ψ_λ be the SGA and H the Hamiltonian function, then*
$sign(align(\psi_\lambda, \nabla H)) = sign(\langle\psi, \nabla H\rangle\langle A^T\psi, \nabla H\rangle)$.

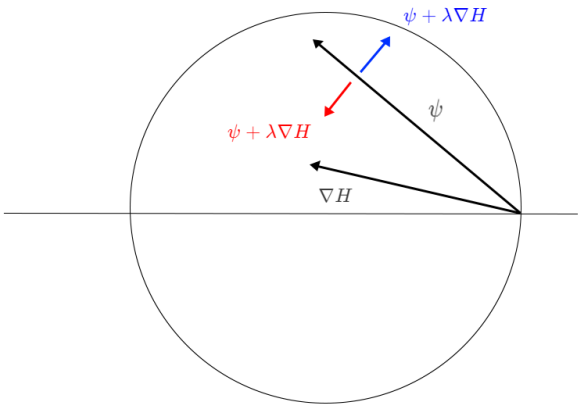

Figure 5: If the angle $\theta(\psi, \nabla H) < 90\,\text{deg}$: In the red case the alignment is positive since ' $\psi_\lambda$points more towards ∇H then ψ' In the blue case the alignment is negative.

Proof. We calculate,

$$cos^2(\theta(\psi_\lambda, w)) = (\frac{\langle \psi_\lambda, \nabla H \rangle}{||\psi_\lambda||\ ||\nabla H||})^2 = \frac{(\psi \nabla H)^2 + 2(\lambda * A^T \psi \nabla H)^T (\psi \nabla H) + (\lambda * A^T \psi \nabla H)^2}{((\psi)^2 + 2(\lambda * A^T \psi)^T (\psi) + (\lambda * A^T \psi)^2)||\nabla H||^2}$$
$$= \frac{(\psi \nabla H)^2 + 2(\lambda * A^T \psi \nabla H)^T (\psi \nabla H) + (\lambda * A^T \psi \nabla H)^2}{((\psi)^2 + (\lambda * A^T \psi)^2)||\nabla H||^2}$$

where the last equality is due to antisymmetry of A. taking derivatives and setting $\lambda = 0$ leaves us with 3.10. □

1. Case.1 We are in a neighborhood of stable critical point.($S \succeq 0$)
   In this case SGA is trying to achieve $\theta(\psi_\lambda, \nabla H) \leq \theta(\psi, \nabla H)$. Since gradient descent on the Hamiltonian function converges to critical points, we try to decrease the angle of ψ and ∇H with the adjustment. For this we calculate, $\langle \psi, \nabla H \rangle = \langle \psi, J^T \psi \rangle = \langle \psi, S^T \psi \rangle \geq 0$, this implies

   $$\langle \psi, \nabla H \rangle \geq 0 \tag{21}$$

   .

   Since 21 the angle $\theta(\psi, \nabla H)$ is positive. The alignment is positive if ψ_λ points more towards ∇H then ψ negative else. To go further from here we apply lemma 3.10

   $$sign(align(\psi_\lambda, \nabla H)) = sign(\langle \psi, \nabla H \rangle \langle A^T \psi, \nabla H \rangle).$$

By 21 this further simplifies to
$sign(+\langle A^T\psi, \nabla H\rangle)$.
Hence

$$\lambda\langle\psi, \nabla H\rangle\langle A^T\psi, \nabla H\rangle > 0 \tag{22}$$

implies $sign(\lambda) = sign(align(...))$, this also covers the case of a instable critical point.

2. Case.2 We are in a neighborhood of instable critical point. In this case SGA is trying to achieve $\theta(\psi_\lambda, \nabla H) \geq \theta(\psi, \nabla H)$ Since 21 the angle $\theta(\psi, \nabla H)$ is negative. In this case the alignment is positive if ψ_λ points less towards ∇H then ψ and negative else.Again working with 22 yields $\theta(\psi_\lambda, \nabla H) \geq \theta(\psi, \nabla H)$. The aligned

   adjustment now reads

$$\psi_\lambda := \psi + sign(\langle\psi, \nabla H\rangle\langle A^T\psi, \nabla H\rangle) * \lambda * A^T\psi \qquad \lambda > 0 \tag{23}$$

   The additional floating point operations for the sign is considerable.

3.3.3 CGD

Schäfer (2019) chapter 2[6]
Competitive Gradient descent does not adjust ψ. A variant of CGD namely A-CGD[11] is implemented in the [6]. Competitive Gradient descent is defined for two NNs. We write $L_1 = f$ and $L_2 = g$. At each step k we consider a bilinear approximation of the game given by,

$$f(\xi_k) + \nabla_\xi f(\xi - \xi_k) + (\xi - \xi_k)^T\nabla_\xi\nabla_\eta f(\eta - \eta_k) + \eta^T\nabla_\eta f(\eta - \eta_k) + \frac{1}{\lambda}(\xi - \xi_k)^T(\xi - \xi_k) \tag{24}$$

$$g(\eta_k) + \nabla_\eta g(\eta - \eta_k) + (\eta - \eta_k)^T\nabla_\eta\nabla_\xi g(\xi - \xi_k) + \xi^T\nabla_\xi g(\xi - \xi_k) + \frac{1}{\lambda}(\eta - \eta_k)^T(\eta - \eta_k). \tag{25}$$

Minimizing 24 with respect to ξ and 25 with respect to η this simplifies to
$\nabla_\xi f(\xi - \xi_k) + (\xi - \xi_k)^T\nabla_\xi\nabla_\eta f(\eta - \eta_k) + \frac{1}{\lambda}(\xi - \xi_k)^T(\xi - \xi_k)$
$\nabla_\eta g(\eta - \eta_k) + (\eta - \eta_k)^T\nabla_\eta\nabla_\eta g(\eta - \eta_k) + \frac{1}{\lambda}(\eta - \eta_k)^T(\eta - \eta_k)$.

The Nash equilibrium can be computed in closed form resulting in
$$\begin{bmatrix}\xi\\ \nu\end{bmatrix} = \begin{bmatrix}\xi_k\\ \nu_k\end{bmatrix} - \lambda * \begin{bmatrix}Id & \lambda * \nabla_\xi\nabla_\nu f\\ \lambda * \nabla_\nu\nabla_\xi g & Id\end{bmatrix}^{-1}\begin{bmatrix}\nabla_\xi f\\ \nabla_\nu g\end{bmatrix}.$$

.

4 Competitive PINNs

4.1 CPINN formulation

In this section we follow (Zeng (2023)[7] chapter 2.1–2.3)[7] Starting from a similar problem 2.1.

$$A[u(x)] = h \qquad z \in \Omega \tag{26}$$

$$u(x) = g(z) \qquad z \in \partial\Omega \tag{27}$$

We approximate u with a neural network $P(x, \theta)$ Indicated in chapter 3 we introduce a second neural network $Z(x, \psi)$ with distinct parameters ψ and a shared loss function L leading us to $min_\theta\ max_\psi\ L(\theta, \psi)$.

Concretely L reads,

$$L(\theta, \psi) := L_{\partial\Omega}(\theta, \psi) + L_\Omega(\theta, \psi). \tag{28}$$

$$L_\Omega(\theta, \psi) := \frac{1}{|R|} \sum_{(\hat{x}^i) \in R} Z(\hat{x}^i)(A[P(\hat{x}^i)]) \tag{29}$$

$$L_{\partial\Omega}(\theta, \psi) := \frac{1}{|\partial R|} \sum_{(x^i) \in \partial R} Z(x^i)(P(x^i) - g(x^i)) \tag{30}$$

Here R and ∂R are the training sets. $L_{\partial\Omega}$ is the loss for the boundary conditions of 26 and L_Ω for h. We can write 4.1 in the form $min_\theta L(\theta, \psi)\ min_\psi -L(\theta, \psi)$ and notice we are in a zero-sum game. According to Zeng (2023)[7] the Nash equilibrium is $(P, Z) = (u, 0)$ where u is the latent solution. From a machine learning perspective this is a min max gan. And Z is the discriminator network.

The optimizer we apply, following the original paper, is adaptive competitive gradient G - ACGD(Schäfer et al. [11]). A variant of CGD where the step sizes are chosen similar to Adam and GMRE(generalized minimal residual algorithm)[**?**] is applied. One ACGD iteration amounts to $2 + 2*$(number of GMRE iterations) forward passes.

4.2 Avoid square condition

The argument given by (Zeng (2023)[7] chapter 2.3) addresses the square in the Mean squared error residual loss.

Lemma 4.1. *Given the above setup, for functions ψ_i, ϕ_i and vectors w, δ in $\mathbb{R}^N$,$\mathbb{R}^M$ let*

$$P := \sum_{i=1}^{N} w_i \psi_i(x) \tag{31}$$

$$Z := \sum_{i=1}^{M} \delta_i \phi_i(x) \tag{32}$$

and suppose A is a linear pde operator, then solving 26 with MSE amounts to a condition number $\kappa(A^2)$, solving 26 with min max4.1 amounts to $\kappa(A)$.

Proof. (Zeng (2023) chapter 2.3[7]) Linearity of A implies $A[P] = \sum_{i=1}^{N} w_i * A[\psi_i(x)]$, hence at training point y_j $\sum_{i=1}^{N} w_i * A[\psi_i(y_j)] = f(y_j)$. This defines a matrix E for training samples $y_1, ..., y_N$,$E_{ij} := A[\psi_i(y_j)]$. MSE applied to $Ew = f$ corresponds to an error function $min_w ||Ew - f||_2^2$ this is a least square problem with solution $w = (E^T E)^1 E^T f$ and condition number $\kappa(E^2)$. Rewriting the problem $min_w\ max_\delta\ \delta(Ew-f)$ or equivalently,

$$\begin{bmatrix} 0 & E^T \\ E & 0 \end{bmatrix} \begin{bmatrix} w \\ \delta \end{bmatrix} = \begin{bmatrix} 0 \\ f \end{bmatrix} \tag{33}$$

The linear system results for instance by computing the respective derivates.
We have $\kappa\left(\begin{bmatrix} 0 & E^T \\ E & 0 \end{bmatrix}\right) = k(E)$. □

Can this in a sense generalize to non-linear pde‘s?

4.3 Poisson equation

In this chapter we test the CPINN formulation with example given in the same paper (Zeng (2023)[7] 3.2).

$$\nabla u = -2sin(t)cos(x), \qquad (t, x) \in [-2, 2] \times [-2, 2] \tag{34}$$

$$u(t, -2) = sin(t)cos(-2) \quad u(-2, x) = sin(-2)cos(x) \qquad t \in [-2, 2], \tag{35}$$

$$u(t, 2) = sin(t)cos(2) \quad u(2, x) = sin(2)cos(x) \qquad x \in [-2, 2]. \tag{36}$$

We notice the A is a linear operator. The losses read

$$L_\Omega(\theta, \psi) = \frac{1}{5000} \sum_{(\hat{t},\hat{x})\in R} Z(\hat{t}, \hat{x})((\partial_x)^2 P(\hat{t}, \hat{x}) + (\partial_y)^2 P(\hat{t}, \hat{x}) + 2sin(\hat{t})cos(\hat{x}))$$

$$L_{\partial\Omega}(\theta, \psi) = \frac{1}{150} \sum_{(t,x)\in\partial R} Z(t, x)(P(t, x) - g(t, x)).$$

This means our Z network has last dimension 2. The PINN network for this example has dimensions
$[2, 50, 50, 50, 1]$ and tanh activation function, the network Z has dimensions $[2, 100, 100, 100, 2]$ and ReLu activation functions.

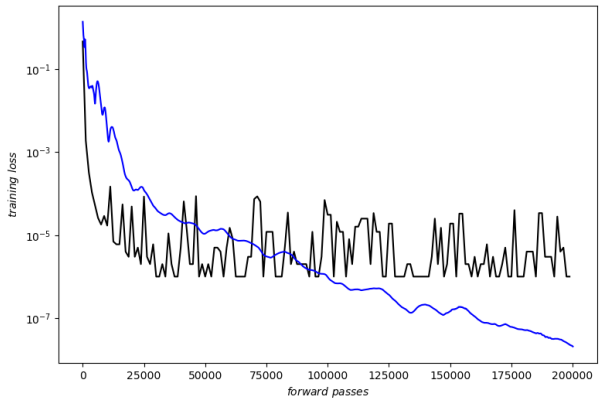

Figure 6: PINN with Adam and CPINN with Gmre-ACGD (CPINN is blue)

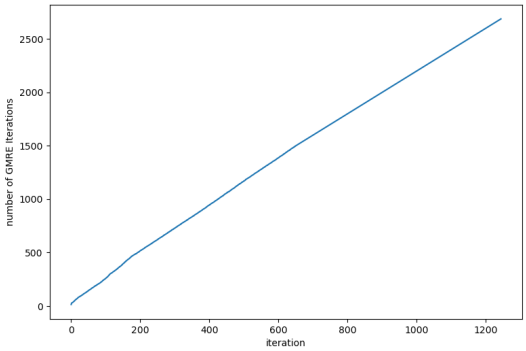

Figure 7: Gmre iterations per ACGD iteration

Further a PINN was implemented according to chapter 2 (adapting Blechschmidt[17]) with 20 000 Adam iterations. To make this somehow comparable we have trained the

CPINN and the PINN for the same number of forward passes and with the same training sets.Forwarded passes were calculated according to 2 + 2∗(number of GMRE iterations). This results in 1235 ACGD iterations. Let u(x,y) be the calculated solution $-2 * np.sin(x) * np.cos(y)$.
The resulting **test** losses read
PINN with Adam : $\|PINN(x,y) - u(x,y)\|_2 = 0.007793986723930779$
CPINN with G-ACGD: $\|PINNGACGD(x,y) - u(x,y)\|_2 = 1.672846e-05$.
It can be noted that 20 000 forward passes is not the end of the training progress.

Further we plotted the training progress 4.3 and number of GMRE iterations per GACGD 4.3.

5 Appendix

5.1 Algorithms

These algorithms can be found in Mescheder(2017)[5],Letcher(2019)[2],Schaefer(2019)[6]

Consensus: while not converged:
$v_{\theta_1} = \nabla_{\theta_1}(\nabla L_1(\theta_1, \theta_2) - \lambda H)$
$v_{\theta_2} = \nabla_{\theta_2}(\nabla L_2(\theta_1, \theta_2) - \lambda H)$
$\theta_1 = \theta_1 + \lambda v_{\theta_1}$
$\theta_2 = \theta_2 + \lambda v_{\theta_2}$

SGA: //losses
$L = l_1, ..., l_n$
//parameters
$W = w_1, ..., w_n$

ψ = gradient(l_i, w_i) for $(l_i, w_i) \ \in (L, W)$
$A^T\psi$ = getsymadj(L,W) //calculation of jacobian is required

if align == True:
∇H = gradient$(\frac{1}{2}||\xi||$,w) for $w \in W$
$\lambda = sign(\frac{1}{n}\langle\psi, \nabla H\rangle\langle A^T\psi, \nabla H\rangle + \epsilon)$ //$\epsilon = 1/10$
else
$\lambda = 1$
endif
Output: $\psi_\lambda := \psi + \lambda * A^T\psi$

CGD: $\theta = (\xi, \eta)$
$L = (f, g)$
$\xi_{k+1} = \xi_k - \lambda(Id - \lambda^2 * \xi^T\nabla_\xi\nabla_\eta f * \nabla_\eta\nabla_\xi g)^{-1}(\nabla_\xi f - \lambda * \nabla_\xi\nabla_\eta f * \nabla_\eta g)$
$\eta_{k+1} = \eta_k - \lambda(Id - \lambda^2 * \eta^T\nabla_\eta\nabla_\xi g * \nabla_\xi\nabla_\eta f)^{-1}(\nabla_\eta g - \lambda * \nabla_\eta\nabla_\xi g * \nabla_\xi f)$

5.2 Acknowledgments

List of people and things this work owes thanks to: parents, supervisor,people from the literature, university of Vienna, laptop.

References

[1] Benoit Liquet , Sarat Moka, Yoni Nazarathy. Mathematical Engineering of Deep Learning,CRC Press, ,2024

[2] Alistair Letcher, David Balduzzi, Sébastien Racaniere, James Martens, Jakob Foerster, Karl Tuyls, and Thore Graepel. Differentiable game mechanics. The Journal of Machine Learning Research, 20(1):3032–3071, 2019

[3] Bruno Després Neural Networks and Numerical Analysis DeGruyterSeries in Applied and Numerical Mathematics ISBN 978-3-11-078312-4, 2020

[4] Maziar Raissi, Paris Perdikaris, and George Em 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. ISSN 0021-9991.

[5] Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. The Numerics of GANs. NeurIPS. 2017. Chapter 4

[6] Florian Schäfer and Anima Anandkumar. Competitive gradient descent. In NeurIPS, 2019 See als: https://f-t-s.github.io/projects/cgd/

[7] Qi Zeng, Yash Kothari, Spencer H. Bryngelson and Florian Schäfer Competitive physics informed networks arXiv:2204.11144v2 2022 Further references

[8] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient Descent Converges to Minimizers. COLT, 2016. Optimizers mentioned:

[9] D. Kingma, J. Ba, Adam: a method for stochastic optimization, 2014, arXiv:1412 .6980.

[10] D.C. Liu, J. Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (1989) 503–528.

[11] Florian Schäfer and Anima Anandkumar. Competitive gradient descent. In NeurIPS, 2019.

[12] Youcef Saad and Martin H Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 7(3):856–869, 1986.

[13] Terven, Juan and Cordova-Esparza, Diana-Margarita and Ramirez-Pedraza, Alfonzo and Chávez Urbiola, Edgar. Loss Functions and Metrics in Deep Learning. A Review. 10.48550/arXiv.2307.02694. (2023)

[14] François Chollet. Deep Learning with Python. Manning Publications Co. ISBN 9781617294433, 2018.

[15] John Taylor and Wenyi Wang and Biswajit Bala and Tomasz Bednarz. Optimizing the optimizer for data driven deep neural networks and physics informed neural networks, arxiv:2205.07430, 2022.

[16] Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality is all you need for training physics-informed neural networks. arXiv:2203.07404, 2022a

[17] Poisson PINN:Jan Blechschmidt under https://github.com/janblechschmidt/PDEsByNNs/ (Tensorflow) (MIT license)
Poisson CPINN: Zeng under https://github.com/comp-physics/CPINN (Pytorch)(MIT license)
Shroedinger PINN : Raissi https://github.com/maziarraissi/PINNs/tree/master/main (Tensorflow) (MIT license)
Shroedinger PINN: Zeng under https://github.com/comp-physics/CPINN (Pytorch)(MIT license)
L-BFGS:Pi-Yueh Chuang https://gist.github.com/piyueh/712ec7d4540489aad2dcfb80f9a54993 (Tensorflow) (MIT license)
GMRES-based ACGD: https://github.com/devzhk/ (Pytorch)(MIT license)
github : https://github.com/josefvie

Notebooks were running on kaggle (https://www.kaggle.com)