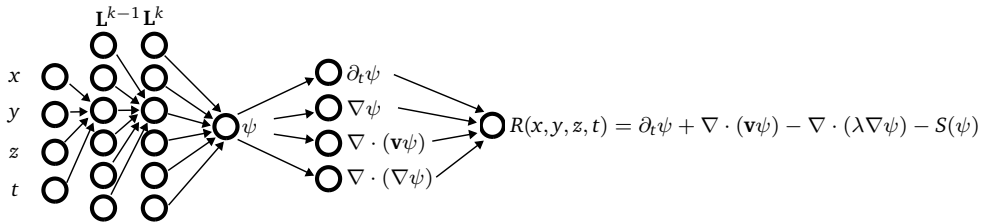
Towards physics-based deep learning in OpenFOAM: Combining OpenFOAM with the PyTorch C++ API

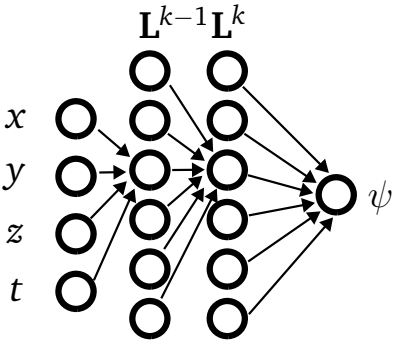
Tomislav Maric (TU Darmstadt), Andre Weiner (TU Braunschweig)

17th OpenFOAM Workshop, 11.07.2022, Cambridge University

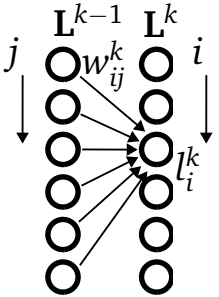
TECHNISCHE UNIVERSITÄT DARMSTADT



- Deep Learning Overview
- Physics-Based Deep Learning Overview
- Combining PyTorch C++ API and OpenFOAM for Physics-Informed Neural Networks



- **Neural Network (NN)** has an input layer (e.g. (x, y, z, t)), $D - 1$ hidden layers, and a (e.g. scalar) output layer ψ.



Hidden layer $\mathbf{L}^k$ is computed from the previous layer

$$l_i^k = \sum_{j=1}^{N_{\mathbf{L}^{k-1}}} w_{ij}^k l_j^{k-1} + b_i^k \tag{1}$$

Einstein's notation (repeated index $\equiv$ dot product)

$$l_i^k = w_{ij}^k l_j^{k-1} + b_i^k \tag{2}$$

Matrix-vector product

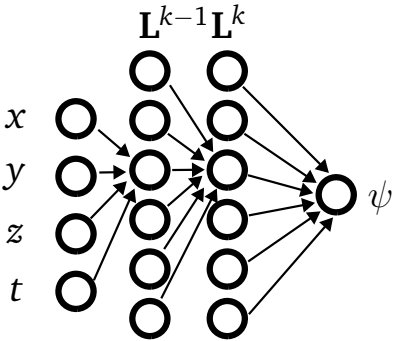
$$\mathbf{L}^k = \mathbf{w}^k \cdot \mathbf{L}^{k-1} + \mathbf{b}^k \tag{3}$$

$\mathbf{w}^k$ is a $N_{\mathbf{L}^k} \times N_{\mathbf{L}^{k-1}}$ matrix.

Adding activation functions to the layers results in the final NN, as a composition of functions

$$\begin{aligned}\psi_\theta^{nn}(\mathbf{u}) &= \mathbf{w}^D \cdot L^{D-1} + b^D = \mathbf{w}^D \cdot \sigma(\mathbf{w}^{D-1} \cdot L^{D-2} + \mathbf{b}^{D-1}) + b^D \\ &= \mathbf{w}^D \cdot \sigma(\mathbf{w}^{D-1} \cdot \sigma(\mathbf{w}^{D-2} \cdot \sigma(\dots \sigma(\mathbf{w}^1 \cdot \mathbf{u} + \mathbf{b}^1)\dots) + \mathbf{b}^{D-2}) + \mathbf{b}^{D-1}) + b^D\end{aligned} \tag{4}$$

- $\mathbf{u} = (x, y, z, t)$ in our example.
- θ are all the weights and biases, $\theta = \{w_{ij}^k, b_i^k\}, k \in [1, D], j \in [1, N_{\mathbf{L}^{k-1}}], i \in [1, N_{\mathbf{L}^k}]$.
- When approximating functions, the last layer is often "linear".

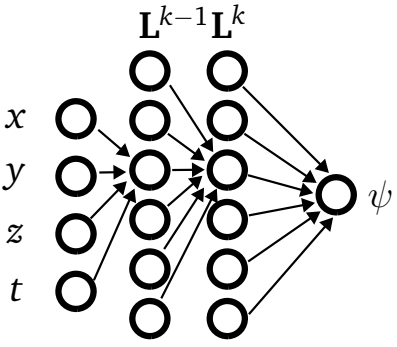


A set of points $\{\mathbf{u}_p\}_{p\in P}$ and their data $\{\psi_p\}_{p\in P}$ can be used to define an error of $\psi_\theta^{nn}(\mathbf{u}_p)$, e.g.

$$e_{MSE}(\theta) = \frac{1}{N_p}\sum_{p=1}^{N_p}(\psi_\theta^{nn}(\mathbf{u}_p) - \psi_p)^2. \tag{5}$$

The network "learns" some θ_M that minimize e_{MSE}.

$$\theta_M = \arg\min_\theta e_{MSE}(\theta) \tag{6}$$



- $\theta_M = \arg\min_\theta e_{MSE}(\theta)$ requires $\min_\theta e_{MSE}(\theta)$

$$\partial_{\theta_i} e_{MSE}(\theta) \rightarrow 0$$

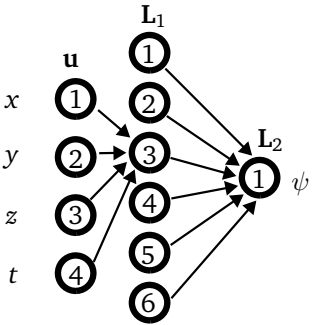
$$\sum_{p=1}^{N_p} (\psi^{nn}_{\theta^m}(\mathbf{u}_p) - \psi_p)\partial_{\theta^m_i}\psi^{nn}_{\theta^m}(\mathbf{u}_p) \stackrel{!}{=} 0 \tag{7}$$

- An approximation is inexact so generally $\psi^{nn}_{\theta^m}(\mathbf{u}_p) - \psi_p \neq 0$, and we strive for $\partial_{\theta^m_i}\psi^{nn}_{\theta^m}(\mathbf{u}_p) \rightarrow 0$.
- Why m? In the beginning $\theta^{m=1}$ is somehow (randomly) initialized and we iteratively (m) improve it to satisfy eq. (7).

TECHNISCHE UNIVERSITÄT DARMSTADT

$$\sum_{p=1}^{N_p} (\psi^{nn}_{\theta^m}(\mathbf{u}_p) - \psi_p) \partial_{\theta^m_i} \psi^{nn}_{\theta^m}(\mathbf{u}_p) \overset{!}{=} 0$$

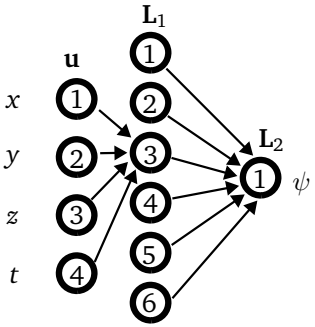
- **Imagine we somehow know** $\partial_{\theta^m_i} \psi^{nn}_{\theta^m}(\mathbf{u}_p)$ ($\nabla_{\theta^m} \psi^{nn}_{\theta^m}(\mathbf{u}_p)$ in vector notation), we can then use gradient descent
$$\theta^{m+1}(\mathbf{u}_p) = \theta^m(\mathbf{u}_p) - \lambda^m \nabla_{\theta^m} \psi^{nn}_{\theta^m}(\mathbf{u}_p) \tag{8}$$
ensuring $e_{MSE}(\theta^{m+1}) \leq e_{MSE}(\theta^m)$ (hopefully $\|\nabla_{\theta^{m+1}} \psi^{nn}_{\theta^m}(\mathbf{u}_p)\|_2 \leq \|\nabla_{\theta^m} \psi^{nn}_{\theta^m}(\mathbf{u}_p)\|_2$), and adapting λ at m to tune the step size.
- New parameters are set as $\frac{1}{N_p} \sum_{p=1}^{N_p} \theta^{m+1}(\mathbf{u}_p)$ (average), or by batch-average, or using radom-subsets of data points.
- Real-world algorithms adapt λ^m differently and are more complex.



If

$$\psi_\theta^{nn}(\mathbf{u}_p) = \mathbf{w}^2 \cdot \sigma(\mathbf{w}^1 \cdot \mathbf{u} + \mathbf{b}^1) + b^2$$

what is $\partial_{w_{13}^1} \psi_\theta^{nn}(\mathbf{u}_p)$?



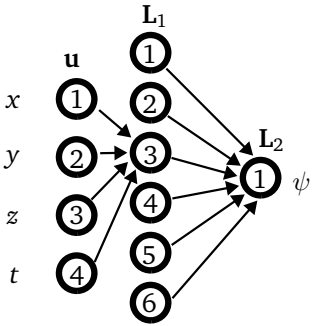
If

$$\psi_\theta^{nn}(\mathbf{u}_p) = \mathbf{w}^2 \cdot \sigma(\mathbf{w}^1 \cdot \mathbf{u} + \mathbf{b}^1) + b^2$$

what is $\partial_{w_{13}^1} \psi_\theta^{nn}(\mathbf{u}_p)$?

$$\partial_{w_{13}^1} \psi_\theta^{nn}(\mathbf{u}_p) = \mathbf{w}^2 \cdot \sigma'(\mathbf{w}^1 \cdot \mathbf{u} + \mathbf{b}^1) u_3$$

- Imagine writing this down for every w_{ij}^k, b_i^k for a deep NN.
- To make things worse, the number of layers and nodes change during "hyperparameter tuning".

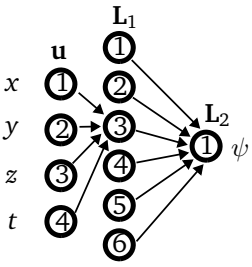


Finite Differences (FD) generalize to arbitrary NN architectures, but don't work, because of **computational costs**.

- $\mathbf{w}^k$ is a $N_{\mathbf{L}^k} \times N_{\mathbf{L}^{k-1}}$ matrix.
- $\mathbf{b}^k$ is a $N_{\mathbf{L}^k}$ vector.
- For $\partial_{\theta_i^m} \psi^{nn}_{\theta^m}(\mathbf{u}_p)$, we need

$$N_\theta := \sum_{i=2}^{D} N_{\mathbf{L}^k}(1 + N_{\mathbf{L}^{k-1}}) \tag{9}$$

finite differences, one for each weight and bias.



Finite Differences (FD) generalize to arbitrary NN architectures, but don't work, because of **Floating-Point (FP) *cancellation* errors**.

- As we converge towards θ^M, forward passes get **very close to each other** $|\psi^{nn}_{\theta^m}(\mathbf{u}_p) - \psi^{nn}_{\theta^{m-1}}(\mathbf{u}_p)| \rightarrow 0$.
- The computer has limited precision, so as weights $\theta^m_i, \theta^{m-1}_i$ get close to each other

```
theta_i_m                = 1.1234567891234569|33333333333
theta_i_m_1              = 1.1234567891234102|22222222222
```

their difference

```
theta_i_m - theta_i_m_1 = 0.0000000000000467|11111111111
```

and, therefore, $|\partial^{FD}_{\theta^m_i}\psi^{nn}_{\theta^m}(\mathbf{u}_p)|$, go to zero quicker than they should, so **NN training stalls**.

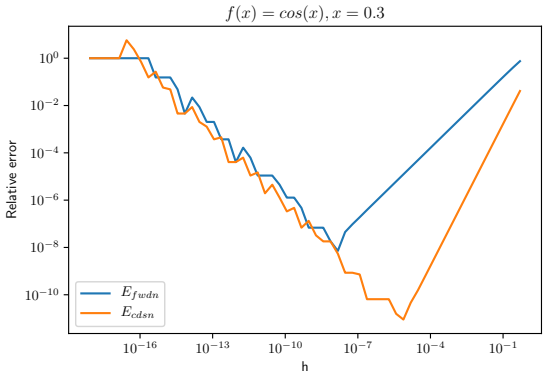
Some Finite Differences

$$f'_{fwd} = \frac{f(x+h) - f(x)}{h} + O(h) \tag{10}$$

$$f'_{cds} = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{11}$$

Important: Finite Differences are **inexact**, order-of accuracy $O(h^p)$.
A great book on FPA is Overton [2001]

- Once $h <= 0.5ulp(x)$ with nearest rounding, full cancellation occurs.
- ulp - units in the last place, $ulp(x) = 2^{-52}2^E$, E is the exponent.



Relative derivative (gradient) error

$$e_n^{fwd,cds} = \frac{|f'_{fwd,cds} - f'_{exact}|}{|f'_{exact}|}. \qquad (12)$$

$h \to 0$, $e_n^{fwd,cds} \to 1$: Floating-Point cancellation errors prevent asymptotic convergence of Finite Differences.

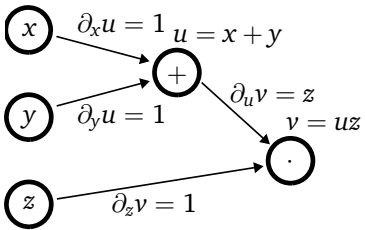
TECHNISCHE UNIVERSITÄT DARMSTADT

- Instead of manually evaluating $\partial_{\theta_i^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p)$, let some software (like Sympy) write down closed-form expressions for you using Symbolic Calculation.
- Doesn't work: huge closed-form expressions are necessary, causing huge memory and CPU overheads.

TECHNISCHE UNIVERSITÄT DARMSTADT

$$f(x,y,z) = (x+y)z = f(v(u(x,y),z))$$

Notation: $\partial_s f = \frac{\partial f(s)}{\partial s}$



Reverse-mode Automatic Differentiation (AD, details in Griewank and Walther [2008]) is the basis for evaluating derivatives for NN training (backpropagation).

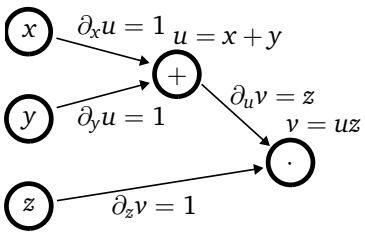
- Mathematic expressions are modeled with an directed acyclic graph (DAG).
- Intermediate results stored in variables.
- The graph's edges can evaluate known partial derivatives w.r.t. intermdediate variables.
- Chain rule is used to compute the partial derivative along the graph:

$$\partial_x f = \partial_v f(v) \partial_u v(u,z) \partial_x u(x,y) = 1z1 = z \qquad (13)$$

TECHNISCHE UNIVERSITÄT DARMSTADT

$$f(x,y,z) = (x+y)z = f(v(u(x,y),z))$$

Notation: $\partial_s f = \frac{\partial f(s)}{\partial s}$



- **Exact**: no Finite Difference-induced Floating-Point cancellation errors, no discretization errors.
- Automatic for arbitrary NN architecture.
- Computationally more efficient than Symbolic Calculations or Finite Differences.
- Responsible for "reviving" Deep Learning.

Deep Learning

Deep Learning aren't a "silver bullet"

TECHNISCHE UNIVERSITÄT DARMSTADT

- Math proves θ^M exists and NNs are universal function approximators, but not how to find it.
- Finding θ^M depends on λ, the NN architecture, and the activation function - **hyperparameters**.
- Hyperparameters are "free" parameters tuned by
  - graduate/Ph.D. students (student descent algorithm)
  - Grid Search, Monte Carlo, Bayesian Optimization: keyword AutoML.
- Once hyperparameters are "tuned", **some** θ^M is found with a minimal e_{MSE} in the best case over a response-surface that **hopefully** models the hyperparameter space well - there is no guarantee θ^M is globally-optimal in terms of data or hyperparameters.
- Training takes **a lot of computational time and resources**.
- **As soon as a form of Stochastic Gradient Descent is used (large data), running the training twice with the same hyperparameters and input data will give a different output from the NN.**

- NNs are function compositions, composing a matrix/vector product, an addition, and a nonlinear (activation) function σ,
  $\psi_\theta^{nn}(\mathbf{u}) = \mathbf{w}^D \cdot \sigma(\mathbf{w}^{D-1} \cdot \sigma(\mathbf{w}^{D-2} \cdot \sigma(\dots \sigma(\mathbf{w}^1 \cdot \mathbf{u} + \mathbf{b}^1) \dots) + \mathbf{b}^{D-2}) + \mathbf{b}^{D-1}) + b^D$,
  for our example scalar function ψ.
- An NN is a function-approximator, "trained" by minimizing an error norm over data, like MSE
  $\sum_{p=1}^{N_p} (\psi_{\theta^m}^{nn}(\mathbf{u}_p) - \psi_p) \partial_{\theta_i^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p) \stackrel{!}{=} 0$
- Approximation generally means $|\psi_{\theta^m}^{nn}(\mathbf{u}_p) - \psi_p| \neq 0$, we aim for $\partial_{\theta_i^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p) \stackrel{!}{=} 0$.
- To reach this, we perform (some form of) gradient descent
  $\theta^{m+1}(\mathbf{u}_p) = \theta^m(\mathbf{u}_p) - \lambda^m \nabla_{\theta^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p)$, resulting in $e_{MSE}(\theta^{m+1}) \leq e_{MSE}(\theta^m)$.
- For gradient descent, we compute $\partial_{\theta_i^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p)$ (gradient components), using Reverse-mode Automatic Differentiation.

Physics-Based Deep Learning

Literature survey (incomplete)

TECHNISCHE UNIVERSITÄT DARMSTADT

Different approaches exist, all extend the idea of function-approximation by NNs with satisfying PDEs. PDEs are built from differential operators, that are constructed from the NN using AD.

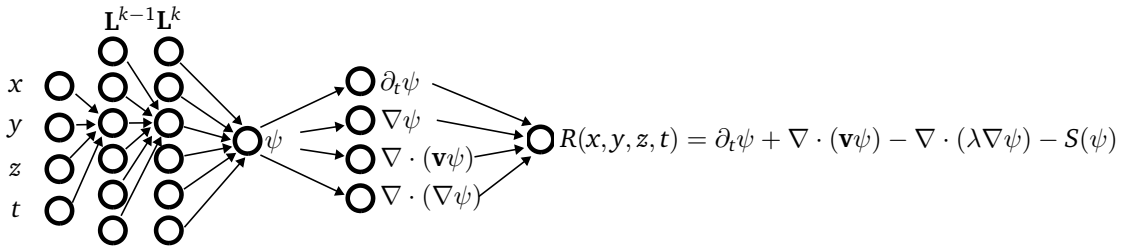
- The idea originated (afaik) with Lagaris et al. [1998].
  - A collocation method with NN as a trial function.
- Geometrically complex boundaries: Lagaris et al. [2000], McFall and Mahan [2009].
- Galerkin method with NN instead of shape functions: Sirignano and Spiliopoulos [2018].
- Raissi et al. [2019], **Physics-Informed Neural Networks (PiNN)s** - collocation MSE for PDEs like Lagaris et al. [1998] + data MSE.
- More alternatives described by Thuerey et al. [2022].

This talk addresses PiNNs.

Physics-Based Deep Learning

PiNNs one slide

TECHNISCHE UNIVERSITÄT DARMSTADT

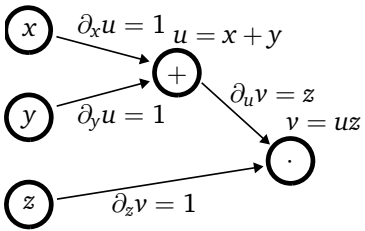


1. Re-use Automatic Differentiation used for NN training, for computing partial derivatives of the forward-pass with respect to NN input to construct PDE operators.
2. Extend the loss function with PDE residuals: the NN learns data and the PDE.

TECHNISCHE UNIVERSITÄT DARMSTADT

$$f(x,y,z) = (x+y)z = f(v(u(x,y),z))$$

Notation: $\partial_s f = \frac{\partial f(s)}{\partial s}$



- Evaluating $\partial_{\theta_i^m} \psi_{\theta^m}^{nn}(\mathbf{u}_p)$ generates partial derivaties w.r.t. intermediate variables.
- The cached partial derivatives are re-used to compute e.g. $\partial_t \psi_{\theta^m}^{nn}(\mathbf{u}_p)$, $\mathbf{u}_p = (x,y,z,t)$, and equivalently for x, y, z,

$$\nabla \psi_{\theta^m}^{nn}(\mathbf{u}_p) = \begin{bmatrix} \partial_x \psi_{\theta^m}^{nn}(\mathbf{u}_p) \\ \partial_y \psi_{\theta^m}^{nn}(\mathbf{u}_p) \\ \partial_z \psi_{\theta^m}^{nn}(\mathbf{u}_p) \end{bmatrix} \tag{14}$$

and have to be cached again for higher-order differential operators like $\nabla \cdot \nabla \psi_{\theta^m}^{nn}(\mathbf{u}_p)$.

TECHNISCHE UNIVERSITÄT DARMSTADT

- The residual of the PDE is computed using differential operators **computed exactly**[1] by the Automatic Differentiation framework - **there are no discretization errors**.
- The residual of a passive scalar transport equation is

$$R_\theta(\mathbf{u}_p) = R_\theta(x,y,z,t) = \partial_t \psi^{nn}_{\theta^m}(\mathbf{u}_p) + \nabla \cdot (\mathbf{v}\psi^{nn}_{\theta^m}(\mathbf{u}_p)) - \nabla \cdot (\lambda \nabla \psi^{nn}_{\theta^m}(\mathbf{u}_p)) - S(\psi^{nn}_{\theta^m}(\mathbf{u}_p)) \qquad (15)$$

  evaluated at collocation points $\{\mathbf{u}_p\}_{p\in[1,\dots,N_p}$.
- Contrary to training the NN only on data with $\partial_{\theta^m_i} \psi^{nn}_{\theta^m}(\mathbf{u}_p)$, **the PiNN residual requires partial derivatives of the NN forward-pass with respect to NN inputs.**

[1] Up to floating-point arithmetic errors.

TECHNISCHE UNIVERSITÄT DARMSTADT

- **One technical detail on how the partial derivatives are evaluated by AD frameworks (e.g. torch::autograd) is relevant for constructing PiNNs in practice.**
- The NN $\psi : \mathbb{R}^k \rightarrow \mathbb{R}^l$, $\psi = \psi(\mathbf{u}, \theta)$, $k = d + 1 + N_\theta$: d for spatial dimensions and $+1$ for time in $\mathbf{u} = (x, y, z, t)$, N_θ for all the weights and biases.
- The loss $e_{MSE} : \mathbb{R}^l \rightarrow \mathbb{R}$, $e_{MSE} = e_{MSE}(\psi(\mathbf{u}, \theta))$.
- **The** $\partial_{\theta_i} e_{MSE}(\psi(\mathbf{u}, \theta))$ **for the gradient descent**, is by chain rule in Einstein's notation

$$\frac{\partial \psi_i}{\partial_{\theta_j}} \frac{\partial e_{MSE}}{\partial \psi_i} = \frac{\partial e_{MSE}}{\partial_{\theta_j}}, \tag{16}$$

in matrix notation

$$J_\theta \psi \cdot J_\psi e_{MSE} \tag{17}$$

is a dot product of two Jacobians - the NN w.r.t θ, the loss w.r.t. NN output.

For our example NN $\psi : \mathbb{R}^4 \to \mathbb{R}$, $\psi(\mathbf{u}) = \psi(x,y,z,t)$ (dropping ^{nn} superscript),

$$J_u\psi(\mathbf{u}_p) = \begin{bmatrix} \partial_x\psi(\mathbf{u_p}) \\ \partial_y\psi(\mathbf{u_p}) \\ \partial_z\psi(\mathbf{u}_p) \\ \partial_t\psi(\mathbf{u}_p) \end{bmatrix} 1, \tag{18}$$

where 1 is the 0-rank tensor (scalar) we use to obtain the elements of $\nabla\psi$ and $\partial_t\psi$,

$$\nabla\psi = \begin{bmatrix} (J_u\psi(\mathbf{u}_p))_1 \\ (J_u\psi(\mathbf{u}_p))_2 \\ (J_u\psi(\mathbf{u}_p))_3 \end{bmatrix}, \quad \partial_t\psi = (J_u\psi(\mathbf{u}_p))_4 \tag{19}$$

Hessian matrix

$$(H_u\psi)^T = (J_u J_u \psi)^T = \left(J_u \begin{bmatrix} \partial_x \psi \\ \partial_y \psi \\ \partial_z \psi \\ \partial_t \psi \end{bmatrix} \right)^T = \begin{bmatrix} \partial_x^2 \psi & \partial_x \partial_y \psi & \partial_x \partial_z \psi \partial_x & \partial_x \partial_t \psi \\ \partial_y \partial_x \psi & \partial_y^2 \psi & \partial_y \partial_z \psi & \partial_y \partial_t \psi \\ \partial_z \partial_x \psi & \partial_z \partial_y \psi & \partial_z^2 \psi & \partial_z \partial_t \psi \\ \partial_t \partial_x \psi & \partial_t \partial_y \psi & \partial_t \partial_z \psi & \partial_t^2 \psi \end{bmatrix} \tag{20}$$

so

$$\nabla \cdot \nabla \psi = \Delta \psi = H^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} [1110] = \partial_x^2 \psi + \partial_y^2 \psi + \partial_z^2 \psi \tag{21}$$

Is eq. (21) valid if $\mathbf{u}_p = (t, x, y, z)$?

Hessian matrix

$$(H_u\psi)^T = (J_u J_u \psi)^T = \left(J_u \begin{bmatrix} \partial_x \psi \\ \partial_y \psi \\ \partial_z \psi \\ \partial_t \psi \end{bmatrix} \right)^T = \begin{bmatrix} \partial_x^2 \psi & \partial_x \partial_y \psi & \partial_x \partial_z \psi \partial_x & \partial_x \partial_t \psi \\ \partial_y \partial_x \psi & \partial_y^2 \psi & \partial_y \partial_z \psi & \partial_y \partial_t \psi \\ \partial_z \partial_x \psi & \partial_z \partial_y \psi & \partial_z^2 \psi & \partial_z \partial_t \psi \\ \partial_t \partial_x \psi & \partial_t \partial_y \psi & \partial_t \partial_z \psi & \partial_t^2 \psi \end{bmatrix} \tag{20}$$

so

$$\nabla \cdot \nabla \psi = \Delta \psi = H^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} [1110] = \partial_x^2 \psi + \partial_y^2 \psi + \partial_z^2 \psi \tag{21}$$

Is eq. (21) valid if $\mathbf{u}_p = (t, x, y, z)$? No - order of partial derivatives changes changes $J_u \psi$ and $H_u \psi$.

TECHNISCHE UNIVERSITÄT DARMSTADT

- Since we're differentiating $\psi_\theta^{nn}(\mathbf{u_p})$ w.r.t $\mathbf{u}_p$, the activation functions should be differentiable.
- Differentiable activation functions increase the number training iterations.

TECHNISCHE UNIVERSITÄT DARMSTADT

- Total PiNN loss is a weighted sum of the data-loss e^d_{MSE} (at data points or collocation points), residual-loss e^r_{MSE}, boundary-condition loss e^b_{MSE}, and initial-value loss e^i_{mse}

$$l_{MSE} = e^d_{MSE} + e^r_{MSE} + e^{bd}_{MSE} + e^{bn}_{MSE} + e^i_{mse} \tag{22}$$

$$e^d_{MSE} = \frac{1}{N_d}\sum_{p=1}^{N_d}(\psi^{nn}(\mathbf{u}_p) - \psi_p)^2, \quad e^{bd}_{MSE} = \frac{1}{N_{bd}}\sum_{q=1}^{N_{bd}}(\psi^{nn}(\mathbf{u}_q) - \psi_q)^2$$
$$e^{bn}_{MSE} = \frac{1}{N_{bd}}\sum_{r=1}^{N_{bn}}(\nabla\psi^{nn}(\mathbf{u}_r))^2, \quad e^i_{MSE} = \frac{1}{N_{bd}}\sum_{s=1}^{N_i}(\psi^{nn}(x,y,z,t=t_0) - \psi_p(x,y,z,t=t_0))^2 \tag{23}$$

- Minimizing the total loss l_{MSE} makes the PiNN satisfy data, the PDE, and initial/boundary conditions **in the least-squares sense**.

Inverse problems

- Recover the solution of a PDE with partially-known boundary and initial conditions, and some noisy measurements.
- Learn a heat transfer coefficient from measurements.

High-dimensional PDEs

- Solving high-dimensional PDEs (finance?, physics) - Finite Differences do not scale for this.

Optimization

- Once trained, a PiNN surrogate model is very fast to evaluate, and should be (way) more accurate than an 1D ROM ODE solution.

- The PiNN loss $l_{MSE} = e^d_{MSE} + e^r_{MSE} + e^{bd}_{MSE} + e^{bn}_{MSE} + e^i_{mse}$ errors have different values and often (very) different gradients.

$$l_{MSE} = \lambda_d e^d_{MSE} + \lambda_r e^r_{MSE} + \lambda_{bd} e^{bd}_{MSE} + \lambda_{bn} e^{bn}_{MSE} + \lambda_i e^i_{mse} \tag{24}$$

  yaay, more free (PiNN loss regularization) "free" parameters to guesstimate. Alternatives: adaptive activation functions by Jagtap et al. [2020], modifying Adam solver by Wang et al. [2021], meta-learning by Psaros et al. [2022].
- Different error contribution may mean e.g. that a PiNN with data-assimilation satisfies the data and the PDE residual better than the BCs and ICs.

TECHNISCHE UNIVERSITÄT DARMSTADT

- PiNNs inherit the curse of hyperparameter dimensionality from DL.
- PiNNs often take many iterations (epochs) to train: differentiable activation functions have diminishing gradients and require small optimizer step sizes.
- Evaluating many PDE differential operators increases computational costs per epoch.
- Yes, a forward pass of a trained NN generates PDE solutions very quickly, but it still takes a lot of (unautomated) effort to get there.

TECHNISCHE UNIVERSITÄT DARMSTADT

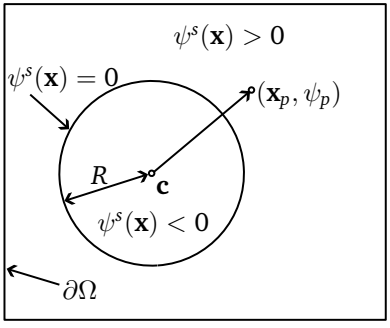
PiNNs are a relatively simple tool worth looking at

- Understand what the NN approximates.
- Understand how the Jacobian is used to construct PDE differential operators.
- Write down the total loss as the sum of: (data), residual, internal, and boundary condition loss.
- Apply adaptive activation functions or something similar to improve training.

If you just want to solve a "normal" PDE, it is overall faster to use a classical numerical method - **there is potential in combining Physics-Based ML with classical numerical methods.**

TECHNISCHE UNIVERSITÄT DARMSTADT

- "Introduction to Scientific Machine Learning 2: Physics-Informed Neural Networks", C. Rackauckas
- Hands-on introduction to Physics-Informed Machine Learning, I. Bilionis, A. Hans
- "When and why Physics-informed Neural Networks fail to train", P. Perdikaris
- "Physics-Informed Neural Networks | Misconceptions", C. Rackauckas



- An implicit sphere is given as a zero-level set

$$\Sigma = \{\mathbf{x} : \psi^e(\mathbf{x}) = 0\} \tag{25}$$

of a signed-distance field to a sphere

$$\psi^e(\mathbf{x}) = \|\mathbf{x} - \mathbf{c}\|_2 - R. \tag{26}$$

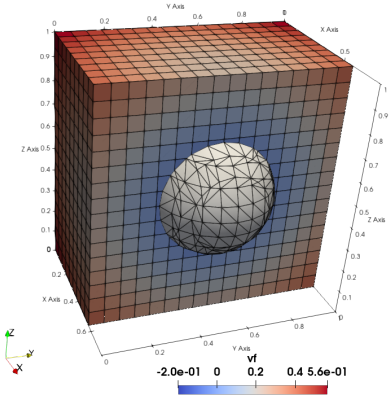
- Approximate the data from points $(\mathbf{x}_p, \psi_p^e)$ randomly sampled within the solution domain Ω, given that

$$\|\nabla \psi_p^e\|_2 = 1 \tag{27}$$

A PiNN with OpenFOAM and PyTorch C++ API

Problem definition II

TECHNISCHE UNIVERSITÄT DARMSTADT



- A unit-box domain $x, y, z \in [0, 1]$, discretized with N_c cells along each coordinate direction.
- Sphere of radius $R = 0.25$ centered at $\mathbf{c} = (0.5, 0.5, 0.5)$.
- **Motivation**: solving $\|\nabla \psi_p^e\|_2 = 1$ is quite complex using the unstructured Finite Volume method, if an approximative solution is good-enough, PiNNs are an option.

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: include order in OpenFOAM

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// libtorch
#include <torch/torch.h>
#include "ATen/Functions.h"
#include "ATen/core/interned_strings.h"
#include "torch/nn/modules/activation.h"
#include "torch/optim/lbfgs.h"
#include "torch/optim/rmsprop.h"

...

// OpenFOAM
#include "fvCFD.H"
```

- Example implemented as an OpenFOAM application that links with the PyTorch C++ API (**libtorch**).
- **Include `libtorch` before `OpenFOAM` headers.**
- **Compile OpenFOAM with C++14 or higher**, in **`wmake/rules/General/C++`**, set **`CC = g++$(COMPILER_VERSION) -std=c++2a`**.

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: `torch::Sequential` Multi-Layer Perceptron

TECHNISCHE UNIVERSITÄT DARMSTADT

```
torch::nn::Sequential nn;
nn->push_back(torch::nn::Linear(3, hiddenLayers[0]));
nn->push_back(torch::nn::Tanh());
for (label L=1; L < hiddenLayers.size(); ++L)
{
    nn->push_back(
        torch::nn::Linear(hiddenLayers[L-1], hiddenLayers[L])
    );
    nn->push_back(torch::nn::Tanh());
}
nn->push_back(
    torch::nn::Linear(hiddenLayers[hiddenLayers.size() - 1], 1)
);
```

- **`torch::Sequential`** used to construct the MLP.
- MLP architecture defined by application options or a dictionary.
  - Parametrization for hyperparameter tuning.

```
//  - Reinterpret OpenFOAM's input volScalarField as scalar* array
volScalarField::pointer vf_data = vf.ref().data();
//  - Use the scalar* (volScalarField::pointer) to view
//    the volScalarField as torch::Tensor without copying data.
torch::Tensor vf_tensor = torch::from_blob(vf_data, {vf.size(), 1});
```

- **`torch::from_blob`** constructs a **view** (interpretation) of data as **`torch::tensor`**.
- This works in OpenFOAM because all tensor fields in OpenFOAM are **`UList<T>`**

```
template<class T>
class UList
{
    // Private Data
        //- Number of elements in UList
        label size_;
        //- Vector of values of type T
        T* __restrict__ v_;
```

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: PiNN gradient

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Compute the prediction from the nn.
vf_predict = nn->forward(cc_training);

// Compute the gradient of the prediction w.r.t. input.
auto vf_predict_grad = torch::autograd::grad(
    {vf_predict},
    {cc_training},
    {torch::ones_like(vf_training)},
    true
);
```

- Following eq. (18), the scalar-valued forward-pass of the NN is differentiated w.r.t. input data a "tensor" of input points $\mathbf{x}_p$.
- For each point $\mathbf{x}_p = (x, y, z)$ we multiply the Jacobian with 1.
- For efficiency reasons (vectorization), calculation is done for all $p \in P$, so we need $\mathbf{v} = [1, 1, 1, 1, 1, \ldots, 1]^T$ in $J_u \cdot \mathbf{v}$, the length of $\mathbf{v}$ is N_p, the number of points.

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: PiNN loss for our problem

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Compute the data mse loss.
auto mse_data = mse_loss(vf_predict, vf_training);

// Compute the gradient mse loss.
auto mse_grad = mse_loss(
    at::norm(vf_predict_grad[0], 2, -1),
    torch::ones_like(vf_training)
);

// Combine the losses into a Physics Informed Neural Network.
mse = mse_data + mse_grad;
```

- The e^r_{MSE} is

$$e^r_{MSE} = \frac{1}{N_p} \sum_{p=1}^{N_r} (\|\nabla \psi^{nn}_\theta(\mathbf{x}_p)\|_2 - 1)^2 \tag{28}$$

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: hyperparameter tuning I

TECHNISCHE UNIVERSITÄT DARMSTADT

Grid Search for OpenFOAM+libtorch: works with other methods as well

- A Jupyter Notebook uses **`subprocess.call`** to run the variations.
- An OpenFOAM application parses hyperparameters as options.
- Results stored as CSV, with **repeated** hyperparameters as columns

	HIDDEN_LAYERS	OPTIMIZER_STEP	MAX_ITERATIONS	DELTA_X	EPOCH	DATA_MSE	GRAD_MSE	TRAININ
0	10,10,10,10	0.0001	3000	0.0625	1	0.164133	0.927430	1.0
1	10,10,10,10	0.0001	3000	0.0625	2	0.154731	0.928236	1.0
2	10,10,10,10	0.0001	3000	0.0625	3	0.148419	0.928779	1.0
3	10,10,10,10	0.0001	3000	0.0625	4	0.143443	0.929211	1.0
...	...	...	...	...	...	...	...	

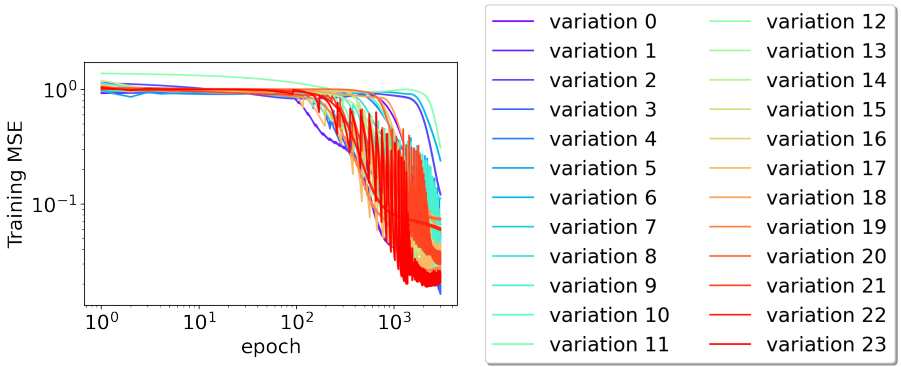
- CSVs are concatenated and the final **`pandas.DataFrame`** can easily be filtered by **repeated** hyperparameters. Avoid the **`pandas.MultiIndex`**, it's not worth the trouble.
- Each parameter vector is uniquely identified with the ID that is part of the file name: **`pinnFoam-00000000.csv`** - connected to the hyperparameters using **`pandas.DataFrame.unique`** on hyperparameter columns.

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: hyperparameter tuning II

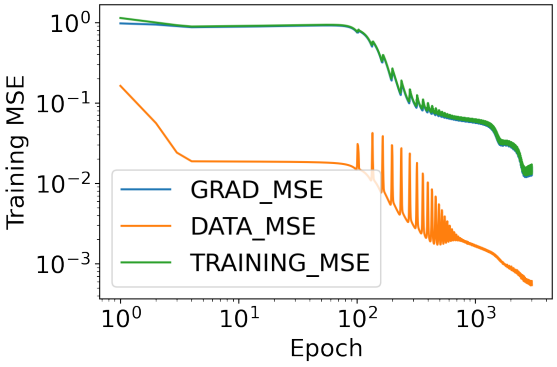
TECHNISCHE UNIVERSITÄT DARMSTADT

Loss MSE Grid Search



A PiNN with OpenFOAM and PyTorch C++ API

Implementation: best hyperparameters



Best hyperparameters:

```
HIDDEN_LAYERS   OPTIMIZER_STEP
20,20,20,20     0.001
MAX_ITERATIONS  TRAINING_ID
3000            3
```

- Just adding different e_{MSE} leads to different gradient flows.
- **These are example runs, actual runs have way more epochs.**
- **`Exercise`**: try to find λ_r and λ_d in

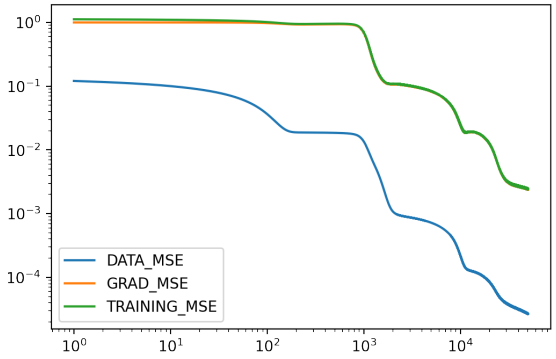
$$l_{MSE} = \lambda_d e^d_{MSE} + \lambda_r e^r_{MSE}$$

A PiNN with OpenFOAM and PyTorch C++ API

Implementation: real-world training

TECHNISCHE UNIVERSITÄT DARMSTADT

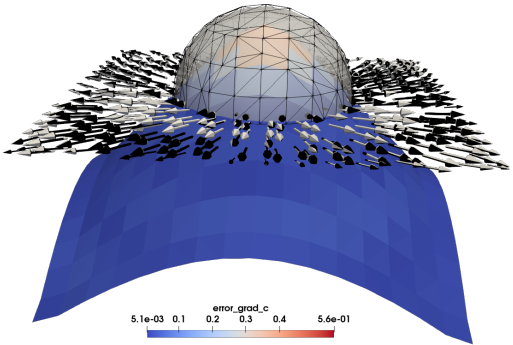
Real-world training takes **a very long time, minutes (!), even with $N_c = 16$ (4096 cells!)**.



A PiNN with OpenFOAM and PyTorch C++ API

Implementation: visualization

TECHNISCHE UNIVERSITÄT DARMSTADT



- Wireframe: $\psi^{nn}_{\theta^M}$ iso-surface.
- Gray surface: ψ^e iso-surface.
- Black and gray arrows: $(\nabla\psi)^e$ and $\nabla\psi^{nn}_{\theta^M}$ gradient vectors.
- Warped surface: $\|(\nabla\psi)^e - \nabla\psi^{nn}_{\theta^M}\|_2$

A PiNN with OpenFOAM and PyTorch C++ API

Source Code and Data

TECHNISCHE UNIVERSITÄT DARMSTADT

- https://gitlab.com/tmaric/ofw17-training-physics-based-dl
- Source Code and Data snapshot [Maric, 2022-07-10].

(My personal) Conclusions on PiNNs

TECHNISCHE UNIVERSITÄT DARMSTADT

- A very promising tool for some challenging problems.
- PiNNs aren't replacing standard numerics any time soon for standard problems.
- Very simple compared to standard numerics: no need to implement very complex algorithms for solving complex (systems of) PDEs.
- Know your Jacobians for PDE differential operator calculation using Automatic Differentiation.
- The PDE differential operators require differentiable activation functions - difficult to train.
- If something goes wrong, guess again - numerical methods can use more analysis.
- Hyperparameter tuning takes a lot of effort: automation makes sense.

References I

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Ameya D. Jagtap, Kenji Kawaguchi, and George Em Karniadakis. Adaptive activation functions accelerate convergence in deep and physics-informed neural networks. *Journal of Computational Physics*, 404:109136, 2020. ISSN 10902716. doi: 10.1016/j.jcp.2019.109136. URL https://doi.org/10.1016/j.jcp.2019.109136. arXiv: 1906.01170 Publisher: Elsevier Inc.

I.E. Lagaris, A.C. Likas, and D.G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, September 2000. ISSN 10459227. doi: 10.1109/72.870037. URL http://ieeexplore.ieee.org/document/870037/.

Isaac Elias Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998. doi: 10.1109/72.712178.

Tomislav Maric. Towards physics-based deep learning in openfoam: Combining openfoam with the pytorch c++ api (source code and data), 2022-07-10. URL https://tudatalib.ulb.tu-darmstadt.de/handle/tudatalib/3527.

References II

K.S. McFall and J.R. Mahan. Artificial Neural Network Method for Solution of Boundary Value Problems With Exact Satisfaction of Arbitrary Boundary Conditions. *IEEE Transactions on Neural Networks*, 20(8):1221–1233, August 2009. ISSN 1045-9227, 1941-0093. doi: 10.1109/TNN.2009.2020735. URL http://ieeexplore.ieee.org/document/5061501/.

Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

Apostolos F Psaros, Kenji Kawaguchi, and George Em Karniadakis. Meta-learning PINN loss functions. *Journal of Computational Physics*, 458:111121, June 2022. ISSN 00219991. doi: 10.1016/j.jcp.2022.111121. URL https://linkinghub.elsevier.com/retrieve/pii/S0021999122001838.

M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378: 686–707, 2019. ISSN 10902716. doi: 10.1016/j.jcp.2018.10.045. URL https://doi.org/10.1016/j.jcp.2018.10.045. Publisher: Elsevier Inc.

Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, December 2018. ISSN 00219991. doi: 10.1016/j.jcp.2018.08.029. URL https://linkinghub.elsevier.com/retrieve/pii/S0021999118305527.

References III

N Thuerey, P Holl, M Mueller, P Schnell, F Trost, and K Um. Physics-based Deep Learning. 2022. arXiv: 2109.05237v2.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, January 2021. ISSN 1064-8275. doi: 10.1137/20M1318043. URL https://epubs.siam.org/doi/10.1137/20M1318043. Publisher: Society for Industrial and Applied Mathematics.