Neural networks and numerical solution of PDE's

Izsák Ferenc

ELTE TTK Institute of Mathematics & AI Research Group

Farkas Miklós Seminar
Budapest, 7th March, 2024

Outline

- Neural networks in general
  - general application fields
  - approximation properties
  - main tools - benefits.
- Application to the numerical solution of PDE's
  - a conventional family of methods,
  - several other approaches,
  - results, main problems, questions.

Neural networks (NN) in general: a figure

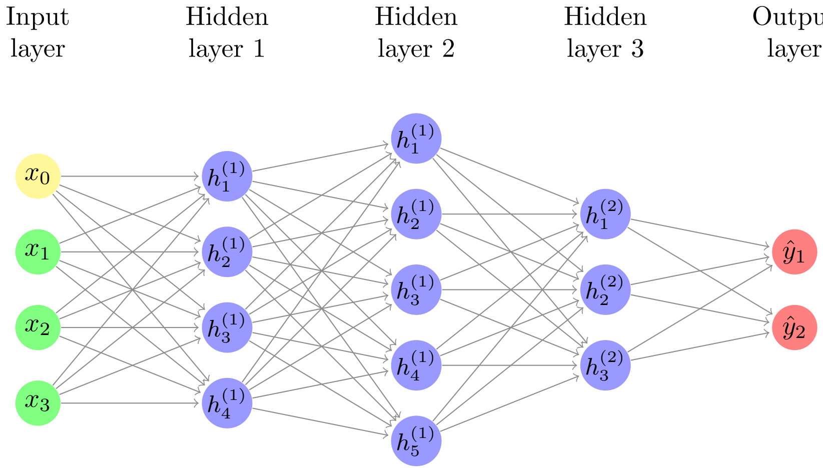


Figure: A simple neural network with dense layers.

Neural networks (NN) in general: formulas

- In a half sentence: a graph structure with some parameters (weights) + activation functions.
- Here: the so-called “feedforward” setting.
- Nodes are organized into “layers”.
  - With two specific ones: input layer, output layer.
  - We have some input and want to compute some output.
- Parameters define affine linear maps between *consecutive* layers.
- Layer-sizes: $s_0, s_1, \dots, s_N$.

Neural networks: the setup with formulas

- Value at layer$\#j$: $\boldsymbol{x}_j$.
- Transformation from layer $\#j-1$ to layer $\#j$:

  $\boldsymbol{x}_{j-1} \mapsto \rho_j(A_j \boldsymbol{x}_{j-1} + \boldsymbol{b}_j) = \boldsymbol{x}_j$.

  - $A_j \in \mathbb{R}^{s_{j-1} \times s_j}$, $\boldsymbol{b}_j \in \mathbb{R}^{s_j}$,
  - $\rho_j : \mathbb{R} \to \mathbb{R}$ given; applied componentwise.
    - In the softwares (MATLAB or Python) this can be chosen from a given family: ReLu, tanh, sigmoid, Id, ...
    - Responsible for nonlinearity.
- The NN can be characterized by $\{\rho_j, A_j, \boldsymbol{b}_j\}_{j=1,2,\dots,N}$.
  - $\{\rho_j\}_{j=1,2,\dots,N}$ and sizes are fixed,
  - the entries of A_j and $\boldsymbol{b}_j$: *parameters* that are tuned/optimized.

Neural networks: the setup - summary, remarks

- In this way, a function is associated to the NN.
- Full notation: $\mathcal{NN}_{A,\boldsymbol{b}} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_N}$.
- Overall aim: find the parameters $A, \boldsymbol{b}$ such that $\mathcal{NN}_{A,\boldsymbol{b}}$ approximates a given "function" $\mathcal{F}$.
- Examples (discrete and continuous):
  - $\mathcal{F} : \{\text{pictures}\} \mapsto \{\text{cat, dog, mouse}\}$
  - $\mathcal{F} : \{\text{medical images}\} \mapsto \{\text{symptom}_0, \text{symptom}_1, \text{symptom}_2\}$
  - $\mathcal{F}(u(0,\cdot) : \text{initial data of a PDE}) = u(t,\cdot)$
- In common words: the NN should "learn" the function $\mathcal{F}$.

Neural networks: the setup - remarks

- First two examples:
  - inputs are, indeed, matrices or rather 3D arrays
  - output-set: $\{(0,0,1),(0,1,0),(1,0,0)\}$
  - many times, also $A_j(\boldsymbol{x}) = \max_k x_k$ is applied.
- Here there is not even a definite function.
  - This is true; therefore, the neural network (or its setup) is called "the model".
  - This is a perfect tool, if there is no model for a phenomenon.

Neural networks: basic properties - mathematical statements

- NN can "learn" any function $\mathcal{F}$:

  Functions of type $\mathcal{NN}_{A,\boldsymbol{b}} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_N}$ can approximate any function $\mathcal{F} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_N}$.
- Name of the corresponding family of statements: universal approximation theorems.
- Why a "family of ..."?
- We can ask:
  - How accurately can approximate a fixed type of neural network our function $\mathcal{F}$?
  - How should we change the setup neural network to approximate our function $\mathcal{F}$?
    - taking larger and larger layers
    - taking more and more layers.

Neural networks: universal approximation theorems

Theorem (Cybenko '90)

For any non-polynomial ρ, $\varepsilon > 0, s_0, s_2 \in \mathbb{N}$, $K \subset \mathbb{R}^{s_0}$ *compact and* $\mathcal{F} \in C(K, \mathbb{R}^{s_2})$ *there are* $s_1 \in \mathbb{N}$ *and matrices* A_1, A_2, *vector* $\boldsymbol{b}_1$ *such that*

$$\sup_{\boldsymbol{x} \in K} \|A_2 \cdot \rho(A_1 \boldsymbol{x} + \boldsymbol{b}_1) - \mathcal{F}(\boldsymbol{x})\| < \varepsilon.$$

- A NN with one (but wide enough) hidden layer with no final activation can approximate any continuous function on a compact set with a given accuracy.

Neural networks: universal approximation theorems

- Similar statements hold if the size of the layers if fixed (maximized) and we can increase the number of them.

- Interesting question: For a given ε how can we achieve this with a minimal number of parameters?
  - In general: no answer for this.
  - It depends on the function to approximate.

- Important question: For a given $\mathcal{F}$ what kind of NN should be used?
  - The most important question.
  - No general answer.

Neural networks: making them work

- In practice, for a given structure, how to choose the best parameters?

  How to learn the function $\mathcal{F}$?

- We should optimize the weights $\{A_j, \boldsymbol{b}_j\}$ to get the best approximation.
- One can recognize it as a fitting problem.
- For this, we should know $\mathcal{F}(\boldsymbol{x}_k)$ for a number of inputs.
  - $\{(\boldsymbol{x}_k, \mathcal{F}(\boldsymbol{x}_k))\}_{k=1,2,\ldots,K}$ - "training set"
  - A number of input - output pairs.

Neural networks: learning procedure

- This is the optimization of parameters; in formulas:

$$\mathrm{LOSS}\left(\left\{\mathcal{F}(\boldsymbol{x}_k) - \mathcal{N}\mathcal{N}_{A,\boldsymbol{b}}(\boldsymbol{x}_k)\right\}_{k=1,2,\dots,K}\right) \xrightarrow{A,\boldsymbol{b}} \min$$

- Here we use a real-valued loss function
  - Common example: $\mathrm{LOSS}(\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K) = \frac{1}{K} \cdot \sum_{k=1}^{K} \|\boldsymbol{w}_k\|^2$.
  - Correlation type losses for discrete data.
  - A family of possible choices in Python or Matlab.
- The engine of the learning: a highly efficient optimization procedure.
  - Analysis comes into the play.

Neural networks: what kind of optimization

- Mostly simple gradient-based algorithms
  - Supported with automatic differentiation.
    - Advance of simple setup of the NN.
  - Adaptive choice of step lengths.
  - Mostly: stochastic gradient algorithms.
    - Epoch: consecutive gradient steps, while all data is used.
- Why just "gradient".
  - For so many parameters avoid computing of second derivatives.
  - Think of a million of parameters.
- Again: a family of possible choices in Python or Matlab.

Move to PDE's: The main conventional setup

- Name: physics informed neural networks (PINN's).
  - Origin: $\approx$ 2017, Karniadakis et. al, MIT
- A chief problem: there is no learning dataset.
  - Therefore, learning is not the conventional one.
  - In some cases, we can construct some learning data.
- The basic setup: we have a time-dependent problem

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) = Lu(t,\boldsymbol{x}) & \boldsymbol{x} \in \Omega,\ t \in (0,T) \\ u(t,\boldsymbol{x}) = u_b(t,\boldsymbol{x}) & \boldsymbol{x} \in \partial\Omega_0 \subset \partial\Omega,\ t \in (0,T) \\ u(0,\boldsymbol{x}) = u_0(t,\boldsymbol{x}) & \boldsymbol{x} \in \Omega \end{cases}$$

  with given functions u_b, u_0 and diff. operator L.
- We perform discretizations:
  - Ω_h - spatial discretization,
  - $t_1, t_2, \dots, t_N$ - time discretization.

Example, geometric setup

- The solution $u : (t, x) \mapsto u(t, x)$ has to be approximated.
- NN-inputs: $(t_k, \boldsymbol{x}_k)$, outputs: $\mathcal{N}\mathcal{N}_{A,\boldsymbol{b}}(t_k, \boldsymbol{x}_k)$.
- Loss function: how much is the equation failed?
  - Line 1, line 2 and line 3 in the equations:
  - $\text{Loss}_1 = \|(\partial_t - L)\mathcal{N}\mathcal{N}_{A,\boldsymbol{b}}(t_k, \boldsymbol{x}_k)\|$ for "interior" $(t_k, \boldsymbol{x}_k)$ inputs.
    - Automated symbolic differentiation of NN's.
  - $\text{Loss}_2 = \|\mathcal{N}\mathcal{N}_{A,\boldsymbol{b}}(t_k, \boldsymbol{x}_k) - u_b(t_k, \boldsymbol{x}_k)\|$ for "boundary" $(t_k, \boldsymbol{x}_k)$ inputs.
  - $\text{Loss}_3 = \|\mathcal{N}\mathcal{N}_{A,\boldsymbol{b}}(0, \boldsymbol{x}_k) - u_0(\boldsymbol{x}_k)\|$ for inputs $(0, \boldsymbol{x}_k)$.
- $\text{Loss} = \text{Loss}_1 + \text{Loss}_2 + \text{Loss}_3$
  - Or similar with squares or with some weights.

PINN's: a computational example

- X. Jin, S. Cai, H. Li, G. Em Karniadakis, NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier–Stokes equations, JCP, 2021.
- Applied to the Navier–Stokes equations
  - conventional and vorticity formulation
- 10 hidden dense layers with 300 neurons
- Altogether $\approx$ 820 000 parameters.
- $\approx$ 8000 epochs
- Initial learning rate $\approx 10^{-3}$, finally $\approx 10^{-3}$
- 100,000 points inside the domain, 26,048 points on the boundary, 147,968 points at the initial time step.
  - 17 time steps

PINN's: a computational example - discussion

- Simulation time:
  - Given just for a smaller problem: 12 times smaller w.r.t. each parameter.
  - This took 20-30 min using 6000 GPUs.
- A number of similar works
  - and a number of corresponding publications.
- This is really brute force
  - with using minimal knowledge on these problems.

NN's and PDE's: any other ideas?

- Main idea: use NN's to enhance the performance
  - *of some compound* of a conventional numerical method.
  - Rather useful for real life problems.
- I do not have a full overview of them:
  - many publications on conferences,
  - many publications on Arxive.
- Two of them will be presented.

An NN-based solver for conservation laws

- The equation to solve:

$$\partial_t u(t,x) + \partial_x(f(u(t,x)) = 0, \quad (t,x) \in (0,T) \times \Omega$$

- For well-posedness: appropriate initial and boundary conditions.
- A model of preservation of the quantity given with u.
- Common examples (for taking vector quantity u):

  Euler's equations, Navier–Stokes equations, shallow water equations
- f: flux of u.
  - can depend on ∇u.
- A number of numerical methods for the solution; they are non-trivial:
  - If we use a linear method for linear equations, then its convergence order w.r.t. time is at most 1. [S.K. Godunov '54]

Sketch of a conventional numerical method

- Discretize first w.r.t. x.
  - Take uniform intervals I_j of length h.
- Introduce: $u_j \approx$ total amount of u on I_j.
  - A system of ODE's for these:
- $\dot{u}_j = -\frac{1}{h}(\hat{f}_{j+\frac{1}{2}} - \hat{f}_{j-\frac{1}{2}})$
  - $\hat{f}_{j+\frac{1}{2}}$: approx of the flux on the right-end of I_j.
- In concrete terms, any of them is OK (Taylor):
  - $f^1_{j+\frac{1}{2}} = \frac{1}{6} \cdot (2f(u_{j-2}) - 7f(u_{j-1}) + 11f(u_j))$
    $f^2_{j+\frac{1}{2}} = \frac{1}{6} \cdot (-f(u_{j-1}) + 5f(u_j) + 2f(u_{j+1}))$
    $f^3_{j+\frac{1}{2}} = \frac{1}{6} \cdot (2f(u_j) + 5f(u_{j+1}) - 1f(u_{j+2}))$

Sketch of the WENO method (continued)

- Choose a weighted sum of these:

  $$\hat{f}_{j+\frac{1}{2}} = \omega_1 \cdot f^1_{j+\frac{1}{2}} + \omega_2 \cdot f^2_{j+\frac{1}{2}} + \omega_3 \cdot f^3_{j+\frac{1}{2}}.$$

  with the weights

  - $\omega_k = \frac{\alpha_k}{\alpha_1+\alpha_2+\alpha_3}$, with $\alpha_k = \frac{d_k}{(\epsilon+\beta_k)^2}$, $\quad k = 1, 2, 3$

- $(d_1, d_2, d_3) = (0.1, 0.6, 0.3)$, β_k: ensure low oscillations.
  - Seems to be rather heuristic but it works.
    - Try to find them instead with a NN.

- In [1] just carefully: instead of β_k: $\beta_k(1 + \delta_{j,k})$
  - optimizing δ_k.

[1]: T. Kossaczká, M. Ehrhardt, M. Günther: Enhanced fifth order WENO Shock-Capturing Schemes with Deep Learning. *Res. Appl. Math.*, 12, 2021.

The NN for optimized WENO approximations

- The authors used the following NN for a Burgers equation:

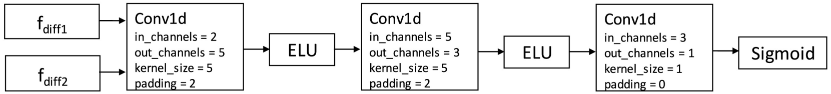


Figure: Optimizing the coefficient δ for the the inputs $f(x_{j+1}) - f(x_{j-1})$ and $f(x_{j+1}) - 2f(x_j) + f(x_{j-1})$.

- In the loss function, they compared some analytic solutions with the rsult of the optimized WENO approach using the above δ.

Another idea: NN-based discretization

- Example in case of the Laplacian.
- Well-known 5-point FD discretization on
  - a uniform 2D *h*-grid
  - gridpoints: $\{\boldsymbol{z}_{j,k}\}$ rácspontokkal.
  - Classic 5-point approximation: $\Delta u(\boldsymbol{z}_{j,k}) \approx$

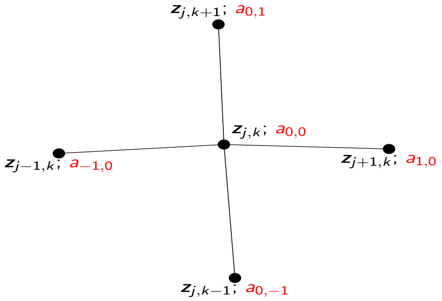
$$\frac{1}{h^2} \cdot \left(u(\boldsymbol{z}_{j-1,k}) + u(\boldsymbol{z}_{j+1,k}) + u(\boldsymbol{z}_{j,k-1}) + u(\boldsymbol{z}_{j,k+1}) - 4u(\boldsymbol{z}_{j,k})\right)$$

- 2nd order in space (w.r.t. both space variables)
- leads to a linear system for solving some Laplacian problem.
  - What happens in case of non-uniform grids??

NN-based FD approximation (continued)

- We are looking for coefficients $\{a_{\boldsymbol{s},j,k}\}$
  - giving accurate approximation of $\Delta u(\boldsymbol{z}_{j,k})$:

$$a_{-1,0}u(\boldsymbol{z}_{j-1,k})+a_{1,0}u(\boldsymbol{z}_{j+1,k})+a_{0,-1}u(\boldsymbol{z}_{j,k-1})+a_{0,-1}u(\boldsymbol{z}_{j,k+1})+a_{0,0}u(\boldsymbol{z}_{j,k})$$


$\boldsymbol{z}_{j,k+1}$; $a_{0,1}$
$\boldsymbol{z}_{j,k}$; $a_{0,0}$
$\boldsymbol{z}_{j-1,k}$; $a_{-1,0}$
$\boldsymbol{z}_{j+1,k}$; $a_{1,0}$
$\boldsymbol{z}_{j,k-1}$; $a_{0,-1}$

A method to compute such coefficients

- Take a fixed geometry.
- Take, e.g., polynomials p of order $0, 1, 1, 2, 2, 2, 3, 3, 3, 3$.
- Find such coefficients
  - that deliver the best approximation of p in the midpoint;
  - this is called the *optimization.*
- We should solve over-determined systems for this
  - Number of unknowns: 5.
  - Number of "equations": 10.
- Summarized: for all local geometry a separate LSQ solver (or another optimization process).

Apply NN instead: how and why?

- Try to learn this optimization step:
  - local geometry $\xrightarrow{NN}$ coefficients $\{a_{\boldsymbol{s},j,k}\}$
- Perform the optimization for many geometries:
  - we obtain a learning set.
  - $\mathcal{N}\mathcal{N}$ should perform faster compared to the optimization
- Possible benefits:
  - can be vectorized,
  - or compute parallel.
- Possible application: moving domains
  - we have to perform space discretization in each time-step.

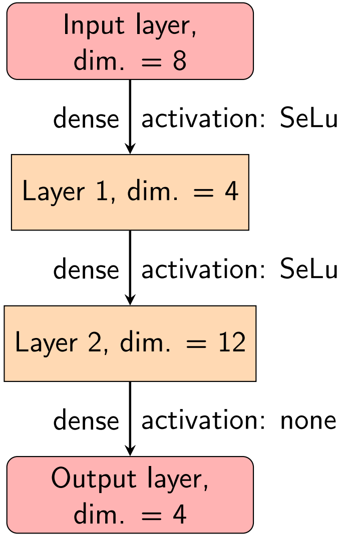
Finite elements: a possible alternative

- It can deal with an arbitrary triangular/tetrahedral grid.
- At the same time:
    - On a simple triangular grid that can be only of first order.
    - Needs an involved data structure.
    - Can hardly be vectorized, or parallel processed.

The NN in concrete terms

- We encode the local geometry into $\mathbb{R}^6$.
  - to compute with less and structured variables.
- Also, we take $\sum a_{\boldsymbol{s},j,k} = 0$
  - ensuring $\Delta(\mathrm{const.}) = 0$.
- Input of the NN:
  - deviation from the code of the standard geometry.
- Output of the NN:
  - the four coefficients $\{a_{1,0}, a_{0,1}, a_{-1,0}, a_{0,-1}\}$.

The structure of the NN



A domain for numerical simulation

- Geometry: only position of the points and their neighbors should be registered.

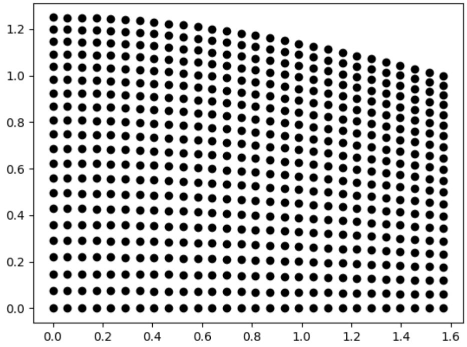


Figure: Pointwise discretization of a wave-shaped realistic domain.

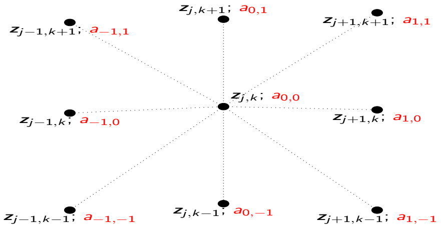
Using the NN-based discretization

- Application to solve a Laplacian problem:
  - Apply pointwise the NN.
  - Get the discretization matrix of the Laplacian.
    - This can be vectorized: `np.apply_along_axis`
  - Solve the corresponding linear system.
- Result:
  - $\approx$ 4-times smaller computational error compared to the computation with the coefficients $1, 1, 1, 1, -4$.

- Published article, poster on this issue.

continuation of this work, present studies

- Increase the accuracy of the approximation
  - try to develop an NN-based 8-point stencil:



- But a bit more structure in the mesh:
  - grid points below each other.

Jelenlegi munka: részeredmények, problémák

- The optimization finds exactly the coefficients for the standard geometry:
  - $1, 4, 1, 4, 1, 4, 1, 4, -20.$
  - A NN using 280 parameters learns quite well,
  - no overfitting.

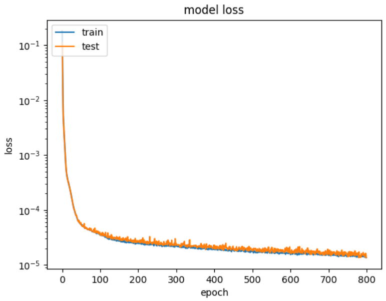


Figure: Training and validation losses.

Recent work: problems, further questions

- Problem: global approximation of the Laplacian is not accurate.
- One should also consider stability issues.

Acknowledgements (in Hungarian)

- A fentiekkel kapcsolatos újabb kutatások a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal támogatásával a Tématerületi Kiválósági Program 2021 - Nemzeti Kiválósági Alprogram „Mesterséges intelligencia, nagy hálózatok, adatbiztonság: matematikai megalapozás és alkalmazások” elnevezésű pályázatának keretében valósultak meg.

Thank you for your attention!