10 Methodological Overview

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

website

book

Contents

10 Methodological Overview

Deep learning in computational mechanics: a review, Herrmann et al. 2024

- **Simulation substitution**
  - Data-driven modelling
    Simulation with graph neural networks; DMD; Transfer learning
  - Physics-informed learning
    Hamiltonian/Lagrangian neural networks; SINDy; (PINNs)
- **Simulation enhancement**
  Input-convex neural networks for material modeling; EUCLID; Neural networks as ansatz function of inverse quantities; Superresolution; Differentiable physics
- **Discretizations as neural networks**
  Hardware acceleration with GPUs; (HiDeNN)
- **Generative approaches**
  Generative design; Realistic data generation; Anomaly detection; Transformers for natural language processing
- **Deep reinforcement learning**
  Control engineering tasks: autonomous flight; robots; Alternative gradient-free optimizer

10.1.1 Data-Driven Modeling

The aim of data-driven modeling is to learn a model in a supervised manner relying on labeled data

$$\{\tilde{x}_i, \tilde{y}_i\}_{i=1}^{m_{\mathcal{D}}}$$

Neural network as approximation of function $y = f(x)$ relating the data

$$\hat{y} = f_{NN}(x; \boldsymbol{\Theta})$$

The quality of prediction is defined in terms of a data misfit function, e.g., the MSE

$$\mathcal{L}_{\mathcal{D}} = \frac{1}{2} \sum_{i=1}^{m_{\mathcal{D}}} ||\hat{y}(\tilde{x}_i; \boldsymbol{\Theta}) - \tilde{y}_i||^2$$

Covered in depth in Chapter 3

10.1.1 Data-Driven Modeling

Covered in depth in Chapter 3

Consider the PDE

$$\mathcal{N}[u;\lambda] = 0$$

Goal is the approximation of a forward operator

$$\hat{u}(x) = F_{NN}(\lambda; x, t; \mathbf{\Theta})$$

Or an inverse operator

$$\hat{\lambda}(x,t) = I_{NN}(u; x, t; \mathbf{\Theta}) \text{ or } \widehat{\mathcal{N}}[u;\lambda] = O_{NN}(u; \lambda; \mathbf{\Theta})$$

For simplicity, we limit ourselves to the forward operator

- With a dataset $\left\{(\tilde{\lambda}_i; \tilde{x}_i, \tilde{t}_i\ ), (\tilde{u}_i)\right\}_{i=1}^{m_{\mathcal{D}}}$, the forward operator is learned
- The quality of prediction is defined in terms of a data misfit function, e.g., the MSE

$$\mathcal{L}_{\mathcal{D}} = \frac{1}{2}\sum_{i=1}^{m_{\mathcal{D}}} ||\hat{u}\big(\tilde{\lambda}_i; \tilde{x}_i, \tilde{t}_i; \mathbf{\Theta}\big) - \tilde{u}_i||^2$$

10.1.1 Data-Driven Modeling

10.1.1.1 Space-Time Approaches

- Time can simply be treated in the same way, as any other spatial dimensions
- In the following, time t is no longer mentioned explicitly, but included in the coordinates x

10.1.1.2 Time-Stepping Procedures

- Time is treated in a discrete manner using relying on a time-stepping procedure

10.1.1 Data-Driven Modeling

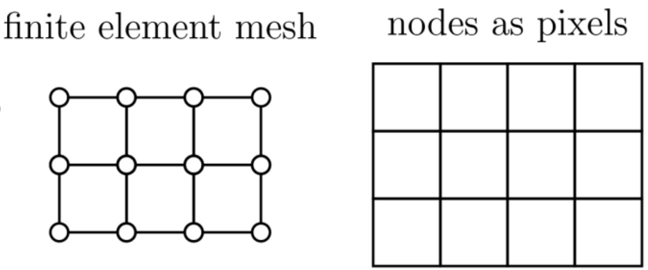
Covered in depth in Chapters 3 & 6

Fully-Connected Neural Networks

$$\hat{u}(x) = F_{FNN}(\lambda; x; \mathbf{\Theta})$$

Image-to-Image Mapping

$$\hat{\boldsymbol{u}}_i = F_{CNN}(\boldsymbol{\lambda}_i; \mathbf{\Theta}) \; or \; \hat{\boldsymbol{u}}_i = F_{GNN}(\boldsymbol{\lambda}_i; \mathbf{\Theta})$$



- where $\hat{\boldsymbol{u}}_i$ and $\boldsymbol{\lambda}_i$ are evaluations of $u(x)$ and $\lambda(x)$
  - on a uniform grid for convolutional neural networks
  - on arbitrary grids for graph neural networks

Model Order Reduction Encoding

- Forward operator is learned on a reduced space $\boldsymbol{\lambda}^{\mathcal{R}} = \phi(\boldsymbol{\lambda})$ with inverse mapping $\boldsymbol{u} = \phi^{-1}(\boldsymbol{u}^{\mathcal{R}})$

$$\hat{\boldsymbol{u}} = \phi^{-1}(\hat{\boldsymbol{u}}^{\mathcal{R}}) = \phi^{-1}\left(F_{NN}(\boldsymbol{\lambda}^{\mathcal{R}}; \mathbf{\Theta})\right) = \phi^{-1}(F_{NN}(\phi(\boldsymbol{\lambda}); \mathbf{\Theta}))$$

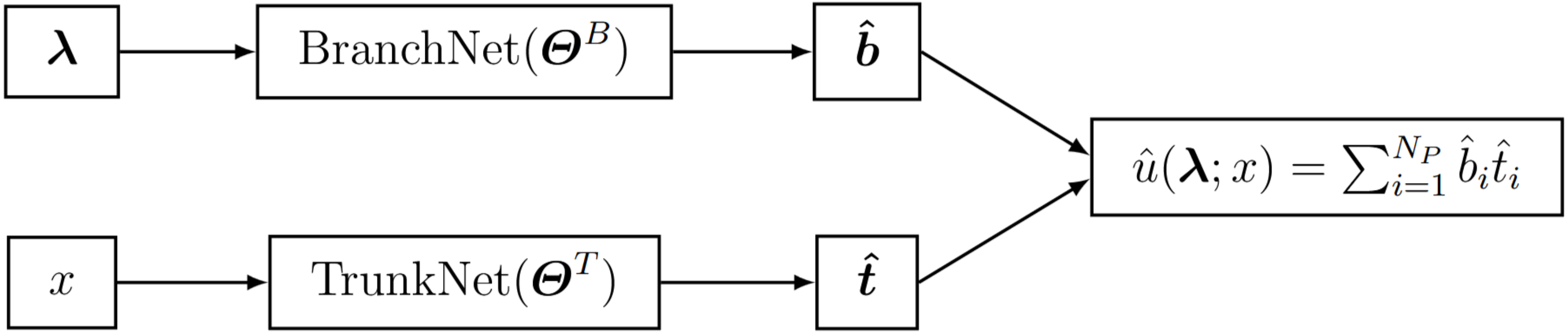
- Dimensional reduction can be performed with principal components analysis, singular value decomposition, autoencoders
- Misfit to data can be formulated on reduced space $\boldsymbol{u}^{\mathcal{R}}$ or full space $\boldsymbol{u}$

10.1.1.1.4 Neural Operators

DeepONet

- Operator prediction is split up into two tasks
  - Prediction of basis functions $\hat{\boldsymbol{t}}_i = F^T_{FNN}(x_i; \boldsymbol{\Theta}^T)$
  - Prediction of coefficients $\widehat{\boldsymbol{b}}_j = F^B_{FNN}(\boldsymbol{\lambda}_j; \boldsymbol{\Theta}^B)$
  - Prediction of solution $\hat{u}_{ij}$ at x_i with coefficients $\boldsymbol{\lambda}_j$ with the scalar product

$$\hat{u}_{ij} = \widehat{\boldsymbol{b}}_j \cdot \hat{\boldsymbol{t}}_i$$



10.1.1.1.4 Neural Operators

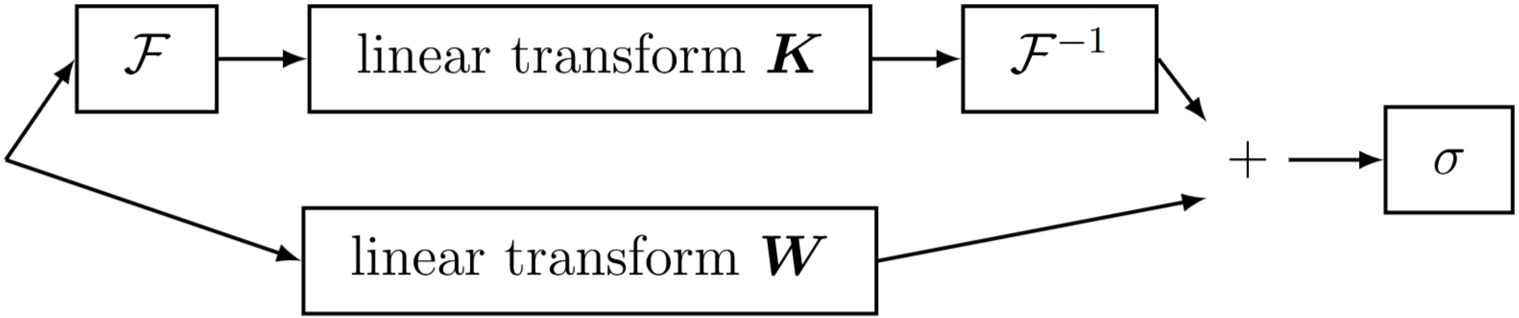
Fourier Neural Operators

- Mapping of coefficients $\boldsymbol{\lambda}$ to solution $\boldsymbol{u}$ is learned on a uniform grid

$$\hat{\boldsymbol{u}}(\boldsymbol{\lambda}) = \mathrm{F}_{\mathrm{FNO}}(\boldsymbol{\lambda}; \boldsymbol{\Theta})$$

- Fourier Neural Operator (FNO) consists of Fourier Layers that operator in Fourier Space (over $\boldsymbol{x}$)

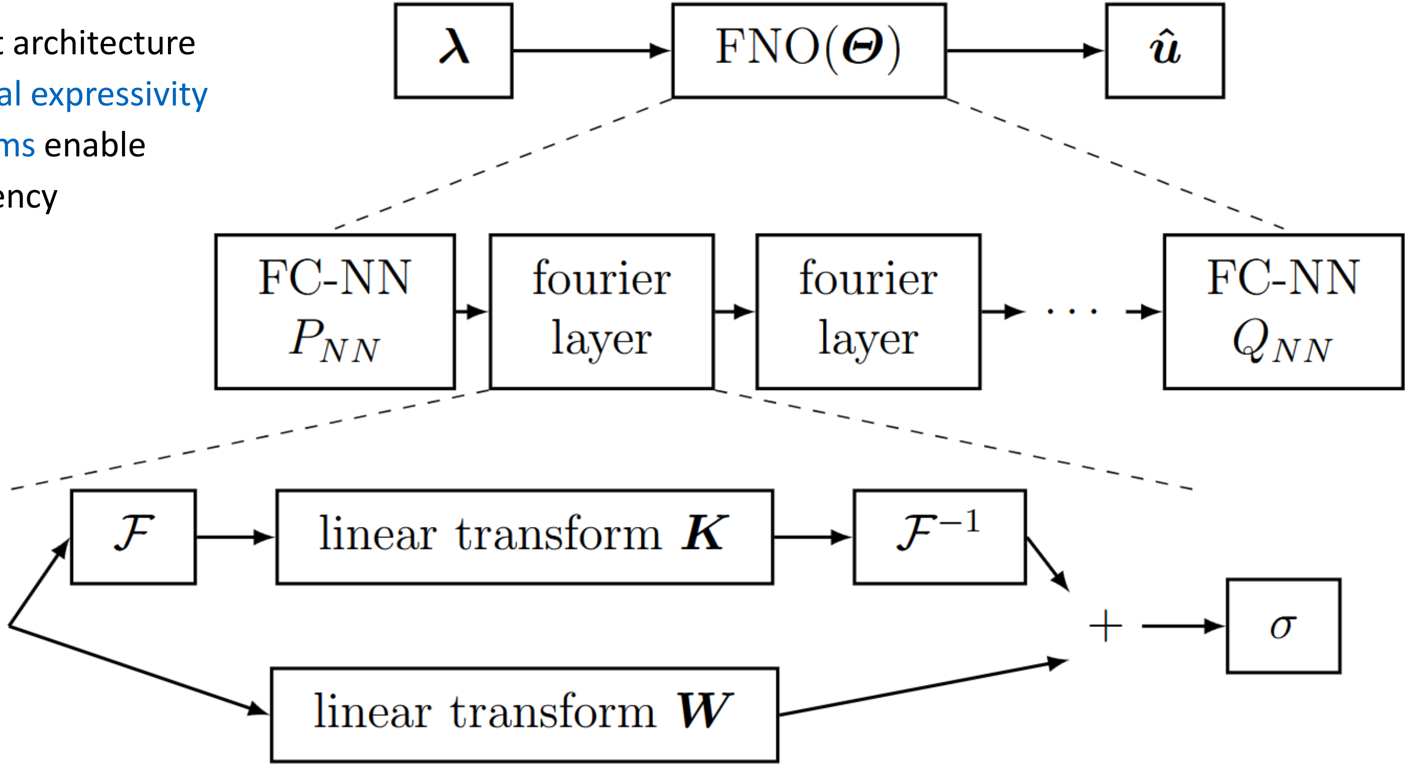
$$\boldsymbol{a}^{(j+1)}(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{a}^{(j)}(\boldsymbol{x}) + \boldsymbol{b} + (\mathcal{F}^{-1}\left[\boldsymbol{K}\mathcal{F}\left(\boldsymbol{a}^{(j)}(\boldsymbol{x})\right)\right])$$



10.1.1.1.4 Neural Operators

Fourier Neural Operators

- Expressive and efficient architecture
  - Enhance the non-local expressivity
  - Fast Fourier transforms enable computational efficiency



10.1.1.2 Time-Stepping Procedures

Time-Stepping Procedures

- Problem is discretized in time with $t_{i+1} = t_i + \Delta t$
- Prediction of time-history with fully-connected neural network

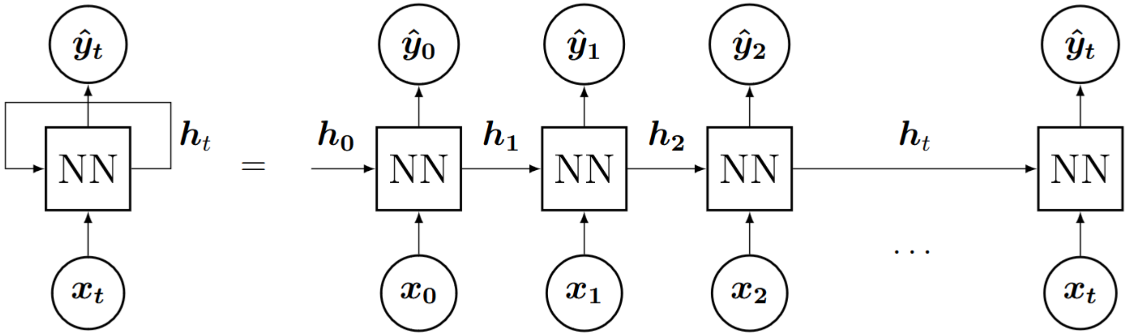
$$\hat{u}(x, t_{i+1}) = F_{FNN}(x, t_i; \boldsymbol{\Theta})$$

- Prediction of time-history with recurrent neural network

$$\{\hat{u}_2, \hat{u}_3, \dots, \hat{u}_N\} = F_{RNN}(x; u_1; \boldsymbol{\Theta})$$

  - or variations, such as LSTM, GRU, transformers

Difference to fully-connected neural network is a hidden/latent state that propagates infromation from prior steps



10.1.1.2 Time-Stepping Procedures

Dynamic Mode Decomposition

- Consider two successive snapshot matrices $\boldsymbol{X} = [\boldsymbol{x}(t_0), \boldsymbol{x}(t_1), \dots, \boldsymbol{x}(t_n)]^T, \boldsymbol{X}' = [\boldsymbol{x}(t_1), \boldsymbol{x}(t_2), \dots, \boldsymbol{x}(t_{n+1})]^T$
- Goal is to identify linear operator $\boldsymbol{A}$ that captures the dynamics

$$\boldsymbol{X}' \approx \boldsymbol{A}\boldsymbol{X}$$

- Solved as a regression using the Moore-Penrose pseudoinverse $\boldsymbol{U}^\dagger$

$$\boldsymbol{A} = \arg\min_{\boldsymbol{A}} ||\boldsymbol{X}' - \boldsymbol{A}\boldsymbol{X}||_F = \boldsymbol{X}'\boldsymbol{X}^\dagger$$

- Dynamical predictions with linear operator $\boldsymbol{A}$

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{A}\boldsymbol{x}(t_i)$$

- Only valid for linear dynamics
- Koopman operator theory states, that it is possible to represent any nonlinear system as a linear one by using an infinite-dimensional Koopman operator $\mathcal{K}$ that acts on a transformed state $g(\boldsymbol{x}(t_i))$

$$g\big(\boldsymbol{x}(t_{i+1})\big) = \mathcal{K}\big[g\big(\boldsymbol{x}(t_i)\big)\big]$$

- In practice, finite dimensional approximation, i.e., also for $g(\boldsymbol{x}(t_i))$
- Constructed with a dictionary of orthonormal basis functions $\boldsymbol{\psi}(\boldsymbol{x})$

10.1.1.2 Time-Stepping Procedures

Dynamic Mode Decomposition with neural networks

- Learn dictionary with a neural network

$$\widehat{\boldsymbol{\psi}}(\boldsymbol{x}) = \psi_{NN}(\boldsymbol{x}; \boldsymbol{\Theta}^{\psi})$$

  - Training on mismatch between predicted state and the true state in the dictionary space

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N} ||\widehat{\boldsymbol{\psi}}\big(\boldsymbol{x}(t_{i+1})\big) - \boldsymbol{A}\widehat{\boldsymbol{\psi}}\big(\boldsymbol{x}(t_i)\big)||_2^2$$

  - Reconstruction using Koopman mode decomposition
- Learn augmented state $\boldsymbol{h}(\boldsymbol{x})$ with autoencoder (learns both encoding and decoding)

$$\boldsymbol{h}(\boldsymbol{x}) = E_{NN}(\boldsymbol{x}; \boldsymbol{\Theta}^{E}), \boldsymbol{x}(\boldsymbol{h}) = D_{NN}(\boldsymbol{h}; \boldsymbol{\Theta}^{D})$$

  - Loss can be formulated in terms of autoencoder reconstruction, linear dynamics, future state prediction

$$||\boldsymbol{x} - D_{NN}(E_{NN}(\boldsymbol{x}; \boldsymbol{\Theta}^{E}); \boldsymbol{\Theta}^{D}||_2^2$$
$$||E_{NN}(\boldsymbol{x}(t_{i+1}); \boldsymbol{\Theta}^{E}) - \boldsymbol{A}E_{NN}(\boldsymbol{x}(t_i); \boldsymbol{\Theta}^{E})||_2^2$$
$$||\boldsymbol{x}(t_{i+1}) - D_{NN}(\boldsymbol{A}E_{NN}(\boldsymbol{x}(t_i); \boldsymbol{\Theta}^{E}); \boldsymbol{\Theta}^{D}||_2^2$$

$\boldsymbol{A}$ is obtained with conventional DMD

10.1.2 Physics-Informed Learning

Incorporation of physical constraints

- via penalty terms, which acts as regularization

$$\tilde{\mathcal{L}} = \mathcal{L} + \lambda||\cdot||^2$$

  - For example physics-informed neural networks as seen in Chapters 4 & 5
- by construction, which reduces the learnable space

$$\tilde{\hat{u}} = f(\hat{u})$$

  - For example physics augmented neural networks for material modeling as seen in Chapter 7

10.1.2 Physics-Informed Learning

Again, we consider **Space-Time Approaches** first

- In general, we consider a PDE with boundary & initial conditions

$$\mathcal{N}[u; \lambda] = 0$$

- And focus on the forward operator $u = F(x)$
- To illustrate the upcoming methods, consider the differential equation of a static elastic bar

$$\frac{d}{dx}\left(EA\frac{du}{dx}\right) + p = 0$$

- with Dirichlet boundary conditions on Γ_D

$$u(x) = g(x)$$

- and/or Neumann boundary conditions on Γ_N

$$EA(x)\frac{du(x)}{dx} = f(x)$$

10.1.1.2.1 Differential Equation Solving

Covered in depth in Chapter 4

Physics-Informed Neural Networks

- Solution to one differential equation is approximated with neural network

$$\hat{u}(x) = F_{FNN}^{u}(x; \mathbf{\Theta}^{u})$$

- Cost function is expressed as residual of the differential equation

$$\mathcal{L}_R = \frac{1}{2}\sum_{i=1}^{m_R} |\mathcal{N}[\hat{u}(x_i); \lambda(x_i)]|^2 = \frac{1}{2}\sum_{i=1}^{m_R}\left(\frac{d}{dx}\left(EA(x_i)\frac{du(x_i)}{dx}\right) + p(x_i)\right)^2$$

- residual of boundary conditions (and initial conditions)

$$\mathcal{L}_{\mathcal{B}} = \frac{1}{2}\sum_{i=1}^{m_{\mathcal{B}_D}} (u(x_i) - g)^2 + \frac{1}{2}\sum_{i=1}^{m_{\mathcal{B}_N}}\left(EA(x_i)\frac{du(x_i)}{dx} - f(x_i)\right)^2$$

- and a data-driven cost

$$\mathcal{L} = \mathcal{L}_{\mathcal{N}} + \mathcal{L}_{\mathcal{B}} + \mathcal{L}_{\mathcal{D}}$$

10.1.1.2.1 Differential Equation Solving

Covered in depth in Chapter 5

Deep Least-Squares Method/Deep Galerkin Method

- Considers the L^2 norm of the residual over the entire domain (instead of individual collocation points)

$$\mathcal{L}_R = \frac{1}{2}\int_{\Omega} |\mathcal{N}[u(x);\lambda]|^2 d\Omega$$

Variational Physics-Informed Neural Networks

- Residual of the weak form instead of the residual of the strong form

$$\mathcal{L}_V = \int_{\Omega} \frac{dw(x)}{dx} EA(x) \frac{du(x)}{dx} d\Omega - \int_{\Gamma_N} w(x) EA(x) \frac{du(x)}{dx} d\Gamma_N - \int_{\Omega} w(x) p(x) d\Omega = 0, \forall w(x)$$

- Test functions $w(x)$ are chosen by the user, e.g., trigonometric or polynomial

Weak Adversarial Networks

- Test functions are modeled with a second neural network

$$\hat{w}(x) = F_{FNN}^{w}(x; \mathbf{\Theta}^w)$$

- Minimax optimization (competition between trial and test functions)

$$\min_{\mathbf{\Theta}^u} \max_{\mathbf{\Theta}^w} \mathcal{L} \text{ with } \mathcal{L} = \mathcal{L}_{\mathcal{V}} + \mathcal{L}_{\mathcal{D}} + \mathcal{L}_{\mathcal{B}}$$

10.1.1.2.1 Differential Equation Solving

Covered in depth in Chapter 5

Deep Energy Method/Deep Ritz Method

- Minimization of the potential energy $\Pi_{\text{tot}} = \Pi_i + \Pi_e$

$$\mathcal{L}_{\mathcal{E}} = \Pi_i + \Pi_e = \frac{1}{2}\int_\Omega EA(x)\left(\frac{du(x)}{dx}\right)^2 d\Omega - \int_\Gamma u(x)EA(x)\frac{du(x)}{dx}d\Gamma - \int_\Omega u(x)p(x)d\Omega$$

- Only applicable to conservative systems, i.e., systems that conserve energy
- <u>Not generally applicable to inverse problems</u>
  - $EA(x)$ going towards $-\infty$ in Ω and towards ∞ on $d\Gamma$ minimizes $\mathcal{L}_{\mathcal{E}}$ to an arbitrary level

10.1.1.2.1 Differential Equation Solving

Extensions: Boundary Conditions

- Enforcement of boundary conditions with penalty term, leads to unbalanced optimization: $\mathcal{L}_\mathcal{N}, \mathcal{L}_B, \mathcal{L}_D$
- Strong enforcement of boundary conditions: a priori satisfaction

$$\hat{u} = G(x) + D(x) F^u_{FNN}(x; \mathbf{\Theta}^u)$$

  - $G(x)$ is a smooth interpolation of boundary conditions
  - $D(x)$ is a signed distance function (zero at boundary)
- For Neumann boundary conditions, prediction of u and derivative $\frac{\partial u}{\partial x}$ with separate neural networks
- For complex domains: learn $G(x), D(x)$ in a supervised manner
- Weighting terms for each loss term $\boldsymbol{\lambda} = \{\lambda_R, \lambda_\mathcal{B}, \lambda_\mathcal{D}\}$
  - Treated as hyperparameters, or learned with minimax optimization

$$\min_{\boldsymbol{\theta}^u} \max_{\lambda} \mathcal{L}$$

- The residual at each collocation point is treated as an equality constraint (augmented Lagrangian)

$$\mathcal{L}_R = \frac{1}{2} \sum_{i=1}^{N_R} \lambda_{R,i} |\mathcal{N}[u(x_i); \lambda(x_i)]|^2$$

10.1.1.2.1 Differential Equation Solving

Extensions: Ansatz

- Use convolutional neural networks instead of fully connected neural networks
  - Irregular geometries: binary encodings, signed distance functions, coordinate transformations
    - Numerical differentiation instead of automatic differentiation necessary
    - Or interpolation with, e.g., splines
- Use graph neural networks instead of fully connected neural networks
- Classical basis, such as FE or IGA: Approximate coefficients with neural network
  - Construct discretization in the classical sense and use discretized residual as loss

$$\mathcal{L}_{\mathcal{F}} = ||\boldsymbol{K}(\boldsymbol{u}^h)\boldsymbol{u}^h - \boldsymbol{F}||_2^2$$

  - Forward problem: learn $\boldsymbol{u}^h$, inverse problem: learn $\boldsymbol{K}$
  - Also interpreted as mapping a neural network to a finite element space (or interpolation)
    - When coefficients are predicted using coordinates
- Parametrization of ansatz with $\boldsymbol{\lambda}$ to learn multiple solutions

10.1.1.2.1 Differential Equation Solving

Extensions: Domain Decomposition

- Split complex problem in multiple simple problems
- hp-vPINNs decompose domain into patches
  - Piece-wise polynomial test functions on each patch & global solution u: separation of integration
- Conservative PINNs utilize one neural network per patch
  - Interface constraints enforce the conversation laws across patches, e.g., flux
  - Allows parallelization and adaptivity (shallow network for smooth, deep network for complex solutions)
- Generalization for any PDE presented as extended PINNs
  - Interface constraint formulated in terms of difference in solution and difference in residual

10.1.1.2.1 Differential Equation Solving

Extensions: Acceleration Methods

- Transfer learning from base task
- Improved sampling strategies
  - Importance sampling (collocation density proportional to residual)
  - Adaptivity: collocation points are added if residual is large
- Derivative of loss should be zero at minimum: include in loss
- Improved formulations of the loss: quality of solution does not necessarily correspond to loss
  - One improvement relies on the H^{-1} norm
- Numerical differentiation instead of automatic differentiation

  In particular for higher order derivatives, as seen in Chapter 5
- Extreme machine learning: all layers except the last one are frozen. The last layer is linear and learned through a least-squares regression
- Adaptive activation functions
- Spiking neural networks

  Specialized hardware to improve efficiency of training neural networks

10.1.2.1.2 Inverse Problems

Discovery of Governing Equations

- AI-Feynman is an automated framework for symbolic regression
- Performs dimensional analysis, polynomial regression and brute force search algorithms sequentially
- If an interpretable equation is not recovered
  - Neural networks as interpolation of the data
  - Test for symmetry, e.g., if $f(x_1, x_2, x_3, \dots) = f(x_1 + a, x_2 + a, x_3, \dots)$ then introduce variable $x_1' = x_2 - x_1$
  - Test for separability, i.e., $f(x_1, x_2) = g(x_1)h(x_2)$, to split equation in simpler parts

10.1.2.2 Time-Stepping Procedures

Time-Stepping Procedures

- For simplicity, we assume a differential equation of the form

$$\frac{\partial u}{\partial t} = \mathcal{N}[u]$$

- For systems, we assume the following form

$$\frac{d\boldsymbol{x}(t)}{dt} = f(\boldsymbol{x}(t))$$

Parareal PINNs

- Temporal domain is split in subdomains.
- Simplified PDE is solved from timestep t_0. PINN corrects solution in all subdomains
- Simplified PDE is solved from timestep t_1. PINN corrects solution in all subdomains
- …

10.1.2.2 Time-Stepping Procedures

PINN: Discrete Time Model

- Relies on the Runge-Kutta method with q stages

$$u^{n+c_i} = u^n + \Delta t \sum_{j=1}^{q} a_{ij} \mathcal{N}[u^{n+c_j}], \qquad i = 1, \dots, q$$

$$u^{n+1} = u^n + \Delta t \sum_{j=1}^{q} b_j \, \mathcal{N}[u^{n+c_j}]$$

- where $u^{n+c_j}(x) = u\left(t^n + c_j \Delta x, x\right)$, which is to be approximated with a neural network

$$\hat{\boldsymbol{u}} = [\hat{u}^{n+c_1}(x), \dots, \hat{u}^{n+c_q}(x), \hat{u}^{n+1}(x)] = F_{NN}(x; \boldsymbol{\Theta})$$

- Loss is computed by rearranging the Runge-Kutta equations using the initial condition u^n

$$\hat{u}^n = \hat{u}_i^n = \hat{u}^{n+c_i} - \Delta\mathrm{t} \sum_{j=1}^{q} a_{ij} \mathcal{N}[\hat{u}^{n+c_j}], \qquad i = 1, \dots, q$$

$$\hat{u}^n = \hat{u}_{q+1}^n = \hat{u}^{n+1} - \Delta t \sum_{j=1}^{q} b_j \mathcal{N}[\hat{u}^{n+c_j}]$$

10.1.2.2.2 Inverse Problems

Discovery of Governing Equations

- Learn right hand-side with neural network $\widehat{\mathcal{N}}[\boldsymbol{u}] = \mathcal{N}_{NN}\left(\boldsymbol{u}; \boldsymbol{\Theta}^{\mathcal{N}}\right)$
  - Minimization of residual of differential equation (PINN)
  - Use with any time-stepping scheme

- Interpolation of observed $\boldsymbol{u}^{\mathcal{M}}$ with neural network $\widehat{\boldsymbol{u}}(x,t) = u_{NN}(x,t; \boldsymbol{\Theta}^{u})$ in a supervised manner
  - Assume a linear network for right hand side with inputs for right hand side

$$\widehat{\mathcal{N}}[\boldsymbol{u}] = \mathcal{N}_{NN}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial x_i}, \frac{\partial \boldsymbol{u}}{\partial x_j}, \dots, \frac{\partial^2 \boldsymbol{u}}{\partial x_i^2}, \dots; \boldsymbol{\Theta}^{\mathcal{N}}\right)$$

  - L^1 regularization to enforce sparsity
  - Due to linearity, the model is interpretable

10.1.2.2.2 Inverse Problems

Discovery of Governing Equations

- PDE-Net: Learns right hand side with specialized convolutional neural network

$$\widehat{\mathcal{N}}[\boldsymbol{u}] = \mathcal{N}_{CNN}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial x_i}, \frac{\partial \boldsymbol{u}}{\partial x_j}, \ldots, \frac{\partial^2 \boldsymbol{u}}{\partial x_i^2}, \ldots; \boldsymbol{\Theta}^{\mathcal{N}}\right)$$

- Each convolution is designed to represent one spatial derivative term
- Achieved with special constraints, such that the neural network is only able to adjust order of approximation
- Forward Euler method

$$\boldsymbol{u}(t_{n+1}) \approx \boldsymbol{u}(t_n) + \Delta t \widehat{\mathcal{N}}[\boldsymbol{u}]$$

- Comparison to measurements $\boldsymbol{u}^{\mathcal{M}}$
- Both coefficients to derivative terms and order of approximation of derivatives are learned

- Identification of system and equation-specific approximations of the derivatives

10.1.2.2.2 Inverse Problems

Covered in depth in Chapter 6

Discovery of Governing Equations

- Sparse Identification of Nonlinear Dynamic Systems (SINDy)
- Snapshot matrices of state $\boldsymbol{X} = [\boldsymbol{x}(t_0), \boldsymbol{x}(t_1), \dots, \boldsymbol{x}(t_n)]^T$ and its time derivative $\dot{\boldsymbol{X}} = \left[\frac{d\boldsymbol{x}(t_0)}{dt}, \frac{d\boldsymbol{x}(t_1)}{dt}, \dots, \frac{d\boldsymbol{x}(t_n)}{dt}\right]^T$
- Formulation as sparse regression problem solved with sequential thresholded least squares/LASSO

$$\dot{\boldsymbol{X}} = \boldsymbol{\Theta}(\boldsymbol{X})\boldsymbol{\Xi}$$

- Expressivity can be increased by coordinate transformations enabling simpler dynamics
  - Similar to DMD, an autoencoder can be used to learn both encoding and decoding
  - Both autoencoder parameters and $\boldsymbol{\Xi}$ are learned by gradient descent
    - L^1 regularization to enforce sparsity
    - Limited interpretability
  - Classical time-stepping scheme can be applied in reduced space

10.1.2.2.2 Inverse Problems

Discovery of Governing Equations

- Consider multistep methods (similar to the PINN discrete time model)

$$\sum_{m=0}^{M} [\alpha_m \boldsymbol{x}_{n-m} + \Delta t \beta_m \boldsymbol{f}(\boldsymbol{x}_{n-m})] = 0$$

- With $M, \alpha_0, \beta_0, \beta_1$ defining the scheme
- Right-hand side is approximated by a neural network

$$\hat{\boldsymbol{f}}(\boldsymbol{x}) = f_{NN}(\boldsymbol{x}; \boldsymbol{\Theta})$$

- Loss is expressed in terms of multistep scheme

$$\mathcal{L} = \frac{1}{N - M + 1} \sum_{n=M}^{N} |\hat{\boldsymbol{y}}_n|^2$$

$$\hat{\boldsymbol{y}}_n = \sum_{m=0}^{M} [\alpha_m \, \boldsymbol{x}_{n-m} + \Delta t \beta_m \hat{\boldsymbol{f}}(\boldsymbol{x}_{n-m})]$$

10.2 Simulation Enhancement

Simulation Chain

- Pre-Processing
- Physical Modeling
- Numerical Methods
- Post-Processing

General Procedure

- Replace a function $\mathrm{y} = f(x)$ in the simulation chain by a neural network
$$\hat{y} = f_{NN}(x; \boldsymbol{\Theta})$$
- The remaining simulation chain remains unchanged

10.2.1 Pre-processing

Pre-Processing

- Geometry extraction
  - Segmentation, e.g., visual crack detection in images
- Mesh generation
  - Prediction of mesh density
- Data preparation
  - Denoising of measurement data
  - Low-frequency extrapolation in seismic data
- Preconditioning
  - Prediction of initial solution for classical solver
  - Transfer learning for neural networks

10.2.1 Physical Modeling

Physical Modeling

- Model Substitution
  - Neural network as surrogate (not interpretable)
  - Incorporation in simulation
- Identification of Model Parameters
  - Under assumption of model structure, neural network identifies the model
  - Neural network is not incorporated in simulation
- Model Identification
  - Neural network discovers model
  - Neural network is not incorporated in simulation

Covered in depth for material modeling in Chapter 7

- Consider the example of consitutitve models: Identify the stress-strain relation

$$\sigma = f(\varepsilon)$$

  - Directly usable in a finite element framework

10.2.1 Physical Modeling

Model Substitution

- Neural network as replacement of stress-strain relation trained in a supervised manner

$$\hat{\sigma} = f_{NN}(\varepsilon; \mathbf{\Theta})$$

- No guarantee, that physical principles are upheld, such as the second law of thermodynamics, material objectivity/frame invariance, material symmetry/isotropy, polyconvexity
- Specialized network architectures to guarantee physical principles by construction
  - Prediction of strain energy from invariants
  - Input-convex neural networks or neural ordinary differential equations for poly-convexity

- Training in a supervised manner with stress-strain data
- Training in an unsupervised manner via incorporation in finite element solver in combination with measurements of simulation using, e.g., modified constitutive relation error

10.2.1 Physical Modeling

Identification of Model Parameters

- Assumption of material model, e.g., linear elasticity $\sigma = C\varepsilon$
  - Prediction of model parameters C from spatially varying Young's modulus $\boldsymbol{E}$

$$\hat{C} = f_{NN}(\boldsymbol{E}; \boldsymbol{\Theta})$$

- Can for example be used for homogenization

10.2.1 Physical Modeling

Model Identification

- Efficient unsupervised constitutive law identification and discovery (EUCLID)
  - Posed as regression problem (inspired by SINDy)
  - Strain energy density ψ expressed in terms of candidate functions $\boldsymbol{Q}(\boldsymbol{I})$ with invariants $\boldsymbol{I}$

$$\psi(\boldsymbol{I}) = \boldsymbol{Q}(\boldsymbol{I})\boldsymbol{\Theta}$$

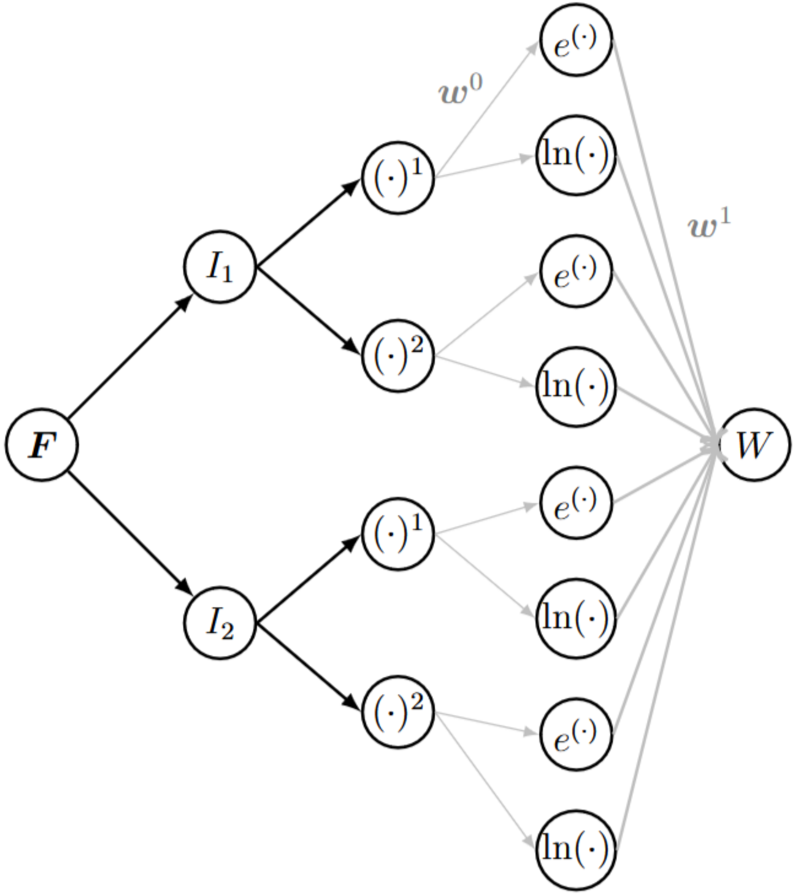
  - Stresses are obtained by deriving $\sigma = \frac{\partial \psi}{\partial \varepsilon}$
  - Residual is formulated in weak form of the momentum balance and solved with fixed-point iteration
  - Extension with neural networks, where $\psi(\boldsymbol{I})$ is learned by ensemble of neural networks instead of candidate functions (interpretability is lost)

10.2.1 Physical Modeling

Model Identification

- Automated discovery through interpretable neural networks
- Strain energy density of example

$$\hat{\psi} = w_0^1 e^{w_0^0 I_1} + w_1^1 \ln(w_1^0 I_1) + w_2^1 e^{w_2^0 I_1^2} + w_3^1 \ln(w_3^0 I_1^2)$$
$$+ w_4^1 e^{w_4^0 I_2} + w_5^1 \ln\left(w_5^0 I_2\right) + w_6^1 e^{w_6^0 I_2^2} + w_7^1 \ln(w_7^0 I_2^2)$$



10.2.3 Numerical Methods

Numerical Methods

- Numerical Quadrature, find optimal positions $\Delta\xi_g$ and weights Δw_g

$$\mathcal{L} = \frac{1}{2}\sum_{g=1}^{N_g} ||f\left(\xi_g^0 + \Delta\xi_g, w_g^0 + \Delta w_g\right) - K_{\text{exact}}^e||_2^2$$

  - Particle Swarm optimization to find optimal $\Delta\xi_g$, Δw_g: train neural network in a supervised manner
- Correction of strain-displacement matrix for distorted elements
- Learn optimal test functions
- Learn optimal timestep based on time-history

10.2.3 Numerical Methods

Numerical Methods

- Multiscale methods
  - Element substructuring
    - Network predicts displacements and stresses from boundary conditions in meta-element
    - Equilibrium is computed through assembly and iterative solution procedure
  - Zooming methods
    - Network predicts global system response
    - Boundary conditions are extracted and used for local analysis

- Active Learning
  - Neural network is trained during employement
  - Neural network is always used and an error estimator asseses the quality of the prediction
  - If prediction is bad, classical method computes solution
  - Every n bad predictions, the neural network is retrained

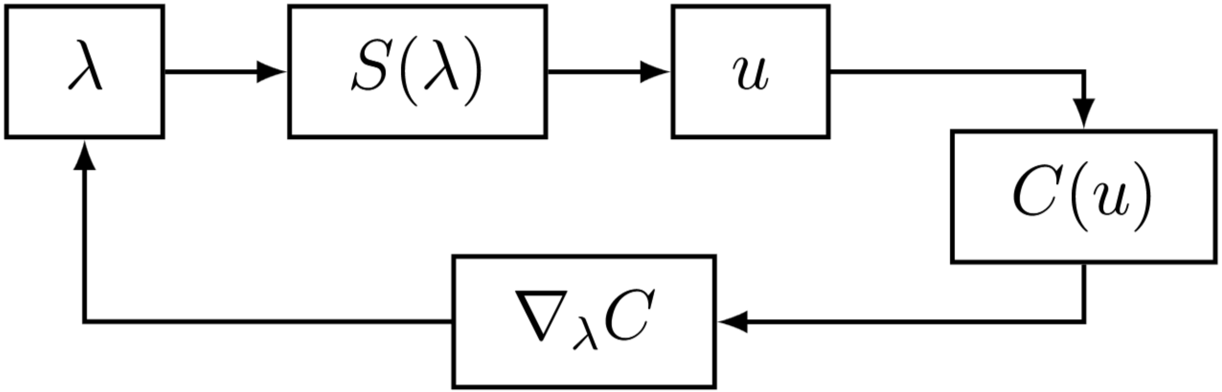
Bauhaus-Universität Weimar

10.2.3 Numerical Methods

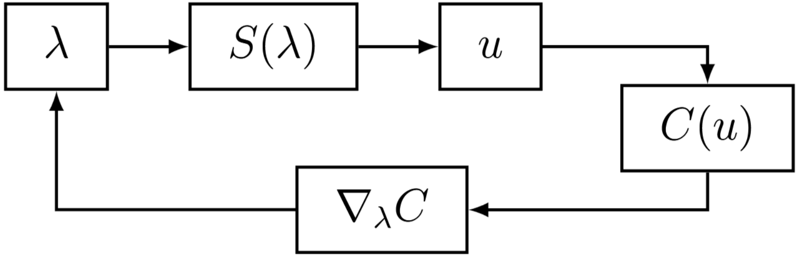
Covered in depth in Chapter 9

Numerical Methods

- Optimization
  - Gradient-based optimization
    - Prediction of state u from parameters λ with forward operator $S(\lambda)$
    - Quality of prediction determined by cost function $C(u)$
    - Gradient of cost function $\nabla_\lambda C$ is used to update parameters λ



10.2.3 Numerical Methods



Numerical Methods

- Injecting deep learning in the optimization
  - Replacement of forward operator S with surrogate model
    - Faster forward prediction
    - Simplified and possibly faster gradient computation
  - Replacement of sensitivity computation $\nabla_\lambda C$ with surrogate model
    - Learned in a supervised manner
    - Learn indirectly by maximizing the improvement of the cost function
      - Possibility for active learning
    - Partial replacement by solving coarse sensitivity problem and refinement with neural networks
  - Direct prediction of update from current state (skip the entire loop by going from λ_i to λ_{i+1})
    - Prediction of final state or intermediate states
  - Neural network as ansatz of λ
    - Neural network acts as regularizer/discovery of better local optima

10.2.4 Post-Processing

Post-Processing

- Modification of results to ensure manufacturability
- Feature extraction
  - Subsequent shape optimization
- Solution enhancement: Coarse to fine to reduce numerical errors
  - Solve problem on coarse and fine grid, train neural network in a supervised manner
    - Forward problems
    - Inverse problems, e.g., increasing resolution of identified designs
    - Sensitivity computation, i.e., solving adjoint problem on coarse grid

10.2.4 Post-Processing

Coarse to fine: tight integration with the solver

- PDE is solved on coarse grid
- Right-hand side is approximated by NNs using gradients as input

$$\hat{\mathcal{N}}[\boldsymbol{u}] = \mathcal{N}_{NN}\left(\boldsymbol{u}, \frac{\partial \boldsymbol{u}}{\partial x_i}, \frac{\partial \boldsymbol{u}}{\partial x_j}, \dots, \frac{\partial^2 \boldsymbol{u}}{\partial x_i^2}, \dots; \boldsymbol{\theta}^{\mathcal{N}}\right)$$

- Gradients are expressed as

$$\frac{\partial^n u}{\partial x^n} \approx \sum_{i=1}^{M} \alpha_i^n u_i$$

- where α_i are predicted by neural network (with constraints)

- Right-hand side is used with time-stepping scheme to predict $u^c(t_{i+1})$ from $u^c(t_i)$
- Loss is defined in terms of coarsened fine scale solution u^f
- Training on fine scale solutions on small domains
- Equation specific discretization of gradients to enable stable time-stepping procedures on coarse grids

10.2.4 Post-Processing

Coarse to fine

- PDE is to be solved on coarse grid
- At each timestep coarse solution $\tilde{\boldsymbol{u}}^c$ is corrected by corrector network

$$\hat{\boldsymbol{u}}^c = C_{NN}(\tilde{\boldsymbol{u}}^c; \boldsymbol{\theta})$$

- Network is corrected using coarsened fine solution $\boldsymbol{u}^f$

- Important detail: during training corrector network is applied after each timestep in the solver.
  - Adjoint state method to obtain gradients through the solver
  - This way, the network is trained the way it later will be employed
  - Outperforms purely supervised approach
  - Methodology is called **differentiable physics**

10.3 Discretizations as Neural Networks

Finite Element Method

- Finite element neural networks
  - Consider the system of equations from a finite element discretization

$$\sum_{j=1}^{N} K_{ij} u_j - b_i = 0, \qquad i = 1,2, \dots, N$$

  - Assuming constant material properties along an element and uniform elements: pre-integration

$$K_{ij} = \sum_{e=1}^{M} \alpha^e W_{ij}^e \text{ with } W_{ij}^e = \begin{cases} w_{ij}^e \text{ if } i,j \in e \\ 0 \text{ else} \end{cases}$$

  - Inserting assembly into system of equations

$$\sum_{j=1}^{N} \left(\sum_{e=1}^{M} \alpha^e W_{ij}^e \right) u_j - b_i, \qquad i = 1,2, \dots, N$$

10.3 Discretizations as Neural Networks

- System of equations

$$\sum_{j=1}^{N}\left(\sum_{e=1}^{M} \alpha^e W_{ij}^e\right) u_j - b_i, \qquad i = 1,2,\dots,N$$

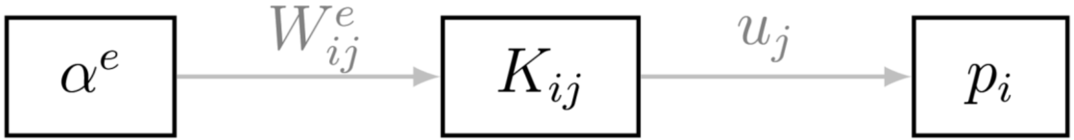
- Comparison to structure of a FNN $a_i^{(l)} = \sigma\left(z_i^{(l)}\right) = \sigma(\sum_{j=1}^{N^{(l)}} a_j^{(l-1)} + b_i^{(l)})$

$$a_i^{(2)} = \sum_{j=1}^{N^{(2)}} W_{ij}^{(1)} a_j^{(1)} = \sum_{j=1}^{N^{(2)}} W_{ij}^{(1)} \left(\sum_{k=1}^{N^{(1)}} W_{jk}^{(0)} a_k^{(0)}\right)$$

- Two layer FNN without activation and bias
- Prediction of forces $\hat{b}_i$ to establish a loss

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}\left|\hat{b}_i - b_i\right|^2$$

- In forward setting u_j is learnable
- In inverse setting α^e is learnable



10.3 Discretizations as Neural Networks

Covered in depth in Chapter 5

Finite Element Method

- Hierarchical deep-learning neural networks (HiDeNNs)
- Shape functions are treated as neural networks constructed from basic building blocks
- Consider one-dimensional linear shape functions

$$N_1(x) = \frac{x - x_2^e}{x_1^e - x_2^e}$$

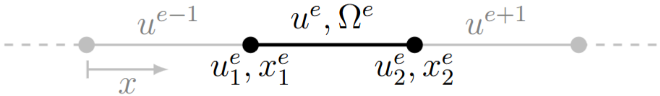
$$N_2(x) = \frac{x - x_1^e}{x_2^e - x_1^e}$$

- Elemental displacement from nodal displacements u_1^e, u_2^e treated as shared neural network weights

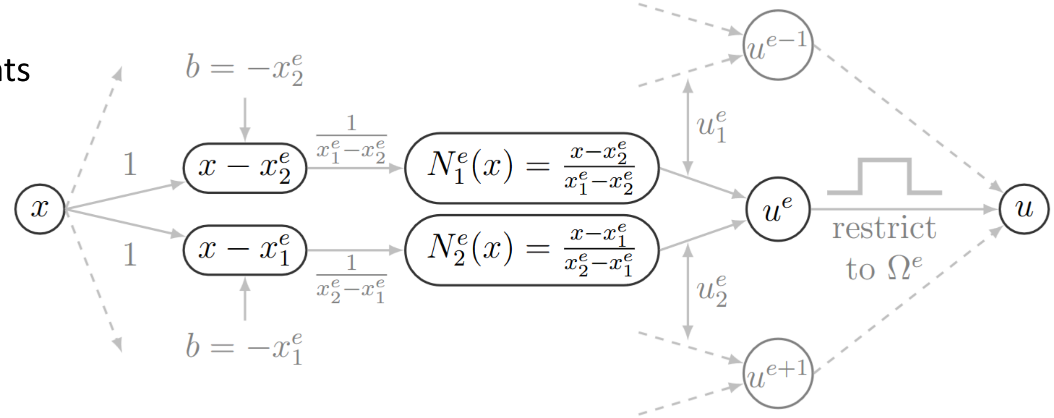
$$u^e = N_1^e(x)u_1^e + N_2^e(x)u_2^e$$

- nodal positions x_1^e, x_2^e and nodal displacements u_1^e, u_2^e are learnable weights

Moving nodal positions during solving is equivalent to *r*-refinement in the finite element method



One-dimensional finite elements



One-dimensional finite elements as NN

10.3 Discretizations as Neural Networks

Finite Difference Method

- Finite difference stencils can be considered as convolutional kernels
- Consider 1D scalar wave equation

$$\rho \frac{\partial^2 u}{\partial t^2} - \rho c^2 \frac{\partial^2 u}{\partial x^2} = f$$

- With a finite difference discretization

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \frac{\Delta t^2 f_i^n}{\rho}$$

- Can be written in terms of convolutional kernels

$$\boldsymbol{u}^{n+1} = 2\boldsymbol{u}^n - \boldsymbol{u}^{n-1} + \boldsymbol{u}^n * \boldsymbol{K}^u + \frac{\Delta t^2 \boldsymbol{f}^n}{\rho}$$

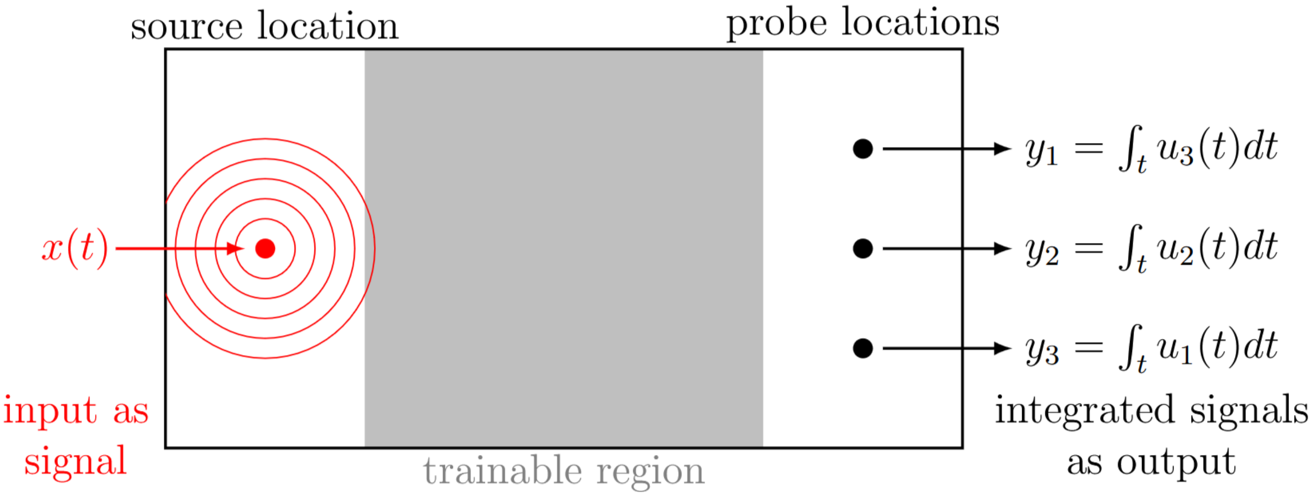
$$\boldsymbol{K}^u = [1, -2, 1]$$

This is not truly a recurrent neural network due to the lack of a hidden state

- Iterative application through time results in a recurrent neural network
- Inverse problem can be posed with these RNNs by solving forward problem and learning input parameters, e.g., ρ, c^2 or f

10.3 Discretizations as Neural Networks

- The discretized wave equation as analog recurrent neural networks
- Input x encoded as signal $x(t)$
- Output y_i measured $y_i(t)$ and integrated at probing locations

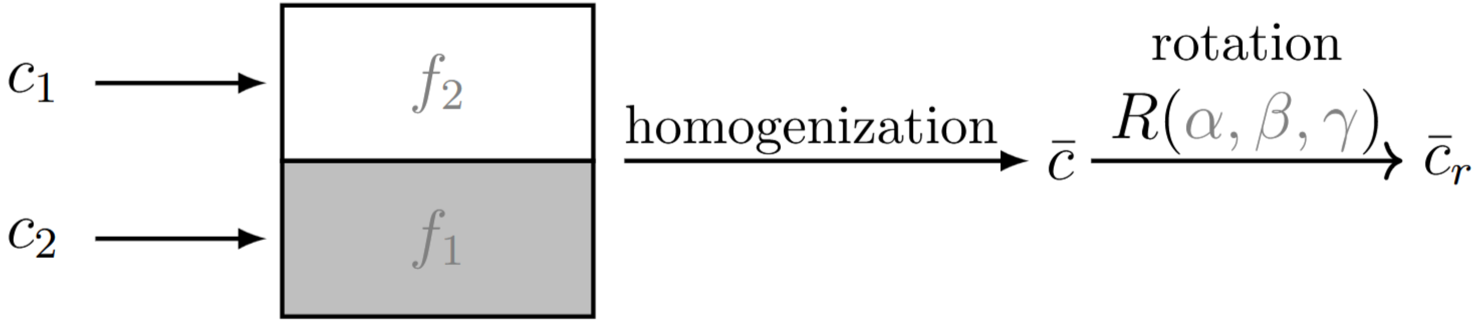


- For example for classification of sounds
- Trainable region could be 3D-printed as an analog neural network for classification

10.3 Discretizations as Neural Networks

Material Discretizations

- Deep material networks construct neural networks from material distributions
  - Basic building blocks from analytical homogenization techniques



  - Output material tensor $\overline{\boldsymbol{C}}_r$ is obtained by two input material tensors $\boldsymbol{C}_1, \boldsymbol{C}_2$, which are homogenized and rotated
  - Learnable parameters are the volume fractions f_1, f_2 and the rotations α, β, γ

  - Application: from stress-strain data from multiple microstructure samples
    - Extraction of material properties of te phases, anisotropic characteristics, or volume fractions

10.3 Discretizations as Neural Networks

Neural Differential Equations

- Evaluation of one recurrent unit in a recurrent neural network

$$\boldsymbol{a}_{t+1} = \boldsymbol{a}_t + f(\boldsymbol{a}; \boldsymbol{\theta})$$

- Can be viewed as an Euler discretization of the following ODE

$$\frac{d\boldsymbol{a}(t)}{dt} = f(\boldsymbol{a}(t), t; \boldsymbol{\theta})$$

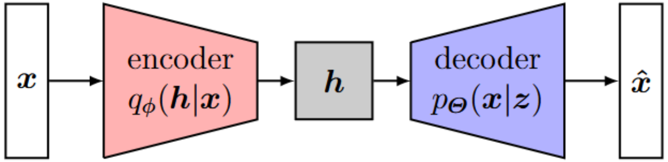
- Instead of using the recurrent evaluations of an RNN, the ODE is considered as a network
  - Input is the initial condition $\boldsymbol{a}(0)$
  - Output is the solution $\boldsymbol{a}(T)$ at time T
  - Gradients are obtained through the adjoint state method in an ODE solver

- Extension to partial differential equations, where convolutional neural networks act as spatial gradients

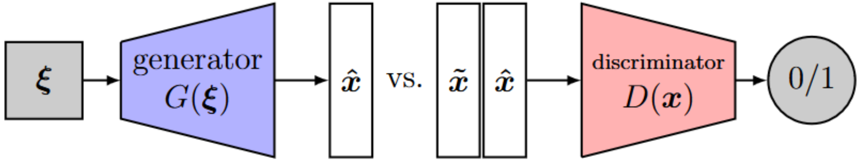
10.4 Generative Approaches

Covered in depth in Chapter 8

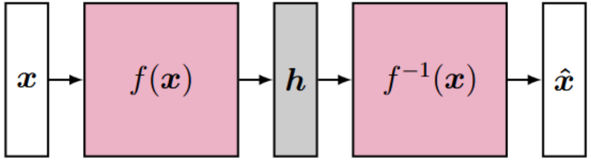
Variational autoencoder



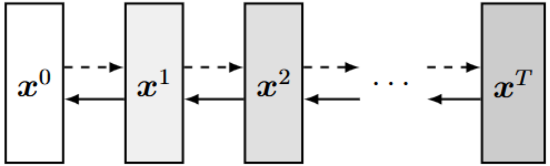
Generative adversarial network



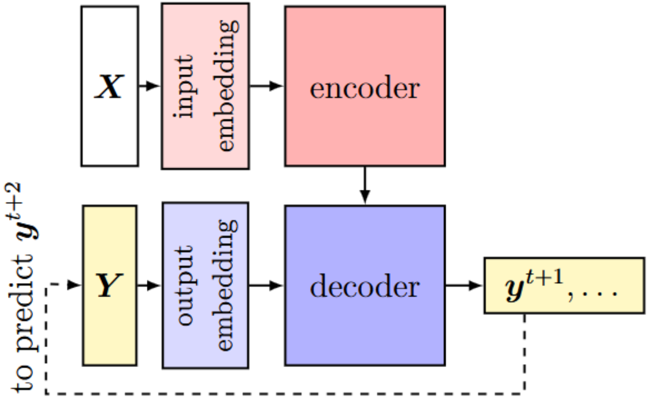
Flow-based model



Diffusion model



Transformer



10.4 Generative Approaches

Covered in depth in Chapter 8

Applications

- Data Generation
  - Microstructures (specialized architectures relying on RNNs/transformers to go from 2D to 3D)
  - Designs
- Optimization
  - Parametrization of design space with subsequent shape optimization
  - Variational autoencoder GANs are popular, due to well-behaving gradients
  - Incorporating vague constraints
    - Design diversity by increasing novelty/creativity
    - Similarity to old designs, due to aethetics or manufacturability
      - Pixel-wise L^1 distance to previous designs
      - Style transfer loss
  - Inverse problems
    - Generator predicts material distribution
    - Discriminator says, if it is correct based on a forward simulation

Generative Approaches

Covered in depth in Chapter 8

Applications

- Conditional Generation
  - Rendered cars from sketches
  - Hierarchical shape generation (child shape considers ist parent shape)
  - Topology optimization from initial fields defined by boundary conditions
  - Generation from physical properties
    - Design with specific compliance
    - Material distribution resulting in a specific seismogram
  - Super-resolution GANs for coarse simulation data (tempoGAN)
  - cycleGANs for invertible functions without need for paired data
- Anomaly Detection
  - Generative approach learns a distribution of structures
    - If the generative method is unable to reconstruct a structure, it must be an anomaly
    - In the case of GAN, a larger discriminator loss indicates an anomaly
    - In the case of an autoencoder, the latent space may also be considered

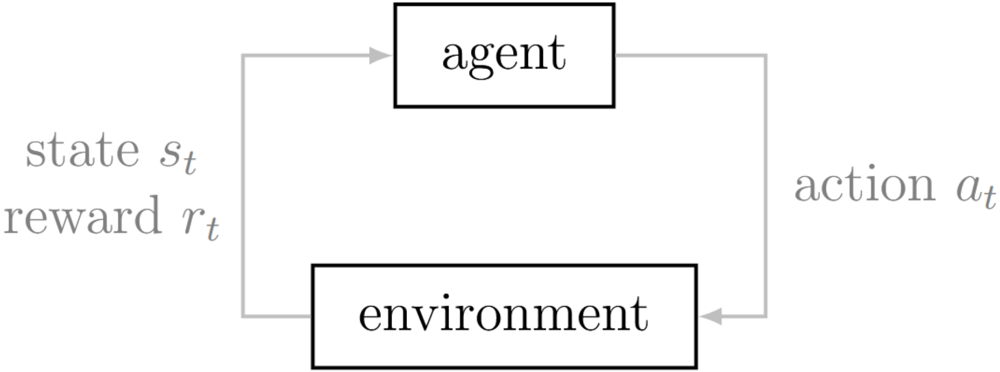
10.5 Deep Reinforcement Learning

- Learning from interaction with environment
  - Agent performs actions A_t
  - Based on actions, environment returns state S_t and reward R_t
- Deep reinforcement learning uses a neural network for the agent

- Value-based methods
  - Estimates value function
  - Take action that maximizes value function

- Policy-based methods
  - Maps states to actions directly

- Actor-critic methods
  - Combine value-based and policy-based methods
  - Actor chooses policy
  - Critic evaluates chosen action (the actor adjusts action according to the evaluations of the critic)



Contents

10 Methodological Overview

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

website

book