3 Neural Networks: Advanced Architectures

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

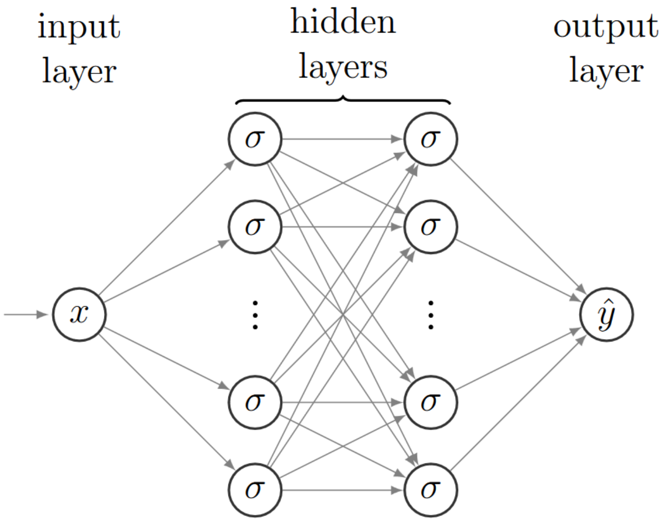
3.8 Approximating the Sine Function

Goal is to approximate

$$f(x) = \sin(2\pi x), x \in [-1, 1]$$

Neural network architecture

- 2 hidden layers
- 50 neurons each
- Activation function $\sigma(z_i) = \tanh(z_i)$
- $f_{NN} = w_3^T \sigma\left(W_2\left(\sigma(w_1^T x + b_1)\right) + b_2\right) + b_3 = \hat{y}$



Data

- Training set contains 40 samples
- Validation set contains 40 samples
- Generated with $y = \sin(2\pi x) + \epsilon, \epsilon = 0.1 \cdot U(-1,1)$ where $U(-1,1)$ is a uniform random distribution in the interval $[-1,1]$
- Test set is the analytical solution

3.8 Approximating the Sine Function

- Goal is to approximate

$$f(x) = \sin(2\pi x), x \in [-1, 1]$$

- Network architecture

$$f_{NN} = w_3^T \sigma\left(W_2\left(\sigma(w_1^T x + b_1)\right) + b_2\right) + b_3 = \hat{y}$$

- Cost function used for training

$$C = \mathcal{L}_D^{(\text{train})} = \frac{1}{m_D^{(\text{train})}} \sum_{i=1}^{m_D^{(\text{train})}} \left(\tilde{y}_i^{(\text{train})} - \hat{y}_i^{(\text{train})}\right)^2$$

- Neural network parameter initialization with a uniform distribution with special bounds
- Optimization with Adam for 10’000 epochs

3.8 Approximating the Sine Function – PyTorch

Neural network definition

```
modules = []
modules.append(torch.nn.Linear(1, 50))
modules.append(torch.nn.Tanh())
modules.append(torch.nn.Linear(50, 50))
modules.append(torch.nn.Tanh())
modules.append(torch.nn.Linear(50, 1))

model = torch.nn.Sequential(*modules)
```

Training data

```
x = torch.rand((40, 1)) * 2 - 1
noise = torch.rand(x.shape) * 0.2 - 0.1
y = torch.sin(2 * torch.pi * x) + noise
```

3.8 Approximating the Sine Function – PyTorch

Cost function definition

```
def costFunction(y, yPred):
    return torch.mean((y - yPred) ** 2)
```

Prediction and cost function evaluation

```
yPred = model(x)
cost = costFunction(y, yPred)
cost.backward()
```

Optimizer definition

```
epochs = 10000
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

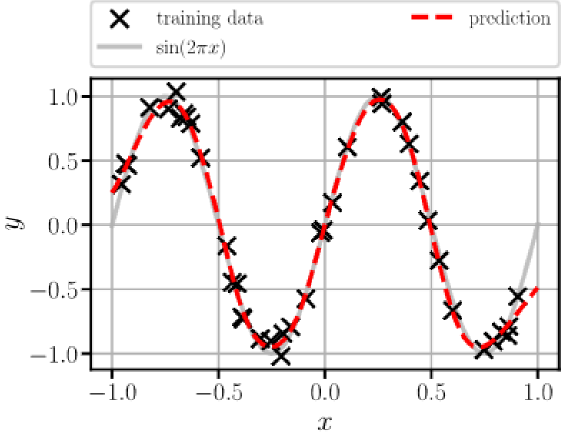
3.8 Approximating the Sine Function – PyTorch

Training loop

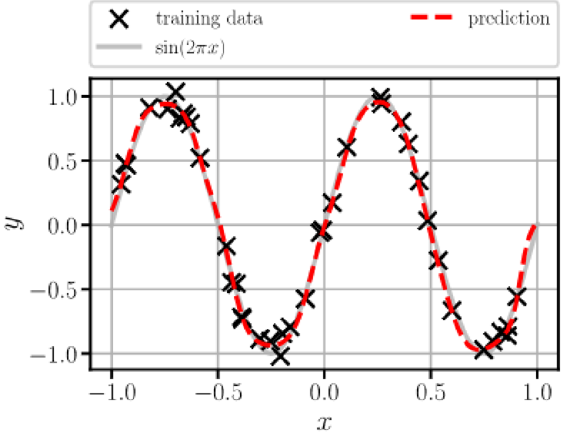
```
for epoch in range(epochs):
    optimizer.zero_grad()
    yPred = model(x)
    cost = costFunction(y, yPred)
    cost.backward()
    optimizer.step()
```

3.8 Approximating the Sine Function – Results

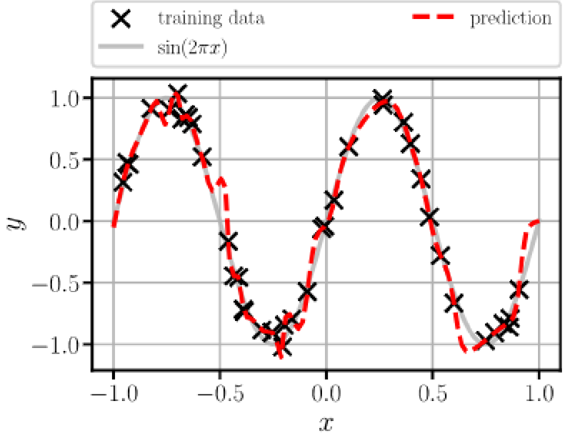
500 epochs



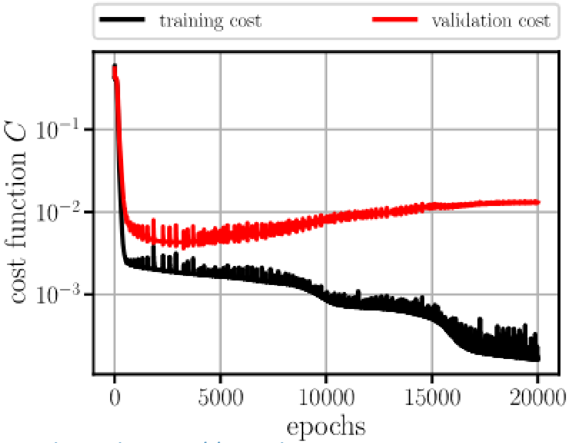
5‘000 epochs



20‘000 epochs

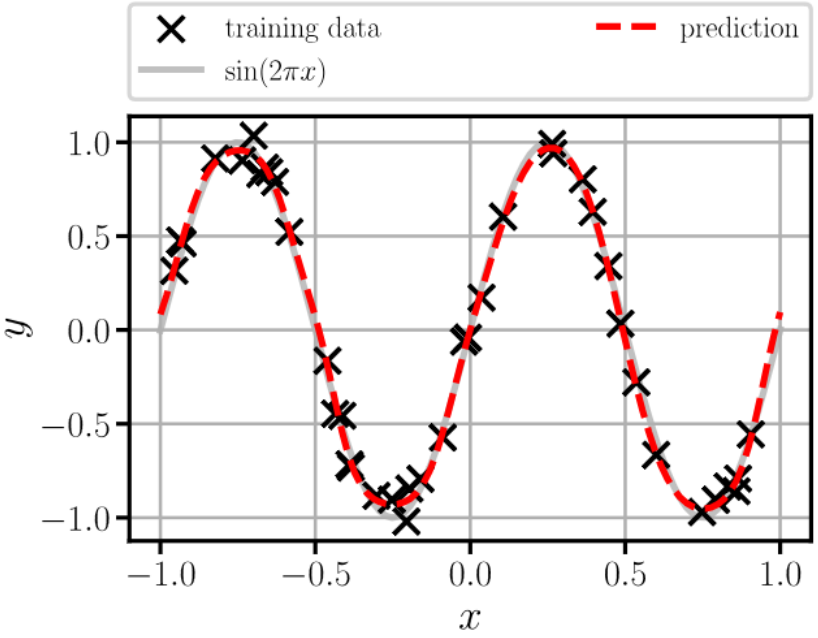


Learning history

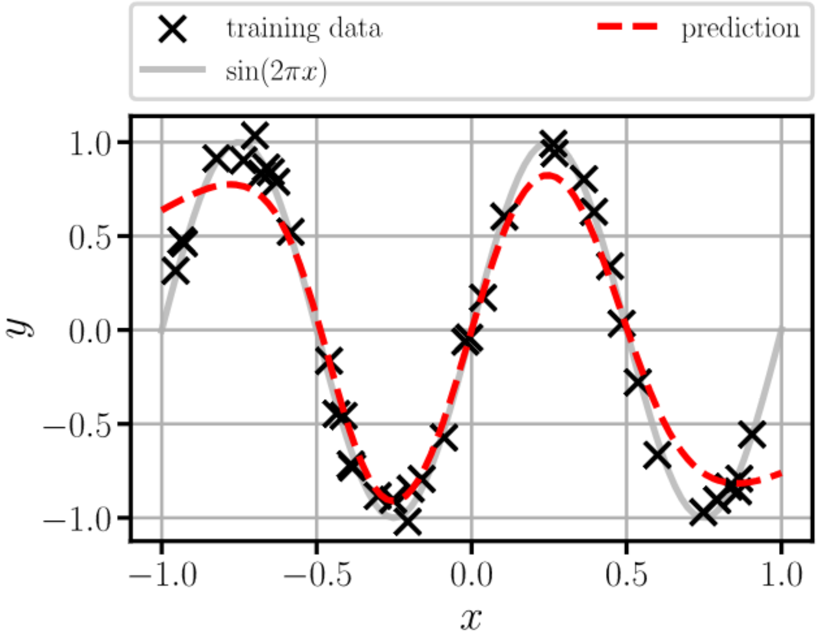


3.8 Approximating the Sine Function – Results

20‘000 epochs with regularization

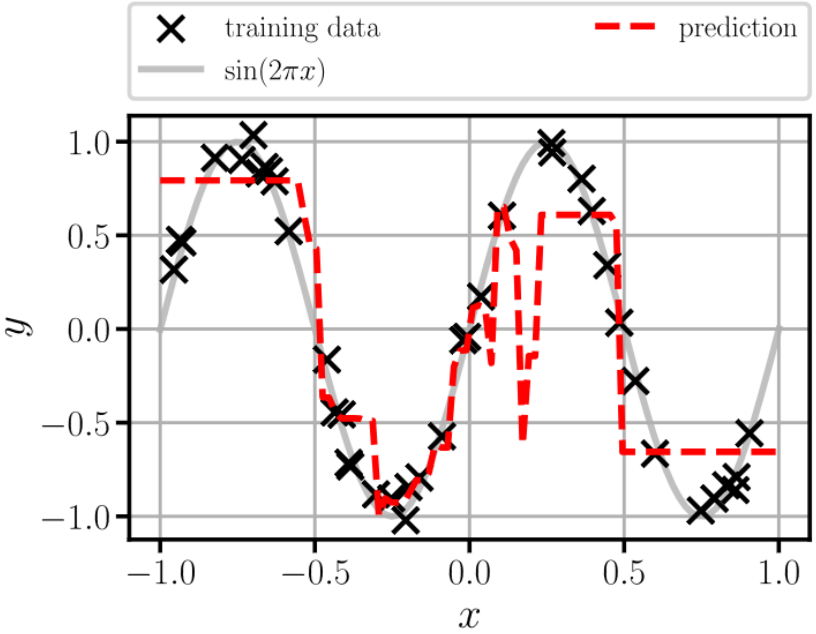


20‘000 epochs without regularization

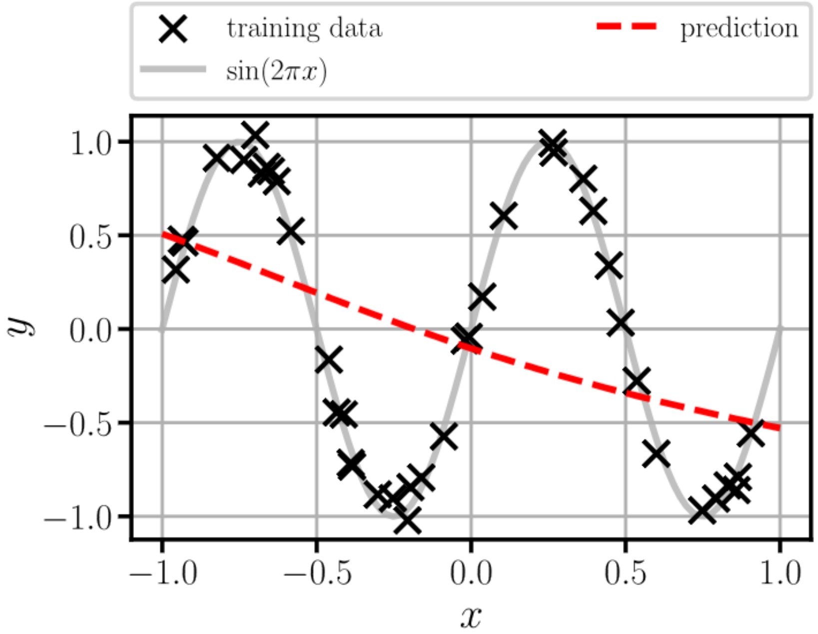


3.8 Approximating the Sine Function – Results

5‘000 epochs with Adam at $\alpha = 0.1$



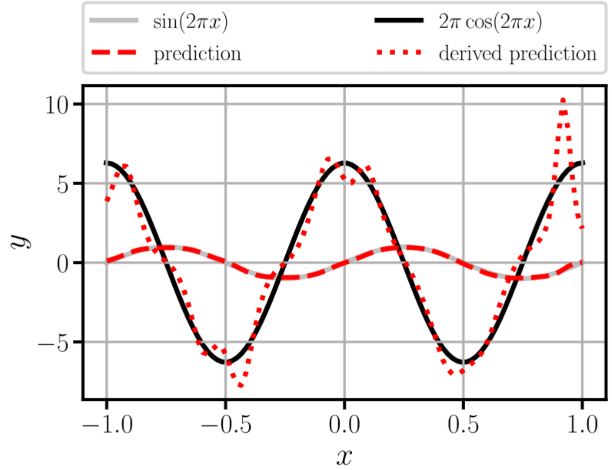
5‘000 epochs with SGD

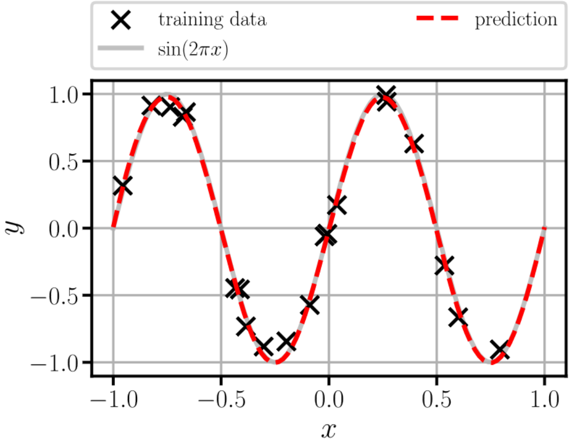


3.8 Approximating the Sine Function – Sobolev

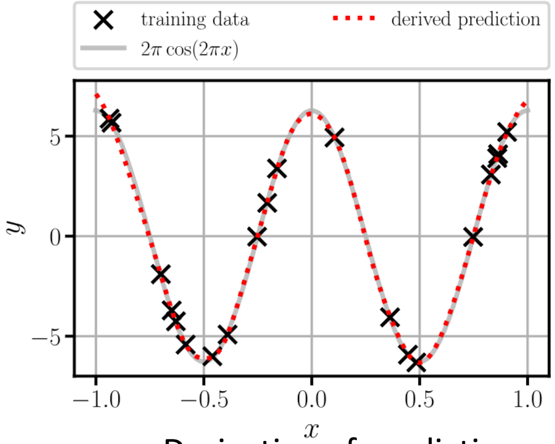
- Derivatives of predictions are typically inaccurate
- **Sobolev training** increases accuracy by incorporating derivatives in training data (higher order derivatives also improve)

```python
def costFunction(y, yPred, dy, dyPred):
     lossy = torch.mean((y – yPred) ** 2)
     lossdy = torch.mean((dy – dyPred) ** 2)
return lossy + lossdy
```





Prediction



Derivative of prediction

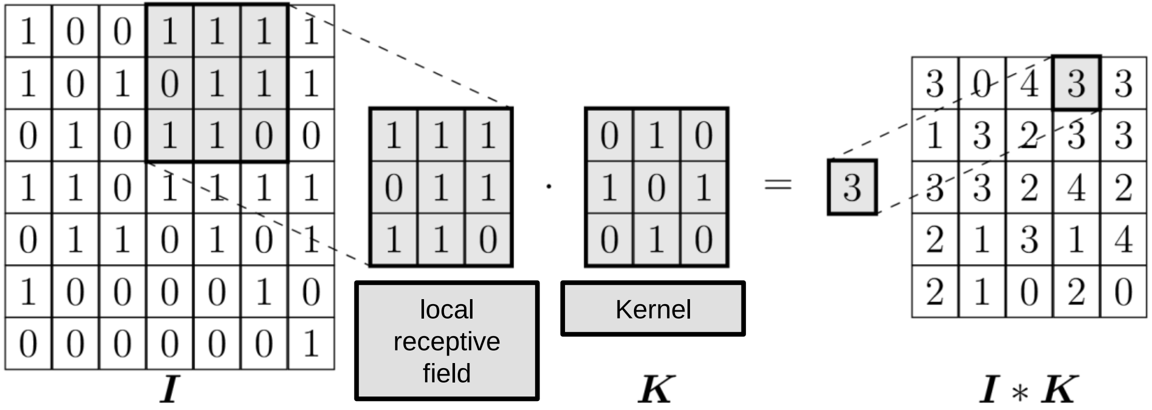
Exercises

E.7 Approximating the Sine Function (C)

- In PyTorch, build a fully connected neural network, that approximates the sine function.

3.9.1 Convolutional Neural Networks

- Originally designed for image processing
- Identify relative positions of features
- Operate with convolutional layers



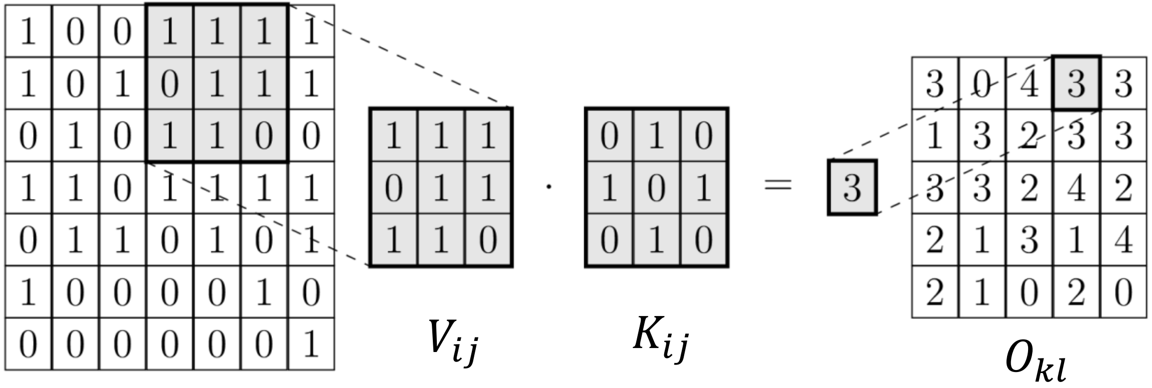
Convolutional layers

consist of filters/kernels $K(w)$ that are applied to an input field I and defined by

- Kernel size
- Number of kernels
- Stride length: how large is the shift of the kernel per step
- Padding: do you embed the input field?

In neural networks: The values in the kernel are not preset. They are learnt by the network and represent the weights.

3.9.1 Convolutional Neural Networks



The convolution $*$ is computed as a combination of the multidimensional dot product, also known as tensordot product:

$$o = V_{ij} \cdot K_{ij}$$

Example (2D convolutional filter applied with a stride length of $s = 1$ and no padding)

$$1 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 = 3$$

This operation is then repeated by shifting the view V_{ij} with the stride s

$$O_{kl} = \sum_{i,j} V_{(i+k)(j+l)} \cdot K_{ij} = V * K$$

3.9.1 Convolutional Neural Networks – Filters

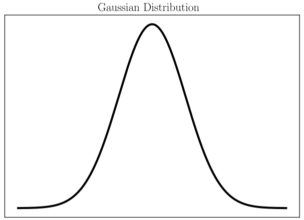
Example filters

Identity filter

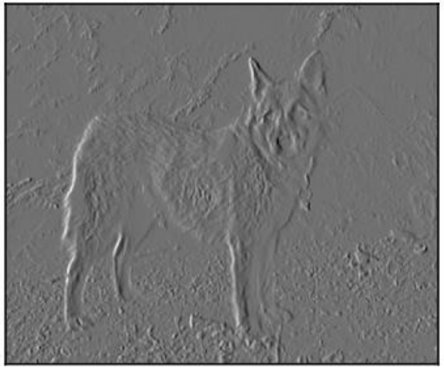
$$\boldsymbol{K} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, s = 1$$

Gaussian filter

$$\boldsymbol{K} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}, s = 1$$



Sobel filter in x-direction



$$\boldsymbol{K} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, s = 1$$

Averaging filter

$$\boldsymbol{K} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, s = 3$$

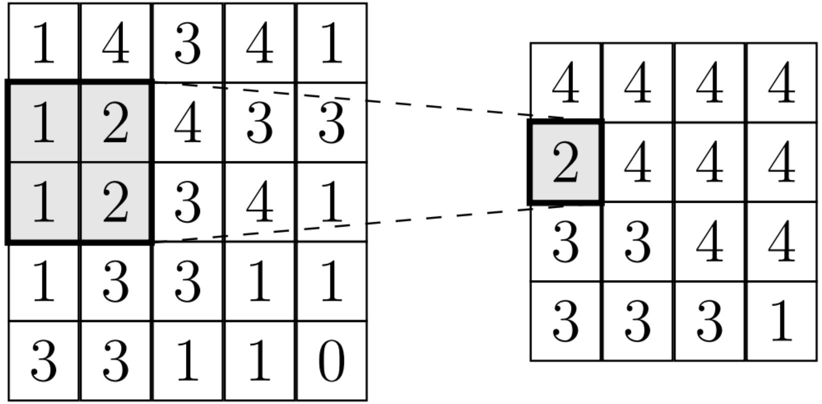
For other filters for gradient computation, see Chapter 9.

finite difference approximation	convolutional filter
$f'(x) = \frac{f(x+\Delta x)-f(x)}{\Delta x}$	$\frac{1}{\Delta x}[-1, 1]$
$f''(x) = \frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}$	$\frac{1}{\Delta x^2}[1, -2, 1]$

3.9.1 Convolutional Neural Networks – Pooling

Pooling layers reduce the spatial size by extracting relevant features (filtering!)

- Pooling types
  - Max Pooling
  - Average Pooling
- Pooling layers are defined by
  - Kernel size
  - Stride length
  - Padding

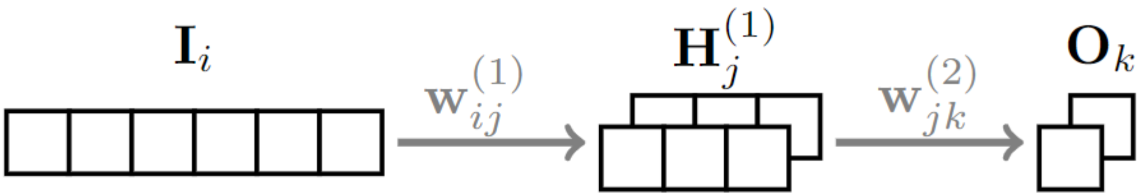


- Example “max pooling”(2D convolutional filter applied with a stride length of $s = 1$ and no padding)

$$\max\{1, 2, 1, 2\} = 2$$

3.9.1 Convolutional Neural Networks – Example

What if we want two features O from one input?



Example with 1D convolutional filters

- Stride length $s = 1$
- Without a bias
- Input (given)

$$\boldsymbol{I} = [0, 1, 3, 2, 5]$$

- Weights from input layer $\boldsymbol{I}$ to first hidden layer $\boldsymbol{H}^{(1)}$ (given)

$$\boldsymbol{w}_{11}^{(1)} = [0, 2, 1], \boldsymbol{w}_{12}^{(1)} = [1, 0, 3]$$

- Forward pass to first hidden layer $\boldsymbol{H}_j^{(1)} = \sum_i \boldsymbol{I}_i * \boldsymbol{w}_{ij}^{(1)}$

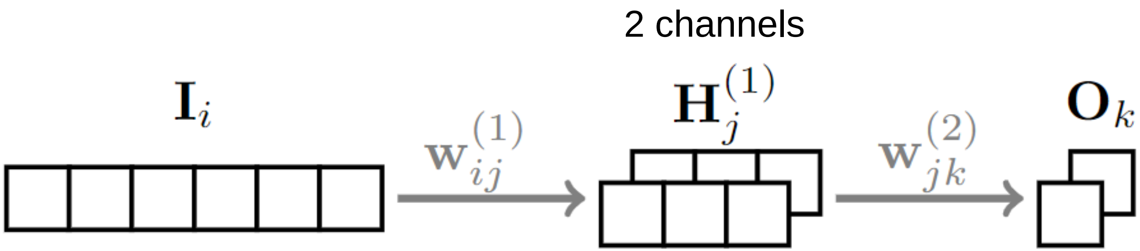
$$\boldsymbol{H}_1^{(1)} = [0 \cdot 0 + 1 \cdot 2 + 3 \cdot 1, 1 \cdot 0 + 3 \cdot 2 + 2 \cdot 1, 3 \cdot 0 + 2 \cdot 2 + 5 \cdot 1] = [5, 8, 9]$$

$$\boldsymbol{H}_2^{(1)} = [0 \cdot 1 + 1 \cdot 0 + 3 \cdot 3, 1 \cdot 1 + 3 \cdot 0 + 2 \cdot 3, 3 \cdot 1 + 2 \cdot 0 + 5 \cdot 3] = [9, 7, 18]$$

3.9.1 Convolutional Neural Networks – Example

Example with 1D convolutional filters

- Stride length $s = 1$
- Without a bias
- Hidden layer $\boldsymbol{H}^{(1)}$ (given)



$$\boldsymbol{H}_1^{(1)} = [5, 8, 9]$$
$$\boldsymbol{H}_2^{(1)} = [9, 7, 18]$$

- Weights from first hidden layer $\boldsymbol{H}^{(1)}$ to output layer $\boldsymbol{O}_k = \sum_j \boldsymbol{H}_j^{(1)} * \boldsymbol{w}_{jk}^{(2)}$ (given)

$$\boldsymbol{w}_{11}^{(2)} = [1, 0, 0], \boldsymbol{w}_{21}^{(2)} = [2, 1, 0]$$
$$\boldsymbol{w}_{12}^{(2)} = [0, 0, 3], \boldsymbol{w}_{22}^{(2)} = [0, 1, 1]$$

- Forward pass to output layer

$$\boldsymbol{O}_1 = [(5 \cdot 1 + 8 \cdot 0 + 9 \cdot 0) + (9 \cdot 2 + 7 \cdot 1 + 18 \cdot 0)] = 30$$
$$\boldsymbol{O}_2 = [(5 \cdot 0 + 8 \cdot 0 + 9 \cdot 3) + (9 \cdot 0 + 7 \cdot 1 + 18 \cdot 1)] = 52$$

3.9.1 Convolutional Neural Networks – Depth

ImageNet Classification with Deep Convolutional Neural Networks, (Alex) Krizhevsky et al. 2012

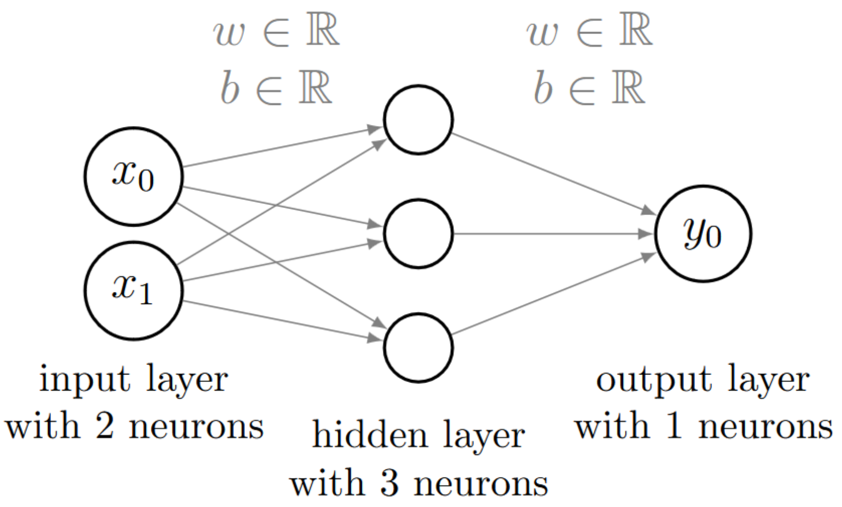
Comparison to fully connected neural network

Number of parameters per weight/kernel

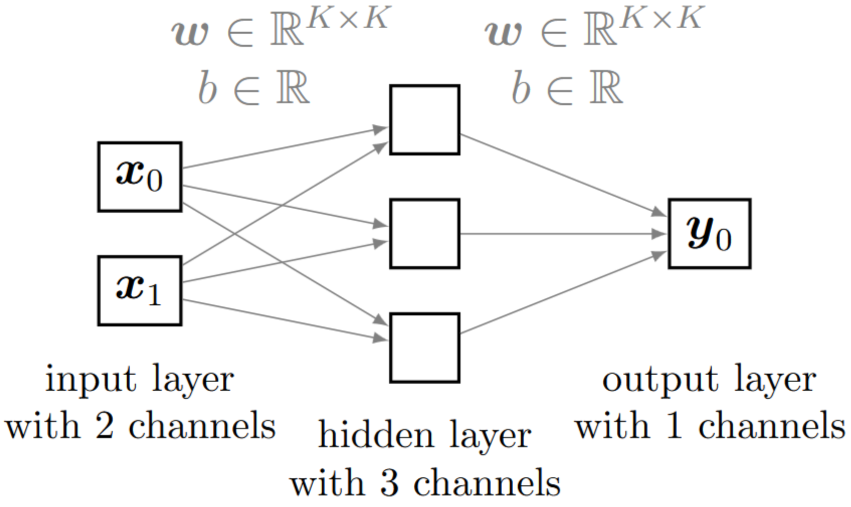
Number of connections (weights)

Number of biases

- Channels are analogous to neurons
- Filters are analogous to linear transformations (connections)
- Number of parameters in fully connected neural network: $1 \cdot (2 \cdot 3 + 3 \cdot 1) + (3 + 1)$
- Number of parameters in convolutional neural network: $(K \cdot K) \cdot (2 \cdot 3 + 3 \cdot 1) + (3 + 1)$



Fully connected neural network



Convolutional neural network

3.9.1 Convolutional Neural Networks – Padding

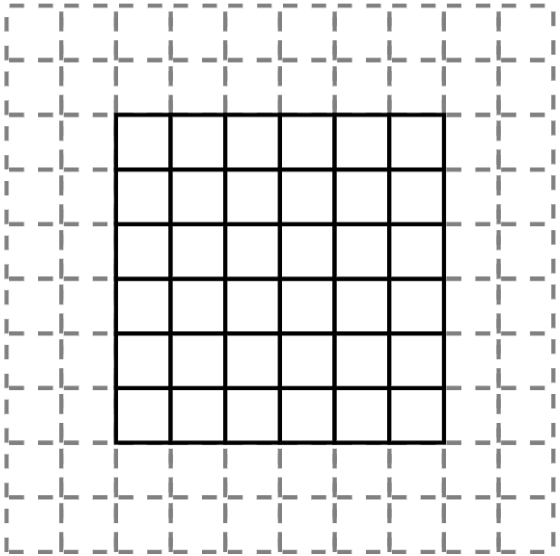
Padding counteracts the shrinking of the output, padding can be used

- Padding types
  - Valid padding (No padding)
  - Zero padding
  - Reflection padding

- The output dimension O after a convolutional layer

$$O = \left\lfloor \frac{I - K + 2P}{s} \right\rfloor + 1$$

- Input dimension I
- Kernel size K
- Padding size P
- Stride length s

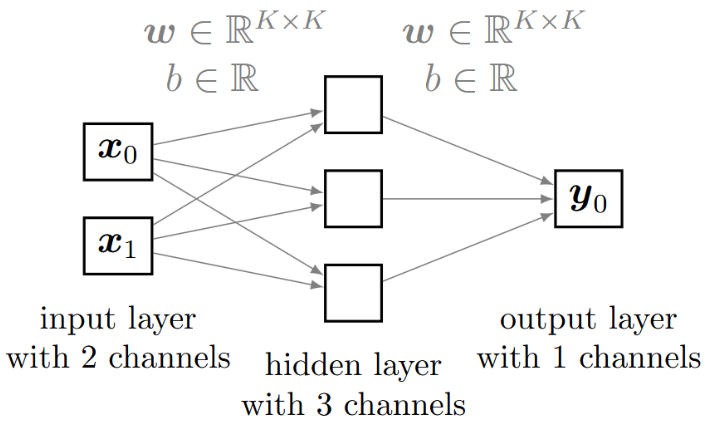


3.9.1 Convolutional Neural Networks – PyTorch

```
class CNN(torch.nn.Module):
    def __init__(self, inputImages, hiddenImages, outputImages):
        super(CNN, self).__init__()
        self.cnn1 = torch.nn.Conv2d(inputImages, hiddenImages, kernel_size=3,
                                    stride=1, padding=1)
        self.cnn2 = torch.nn.Conv2d(hiddenImages, outputImages, kernel_size=3,
                                    stride=1, padding=1)
        self.activation = torch.nn.ReLU()

    def forward(self, x):
        y = self.activation(self.cnn1(x))
        y = self.cnn2(y)
        return y


model = CNN(2, 3, 1) # input has size
                       (1, 2, imageSize, imageSize)
```



Exercises

E.11 Convolutional Neural Network (P & C)

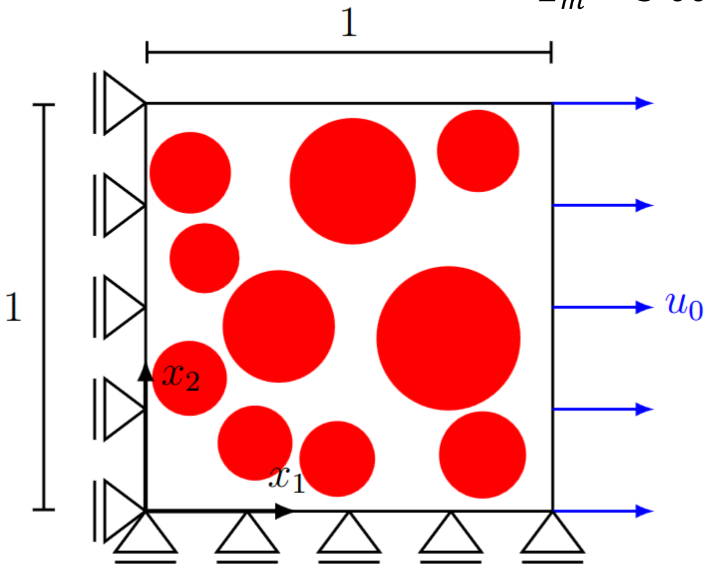
- Familiarize yourself with convolutional neural networks through pen-and-paper forward propagation and through the application of different convolutional filters to images (using PyTorch).

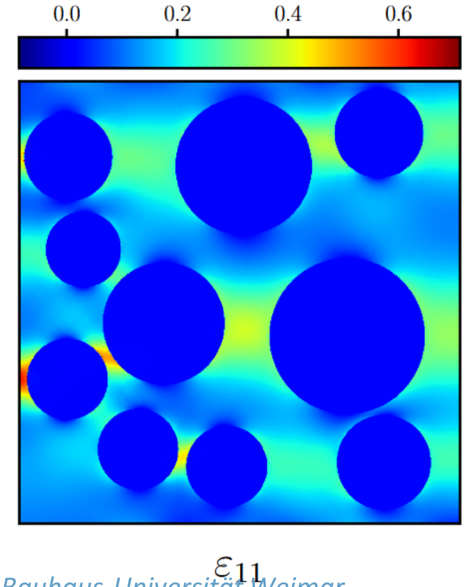
3.9.1 Convolutional Neural Networks – Example

Learning strain distributions from data

- Goal is to make the neural network predict the strain of a 2D domain under a uni-axial load
- Domain is defined by fiber material (red) and matrix material (white) with the elastic properties

$$E_f = 85'000, \nu_f = 0.22$$
$$E_m = 3'000, \nu_m = 0.4$$

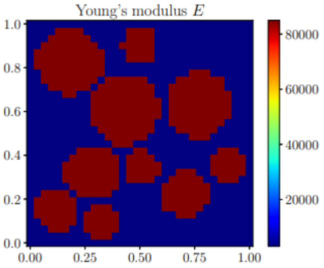


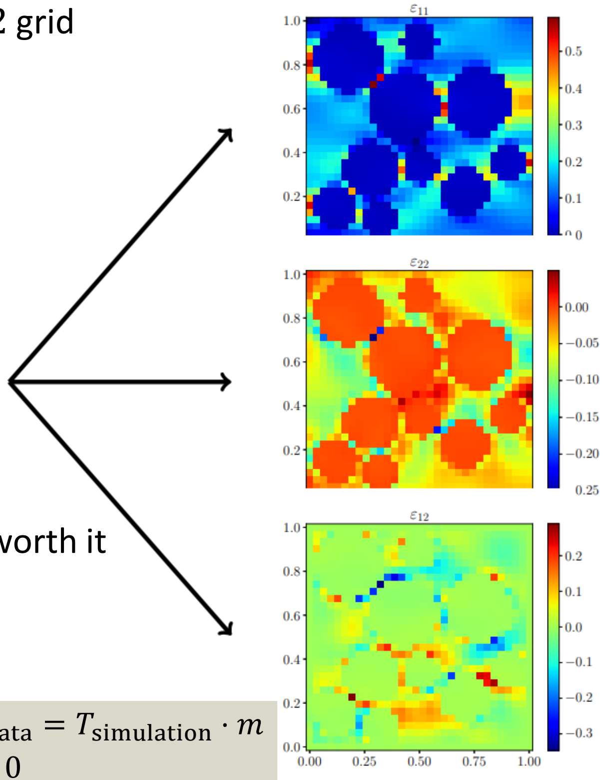


3.9.1 Convolutional Neural Networks – Example

- To reduce the complexity of the task, the domain is discretized in 32×32 grid
- Strain prediction from the underlying material distribution
- As neural network architecture, a U-Net (see Chapter 8) is used

U-Net: Convolutional Networks for Biomedical Image Segmentation, Ronneberger et al. 2015

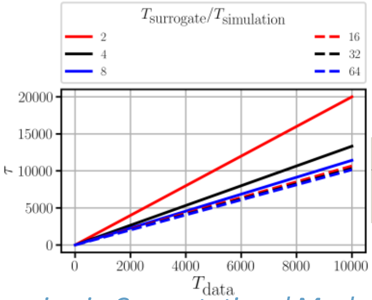




- When training a surrogate model, the breakeven threshold τ must be considered

$$\tau = \frac{T_{\text{data}} + T_{\text{train}}}{T_{\text{simulation}} + T_{\text{surrogate}}}$$

- Which represents the number of surrogate evaluations to make training worth it
- T is the time of
  - Data collection (T_{data})
  - Training (T_{train})
  - 1 Simulation ($T_{\text{simulation}}$)
  - 1 Surrogate evaluation ($T_{\text{surrogate}}$)



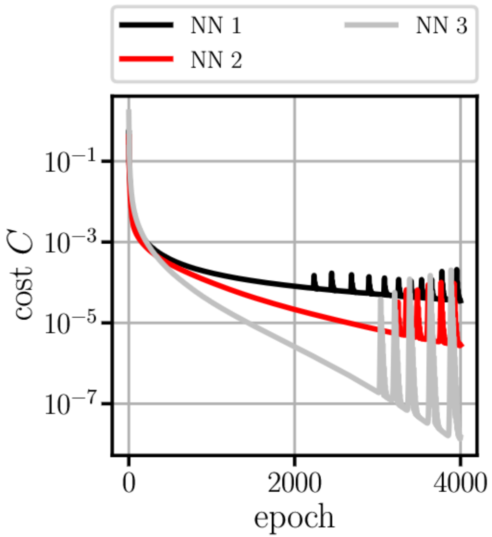
Assuming $T_{\text{data}} = T_{\text{simulation}} \cdot m$ and $T_{\text{train}} \approx 0$

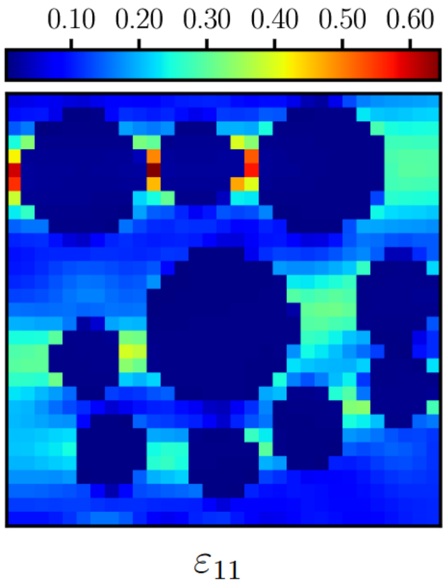
3.9.1 Convolutional Neural Networks – Example

Model selection (make sure that model can learn the training data by heart)

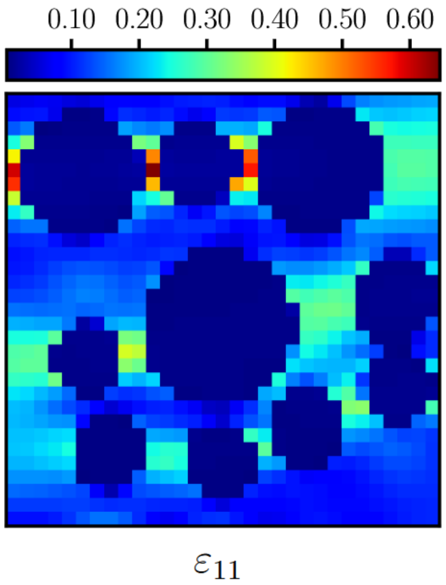
- NN 1: U-net with 5 convolutional layers: 94‘978 parameters
- NN 2: feed-forward convolutional neural network with 4 layers: 38‘405 parameters
- NN 3: U-net followed by 3 convolutional layers: 154‘471 parameters

- Training with 1 datapoint





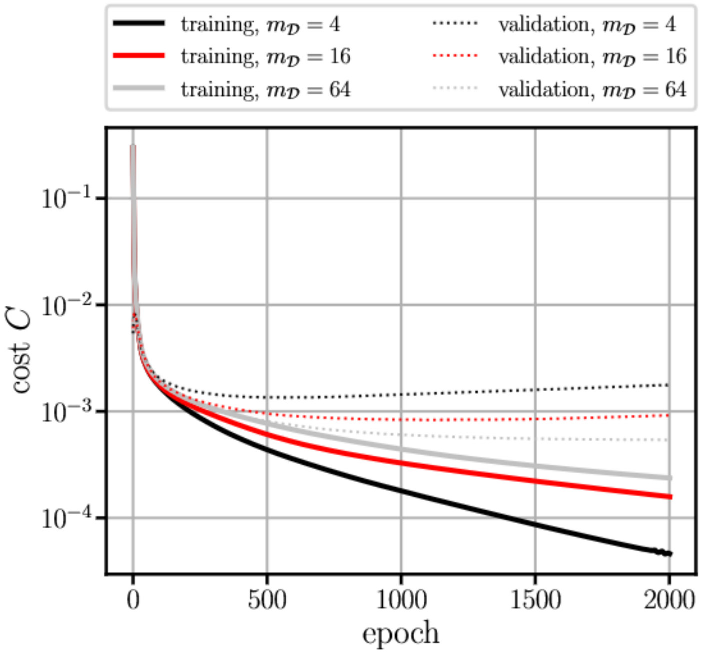
Prediction with NN 2



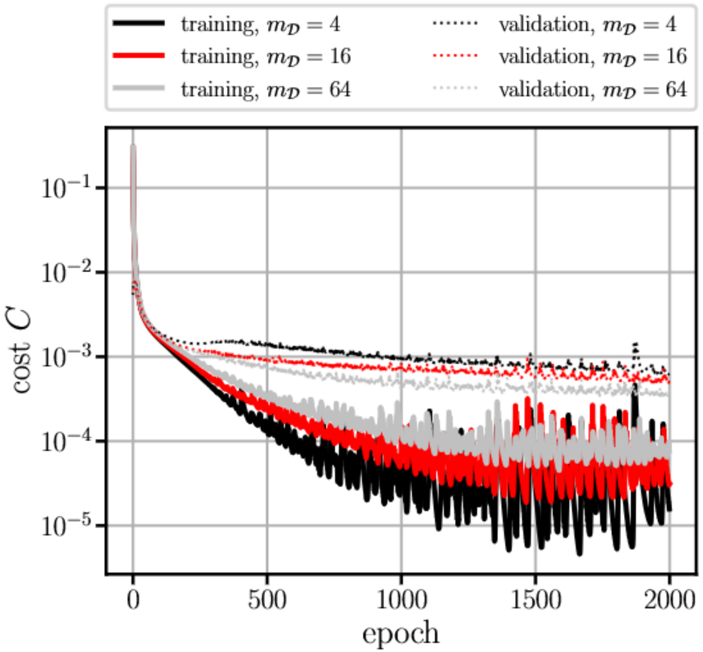
Ground truth

3.9.1 Convolutional Neural Networks – Example

With selected model (NN 2), increase the training data (and potentially tune the architecture & optimizer)



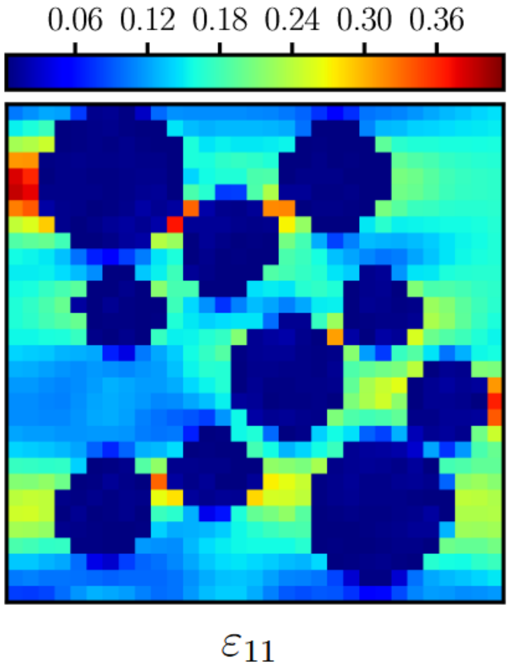
Without regularization



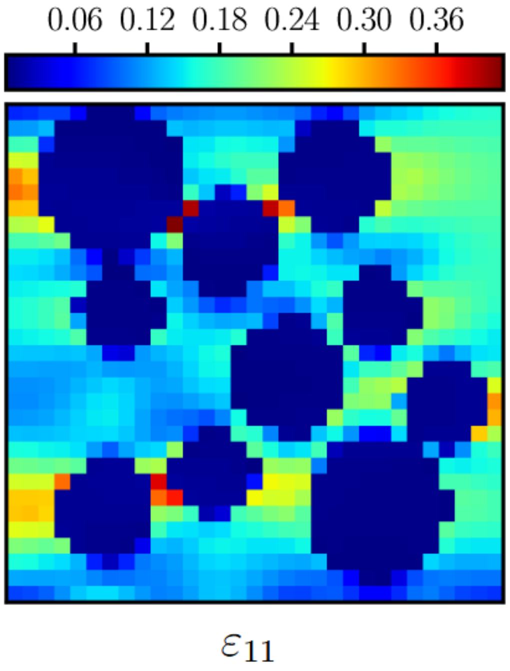
With regularization

3.9.1 Convolutional Neural Networks – Example

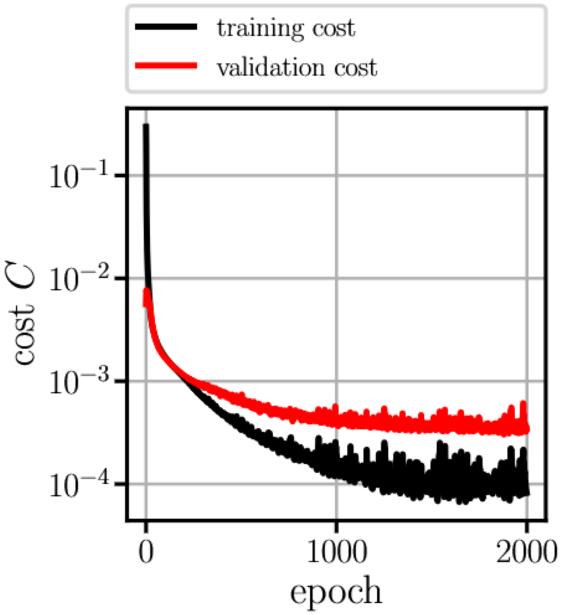
Prediction on unseen data (i.e., validation data)



Prediction



Ground truth

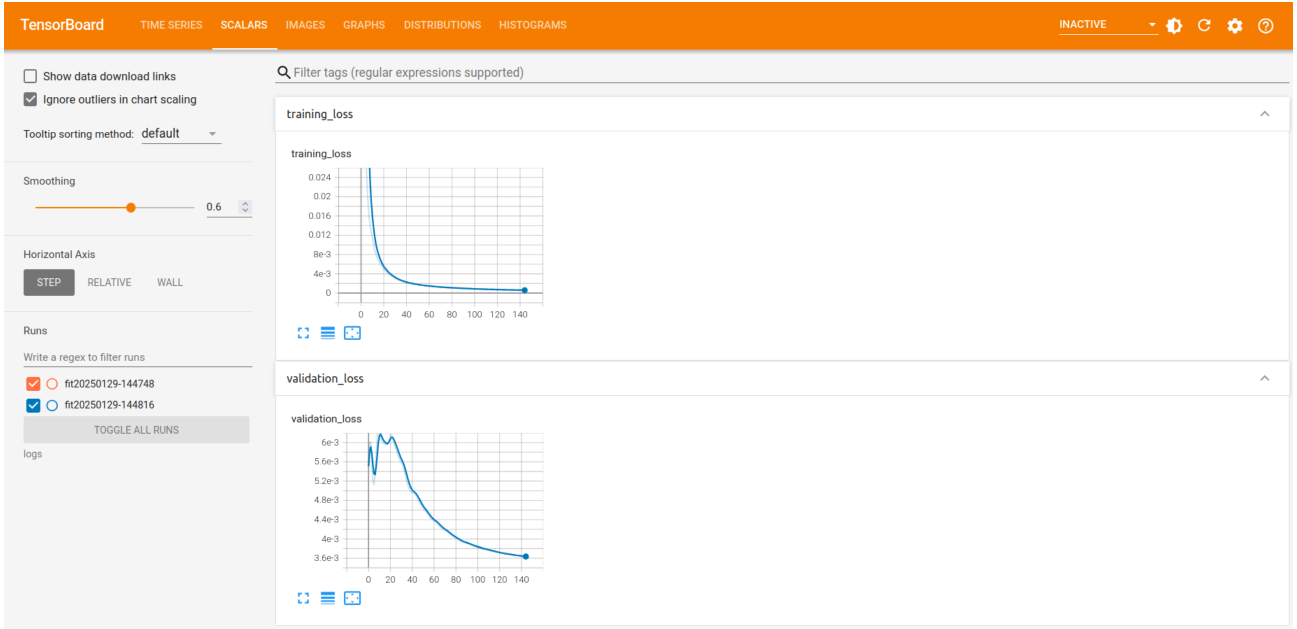


Learning history

3.9.1 Convolutional Neural Networks – TensorBoard

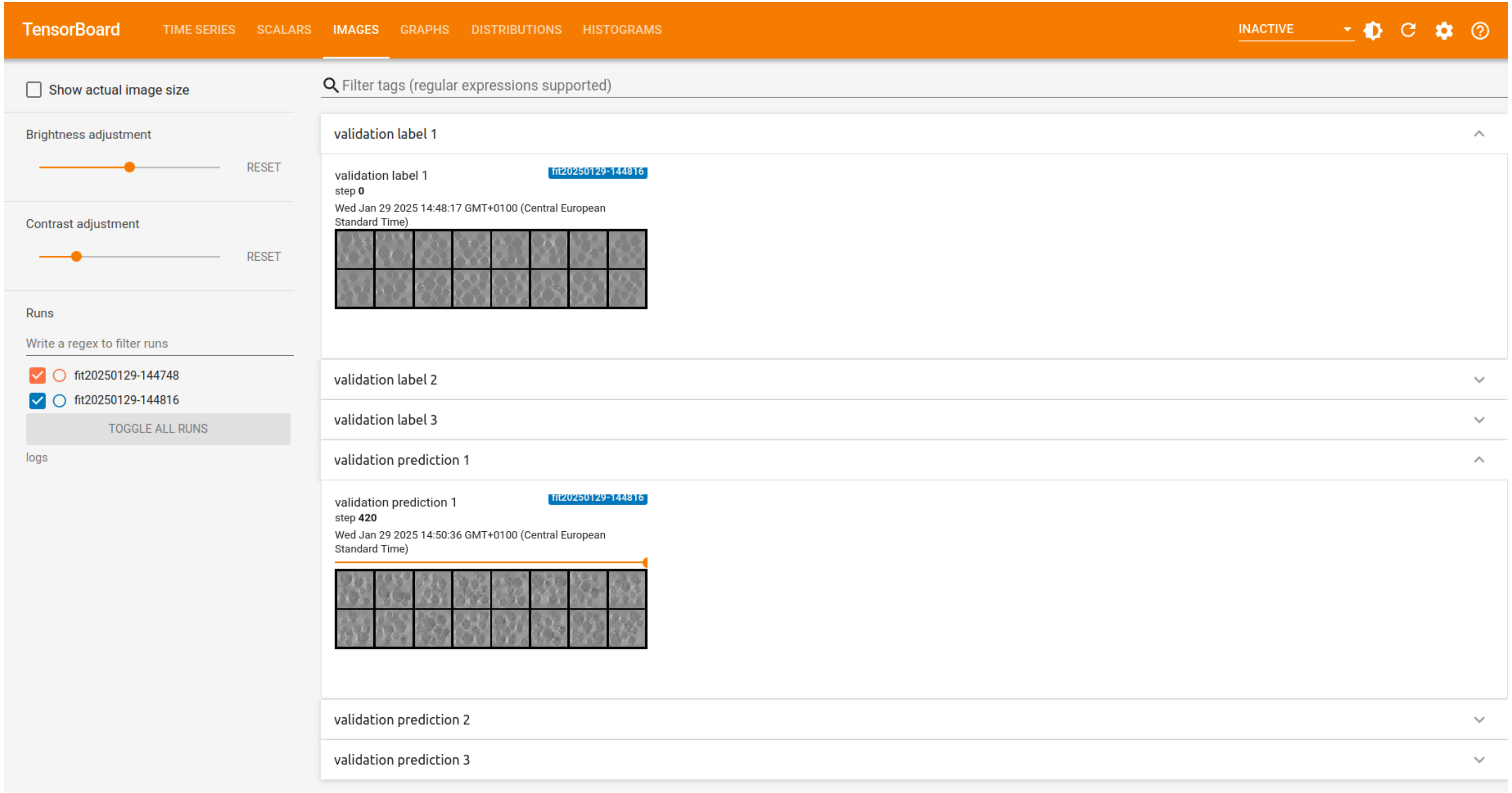
TensorBoard is a visualization tool for TensorFlow (& other machine learning libraries, e.g., PyTorch)

- Beneficial for debugging and hyperparameter tuning/model selection
- Includes features such as graphs, histograms, output visualization



3.9.1 Convolutional Neural Networks – TensorBoard

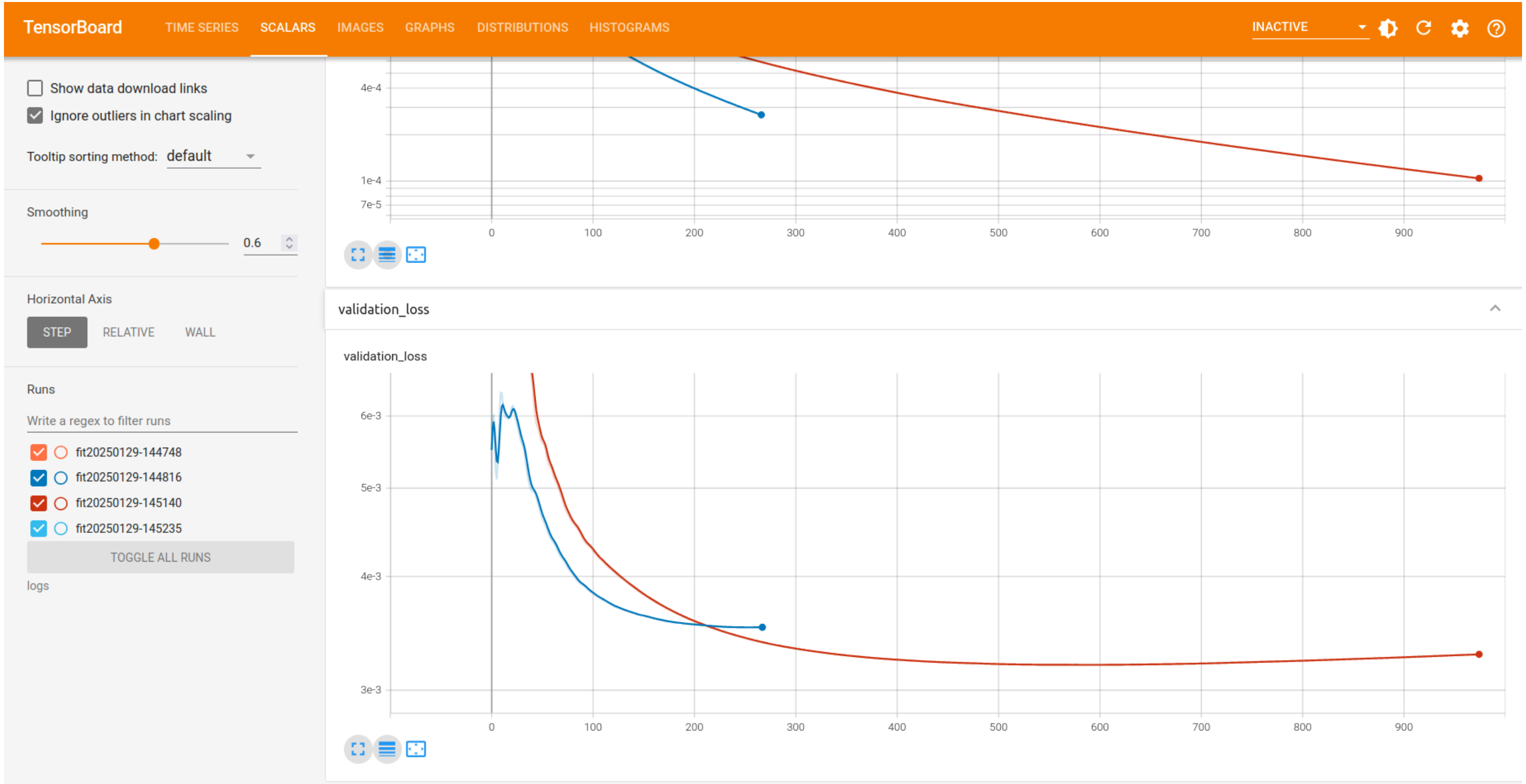
Track the progress in prediction quality by comparing outputs with labels



3.9.1 Convolutional Neural Networks – TensorBoard

Bauhaus-Universität Weimar

Compare different architectures and hyperparameters through different training runs



Exercises

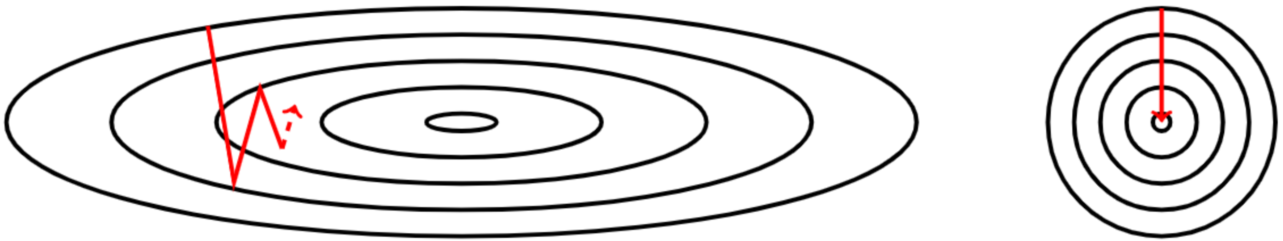
E.12 Learning Strain Distributions (C)

- Train a convolutional neural network as a surrogate, that maps the Young‘s modulus distribution to strain distributions. Experiment with different hyperparameters and different network architectures. For this, use TensorBoard.

Normalization

Normalization helps the optimization procedure

- Ensures, that features have similar scales



- Input normalization centering the mean to zero and establishing a unit standard deviation

$$\hat{x}_i = \frac{x_i - \mu}{\sigma}$$

- where

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \, , \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

Batch Normalization

Batch Normalization normalizes the activations between the layers to help the optimization

$$y = \frac{x - \mu(x)}{\sigma(x) + \epsilon} \cdot \gamma + \beta$$

Batch Normalization is composed of two parts

a) Centering the mean around zero and normalizing the standard deviation to a unit standard deviation

$$\tilde{x} = \frac{x - \mu(x)}{\sigma(x) + \epsilon}$$

b) scaling and shifting by an element-wise modification with the trainable parameters γ, β

$$y = \tilde{x} \cdot \gamma + \beta$$

- This increases the model capacity further and makes it possible for the network to learn an optimal normalization

Batch normalization is:

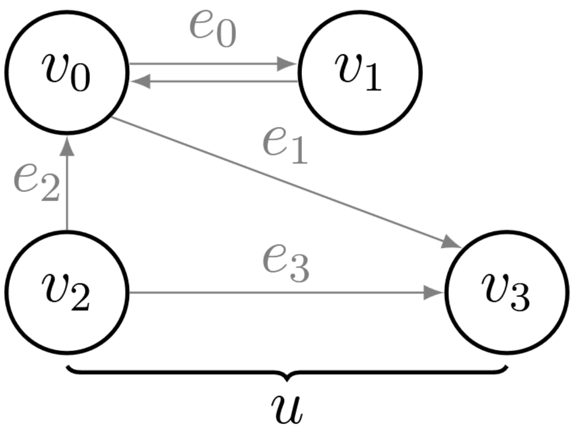
- typically applied after the activation function, but before is also possible
- Almost always encountered when using convolutional neural networks (alternate normalizations possible)

In PyTorch it can be defined with torch.nn.BatchNorm1d or torch.nn.BatchNorm2d

3.9.2 Graph Neural Networks

- Convolutional neural networks are limited to data aligned on rectangular & uniform grids
- **Graph neural networks** are a generalization to general graphs

- **Graph**
  - Nodes/vertices v_i
  - Edges e_i
  - Global attribute u



- Many different architectures exist, (message passing networks, graph convolutional networks, graph transformers…)
- We will focus on **message passing networks**, which consider the following invariant

$$v_i^s \xrightarrow{e_i} v_i^r$$

Relational inductive biases, deep learning, and graph networks, Battaglia et al. 2018

- Each edge e_i has a sender v_i^s and receiver node v_i^r

3.9.2 Graph Neural Networks

- Edges are updated through the **first** fully connected neural network $e_i' = f^e(e_i, v_i^s, v_i^r, u)$
- Nodes are updated by aggregating connecting edges $\bar{e}_i$ (e.g., max or sum): **second** fully connected neural network $v_i' = f^v(\bar{e}', v_i, u)$
- The global attribute is updated by aggregating all edges and nodes: **third** fully connected neural network $u' = f^u(\bar{e}', \bar{v}', u)$

- Application to mesh-based simulation

Learning Mesh-Based Simulation with Graph Networks, Pfaff et al. 2020

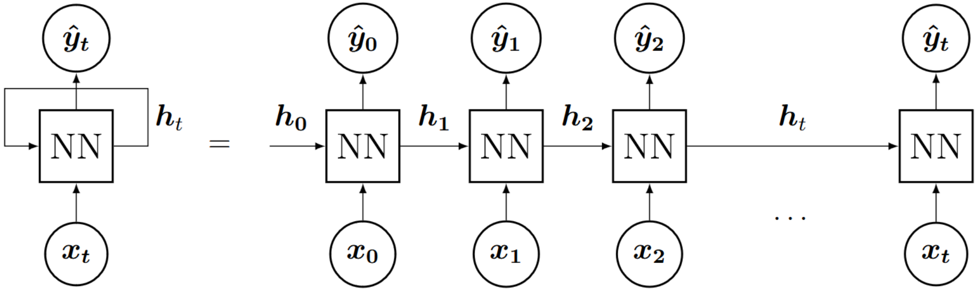
Algorithm 8 Graph block in a message passing neural network

Require: Graph consisting of nodes $\boldsymbol{v}$, edges $\boldsymbol{e}$ and a global attribute u.
  for all edges e_i **do**
    Update edges: $e_i' = f^e(e_i, v_i^s, v_i^r, u; \boldsymbol{\Theta}_e)$
  end for
  for all nodes v_i **do**
    Find all edges connecting to node v_i: $\boldsymbol{e}_i'$
    Aggregate adjacent edges: $\bar{e}_i' = \rho^{e \to v}(\boldsymbol{e}_i')$
    Update nodes: $v_i' = f^v(\bar{e}_i', v_i, u; \boldsymbol{\Theta}_v)$
  end for
  Aggregate all edges $\boldsymbol{e}$: $\bar{e}' = \rho^{e \to u}(\boldsymbol{e})$
  Aggregate all nodes $\boldsymbol{v}$: $\bar{v}' = \rho^{v \to u}(\boldsymbol{v})$
  Update global attribute: $u' = f^u(\bar{e}', \bar{v}', u; \boldsymbol{\Theta}_u)$

3.9.6 Recurrent Neural Networks

Recurrent Neural Networks reuse information from previous states by

- looping over itself to generate sequences.



- Equivalent to using multiple copies of the same network with a hidden state h_t (unrolled recurrent neural network)
$$h_t = \sigma(W_x x_t + b_x + W_h h_{t-1} + b_h)$$
- W_x, W_h are the weights and b_x, b_h the biases
- y_t is obtained by a (learnable) mapping between h_t and y_t
- Useful for applications with sequential data, such as speech recognition, language modeling, translation
- In practice, only able to connect recent information with each other, e.g., *"We used too many lemons for our lemonade ... We did not like the lemonade, it was too sour"* poses a problem, as the cause for the is too far from the outcome
- Overcome by long short-term memory networks (LSTMs), gated recurrent units (GRUs), and transformers

3.9.6 Recurrent Neural Networks - PyTorch

```
class RNN(torch.nn.Module):
    def __init__(self, inputSize, hiddenStateSize, outputSize):
        super(RNN, self).__init__()
        self.rnn = torch.nn.RNN(inputSize, hiddenStateSize, nonlinearity=‘relu’,
num_layers=2)
        self.linear = torch.nn.Linear(hiddenStateSize, outputSize)

    def forward(self, x):
        h, _ = self.rnn(x) # unrolls the RNN
        y = self.linear(h) # transforms hidden layer to output layer
        return y


inputSize = 1
hiddenStateSize = 500
outputSize = 1
model = CNN(inputSize, hiddenStateSize, outputSize)
```

Recurrent neural networks do not only have to be combined with fully connected neural networks, convolutional neural networks are for example also possible

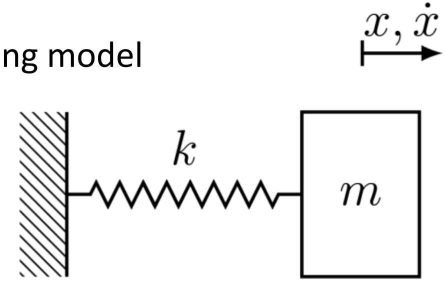
3.9.8 Physics-Inspired Architectures for Dynamics

- The previous architecture learn physical behavior only from data
- Governing laws can be incorporated into neural networks
  - Through training (via penalty terms in the cost function, i.e., weak enforcement/regularization; see Chapters 4 & 5)
  - Via the neural network architecture (by constraining the learnable space, i.e., strong enforcement; see Chapter 7)

- Strong enforcement of physics
  - Dynamics
    - **Hamiltonian neural networks**
    - **Lagrangian neural networks**
  - Constitutive modeling
    - Input-convex neural networks (see Chapter 7)
  - Boundary conditions
    - Strong enforcement of boundary conditions (see Chapters 4 & 5)

3.9.8 Physics-Inspired Architectures for Dynamics

Consider a one-dimensional mass-spring model



Desribed by the ordinary differential equation

$$m\ddot{x}(t) + kx(t) = 0$$

And the solution

$$x(t) = A\sin(\omega t + \phi)$$

Where $\omega = \sqrt{k/m}$ is the natural frequency and A, ϕ are determined by the initial conditions $x(0), \dot{x}(0)$

Alternatively the system can be described by

- Hamiltonian Mechanics
- Euler-Lagrange Equation

3.9.8.2 Hamiltonian Mechanics

In Hamiltonian mechanics, systems are described by coordinate pairs (position & momentum, i.e., $\boldsymbol{p}(t) = m\dot{\boldsymbol{x}}(t)$)

$$[\boldsymbol{x}(t), \boldsymbol{p}(t)]$$

Scalar Hamiltonian $\mathcal{H}\big(\boldsymbol{x}(t), \boldsymbol{p}(t)\big)$ represents the system's total energy Π_{tot} and fulfills

$$\frac{d\boldsymbol{x}}{dt} = \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}},$$

$$\frac{d\boldsymbol{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \boldsymbol{x}}$$

Given the Hamiltonian, the time evolution of $\boldsymbol{x}, \boldsymbol{p}$ can be computed via integration

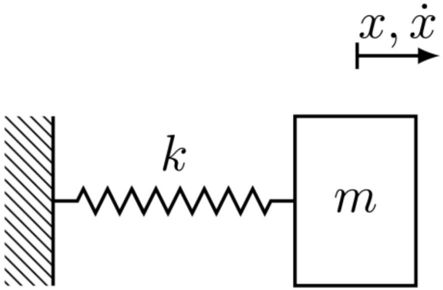
$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \int_t^{t+\Delta t} \left.\frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}\right|_\tau d\tau$$

$$\boldsymbol{p}(t + \Delta t) = \boldsymbol{p}(t) - \int_t^{t+\Delta t} \left.\frac{\partial \mathcal{H}}{\partial \boldsymbol{x}}\right|_\tau d\tau$$

Which can, e.g., be discretized with a forward Euler scheme

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \left.\frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}\right|_t \Delta t, \qquad \boldsymbol{p}(t + \Delta t) = \boldsymbol{p}(t) - \left.\frac{\partial \mathcal{H}}{\partial \boldsymbol{x}}\right|_t \Delta t$$

3.9.8.3 Hamiltonian Neural Networks



For the one-dimensional mass-spring model, the Hamiltonian is known as

$$\mathcal{H} = \Pi_{\text{tot}} = \Pi_{\text{kin}} + \Pi_{\text{pot}} = \frac{1}{2} m\dot{x}^2 + \frac{1}{2} kx^2$$

But what if the Hamiltonian is unknown, but the system must obey Hamiltonian mechanics?

Hamiltonian neural networks

- Data triplets $\{\tilde{\boldsymbol{x}}(t_i), \dot{\tilde{\boldsymbol{x}}}(t_i), \ddot{\tilde{\boldsymbol{x}}}(t_i)\}_{i=1}^{m}$ (enable computation of $\tilde{\boldsymbol{p}}(t_i), \dot{\tilde{\boldsymbol{p}}}(t_i)$)
- Mapping to the Hamiltonian is to be learned $\widehat{\mathcal{H}} = \mathcal{H}(\boldsymbol{x}(t), \boldsymbol{p}(t); \boldsymbol{\Theta})$
- From $\widehat{\mathcal{H}}$, $d\widehat{\boldsymbol{x}}/dt$ and $d\widehat{\boldsymbol{p}}/dt$ can be computed $\frac{d\boldsymbol{x}}{dt} = \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}, \frac{d\boldsymbol{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \boldsymbol{x}}$
- A comparison of $d\widehat{\boldsymbol{x}}/dt$ and $d\widehat{\boldsymbol{p}}/dt$ to the corresponding data in the data triplets, enables a cost function

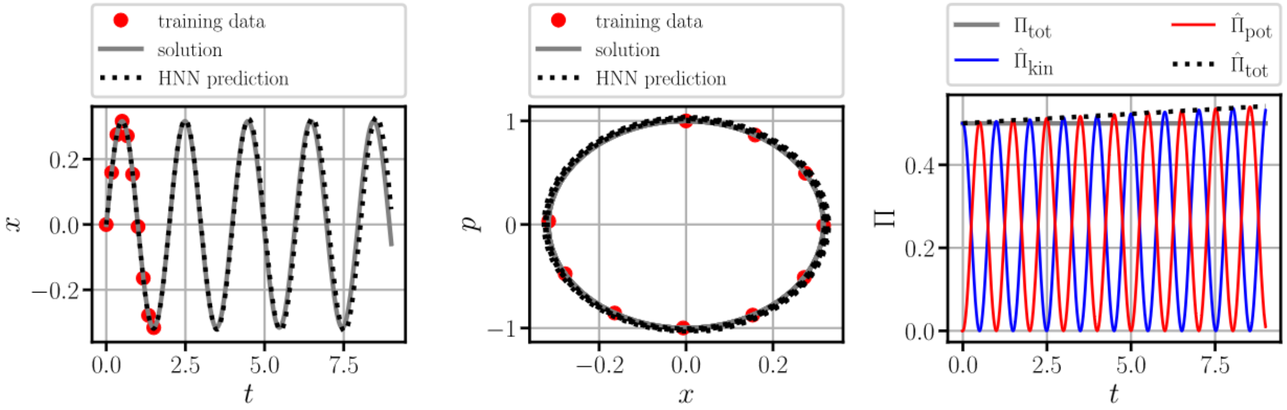
$$C = \frac{1}{m} \sum_{i=1}^{m} \left(\left\| \frac{d\tilde{\boldsymbol{x}}(t_i)}{dt} - \frac{\partial \widehat{\mathcal{H}}(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{p}}(t_i); \boldsymbol{\Theta})}{\partial \tilde{\boldsymbol{p}}(t_i)} \right\|^2 + \left\| \frac{d\tilde{\boldsymbol{p}}(t_i)}{dt} + \frac{\partial \widehat{H}(\tilde{\boldsymbol{x}}(t_i), \tilde{\boldsymbol{p}}(t_i); \boldsymbol{\Theta})}{\partial \tilde{\boldsymbol{x}}(t_i)} \right\|^2 \right)$$

- Prediction of the trajectory $\widehat{\boldsymbol{x}}(t)$ beyond the datapoints seen in the data triplets, possible through forward Euler

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \left. \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}} \right|_t \Delta t, \qquad \boldsymbol{p}(t + \Delta t) = \boldsymbol{p}(t) - \left. \frac{\partial \mathcal{H}}{\partial \boldsymbol{x}} \right|_t \Delta t$$

3.9.8.3 Hamiltonian Neural Networks - Results

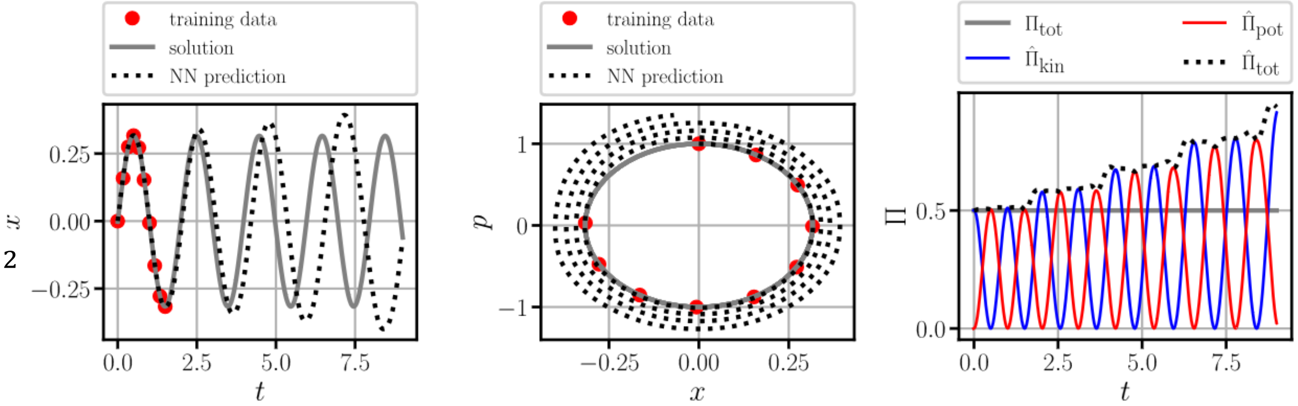
Hamiltonian neural network



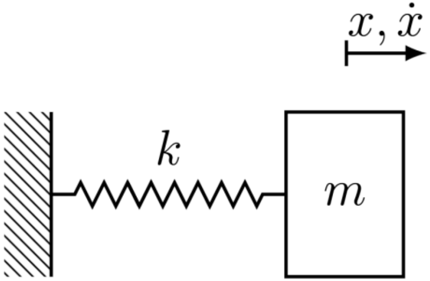
Standard neural network

$$\begin{pmatrix} \hat{\dot{x}} \\ \hat{\dot{p}} \end{pmatrix} = f(x, p; \boldsymbol{\Theta})$$

$$C = \frac{1}{m} \sum_{i=1}^{m} (\tilde{\dot{x}} - \hat{\dot{x}})^2 + (\tilde{\dot{p}} - \hat{\dot{p}})^2$$



3.9.8.4 Euler-Lagrange Equation



The **Lagrangian framework** offers a more general framework than Hamiltonian mechanics

The Lagrangian of the mass-spring system is defined as

$$L = \Pi_{\text{kin}} - \Pi_{\text{pot}} = \frac{1}{2} m \dot{x}^2 - \frac{1}{2} k x^2$$

Which obeys the Euler-Lagrange equation (ensuring the principle of least action)

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{x}} = \frac{\partial L}{\partial x}$$

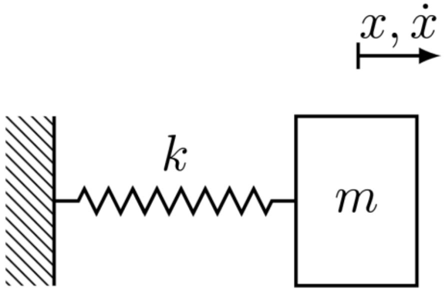
The acceleration $\ddot{x}$ can be obtained by rewriting the Euler-Lagrange equation

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{x}} = \frac{\partial^2 L}{\partial x \partial \dot{x}} \dot{x} + \frac{\partial^2 L}{\partial \dot{x}^2} \ddot{x} = \frac{\partial L}{\partial x}$$

$$\ddot{x} = \left(\frac{\partial^2 L}{\partial \dot{x}^2} \right)^{-1} \left(\frac{\partial L}{\partial x} - \frac{\partial^2 L}{\partial x \partial \dot{x}} \dot{x} \right)$$

Similar as for the Hamiltonian mechanics, the evolution of $x(t)$ can be computed by integrating $\dot{x}, \ddot{x}$

3.9.8.5 Lagrangian Neural Networks

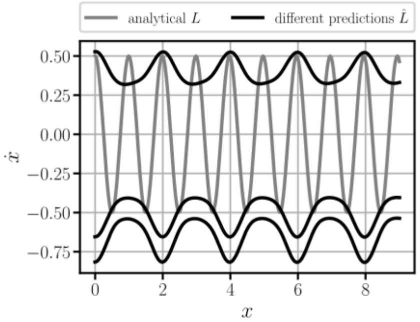


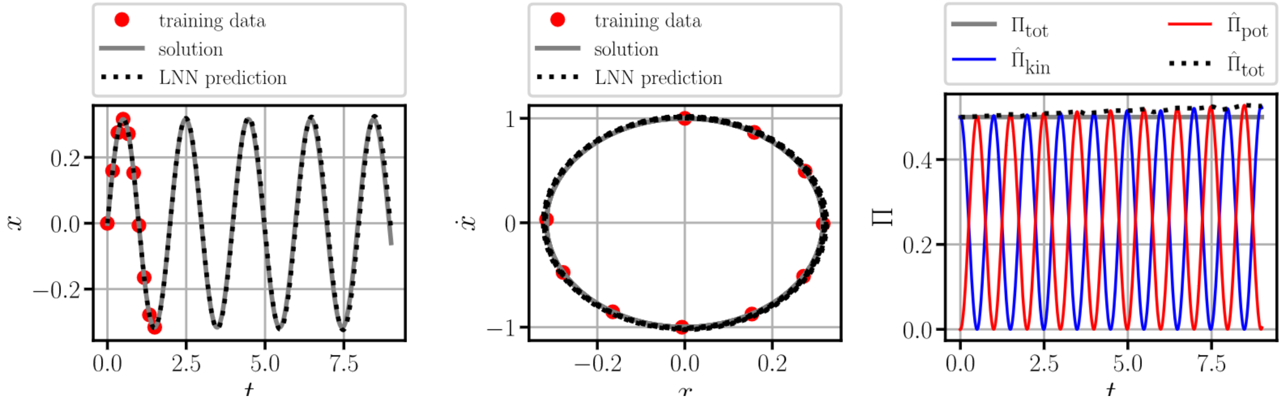
- Data triplets $\{\tilde{x}(t_i), \tilde{\dot{x}}(t_i), \tilde{\ddot{x}}(t_i)\}_{i=1}^{m}$
- Mapping to the Lagrangian is to be learned $\hat{L} = L(x(t), \dot{x}(t); \boldsymbol{\Theta})$
- The acceleration $\hat{\ddot{x}}$ can be computed from $\ddot{x} = \left(\frac{\partial^2 L}{\partial \dot{x}^2}\right)^{-1} \left(\frac{\partial L}{\partial x} - \frac{\partial^2 L}{\partial x \partial \dot{x}} \dot{x}\right)$
- With the predicted acceleration a cost function can be formulated as

$$C = \frac{1}{m} \sum_{i=1}^{m} \left\| \tilde{\ddot{x}}(t_i) - \hat{\ddot{x}}\left(\hat{L}; \tilde{x}(t_i), \tilde{\dot{x}}(t_i)\right) \right\|^2$$

3.9.8.5 Lagrangian Neural Networks - Results

Lagrangian neural network



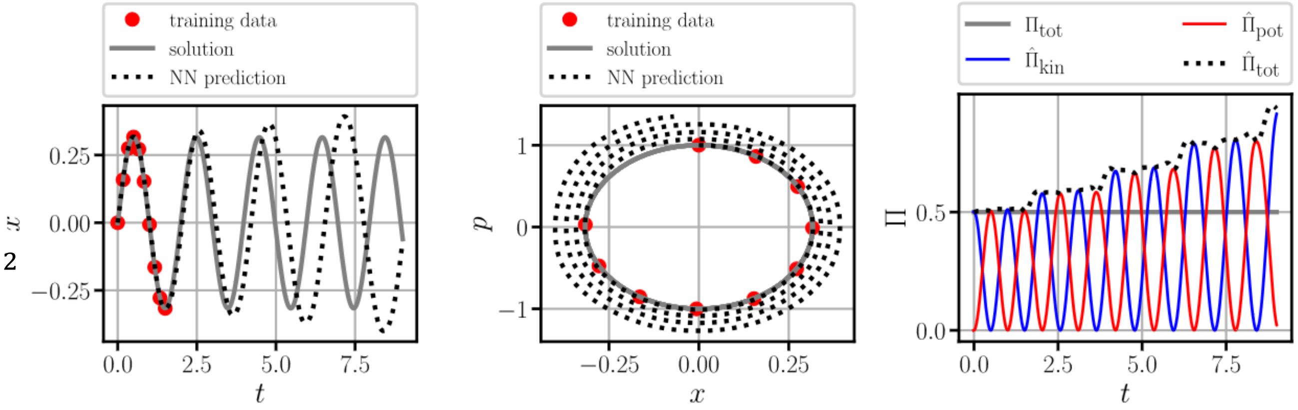


The Lagrangian is not unique

Standard neural network

$$\begin{pmatrix} \hat{\dot{x}} \\ \hat{\dot{p}} \end{pmatrix} = f(x, p; \mathbf{\Theta})$$

$$C = \frac{1}{m} \sum_{i=1}^{m} (\tilde{\dot{x}} - \hat{\dot{x}})^2 + (\tilde{\dot{p}} - \hat{\dot{p}})^2$$



Exercises

E.13 Hamiltonian & Lagrangian Neural Networks (C)

- Implement a Hamiltonian and a Lagrangian neural network and apply it to the mass-spring model. Compare the generalization capabilities to a standard neural network.

Contents

3 Neural Networks: Advanced Architectures

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