Model order reduction using artificial neural networks

Hendrik Kleikamp

October 7, 2021 – Afternoon session

Institute for Analysis and Numerics - WWU Münster

Mathematical background

- **Starting point:** Given a full-order model $\mu \mapsto u(\mu)$ and a reduced space V_N with orthonormal basis Ψ_N, how to compute a fast and reliable reduced model without affine decomposition of operators (i.e. without using EIM for instance)?

Mathematical background

- **Starting point:** Given a full-order model $\mu \mapsto u(\mu)$ and a reduced space V_N with orthonormal basis Ψ_N, how to compute a fast and reliable reduced model without affine decomposition of operators (i.e. without using EIM for instance)?

- **Goal:** Approximate the map $\pi_N : \mathcal{P} \to \mathbb{R}^N$, given as

$$\pi_N(\mu) = u_{\text{proj}}(\mu) \in \mathbb{R}^N,$$

where $u_{\text{proj}}(\mu)$ holds the coefficients of the **orthogonal** projection of the full-order solution $u(\mu)$ onto the reduced space V_N.

Mathematical background

- **Starting point:** Given a full-order model $\mu \mapsto u(\mu)$ and a reduced space V_N with orthonormal basis Ψ_N, how to compute a fast and reliable reduced model without affine decomposition of operators (i.e. without using EIM for instance)?

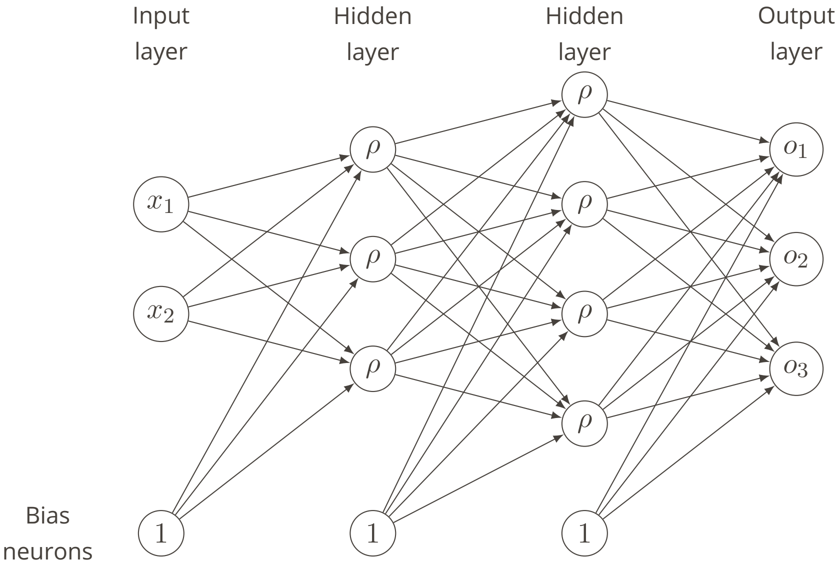
- **Goal:** Approximate the map $\pi_N \colon \mathcal{P} \to \mathbb{R}^N$, given as

$$\pi_N(\mu) = u_{\text{proj}}(\mu) \in \mathbb{R}^N,$$

  where $u_{\text{proj}}(\mu)$ holds the coefficients of the **orthogonal** projection of the full-order solution $u(\mu)$ onto the reduced space V_N.

- **Idea:** Use a neural network to approximate π_N.

Example: Feed-forward neural networks



Definition: Feed-forward neural networks

Components of a neural network Φ:

- Number of layers L
- Number of neurons $N_1, \dots, N_{L-1}$ in the $L-1$ hidden layers
- Input size N_0; Output size N_L
- Matrices $A_1, \dots, A_L$ with $A_i \in \mathbb{R}^{N_i \times N_{i-1}}$ (weights)
- Vectors $b_1, \dots, b_L$ with $b_i \in \mathbb{R}^{N_i}$ (biases)
- Activation function $\rho\colon \mathbb{R} \to \mathbb{R}$ (ReLU, $\tanh$, $\dots$)

Definition: Feed-forward neural networks

Components of a neural network Φ:

- Number of layers L
- Number of neurons $N_1, \dots, N_{L-1}$ in the $L-1$ hidden layers
- Input size N_0; Output size N_L
- Matrices $A_1, \dots, A_L$ with $A_i \in \mathbb{R}^{N_i \times N_{i-1}}$ (weights)
- Vectors $b_1, \dots, b_L$ with $b_i \in \mathbb{R}^{N_i}$ (biases)
- Activation function $\rho\colon \mathbb{R} \to \mathbb{R}$ (ReLU, $\tanh$, …)

For an input $x \in \mathbb{R}^{N_0}$, the output of Φ is given by

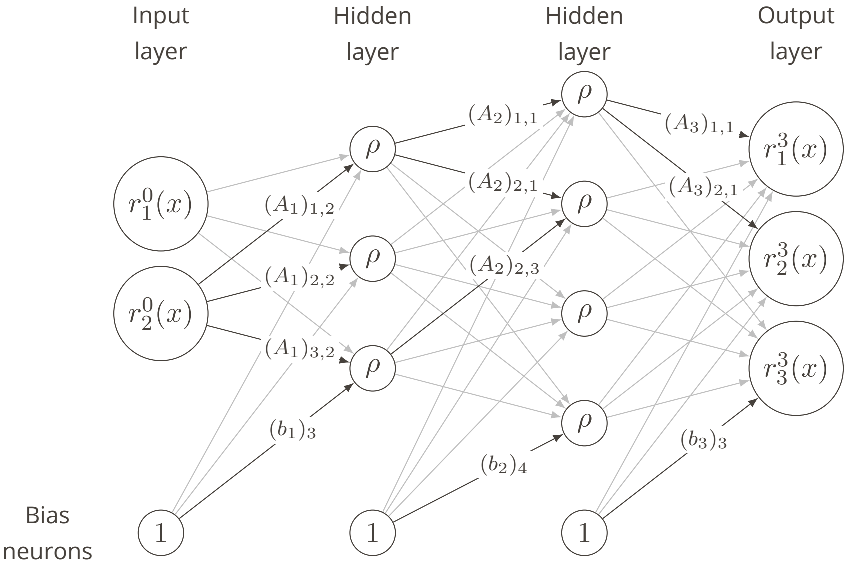
$$\Phi(x) = A_L r^{L-1}(x) + b_L,$$

where

$$\begin{aligned} r^i(x) &= \rho^*(A_i r^{i-1}(x) + b_i), \qquad i = 1, \dots, L-1, \\ r^0(x) &= x, \end{aligned}$$

with ρ^* being the component-wise application of ρ.

Visualization of the definition: Feed-forward neural networks



Neural network training in a nutshell

- **Task:** Adjust weights and biases such that Φ approximates a function f.
- **Problem:** Typically, f is not given explicitly, but we can only compute samples of f.

Neural network training in a nutshell

- **Task:** Adjust weights and biases such that Φ approximates a function f.
- **Problem:** Typically, f is not given explicitly, but we can only compute samples of f.
- **Idea:** Choose training set $x_1,\dots,x_n$, compute $f(x_1),\dots,f(x_n)$, and minimize error

$$E(\Phi) = \frac{1}{n}\sum_{i=1}^{n} l(\Phi(x_i), f(x_i))$$

  with respect to the weights and biases of Φ.
  Here, $l\colon \mathbb{R}^{N_L}\times\mathbb{R}^{N_L}\to\mathbb{R}$ denotes a loss function, for instance $l(y,\tilde{y}) = \|y-\tilde{y}\|_2^2$.
- **Optimization of E:** Compute gradient of E with respect to weights and biases using backpropagation and apply (variant of) gradient descent algorithm to gradually minimize E.

Pseudo code for neural network training

for $r = 1, \ldots, N_{\text{restarts}}$ **do**
  initialize weights and biases randomly
  for $e = 1, \ldots, N_{\text{epochs}}$ **do**
    create batches of training data
    `/* Training */`
    for $b = 1, \ldots, N_{\text{batches}}$ **do**
      apply optimization step with loss function for batch b
    end
    `/* Validation */`
    compute validation loss
    if early stopping criterion is fulfilled **then**
      stop training and perform next restart
    end
  end
  update neural network with lowest validation loss if necessary
end

General information on the implementation in pyMOR

- Reduced basis Ψ_N via POD of snapshots $u(\mu_1),\dots,u(\mu_n)$.
- Neural networks implemented and trained using PyTorch (`https://pytorch.org/`).
- Training with snapshots $u(\mu_i)$ projected on reduced space V_N:

$$\{(\mu_i,\underbrace{u_{\text{proj}}(\mu_i)}_{=\pi_N(u(\mu_i))})\in\mathcal{P}\times\mathbb{R}^N : i=1,\dots,n\}$$

- Validation phase to assess generalization ability.
- Early stopping and multiple restarts of training.
- Customizable training routine (optimizer, epochs, learning rate, ...).
- **Everything hidden in a reductor!**

Let's look at some code ...

```
class NeuralNetworkReductor(BasicObject):
    def __init__(self, fom, training_set, validation_set=None,
                 validation_ratio=0.1, basis_size=None,
                 rtol=0., atol=0., l2_err=0., pod_params=None,
                 ann_mse='like_basis'):
        ...

    def reduce(self, hidden_layers='[(N+P)*3, (N+P)*3]',
               activation_function=torch.tanh,
               optimizer=optim.LBFGS, epochs=1000,
               batch_size=20, learning_rate=1., restarts=10,
               seed=0):
        ...

    def reconstruct(self, u):
        ...
```

A bit more code ...

```python
def train_neural_network(training_data, validation_data,
                         neural_network, training_parameters={}):
    """Training algorithm for artificial neural networks.

    Trains a single neural network using the given training and
    validation data."""
    ...

def multiple_restarts_training(training_data, validation_data,
                               neural_network, target_loss=None,
                               max_restarts=10,
                               training_parameters={}, seed=None):
    """Algorithm that performs multiple restarts of neural network
       training.

    This method either performs a predefined number of restarts and
    returns the best trained network or tries to reach a given target
    loss and stops training when the target loss is reached."""
    ...
```

The NeuralNetworkReductor in action

```
from pymor.basic import *
# Set up the problem and the full-order model
problem = StationaryProblem(...)
fom, _ = discretize_stationary_cg(problem)
parameter_space = fom.parameters.space(...)
# Sample randomly for training and test set
training_set = parameter_space.sample_uniformly(...)
validation_set = parameter_space.sample_randomly(...)
# Set up reductor and compute the reduced-order model
from pymor.reductors.neural_network import NeuralNetworkReductor
reductor = NeuralNetworkReductor(fom, training_set, validation_set,
                                 l2_err=..., ann_mse=...)
rom = reductor.reduce(restarts=...)
```

More details:

https://docs.pymor.org/main/tutorial_mor_with_anns.html

Variants of the `NeuralNetworkReductor`

Instationary models:

Treat time as an additional parameter and apply same procedure as for stationary problems, i.e. approximate the map $(\mu, t) \mapsto u_N(\mu, t) \in \mathbb{R}^N$ by a neural network.

Usage shown in a demo:

`https://github.com/pymor/pymor/blob/2021.1.x/src/pymordemos/neural_networks_instationary.py`

Variants of the `NeuralNetworkReductor`

Instationary models:

Treat time as an additional parameter and apply same procedure as for stationary problems, i.e. approximate the map $(\mu, t) \mapsto u_N(\mu, t) \in \mathbb{R}^N$ by a neural network.

Usage shown in a demo:

`https://github.com/pymor/pymor/blob/2021.1.x/src/pymordemos/neural_networks_instationary.py`

Statefree outputs:

Learn map from parameter space to output directly without computing a (reduced) state, i.e. given an output functional $\mathcal{J}(\mu) := J(u(\mu), \mu)$, approximate $\mathcal{J}$ by a neural network instead of using a reduced order model to obtain $u_N(\mu)$ and computing $\mathcal{J}(\mu) \approx J(u_N(\mu), \mu)$ afterwards.

Usage shown in the tutorial:

`https://docs.pymor.org/main/tutorial_mor_with_anns.html`

All available neural network-based reductors

Stationary problems:

- `NeuralNetworkReductor`
  Approximates map from parameter to reduced state.
- `NeuralNetworkStatefreeOutputReductor`
  Approximates map from parameter to output. *

Instationary problems:

- `NeuralNetworkInstationaryReductor`
  Approximates map from parameter and time to reduced state.
- `NeuralNetworkInstationaryStatefreeOutputReductor`
  Approximates map from parameter and time to output. *

* **New in release 2021.1!**

Remarks on the method

Advantages:

- Non-intrusive method.
- Parameter separability is not required.
- Very fast during online computations.
- Orthogonal projection produces smaller error than Galerkin projection (constant from the Lemma of Céa).

Disadvantages:

- Neural network produces additional error (beside the error of the reduced space).
- Finding a proper neural network architecture is crucial to obtain good results.
- Training in the offline phase might be expensive.

References

Jan S. Hesthaven and Stefano Ubbiali: Non-intrusive reduced order modeling of non-linear problems using neural networks.
J. Comput. Phys., 363:55-78, 2018.

Qian Wang, Jan S. Hesthaven, and Deep Ray: Non-intrusive reduced order modeling of unsteady flows using artificial neural networks with application to a combustion problem.
J. Comput. Phys., 384:289-307, 2019.

Are there questions?