Model order reduction using artificial neural networks

Hendrik Kleikamp

August 25, 2022 – Morning session

Institute for Analysis and Numerics – WWU Münster

Mathematical background

- **Two scenarios:**
  1. Given a full-order model $\mu \mapsto u(\mu)$, but no affine decomposition of operators.
  2. Given only a set $\{(\mu_i, u(\mu_i))\}_{i=1}^n$ of snapshots with corresponding parameter values, i.e. purely data-driven setting.

Mathematical background

- **Two scenarios:**
  1. Given a full-order model $\mu \mapsto u(\mu)$, but no affine decomposition of operators.
  2. Given only a set $\{(\mu_i, u(\mu_i))\}_{i=1}^n$ of snapshots with corresponding parameter values, i.e. purely data-driven setting.
- **Goal:** Approximate the map $\pi_N : \mathcal{P} \to \mathbb{R}^N$, given as

$$\pi_N(\mu) = \underline{u}_N(\mu) \in \mathbb{R}^N,$$

where $\underline{u}_N(\mu)$ holds the coefficients of the **orthogonal** projection $u_N(\mu)$ of the full-order solution $u(\mu)$ onto the basis Ψ_N of a reduced space V_N.

Mathematical background

- **Two scenarios:**
  1. Given a full-order model $\mu \mapsto u(\mu)$, but no affine decomposition of operators.
  2. Given only a set $\{(\mu_i, u(\mu_i))\}_{i=1}^{n}$ of snapshots with corresponding parameter values, i.e. purely data-driven setting.
- **Goal:** Approximate the map $\pi_N \colon \mathcal{P} \to \mathbb{R}^N$, given as

$$\pi_N(\mu) = \underline{u}_N(\mu) \in \mathbb{R}^N,$$

  where $\underline{u}_N(\mu)$ holds the coefficients of the **orthogonal** projection $u_N(\mu)$ of the full-order solution $u(\mu)$ onto the basis Ψ_N of a reduced space V_N.

$\longrightarrow$ No operators required, no Galerkin projection, only solution snapshots!

Mathematical background

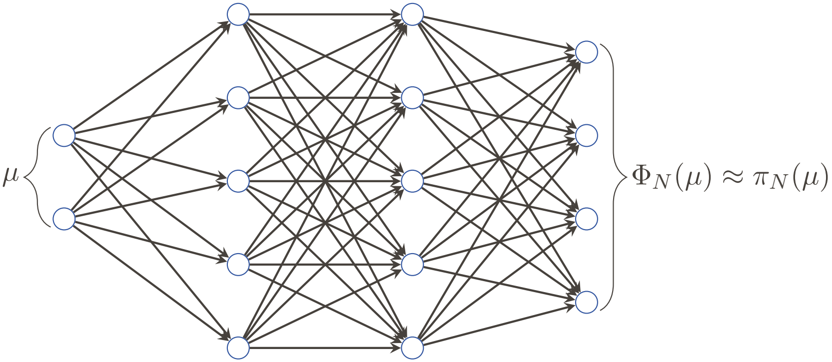
- **Two scenarios:**
  1. Given a full-order model $\mu \mapsto u(\mu)$, but no affine decomposition of operators.
  2. Given only a set $\{(\mu_i, u(\mu_i))\}_{i=1}^n$ of snapshots with corresponding parameter values, i.e. purely data-driven setting.
- **Goal:** Approximate the map $\pi_N \colon \mathcal{P} \to \mathbb{R}^N$, given as

$$\pi_N(\mu) = \underline{u}_N(\mu) \in \mathbb{R}^N,$$

  where $\underline{u}_N(\mu)$ holds the coefficients of the **orthogonal** projection $u_N(\mu)$ of the full-order solution $u(\mu)$ onto the basis Ψ_N of a reduced space V_N.
- **Idea:** Use a neural network Φ_N to approximate π_N.

  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.

Mathematical background – Visualization of the approach



Mathematical background – Error analysis

Components:

- High-fidelity space V with $\dim V = n$.
- Reduced subspace $V_N \subset V$ of dimension $\dim V_N = N \ll n$.
- Orthonormal basis Ψ_N of V_N.
- Matrix $\underline{\Psi}_N \in \mathbb{R}^{n\times N}$ with orthonormal columns formed by elements from Ψ_N.
- Approximation Φ_N of the map π_N, where

$$\pi_N(\mu) := \underline{\Psi}_N^\top u(\mu) = \underline{\Psi}_N^\top u_N(\mu) = \underline{u}_N(\mu).$$

Mathematical background – Error analysis

Components:

- High-fidelity space V with $\dim V = n$.
- Reduced subspace $V_N \subset V$ of dimension $\dim V_N = N \ll n$.
- Orthonormal basis Ψ_N of V_N.
- Matrix $\underline{\Psi}_N \in \mathbb{R}^{n \times N}$ with orthonormal columns formed by elements from Ψ_N.
- Approximation Φ_N of the map π_N, where

$$\pi_N(\mu) := \underline{\Psi}_N^\top u(\mu) = \underline{\Psi}_N^\top u_N(\mu) = \underline{u}_N(\mu).$$

Error estimate for the neural network-based approach:

$$\|u(\mu) - \underline{\Psi}_N \Phi_N(\mu)\| \leq \|u(\mu) - \underline{\Psi}_N \pi_N(\mu)\| + \|\underline{\Psi}_N \pi_N(\mu) - \underline{\Psi}_N \Phi_N(\mu)\|$$

Mathematical background – Error analysis

Components:

- High-fidelity space V with $\dim V = n$.
- Reduced subspace $V_N \subset V$ of dimension $\dim V_N = N \ll n$.
- Orthonormal basis Ψ_N of V_N.
- Matrix $\underline{\Psi}_N \in \mathbb{R}^{n\times N}$ with orthonormal columns formed by elements from Ψ_N.
- Approximation Φ_N of the map π_N, where

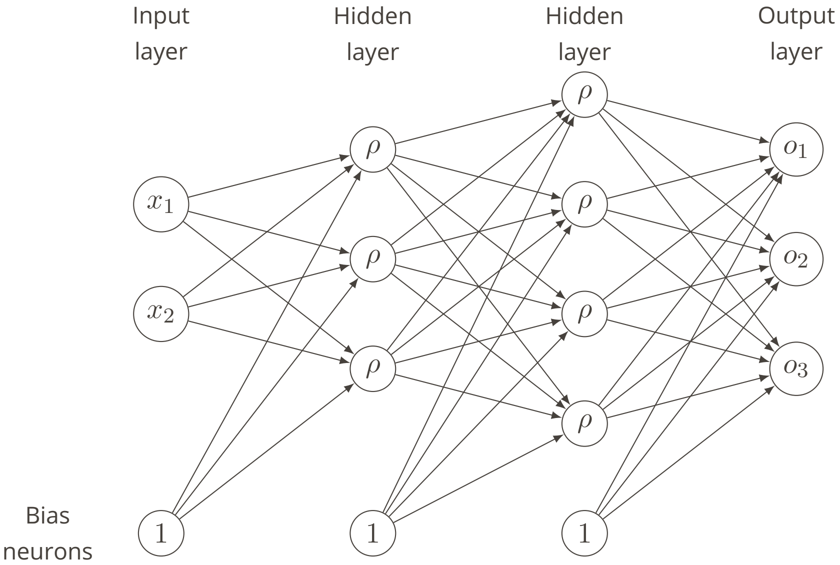
$$\pi_N(\mu) := \underline{\Psi}_N^\top u(\mu) = \underline{\Psi}_N^\top u_N(\mu) = \underline{u}_N(\mu).$$

Error estimate for the neural network-based approach:

$$\|u(\mu) - \underline{\Psi}_N \Phi_N(\mu)\| \leq \|u(\mu) - \underline{\Psi}_N \pi_N(\mu)\| + \|\underline{\Psi}_N \pi_N(\mu) - \underline{\Psi}_N \Phi_N(\mu)\|$$

$$= \underbrace{\|u(\mu) - u_N(\mu)\|}_{\substack{\text{best-approximation error in } V_N \\ \text{(orthogonal projection)}}} + \underbrace{\|\pi_N(\mu) - \Phi_N(\mu)\|}_{\substack{\text{approximation error} \\ \text{of the neural network}}}$$

Example: Feed-forward neural networks



Definition: Feed-forward neural networks

Components of a neural network Φ:

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

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, $\tanh$, $\dots$)

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

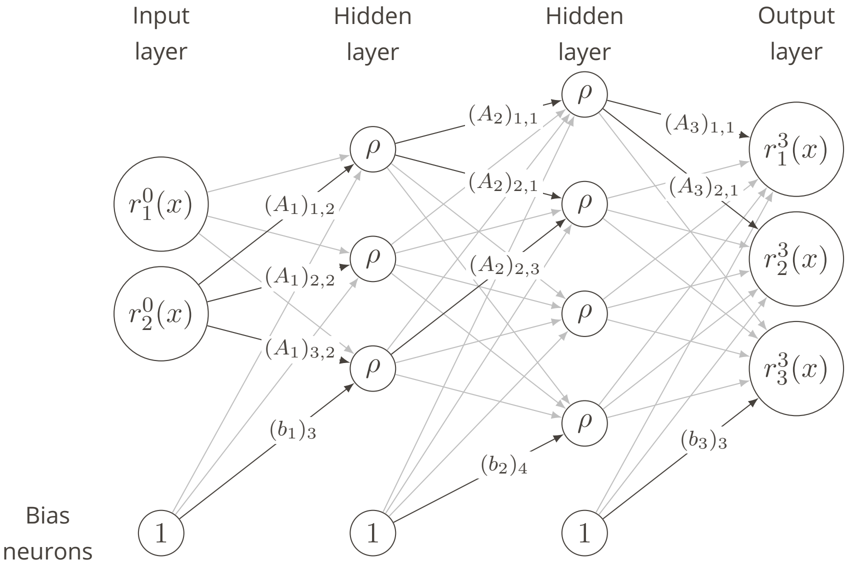
$$\Phi(x) = A_L r^{L-1}(x) + b_L,$$

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 loss

$$E(\Phi) = \frac{1}{n} \sum_{i=1}^{n} \|\Phi(x_i) - f(x_i)\|_2^2$$

with respect to the weights and biases of Φ.

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 loss
$$E(\Phi) = \frac{1}{n}\sum_{i=1}^{n}\|\Phi(x_i) - f(x_i)\|_2^2$$
with respect to the weights and biases of Φ.
- **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**

Pseudo code for neural network training

for $r = 1, \dots, N_{\text{restarts}}$ **do**

  initialize weights and biases randomly

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**

Pseudo code for neural network training

```
for r = 1, ..., N_restarts do
    initialize weights and biases randomly
    for e = 1, ..., N_epochs do
        create batches of training data
        /* Training */
        for b = 1, ..., N_batches do
            apply optimization step with loss function for batch b
        end
```

Pseudo code for neural network training

```
for r = 1, ..., N_restarts do
    initialize weights and biases randomly
    for e = 1, ..., N_epochs do
        create batches of training data
        /* Training */
        for b = 1, ..., N_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
```

Pseudo code for neural network training

```
for r = 1, ..., N_restarts do
    initialize weights and biases randomly
    for e = 1, ..., N_epochs do
        create batches of training data
        /* Training */
        for b = 1, ..., N_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 $u(\mu_1), \ldots, u(\mu_n)$.

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/`).

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:

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

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:

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

- Validation phase to assess generalization ability.

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:

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

- Validation phase to assess generalization ability.
- Early stopping and multiple restarts of training.

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:

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

- Validation phase to assess generalization ability.
- Early stopping and multiple restarts of training.
- Customizable training routine (optimizer, epochs, learning rate, ...).

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:

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

- 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!**

Some of the new features in release 2022.1

- Support for learning rate schedulers.

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.
- Scaling of inputs and outputs.

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.
- Scaling of inputs and outputs.
- Regularization of weights and biases.

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.
- Scaling of inputs and outputs.
- Regularization of weights and biases.
- Weighted loss function (output components are weighted by respective singular values).

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.
- Scaling of inputs and outputs.
- Regularization of weights and biases.
- Weighted loss function (output components are weighted by respective singular values).
- Logging of current training and validation loss.

Some of the new features in release 2022.1

- Support for learning rate schedulers.
- Customization of early stopping scheduler.
- Scaling of inputs and outputs.
- Regularization of weights and biases.
- Weighted loss function (output components are weighted by respective singular values).
- Logging of current training and validation loss.

- **Purely data-driven usage of neural network reductors.**

Let's look at some code ...

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

    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.,
                 loss_function=None, restarts=10,
                 lr_scheduler=optim.lr_scheduler.StepLR,
                 lr_scheduler_params={'step_size': 10, 'gamma': 0.7},
                 es_scheduler_params={'patience': 10, 'delta': 0.},
                 weight_decay=0., log_loss_frequency=0, seed=0):
        ...

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

A bit more code ...

```
def train_neural_network(training_data, validation_data,
            neural_network, training_parameters={},
            scaling_parameters={}, log_loss_frequency=0):
    """Training algorithm for artificial neural networks."""
    ...

def multiple_restarts_training(training_data, validation_data,
            neural_network, target_loss=None,
            max_restarts=10, log_loss_frequency=0,
            training_parameters={}, scaling_parameters={}, seed=None):
    """Algorithm that performs multiple restarts of neural network
       training."""
    ...
```

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

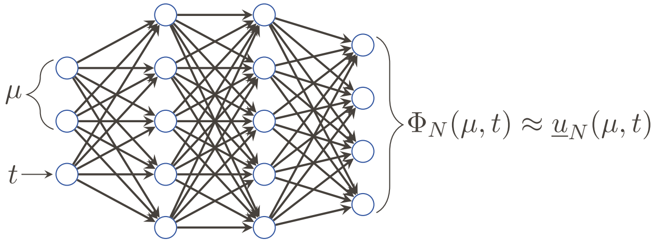
Variants of the `NeuralNetworkReductor`

Instationary models:

Treat time as an additional parameter and apply same procedure as for stationary problems, i.e. approximate the map $(\mu, t) \mapsto \underline{u}_N(\mu, t) \in \mathbb{R}^N$ by a neural network $\Phi_N : \mathcal{P} \times [0, \infty) \to \mathbb{R}^N$.

Usage shown in a demo:

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



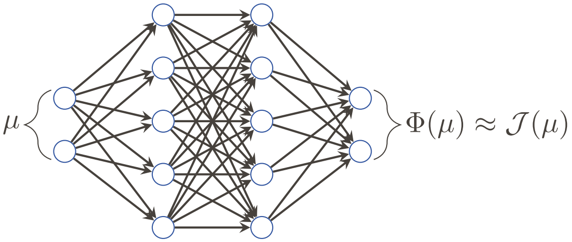
Variants of the `NeuralNetworkReductor`

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 Φ 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.

Navier-Stokes equations as an example for an instationary problem

Components:

- Velocity $u\colon \mathbb{R}^d \to \mathbb{R}^d$
- Pressure $p\colon \mathbb{R}^d \to \mathbb{R}$
- Reynolds number Re > 0 (parameter of the system)

Navier-Stokes equations as an example for an instationary problem

Components:

- Velocity $u\colon \mathbb{R}^d \to \mathbb{R}^d$
- Pressure $p\colon \mathbb{R}^d \to \mathbb{R}$
- Reynolds number Re > 0 (parameter of the system)

Navier-Stokes equations:

$$\begin{aligned} \partial_t u + (u \cdot \nabla) u - \frac{1}{\mathrm{Re}} \Delta u + \nabla p &= 0 \\ \nabla \cdot u &= 0 \end{aligned}$$

Navier-Stokes equations as an example for an instationary problem

Components:

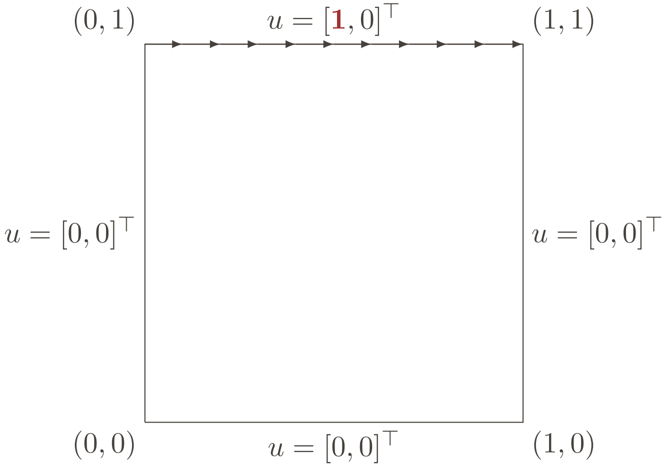
- Velocity $u\colon \mathbb{R}^d \to \mathbb{R}^d$
- Pressure $p\colon \mathbb{R}^d \to \mathbb{R}$
- Reynolds number Re > 0 (parameter of the system)

Navier-Stokes equations:

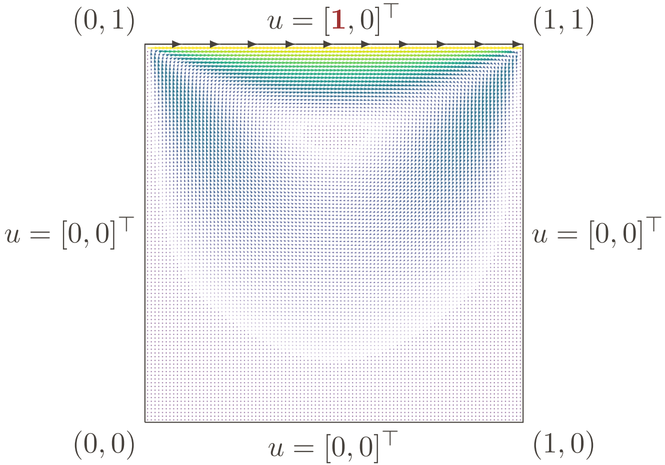
$$\underbrace{\partial_t u}_{\text{time variation}} + \underbrace{(u \cdot \nabla)u}_{\text{convection}} - \underbrace{\frac{1}{\mathrm{Re}}\Delta u}_{\text{diffusion}} + \underbrace{\nabla p}_{\substack{\text{internal force,}\\ \text{pressure differences}}} = \underbrace{0}_{\substack{\text{no external force}\\ \text{(for simplicity)}}}$$

incompressibility condition: $\nabla \cdot u = 0$

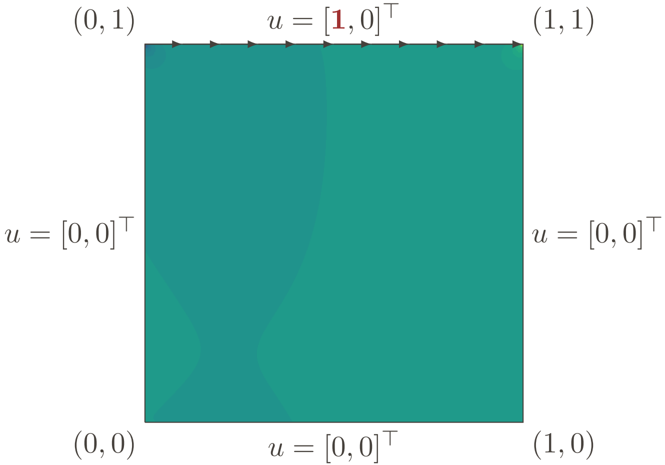
Lid-driven cavity problem



Lid-driven cavity problem



Lid-driven cavity problem



Discretization using FEniCS (see also the demo for full code)

```
from pymor.bindings.fenics import FenicsVectorSpace, FenicsOperator, FenicsMatrixOperator
from pymor.algorithms.timestepping import ImplicitEulerTimeStepper

import dolfin as df

# create square mesh
mesh = df.UnitSquareMesh(n, n)

# create Finite Elements for the pressure and the velocity
P = df.FiniteElement('P', mesh.ufl_cell(), 1)
V = df.VectorElement('P', mesh.ufl_cell(), 2, dim=2)
# create mixed element and function space
TH = df.MixedElement([P, V])
W = df.FunctionSpace(mesh, TH)

# extract components of mixed space
W_p = W.sub(0)
W_u = W.sub(1)

# define trial and test functions for mass matrix
u = df.TrialFunction(W_u)
psi_u = df.TestFunction(W_u)

# assemble mass matrix for velocity
mass_mat = df.assemble(df.inner(u, psi_u) * df.dx)
```

Discretization using FEniCS (see also the demo for full code)

```
# define trial and test functions
psi_p, psi_u = df.TestFunctions(W)
w = df.Function(W)
p, u = df.split(w)

# set Reynolds number, which will serve as parameter
Re = df.Constant(1.)

# define walls
top_wall = "near(x[1], 1.)"
walls = "near(x[0], 0.) | near(x[0], 1.) | near(x[1], 0.)"

# define no slip boundary conditions on all but the top wall
bcu_noslip_const = df.Constant((0., 0.))
bcu_noslip  = df.DirichletBC(W_u, bcu_noslip_const, walls)
# define Dirichlet boundary condition for the velocity on the top wall
bcu_lid_const = df.Constant((1., 0.))
bcu_lid = df.DirichletBC(W_u, bcu_lid_const, top_wall)

# fix pressure at a single point of the domain to obtain unique solutions
pressure_point = "near(x[0],  0.) & (x[1] <= " + str(2./n) + ")"
bcp_const = df.Constant(0.)
bcp = df.DirichletBC(W_p, bcp_const, pressure_point)

# collect boundary conditions
bc = [bcu_noslip, bcu_lid, bcp]
```

Discretization using FEniCS (see also the demo for full code)

```
mass = -psi_p * df.div(u)
momentum = (df.dot(psi_u, df.dot(df.grad(u), u)) - df.div(psi_u) * p
            + 2.*(1./Re) * df.inner(df.sym(df.grad(psi_u)), df.sym(df.grad(u))))
F = (mass + momentum) * df.dx

df.solve(F == 0, w, bc)

# define pyMOR operators
space = FenicsVectorSpace(W)
mass_op = FenicsMatrixOperator(mass_mat, W, W, name='mass')
op = FenicsOperator(F, space, space, w, bc,
                    parameter_setter=lambda mu: Re.assign(mu['Re'].item()),
                    parameters={'Re': 1})

# timestep size for the implicit Euler timestepper
dt = 0.01
ie_stepper = ImplicitEulerTimeStepper(nt=nt)

# define initial condition and right hand side as zero
fom_init = VectorOperator(op.range.zeros())
rhs = VectorOperator(op.range.zeros())
# define output functional
output_func = VectorFunctional(op.range.ones())

# construct instationary model
fom = InstationaryModel(dt * nt, fom_init, op, rhs, mass=mass_op, time_stepper=ie_stepper,
                        output_functional=output_func, visualizer=FenicsVisualizer(space))
```

Neural network reductor using the FEniCS-based FOM

Constructing the reduced order model is now similar to before!

```
parameter_space = fom.parameters.space(1., 50.)

training_set = parameter_space.sample_uniformly(training_samples)
validation_set = parameter_space.sample_randomly(validation_samples)

reductor = NeuralNetworkInstationaryReductor(fom, training_set,
     validation_set, basis_size=10, scale_outputs=True, ann_mse=None)
rom = reductor.reduce(hidden_layers='[30, 30, 30]', restarts=0)
```

Purely data-driven usage of reductors without an actual FOM

- Set `fom=None` when initializing the reductor.

Purely data-driven usage of reductors without an actual FOM

- Set `fom=None` when initializing the reductor.
- Use a list of pairs of parameters and solution snapshots (in form of `VectorArrays`) as `training_set`.

Purely data-driven usage of reductors without an actual FOM

- Set `fom=None` when initializing the reductor.
- Use a list of pairs of parameters and solution snapshots (in form of `VectorArrays`) as `training_set`.
- No access to operators or high-fidelity solver code required!

Purely data-driven usage of reductors without an actual FOM

- Set `fom=None` when initializing the reductor.
- Use a list of pairs of parameters and solution snapshots (in form of `VectorArrays`) as `training_set`.
- No access to operators or high-fidelity solver code required!
- Solution snapshots might even be written to disk and read from file!

Purely data-driven usage of reductors without an actual FOM

- Set `fom=None` when initializing the reductor.
- Use a list of pairs of parameters and solution snapshots (in form of `VectorArrays`) as `training_set`.
- No access to operators or high-fidelity solver code required!
- Solution snapshots might even be written to disk and read from file!
- An example is again shown in the demo under `https://github.com/pymor/pymor/blob/main/src/pymordemos/neural_networks.py`

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).

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.

Thank you for your attention!

Are there questions?