Convolutional Neural Network for Fluid Dynamics

Thesis Project

Sylle Hoogeveen
4445082

MSc Applied Mathematics Student

Department of Electrical Engineering,
Mathematics & Computer Science

March 15, 2022

Contents

Nomenclature

- NN - Neural network
- ROM - Reduced Order Model
- PINN - Physics Informed Neural Network
- PRNN - Physics Reinforced Neural Network
- CNN - Convolutional Neural Network
- POD - Proper Orthogonal Decomposition
- PCA - Principal Component Analysis
- PGD - Proper Generalized Decomposition
- RD - Reduced Basis
- ML - Machine Learning
- LSTM - Long-Short Term Memory
- SVD - Singular Value Decomposition
- CAE - Convolutional Autoencoder
- RNN - Recurrent Neural Network
- GRU - Gated Recurrent Unit

1 Introduction

1.1 Relevance

1.2 Aim and Objectives

The aim of this study is to construct a single reduced order model for time-dependent incompressible flow that can account for different boundary conditions, material parameters and geometries (computational domains). As test case, blood flow during one heartbeat will be used. The objectives are high (need to specify?) accuracy, measured as difference between the predicted velocity field and simulated velocity field, and speed-up between the OpenFoam simulations and machine learning predictions.

2 | Preliminaries & Related work

In this chapter relevant theory and related research is discussed to set the framework for this study.

2.1 Reduced order methods

Reduced order models (ROMs) aim to capture the most important features of a physical phenomena being simulated, whilst reducing the computational load. The increase in complexity of mathematical models in an attempt to approximate reality and desire to have near real-time simulations have emphasized the need for such strategies. There are two approaches to this task. The first approach is to simplify the underlying physics (known as operational based reduction methods) for example by making assumptions on certain parameters or symmetry. The second approach is discretizing the continuous equations and thereafter reducing the model, most commonly projection-based [1].

Some well known projection-based methods are proper orthogonal decomposition (POD), reduced basis (RD), Proper Generalized Decomposition (PGD) and Principal Component Analysis (PCA). ***explain methods?***. The drawback of using these methods is that they usually rely on linear basis functions constructed by Singular Value Decomposition (SVD) [2] or eigenvalue decomposition, whilst the underlying dynamics are often non-linear. Also, due to discretizing, they depend on fixed computational domain (fixed geometry), although work has been done on using reduced order methods on moving objects inside the domain [3], [4]. Although many interesting research can be found on the use of classic ROMs especially in the field of (cardio-vascular) fluid simulations, this research is limited to discussing them further only when used in combination with machine learning techniques.

2.2 Machine Learning framework

Instead of using classical numerical methods, neural networks can be used to learn non-linear relations, produce lower dimension representations and perform time evolution of physical phenomena governed by differential equations. Two types of network that could be used for creating a reduced order representation are convolutional neural networks and autoencoders. For time evolution, standard neural networks and recurrent neural networks can be used. To ensure the networks take into account the underlying physics and increase the chance the networks output is physically relevant, a network can be made physics informed. All the above concepts will be discussed in this section.

2.2.1 Neural Network

A neural network (NN) is a general machine learning method that has many purposes, from regression and classification to creating embeddings. A NN is built by three sets of neurons: input neurons, (multiple layers of) hidden neurons and output neurons. The inputs of each neuron are multiplied by a weight, summed and passed though an activation function to determine the output of a neuron. This activation function can be linear, however this would entail that multiple layers of a network can be collapsed back to one. Hence, to learn more complex relations, non-linear activation functions are used. How the neurons are connected determines the architecture of a NN.

The training process in which the weights of the network are learned can be supervised or unsupervised. This research will be limited to supervised learning. In supervised learning, inputs are passed through the network and the output is compared to a 'ground truth'. How these are compared is determined by the loss function, usually the mean squared error (MSE) is used. Hence the weights are adjusted such that a loss function is minimized. How much and in what direction the weights are adjusted is determined by the optimizer. For deep neural networks (NNs with many layers), Adam is the best option [5].

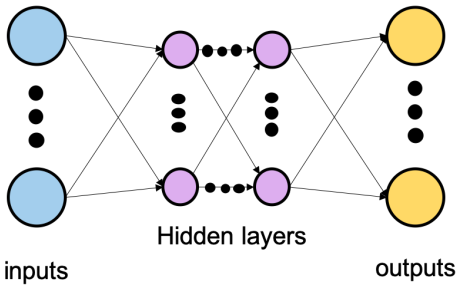

Figure 2.1: Neural Network

Choosing the architecture of a NN is a trade-off between making the NN complex enough such that the relation between input and output can be learned (prevent underfitting) and keeping the NN simple enough to be able to generalize (prevent overfitting). Generalizability refers to the ability of the NN to make predictions on input data that does not belong to the training data set, i.e. is not seen by the network before. There are multiple regularization techniques, including early stopping of training process, parameter norm penalties such as L1 and L2 regularization and adding drop-out probability to layers of neurons. ***For further details on how NN work, optimizers, activation functions and regularization techniques the reader is referred to appendix ... or just not discussed in this work***

The most naive way of predicting future time steps with machine learning is using a standard neural network. With a NN, no assumption on the input data is made and there is no mechanism for 'memorizing' previous input data. However, this approach is still valuable to explore, as making no assumptions on the input data also means the network has all flexibility to learn undiscovered patterns. Moreover, less complexity in the network is desirable as it is computationally and memory-wise more efficient.

Using a NN for time evolution in combination with using ML techniques for reducing order was explored by Kim et al., who firstly trained an autoencoder to create a latent space representation of smoke and fluid simulations. Thereafter, they implemented a NN to find the subsequent latent representations. As input for the NN, they used the latent representation found by the autoencoder concatenated with a control vector difference between user input parameters. The output of the network is the difference between the current latent space representation and the next latent representation. Hence the new reduced order representation can be found by adding the output of the network to the previous reduced order representation [2].

2.2.2 Convolutional Neural Network

A convolutional NN is characterized by convolution kernels that slide across the input features creating feature maps.

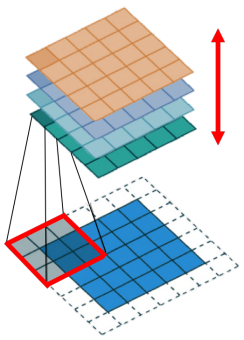

Figure 2.2: Convolution

As depicted in figure 2.2 by the red square, a convolution kernel has a kernel size (in this case 3x3) on which it performs a filter operation. The kernel also has a stride, defined as the amount of spaces the kernel shifts per step. The depth of the kernel (or filter) is the amount of different filter operations it performs on the input data, and thus how many feature maps the layer creates. This is depicted in figure 2.2 by the red arrow. The different filters uncover relations in the input data. The filters are the weights of the network, and are learned during training phase. Using a CNN versus a NN reduces the number of parameters to be learned in a network, as the neurons share weights, i.e. the same filter/kernel is applied to every input neuron to create one feature map.

A CNN can consist of multiple convolutional layers, each reducing the dimension of the input by kernelsize -1 along each axis, if no padding is used. Further reduction of the dimension is achieved by adding Max Pooling layers, which is a operation that returns the max value within it's pool (same as kernel). By reducing the dimension and thus the total number of weights, max pooling layers reduce the computational and memory usage of the CNN.

Lopez-Martin et al. created one Deep NN with 3D convolutional layers to reduce dimensionality of velocity fields generated by a synthetic jet simulation. Three dimensions in this case are x,y and time. The first convolutional layers were used to find temporal-spacial relations and a reduced order representation. The final layers of the network were used for time evolution. This was achieved by adding a convolutional layer that controls the final depth dimension of the feature space to be the number of time steps to be predicted. This tensor is reshaped such that it can be broken into the number of time steps to be predicted vectors. These vectors are then used as input to the fully connected layers, which perform the time evolution [6].

2.2.3 Autoencoder

An autoencoder is a type over NN that consists of an encoder and decoder part. The encoder part of the network tries to find an embedding or lower dimension representation (depicted in orange in figure 2.3) of the input data. This is achieved by reducing the amount of neurons in each consecutive layer (purple part figure 2.3) until the amount of neurons is equal to the desired latent dimension size. The decoder part of the network (light blue in figure 2.3) then tries to decode the latent representation back to the full input dimensionality. This process is used in various applications, for example denoising input data.

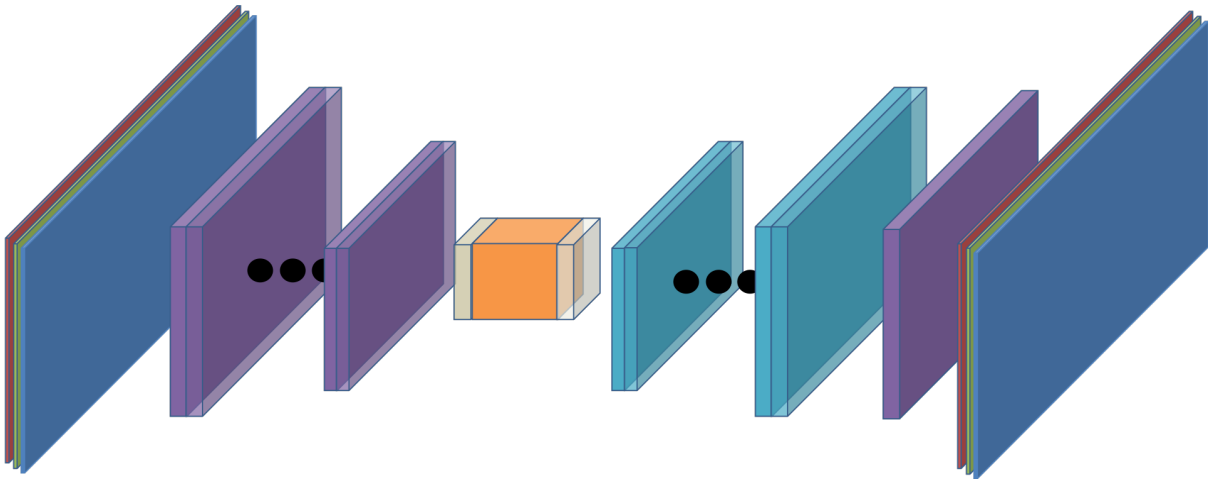

Figure 2.3: Autoencoder structure for RGB image

Also for the purpose of creating reduced order models autoencoders have been used. Simpson et al used a simple autoencoder, consisting of layers of normal neuron, to find a reduced order representation of systems under forcing. It was noted that dynamics not captured in the training data were not captured by the produced ROM. Additionally, for high dimensionality input, a convolutional autoencoder was suggested to prevent training time and amount of training data issues [7].

A convolutional autoencoder (CAE) is an autoencoder network structure, using convolutional layers as explained in the sections above. Kim et al. [2] and Eichinger et al [8] used this architecture. Opposed to other papers reviewed, Eichinger et al. studied steady state fluid flow, using a binary and signed distance function representations of the input domain and retrieving the velocity field as output. They found a speedup in the order of 100 comparing their CAE with OpenFoam simulations [8].

Kim et al. used an CAE network to generate smoke and fluid simulations for graphical implementations (games, videos). Their approach led to significant speedup while maintaining accuracy for a variety of fluid behaviour. However, the ability to reconstruct physically accurate scenarios depended heavily on the how closely the scenarios matched the input training data [2].

Due to the depth of their networks, the vanishing gradient problem can occur. The large number of steps in backpropagation though the network results in the gradient becoming very small. Hence the update of the weights becomes very small. In other words, learning halts. To prevent this, both studies implemented residual connections, allowing information to pass certain layers of the network and thus increasing the gradient.

A (different) Kim et al. proposed using a shallow masked autoencoder instead of a deep convolutional autoencoder to improve efficiency. Shallow in this case means the encoder and decoder only consisted of one hidden layer. This was possible as they developed a hyper-reduction technique exploiting classic numerical methods for solving PDEs/ODEs. The hidden layer and output layer of the decoder were sparsely connected by multiplying the weight matrix of this connection by a mask matrix. The mask matrix consists of zeros and ones, and was constructed to reflect local connectivity as in the central difference scheme of the Finite Difference Method [9].

However, using such a mask matrix makes the autoencoder problem (PDE/ODE and computational domain) specific.

2.2.4 Recurrent Neural Network

Recurrent NNs can be built with different unit types: standard units, long short-term memory (LSTM) units or Gated Recurrent Units (GRUs). (Also RNNs can have a specific architecture called an Encoder-Decoder network with or without attention mechanism.) All these techniques will be discussed in the following sections.

Recurrent Neural Networks have input, hidden and output neurons similar to a standard neural network. However, the neurons are divided in groups belonging to a fixed time step. The input neurons are connected to neurons in the hidden layer belonging to the same time step. Also the neurons within the hidden layer are fully connected to hidden neurons with the same assigned time step and one way connected to hidden neurons of the next time step. The output neurons are again only connected to hidden neurons belonging to the same time step.

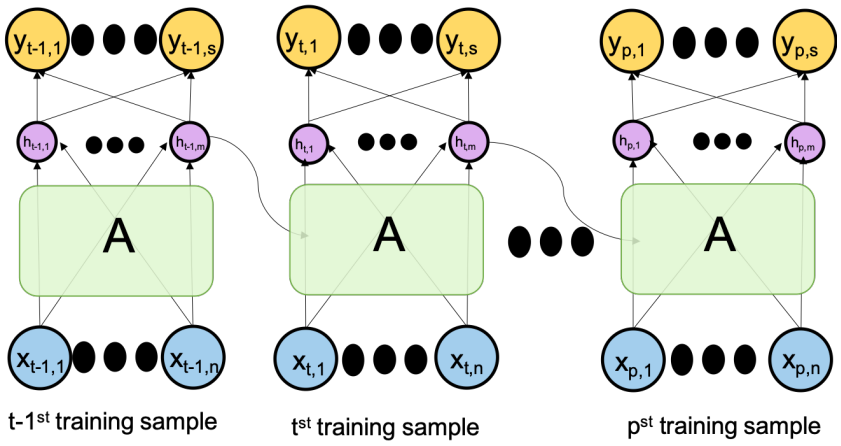

Figure 2.4: RNN with n input neurons, m hidden state variables, s output neurons and p time steps

Standard Unit

A standard unit or standard RNN cell has two inputs: x_t, the new part of a sequence and h_{t-1} the hidden state of the previous part of the sequence. These are concatenated, multiplied by the weight matrix, added with the bias vector and the results is scaled between [-1,1] by the hyperbolic tangent activation function. The outcome is a new hidden state: $h_t = tanh(W * x_t \frown h_{t-1} + b)$, which will be used in the next time step. Note that the weight matrix is constant for each time step.

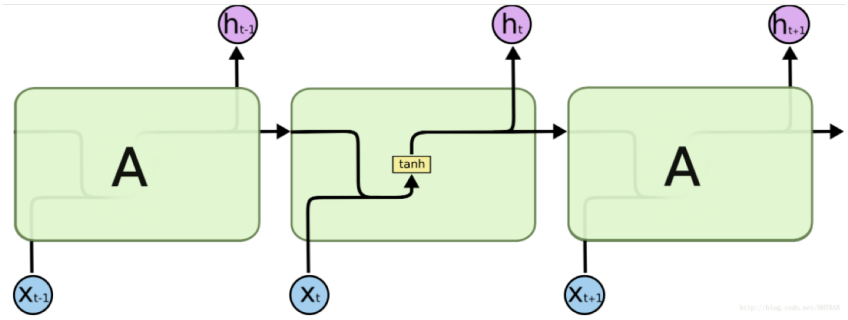

Figure 2.5: Standard unit RNN, figure from:

However, the vanishing gradient problem occurs when the sequence is very long. A large sequence length is similar to a deep NN, as the gradient is backpropagated many times though the network. Again, this leads to very small to no weight updates and learning halts.

Long Short-Term Memory Unit

The LSTM unit was developed in ... to solve the vanishing gradient problem of the RNNs with standard units. The LSTM unit has three inputs instead for two. Again the new part of the sequence x_t and hidden state of the previous part of the sequence h_{t-1} are used. In addition, there is a second state vector which is the cell state or LSTMs memory c_{t-1}. The cell state passes through the LSTM unit with less computation, making it easier to pass through unchanged [10]. Similar to the idea of adding residual connections in deep NNs, this helps preserve the gradient. In the LSTM unit the three gates are computed from the concatenation $x = x_t^\frown h_{t-1}$, each having their own weight matrix:

- Forget gate $f = \sigma(W_f x + b_f)$
- Update gate $u = \sigma(W_u x + b_u)$
- Output gate $o = \sigma(W_o x + b_o)$

Here, σ is the sigmoid function which returns values between 0 and 1, killing input close to 0 and letting input close to 1 pass nearly unchanged. The new cell state is computed by multiplying the previous cell state by what was learned to forget (the forget gate) and adding this to what was learned to be remembered (update gate times $tanh(W_c * x + b_c)$): $c_t = f * c_{t-1} + u * tanh(W_c * x + b_c)$. The new hidden state is computed by the cell state scaled between [-1,1] by tanh function multiplied by what was learned to be exposed to the hidden state (output gate): $h_t = o * tanh(c_t)$. This gives the ability to control what to forget from the cell state, what to store from the input in the cell state and what part of the cell state to expose to the hidden state at a given point in time [10].

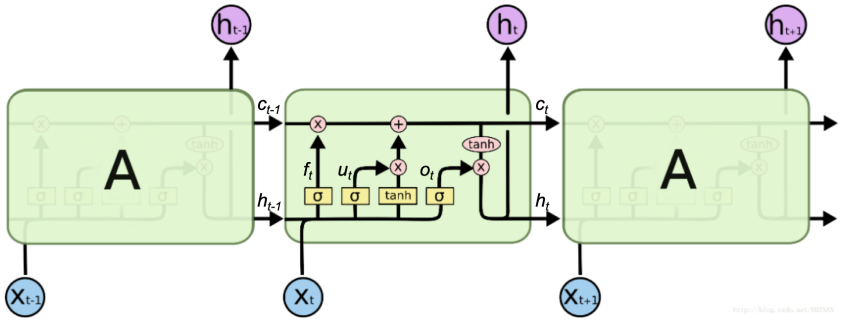

Figure 2.6: LSTM unit RNN, figure from:

RNNs with LSTM units have been used in combination with classical reduced order methods for temporal evolution in simulations of physical phenomena. For example, Hu et al. [11] used POD and SVD together with an LSTM network for predicting flood induced conditions. They reported a maintained accuracy in comparison to the full model while CPU cost was reduced three orders of magnitude. Also RNNs with LSTM units have been used in combination with machine learning techniques for order reduction. Pawar et al. also merged a classical reduced order approach, POD with Galerkin projection, with LSTM network for time evolution in fluid mechanics simulations. Simpson et al. [7] trained an autoencoder for order reduction and LSTM network to predict the response of dynamical systems in the latent space based on forcing time histories.

Gated Recurrent Unit

GRUs are an evolution of LSTM units and introduced quite recently in 2014 [10]. By dropping the output gate and only using two gates:

- Reset gate $r = \sigma(W_r x + b_r)$
- Update gate $z = \sigma(W_z x + b_z)$

The reset gate is similar to the forget gate in the LSTM unit and is to decide what to forget. The update gate is to decide what should be passed to the output. First a proposed new hidden state is computed by multiplying the old hidden state with the reset gate, concatenating this with the new part of the sequence and scaling between [-1,1] with the tanh function: $\hat{h}_t = tanh(W x_r^\frown (r * h_{t-1})$. Thereafter, the actual new hidden state is computed by taking a linear sum between the previous hidden state and the proposed new hidden state, in which the reset gate determines how much of the hidden state is updated: $h_t = (1 - z_t)h_{t-1} + z_t\hat{h}_t$ [10].

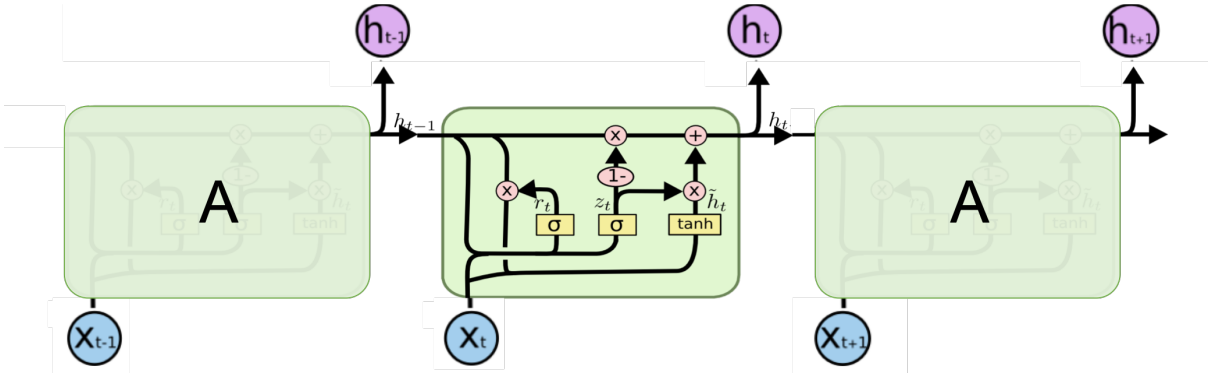

Figure 2.7: GRU RNN. figure from

With only two gates, there is no mechanism to control which parts of the state is exposed to the output, but the GRU has a less complex structure. This makes the units computationally more efficient while it was proven to have comparable performance to LSTM units [12].

To the author's knowledge, an RNN with GRUs for time evolution has not been used in combination with classical or machine learning reduced order techniques. Hence, this is a novel approach which will be explored in this research. *no papers found, novel approach?*

2.2.5 Physics informed Neural Networks

The idea of making neural networks for solving problems involving PDEs 'physics informed' was explored by Raissi et al. [13]. Information about the underlying physics of the problem can be inserted into the NN by constructing activation and loss functions specific for the underlying differential operator. All network types before mentioned thus can be physics informed, if the activation and/or loss function is tailored to the PDE.

Chen et al. created a neural network to predict the projection coefficients of the POD-Galerkin projection (PDNN), a neural network to predict the reduced order solution (PINN) and a neural network to predict the projection of high-fidelity solution on the reduced space (PRNN) by incorporating appropriate terms into the loss function [14].

To ensure conservation of mass (thus non-divergence) for incompressible fluid dynamics, Kim et al. introduced a stream function based loss function in the decoder part of their network. In the stream loss function as described in equation 2.1, $G(\mathbf{c})$ is the output of the network and $\mathbf{u_c}$ is a simulation sample from the data set.

$$L_G(\mathbf{c}) = \lambda_{\mathbf{u}}||\mathbf{u_c} - \nabla \times G(\mathbf{c})||_1 + \lambda_{\nabla \mathbf{u}}||\nabla \mathbf{u_c} - \nabla(\nabla \times G(\mathbf{c}))||_1 \tag{2.1}$$

Making the stream function of the model output the reconstruction target, which is divergence free by construction ($\nabla \cdot (\nabla \times G(\mathbf{c})) = 0$) and ensuring the derivatives also match.

Above approach of pushing a solution towards one that obeys the physical laws by modification of loss functions can be seen as soft physical constraining the network. Physical laws are not enforced at all times but are encouraged. Mohan et al. took a different approach, embedding hard physical constraints in the neural network architecture. This was done by adding non-trainable layers with physical input after a CAE structure. The decoder in this network has the vector potential as output. The non-trainable layers consists of a layer enforcing BCs with ghost cells, a layer which computes all spatial derivatives and a last layer to compute the curl on the vector potential field.

Physics can also be injected in a time-series forecasting networks. For example, by taking a reduced order representation, created either by classical methods or ML techniques, and concatenating this with computed hidden state vectors of a recurrent neural network [15].

2.3 Other Machine Learning use in CFD

Machine learning has not only been used to produce reduced order representations and perform time evolution. Other attempts have been made using ML to combat increased complexity of mathematical models and create more computationally efficient algorithms. For example, the accuracy of a coarse grid CFD simulation was enhanced using ML [16]. ***Explore a little further for background knowledge***.

2.4 Cardio-vascular modelling

As use case this research will apply ML ROM on cardio-vascular simulations. To generate simulation data for training, validation and testing, some background knowledge on cardio-vascular modelling is necessary and given in this section.

2.4.1 Mathematical representation

The flow of blood was modeled by Navier-Stokes equations as proposed in [17], adapted from 3D to 2D.

$$\begin{aligned} \nabla \cdot \mathbf{u} &= 0 \\ \frac{\partial}{\partial t}(\mathbf{u}) + (\nabla \mathbf{u})\mathbf{u} + \nabla \frac{p}{\rho} - \nabla \cdot (2\nu_2 D(\mathbf{u})) &= 0 \end{aligned} \tag{2.2}$$

In which $\mathbf{u}$ is the velocity, p is the mean normal stress ($p = -\frac{1}{2}(tr\mathbf{T})$ with $\mathbf{T} = -p\mathbf{I} + 2\mu\mathbf{D}$ the fluid stress), ν is the kinematic viscosity ($\nu = \frac{\mu}{\rho}$, with μ the dynamic viscosity) and $\mathbf{D}$ the strain-rate tensor ($\mathbf{D} = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)$)). The dynamic viscosity is reported between 3.5 and 5.5 cP, which is equal to 0.003 and 0.0055 $N * \frac{s}{m^2}$. However, in reality, the viscosity of blood has a much larger range dependent on hemodynamic conditions [18]. Similarly, blood density differs depending on gender and moreover, body position. It was set to the average blood density of 1060 $\frac{kg}{m^3}$ [19]. The artery walls are modeled as rigid, which is again simplification of reality. To model artery walls more accurately, hyperelastic and viscoelastic material models can be incorporated [20].

3 | Methodology

3.1 Data Generation

3.1.1 Numerical Method

To solve the mathematical problem stated in the previous section. OpenFoam software was used. In particular the IcoFoam solver, for incompressible, Newtonian fluids was implemented. Blood is a non-Newtonian fluid, but for the scope of this research this simplification was made.

Equations

The IcoFoam solver solves the following equation:

$$\begin{aligned} \nabla \cdot \mathbf{u} &= 0 \\ \frac{\partial}{\partial t}(\mathbf{u}) + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) - \nabla \cdot (\nu_1 \nabla \mathbf{u}) &= -\nabla p_k \end{aligned} \tag{3.1}$$

In which $\mathbf{u}$ is the velocity in $\frac{m}{s}$ and p_k is the kinematic pressure ($p_k = \frac{p}{\rho}$) in $\frac{m^2}{s^2}$. Equations 2.2 and 3.1 are equivalent as shown below.

$$\nabla\cdot(\mathbf{u} \otimes \mathbf{u}) = \nabla \cdot \begin{bmatrix} u_x \\ u_y \end{bmatrix} [u_x \; u_y] = \nabla \cdot \begin{bmatrix} u_x^2 & u_x u_y \\ u_x u_y & u_y^2 \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x}(u_x^2) + \frac{\partial}{\partial y}(u_x u_y) \\ \frac{\partial}{\partial y}(u_y^2) + \frac{\partial}{\partial x}(u_x u_y) \end{bmatrix} = \begin{bmatrix} 2u_x \frac{\partial u_x}{\partial x} + u_x \frac{\partial u_y}{\partial y} + u_y \frac{\partial u_x}{\partial y} \\ 2u_y \frac{\partial u_y}{\partial y} + u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_x}{\partial x} \end{bmatrix}$$

$$= \begin{bmatrix} u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} + u_x(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y}) \\ u_y \frac{\partial u_y}{\partial y} + u_x \frac{\partial u_y}{\partial x} + u_y(\frac{\partial u_y}{\partial y} + \frac{\partial u_x}{\partial x}) \end{bmatrix} \overset{1}{=} \begin{bmatrix} u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} \\ u_y \frac{\partial u_y}{\partial y} + u_x \frac{\partial u_y}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix} = (\nabla \mathbf{u})\mathbf{u}$$

1) Use $\nabla \cdot \mathbf{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} = 0$

$$\nabla \cdot (\nu_1 \nabla \mathbf{u}) = \nabla \cdot (\nu_1 \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} \end{bmatrix}) = \nu_1 \begin{bmatrix} \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_y}{\partial x \partial y} \\ \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_x}{\partial x \partial y} \end{bmatrix} \overset{2}{=} \nu_1 \begin{bmatrix} \frac{\partial^2 u_x}{\partial x^2} \\ \frac{\partial^2 u_y}{\partial y^2} \end{bmatrix} \overset{3}{=} \nu_2 \begin{bmatrix} 2\frac{\partial^2 u_x}{\partial x^2} \\ 2\frac{\partial^2 u_y}{\partial y^2} \end{bmatrix}$$

$$\overset{2}{=} \nu_2 \begin{bmatrix} 2\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_y}{\partial x \partial y} + \frac{\partial^2 u_x}{\partial y^2} \\ 2\frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_x}{\partial x \partial y} + \frac{\partial^2 u_y}{\partial x^2} \end{bmatrix} = \nabla \cdot (\nu_2 \begin{bmatrix} 2\frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \\ \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} \end{bmatrix}) = \nabla \cdot (\nu_2 (\begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} \end{bmatrix} + \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} \\ \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} \end{bmatrix}))$$

$$= \nabla \cdot (2\nu_2 * \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T))) = \nabla \cdot (2\nu_2 \mathbf{D}(\mathbf{u}))$$

2) Use $\frac{\partial u_x}{\partial y} = \frac{\partial u_y}{\partial x} = 0$
3) Take $\nu_1 = 2\nu_2$

Boundary conditions

For the velocity component a time dependent function was generated to simulate blood flow during one heartbeat based on measurements **??**. The blue function in graph 3.1 was converted to tabular values and given as 'UniformFixedValue' type boundary condition (BC) for the inlet boundary. The outlet condition for velocity was set to zero gradient, defined as the Neumann BC: $\frac{\partial \mathbf{u}}{\partial t} = 0$. *Note that here the assumption is made that the velocity is fully developed and constant at the outlet boundary.* No slip boundary conditions were implemented for all walls. This is defined as the Dirichlet BC: $\mathbf{u} =$

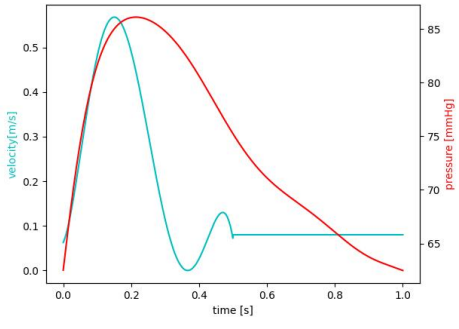

Figure 3.1: time dependent velocity inlet and pressure outlet boundary condition

0.

The pressure boundary conditions were all defined as zero gradient, except for the outlet boundary. Thus it is assumed that the walls are rigid and the force exerted by the wall on the fluid is equal to the force exerted by the fluid on the wall. *Also the assumption is made that the pressure is constant at the inlet boundary.* The outlet BC is a again a function of time, as shown in graph 3.1 by the red line, created to simulate the aortic pressure during one heartbeat. The graph was again converted to tabular values and given as 'UniformFixedValue' type BC in OpenFoam solver.

Solver

IcoFoam was used –> courant number explodes for small geometries :(

try a single phase solver that can adjust the time-step width dt, based on the maximum Courant number (CFL) defined in constant/controlDict. icoFoam cannot do that and only uses a fixed timestep.

3.1.2 Geometries

Four different geometries were created with the GMSH python library by the following steps:

1. define 2D mesh by creating points, lines and a surface in the xy-plane
2. extrude the surface in the z-direction by one cell (this is necessary as OpenFoam only takes 3D meshes)
3. add all surfaces to appropriate Physical Groups and name these groups
4. save the mesh as .msh2 file (this is compatible with gmshToFoam function)
5. in openfoam, run gmshToFoam on mesh file
6. set front and back patches to empty type in constant/boundary file, set inflow and outlflow boundaries to patch type and set other boundaries to wall type
7. define boundary conditions in dict files in 0 folder
8. run icoFoam as solver
9. convert output to VTK files by the "foamToVTK -allPatches" command

All code to create the meshes can be found in [GITHUB REPO].

From Paraview visualization of the VTK files, png images where stored for every time step on a 1578x952 pixel grid, these were cropped to .... pixel grid such that only a rectangular domain of computation remained, as shown in figure ... Note that the inflow boundary is always at the top.

3.1.3 Data Generation Pipeline

Combining all sections above, an automated data pipeline was created such that input for the NNs could be generated swiftly. The pipeline takes geometry parameters, such as channel width and bend angles if applicable, and boundary condition parameters, for example peak velocity and blood pressure, as input. The geometry input parameters were used to create an unstructured mesh and with the boundary condition input parameters the time dependent velocity inlet and pressure outlet values were generated. These were all loaded into the openfoam Docker image. In the Docker image, the GMSH mesh was converted to a Foam polymesh and the dict files were adjusted as described in above section. Next IcoFoam solver was run creating Foam output. This Foam output was converted to VTK files and thereafter converted again to PNG image files. The PNG image files are the output of the data generation pipeline and input for the NNs. The whole process is depicted in the figure below.

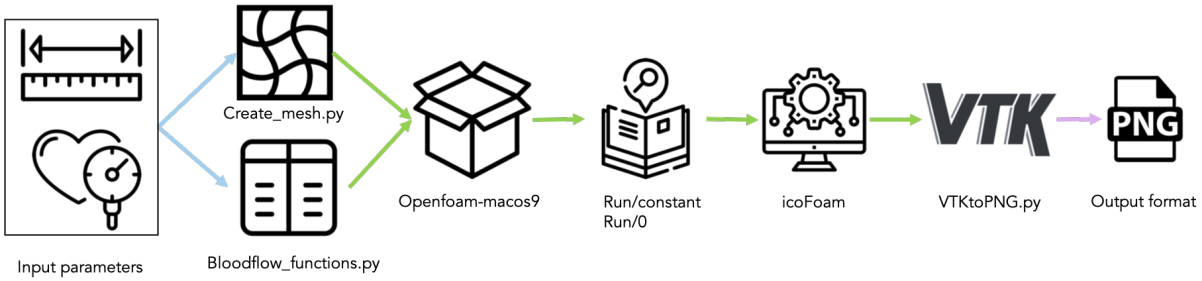

Figure 3.2: Data Generation Pipeline, icons from: Flaticons.com

3.2 Network Architectures

Taking into account the possibilities discussed in chapter 2, first of all a convolutional autoencoder was chosen for order reduction, given the high dimensionality of the input data and desired flexibility in the computational domain. Two network architectures are proposed: an autoencoder with a NN for time evolution and an autoencoder with a RNN using GRUs for time evolution.

Secondly, a 3D convolutional network with dimensions (x,y,t) was explored.

3.2.1 Auto-encoder

3.2.2 Neural Network

3.2.3 Recurrent Neural Network with GRUs

3.3 Tuning, Training and Testing protocols

4 | Results

5 | Discussion

6 | Conclusions and Recommendations

Bibliography

[1] Gianluigi Rozza Alfio Quarteroni, ed. *Reduced Order Methods for Modeling and Computational Reduction*. Springer, Cham, 2014. DOI: 10.1007/978-3-319-02090-7.

[2] Byungsoo Kim et al. "Deep Fluids: A Generative Network for Parameterized Fluid Simulations". In: *Computer Graphics Forum* 38 (2 May 2019), pp. 59–70. ISSN: 14678659. DOI: 10.1111/cgf.13619.

[3] My Ha Dao. *Projection-Based Reduced Order Model for Simulations of Nonlinear Flows with Multiple Moving Objects*.

[4] Zoran Popovic Adrien Treuille Andrew Lewis. "Model reduction for real-time fluids". In: *ACM Trans. Graph.* 25 (July 2006), pp. 826–834. DOI: 10.1145/1141911.1141962.

[5] Jimmy Ba Diederik P. Kingma. "Adam: A Method for Stochastic Optimization". In: 2015. DOI: 10.48550/arXiv.1412.6980.

[6] Manuel Lopez-Martin, Soledad Le Clainche, and Belen Carro. "Model-free short-term fluid dynamics estimator with a deep 3D-convolutional neural network". In: *Expert Systems with Applications* 177 (2021). ISSN: 09574174. DOI: 10.1016/j.eswa.2021.114924.

[7] Thomas Simpson, Nikolaos Dervilis, and Eleni Chatzi. "Machine Learning Approach to Model Order Reduction of Nonlinear Systems via Autoencoder and LSTM Networks". In: *Journal of Engineering Mechanics* 147 (10 2021). ISSN: 0733-9399. DOI: 10.1061/(asce)em.1943-7889.0001971.

[8] Axel Klawonn Matthias Eichinger Alexander Heinlein. *Technical Report Series Center for Data and Simulation Science Surrogate Convolutional Neural Network Models for Steady Computational Fluid Dynamics Simulations SURROGATE CONVOLUTIONAL NEURAL NETWORK MODELS 1 FOR STEADY COMPUTATIONAL FLUID DYNAMICS 2 SIMULATIONS*. 2016.

[9] Youngkyu Kim et al. "A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoder". In: *Journal of Computational Physics* 451 (2022). ISSN: 10902716. DOI: 10.1016/j.jcp.2021.110841.

[10] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[11] R. Hu et al. "Rapid spatio-temporal flood prediction and uncertainty quantification using a deep learning method". In: *Journal of Hydrology* 575 (Aug. 2019), pp. 911–920. ISSN: 00221694. DOI: 10.1016/j.jhydrol.2019.05.087.

[12] Shudong Yang, Xueying Yu, and Ying Zhou. "LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example". In: June 2020, pp. 98–101. DOI: 10.1109/IWECAI50956.2020.00027.

[13] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019). ISSN: 10902716. DOI: 10.1016/j.jcp.2018.10.045.

[14] Wenqian Chen et al. "Physics-informed machine learning for reduced-order modeling of nonlinear problems". In: *Journal of Computational Physics* 446 (2021). ISSN: 10902716. DOI: 10.1016/j.jcp.2021.110666.

[15] Suraj Pawar et al. "Model fusion with physics-guided machine learning: Projection-based reduced-order modeling". In: *Physics of Fluids* 33 (6 2021). ISSN: 10897666. DOI: 10.1063/5.0053349.

[16] Dmitrii Kochkov et al. "Machine learning–accelerated computational fluid dynamics". In: *Proceedings of the National Academy of Sciences* 118.21 (2021), e2101784118. DOI: 10.1073/pnas.2101784118. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2101784118. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2101784118.

[17] H. Švihlová et al. "Determination of pressure data from velocity data with a view toward its application in cardiovascular mechanics. Part 1. Theoretical considerations". In: *International Journal of Engineering Science* 105 (2016). ISSN: 00207225. DOI: 10.1016/j.ijengsci.2015.11.002.

[18] Elie Nader et al. "Blood Rheology: Key Parameters, Impact on Blood Flow, Role in Sickle Cell Disease and Effects of Exercise". In: *Frontiers in Physiology* 10 (2019). ISSN: 1664-042X. DOI: 10.3389/fphys.2019.01329. URL: https://www.frontiersin.org/article/10.3389/fphys.2019.01329.

[19] Micheal Shmukler. *The Physics Factbook*. 2004. Chap. Density of Blood. URL: https://web.archive.org/web/20060919051122/http://hypertextbook.com/facts/2004/MichaelShmukler.shtml.

[20] Alexander Heinlein. *Parallel Overlapping Schwarz preconditioners and multiscale discretizations with applicationsto fluid-structure interaction and highly heterogeneous problems*. 2016.

[21] Amirhossein Arzani and Scott T.M. Dawson. *Data-driven cardiovascular flow modelling: Examples and opportunities*. 2021. DOI: 10.1098/rsif.2020.0802.

[22] H. Švihlová et al. "Determination of pressure data from velocity data with a view towards its application in cardiovascular mechanics. Part 2. A study of aortic valve stenosis". In: *International Journal of Engineering Science* 114 (2017). ISSN: 00207225. DOI: 10.1016/j.ijengsci.2017.01.002.

[23] Amir Barati Farimani, Joseph Gomes, and Vijay S Pande. *Deep Learning the Physics of Transport Phenomena*. 2017.

[24] M. Cheng et al. "An advanced hybrid deep adversarial autoencoder for parameterized nonlinear fluid flow modelling". In: *Computer Methods in Applied Mechanics and Engineering* 372 (Dec. 2020). ISSN: 00457825. DOI: 10.1016/j.cma.2020.113375.