GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Max Planck Institute for
Dynamics and Self-Organization

Masterarbeit

Locating Excitation Sites and Reconstructing Spiral Waves from Multichannel-ECG Signals using Artificial Neural Networks

angefertigt von

Roland Stenger

aus Erlangen

am Max-Planck-Institut für Dynamik and Selbstorganisation

Bearbeitungszeit: 8. Oktober 2020 bis 22. Mai 2021

Betreuer: Baltasar Rüchardt

Erstgutachter: Prof. Dr. Ulrich Parlitz

Zweitgutachter: Prof. Dr. Alexander Ecker

Acknowledgments

First of all, I want to thank my supervisors Ulrich Parlitz and Baltasar Rüchardt for their outstanding support. I am really grateful for the fascinating insight into the research of the group at this Max-Planck-Institute, and that it was possible for me to work on my master's thesis there. Furthermore, I want to thank Stefan Luther and Ulrich Parlitz for giving me the opportunity for writing my master thesis at their research group. Also I want to thank Alexander Ecker for his efforts as second corrector.

I felt very comfortable in the group and enjoyed the pleasant atmosphere, which I noticed especially in the weekly group meetings. The discussions with everybody were always very constructive and furthermore very supportive for my own research and to gain a broader picture of the research.

In general, the communication was excellent and I cannot emphasize enough how helpful it was that I could always rely on the support of Ulrich and Baltasar. It was also possible to continue this good communication in the home office without any problems. Feedback on my questions towards Baltasar and Ulrich, with whom I discussed and worked out most of it, usually came within a very short time.

I am glad about my fellow students and good friends Ansgar Adler, Stefan Kiebacher and Tobias Strübing who shared the time with me during the whole Bachelor and Master of Physics in Göttingen, without whom the studies would have been much less fun. Finally, I want to thank my family, who have been a huge support for me during the whole time of my studies. I am very aware of this fortunate circumstance.

Symbols, notation and abbreviations

Machine learning

Adam Adam optimizer, short for Adaptive Moment Estimation

ANN Artificial neural network

C-LSTM Convolutional long-short-term memory

CNN Convolutional neural network

DNN Deep neural network

lr Learning rate

LU Length unit

LSTM Long-short-term memory

MAE Mean absolute error

MSE Mean squared error

ReLU Rectified Linear Unit

RNN Recurrent neural network

seq2seq Sequence-to-sequence

SGD Stochastic gradient descent

ST-LSTM Spatio-temporal long-short-term memory

Biology

AV (node) Atrioventricular node

SA (node) Sinoatrial node

Diverse

CT Computed tomography

ECG Electrocardiography

FEM Finite element method

ECG Inverse Electrocardiography

Contents

CHAPTER 1

Introduction

The heart is essential for human survival by maintaining two blood circuits within the organism. Oxygen-rich blood is sent from the lungs to the organs, and oxygen-depleted blood is sent from the organs to the lungs. The heart makes this possible through the coordinated contraction control of its cardiac muscle cells, from which a pump effect results. The contraction, respectively the excitation of a cardiac muscle cells is caused by an action potential, which is induced by the sinus node. The potential is transmitted from cell to cell whereby a voltage is measurable. This electrical activity creates spatio-temporal excitation patterns inside the heart as well as on its surface. Arrhythmia can be understood as *dynamical disease*, where the cardiac dynamics shows abnormal chaotic behavior as a result of which an effective pump mechanism can fail [7]. Therefore, the identification of such dynamics is an important diagnostic procedure.

This work consists of two similar numerical experiments that attempt to reconstruct the electrical activity of the heart using methods from the field of machine learning. The first experiment employs with simulated electrocardiography (ECG) signals from a simulated heart by processing them with a neural network to obtain information about the electrical activity at the surface of the heart. It does so by distinguishing a variety of different excitation patterns, based on a sequence of the recordings from multiple ECG electrodes. The ECG signals are influenced only by the surface dynamics of the heart, making the experiment an attempt to test the feasibility of neural networks to characterize these surface dynamics.

The second (numerical) experiment is about an approach to reconstruct excitation patterns under the surface, based only on the temporal development of the electrical activity at the surface. Here the 3D electrical activity is simulated as well. However,

the simulation takes place in a simplified geometry (a cube). It tests the principle feasibility of such predictions from neural networks. After identifying boundaries of the predictions of the neural networks independent of the geometry, the transfer of this experiment to more complicated geometry such as the heart is subject of the future research.

Since both experiments have in common that they use methods from the field of machine learning, some relevant techniques are firstly introduced in chapter 2. This includes an introduction into deep neural networks, a type of non-linear functions with a large computational graph, as well as the *training process* of such functions, which stands for the adjustment of the neural networks immanent parameters, with respect to a given problem. It follows the documentation of the two experiments in chapters 3 and 4. Since both experiments are numerical, the process of data generation by simulations is explained in each case. In addition, there is a brief description of the mathematical models for simulating spatio-temporal excitation. Furthermore the exact implementation of the neural networks to solve the concerning problems in each experiment is explained.

All simulations, training processes of the neural networks and evaluations are carried out using the Python programming language. The Python libraries Keras and Pytorch are used to define the computational graphs of the neural networks and to train them. All scripts and programs of both experiments can be found on github at https://github.com/RolandGit95/HeartActivity.

CHAPTER 2

Deep learning and neural networks

Concepts that fall under the definition of *deep learning* become more and more popular, since a wide range of tools are based on it, which often achieve state-of-the-art results. Important fields are, for example, language processing, like machine translation [20] or speech recognition [3] in which deep learning algorithms with respective deep neural networks achieve state-of-the-art results. Also for games like Chess or Go, tools, based on deep learning are the strongest players [32]. Another example for a breakthrough in recent time is a tool called AlphaFold2 which „solved“ the problem of protein folding [38], according to the problem definition of the CASP-competition [26] in which AlphaFold2 achieved the best result in 2020. It can be said that deep learning is an important topic that has made possible a large number of breakthroughs in a wide range of research areas.

This chapter describes the implementation of deep learning models to process the data in the two experiments. Special architectures like *convolutional neural networks* (CNNs) and *recurrent neural networks* (RNNs) are used to address the given problems. The mathematics behind these neural networks, as well as the implementation is part of this chapter.

The term deep learning, which is a sub field of machine learning, refers to the concepts of software development, where a computer is not directly programmed to solve a problem but is given instructions on how to find a good solution. The approach, that falls into the area of deep learning, is the training of deep neural networks (DNN). DNNs are a type of artificial neural network which are characterized by a big number of composed functions, so-called *layers*, which are included in the neural network to build a large computational graph. A typical example of a neural network, regardless if it is deep or not, is the feed-forward network. It consists of the

combination of several similar layers, where each layer is built by a combination of a linear function the *activation function*, which in non-linear. Each layer provides a new representation of the input which passes through the network. With each layer, the representation can be developed in a way, that it finally provides a meaningful representation, with respect to a given problem. The search for this meaningful representation can succeed by adjusting the parameters of the network in the training process. In the following section, the mathematical basics behind this concept and important variations of the mentioned layers are introduced.

2.1 Artificial Neural networks

Artificial neural networks are inspired by biological neural systems. Yet strongly simplified, the terminology is oriented on parts and structures of the brain. A first approach for artificial neural networks was introduced by Warren McCulloch and Walter Pitts 1943 in their paper „A logical calculus of the ideas immanent in nervous activity“ [23], where they explain their idea of a computational model which could mathematically represent how animal brains work on the scale of neurons to perform tasks. Their neural network is built from binary units with one or multiple binary inputs. It is able to perform simple binary classifications respectively logical computations. However, a more general and nowadays more common description of a neuron is based on continuous calculations, whose mathematical concepts are explained in the following.

2.1.1 Mathematical computations in artificial neural networks

The smallest functional unit of an artificial neural network is the neuron. As mathematical function f it takes a number of weighted scalars and outputs the sum of them. It is an affine linear function, represented by a weight-vector $\mathbf{w}$ and a bias b. The computations of a neuron are

$$y = f_{\text{neuron}}(\mathbf{x}) \equiv \mathbf{w}^T\mathbf{x} + b, \tag{2.1}$$

where $\mathbf{x} = (x_1, ...x_n)^T \in \mathbb{R}^n$ is a n-dimensional input of the neuron and the bias $b \in \mathbb{R}$ is a 1-dimensional scalar.

However, a layer in a feed forward neural network can contain a bunch of neurons, where computation for all neurons thus is written in the form

$$\mathbf{y} = f(\mathbf{x}) \equiv \underline{\underline{\mathbf{w}}}\mathbf{x} + \mathbf{b}, \tag{2.2}$$

with the weight-matrix $\underline{\underline{\mathbf{w}}} \in \mathbb{R}^{n \times m}$ and bias-vector $\mathbf{b} \in \mathbb{R}^m$ with the number of neurons m. This computation builds with an activation-function A a single layer in a feed-forward network. Typically, activation-functions are the logistic (or sigmoid) function

$$\sigma(y) = \frac{1}{1 + \exp(-y)}, \tag{2.3}$$

or the rectified linear unit (ReLU-function)

$$\mathrm{ReLU}(y) = \max(0, y), \tag{2.4}$$

to mention two of the most frequently used. Furthermore, in this work, the identity function is used in the neural network within the first experiment, which is defined as

$$\mathrm{ID}(y) = y. \tag{2.5}$$

The computation within a single layer can be written as

$$\mathbf{y} = A \circ f(\mathbf{x}). \tag{2.6}$$

Note that the activation-function computes every scalar within an input-vector $\mathbf{y}$ independently. In figure 2.1 the analogy to a biological neural network is visualized, where the weight matrix is represented by connections between neurons and a neuron is represented by the nodes (blue circles). In this case, a neural network is shown where three layers are composed.

A key characteristic of a neural network is that the weight matrices and biases are adjusted in a way that the network gives a useful output $\mathbf{y}$ regarding a given

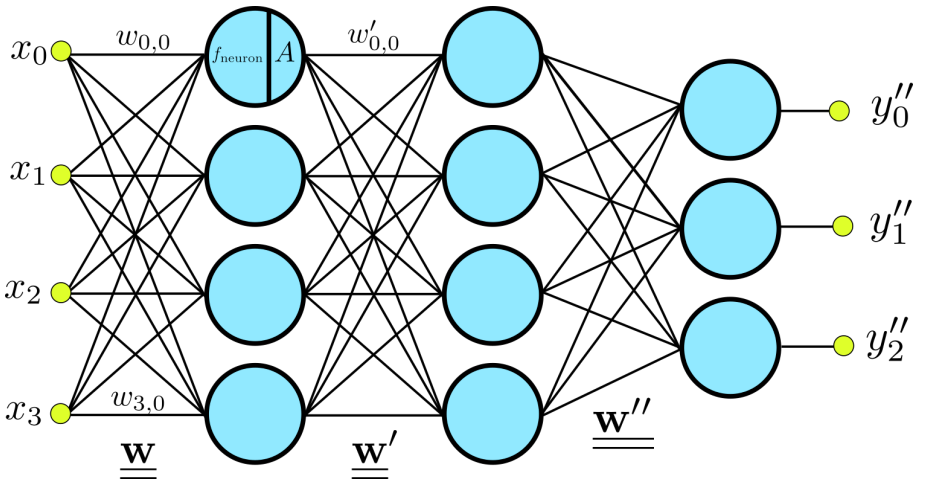

Fig. 2.1: Schematic view of the computation within a feed forward network, a simple neural network architecture. The parameter $\underline{\underline{\mathbf{w}}}$ represents the weight matrices, where $w_{i,j}$ is an element of the weight-matrix $\underline{\underline{\mathbf{w}}}$.

problem. The parameter will be adjusted in the training process whose fundamentals are introduced in the following section.

2.1.2 Training of neural networks

The search for the network's optimal parameter adjustment is called training process. The aim of training a neural network is that the network performs well on yet *unseen* data when the performance is measured by a loss-function, which computes the discrepancy between predictions and the *true values*. Predictions are the output of the neural network, previously noted as $\mathbf{y}$ while the *true values* are the aim for what the network should predict, based on a given input. Mostly, the training is a minimization process by applying a distance metric such as mean squared error as loss-function to calculate the validity of the prediction.

The kind of training process where the input and output is known is called supervised learning. Other types are unsupervised learning on which the data is not given and the model has find pattern in the input data, and reinforcement learning where an *agent* is getting feedback from actions which are tested out. However, since this work contains two experiments which focus only on supervised learning, here only this kind is considered in the explanation. While the training process stands for the adjustment of the neural networks parameter, the adjustment is performed by algorithms, the *optimizers*.

2.1.2.1 Optimizer

Optimizers are iterative algorithms to find the optimal network parameters which minimize a loss function L that quantifies the discrepancy between the network's prediction and the desired outcome[1]. With each iteration in the optimizer algorithm, a sub set or the whole training dataset, the *batch*, is passed to the loss function, whose gradient with respect to the networks parameter θ is then used to update these parameters. The gradient calculation is performed by the back-propagation algorithm [30] in the parameter space. There exists a lot of different optimizers respectively update rules, from which one of the most basic one called *gradient descent* is introduced here, to give an idea about the general functionality of optimizers. Every optimizer of this kind is based on an update rule including the gradient of a loss function. An update in the *gradient descent* algorithm is computed according to

$$\theta_{\text{new}} = \theta + \eta \cdot \nabla_\theta L(f(x|\theta), y), \tag{2.7}$$

where η is the learning rate, a quantity to adjust how much the network parameters θ can change within an update. The right choice of this parameter is critical for convergence of the network parameters [41]. Here, x is the input for the neural network, where its dimensionality depends on the neural network.

There are three types of gradient descent methods which differ in the amount of data respectively the batch size, to be passed through one iteration/update. If x contains one *data point*, the optimizer performs a parameter update with every single training example, which is called *stochastic gradient descent* (SGD). In contrast to that is the *batch gradient descent* method (BGD) which calculates the gradient on the whole dataset to perform a parameter update. The third possibility is a compromise of both, where the optimizer only uses a part of the dataset, which contains more than one single example to perform one parameter update. This method is called *mini batch gradient descent*. In this work, regardless of the kind of optimizer, a mini batch is always used in the training processes.

Probably the most famous optimizer in machine learning is the Adaptive Moment Estimation optimizer (Adam) [19], which is specifically designed to train neural net-

[1] Dependent on the loss function, the goal could also be the maximization. Nonetheless every loss function can be reformulated so that the training process is again a minimization task, without changing the properties of the loss function.

works. Firstly introduced in 2014 it showed big performance gains in training speed. In this work, the Adam optimizer is used for every training process in both experiments.

Overfitting

When training a neural network on a dataset, one wants a loss function to get smaller during the process, but also the ability to generalize what it has learned from the training data on yet unseen data. A problem can appear if a model relies too much on the training data that it even considers secondary patterns like noise. For a neural network which should be able to generalize well, this is a disadvantage. The right graphic of figure 2.2 shows an example, where a model which performs a regression on the dataset fits very well on the known data (red points), but at the same time it is not able to capture the most dominant trend which is better handled by a polynomial regression of order 4 (figure in the middle). Overfitting can appear if the model is too complex (polynomial regression of order 16 instead of a less complex order 4), and/or the model was trained for too many iterations. This is not an example of an overfitting neural network, but the principles which are visualized are the same.

Underfitting

The problem of underfitting might be regarded as the opposite of overfitting, which appears if the model is not capable of processing the complexity of the dataset. Reasons for this can be a model which is not trained enough that it did not learn *enough* from the data. Otherwise the model could be *too simple*. An example is visualized in figure (2.2, left). The model is not able to capture the main trend as well. This is due to the fact that in this case a linear regression was chosen as model.

To address these problem, there are a few regularization techniques which are also applied to the neural networks in this work.

2.1.3 Regularization against over- and underfitting

The following methods intend to prevent overfitting and underfitting and can be divided into two areas: On the one hand, methods that are implemented in the form of additional layers in a neural network, such as dropout and batch-normalization, are introduced. On the other hand, there are rules that only take effect during the training process. The difference is that some methods co-define the computational graph and may have trainable parameters, while other methods are applied during

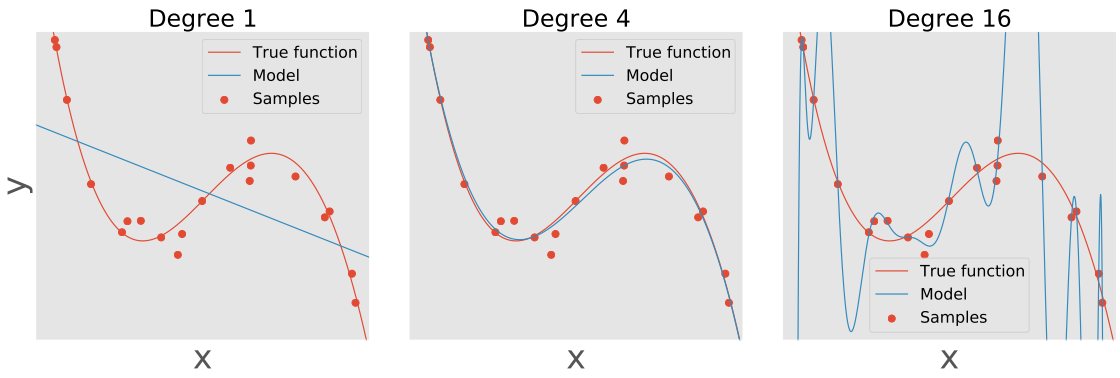

Fig. 2.2: Examples of underfitting (left) and overfitting (right) on a dataset whose (noisy) data samples underlie a polynomial function of order 4. The blue line in each figure shows the curve of a polynomial regression of a certain degree.

the training process and are detached from the computational graph. All of these methods appear in at least one of the two experiments from the chapters in 3 and 4.

2.1.3.1 Dropout

By ignoring a set of neurons, a neural network can imitate *sub networks* and is therefore able to have a dynamic architecture. The regularization technique called *dropout* randomly removes neurons during any execution of the network as visualized in figure 2.3. Firstly introduced by N. Srivastava et al. [34], dropout can help to prevent overfitting by not allowing the network to have a rigid architecture which may be too oriented to the training data.

2.1.3.2 Batch normalization

Batch normalization can increase the stability of a neural network by normalizing the input of single layers in the neural network. It can increase the training speed by being able to handle bigger learning rates from the optimizer. Deep neural networks are sensitive to the choice of initial weights and the training algorithm respectively the choice of the learning rate. Batch normalization increases the robustness with regard to these choices [18].

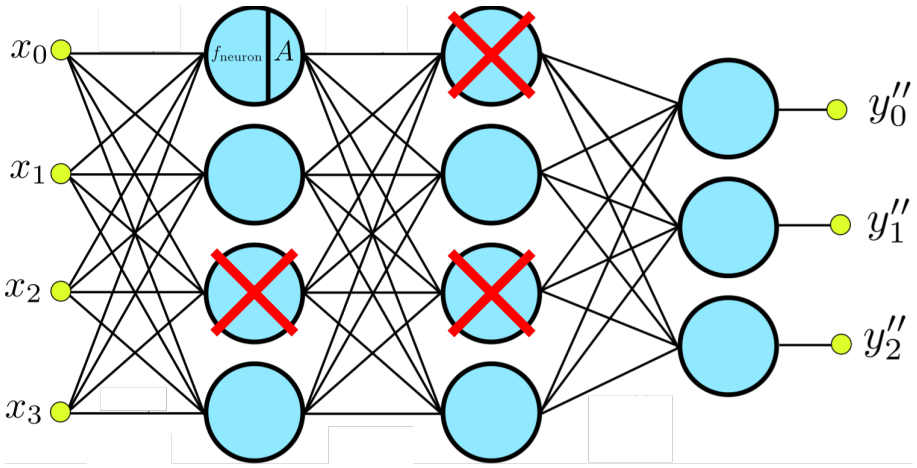

Fig. 2.3: Schematic view of the computations within a feed forward network with an applied dropout rule.

2.1.3.3 Early stopping

While the two previous regularization techniques require to be implemented with the neural network, early stopping is a rule that takes effect during the training process. It interrupts the training process as soon as there is no improvement in the loss for the test set for a longer period, a so-called *plateau*.

2.1.3.4 Reduce learning rate on plateau

This method also refers to the ongoing training process, where the aim is to reduce the learning rate as soon as a metric (loss function) does not improve over a certain number of training iterations.

2.2 Convolutional neural networks

In the same way neural networks have been inspired by biological brains, convolutional neural networks (CNNs) might have gained their inspiration from the brain's visual cortex. They are commonly designed for tasks in computer vision and some state-of-the-art results are performed by neural network designs which includes convolutional operations. Examples are *EfficientNet* in image classification [37] or *EfficientPS* in semantic segmentation [25]. Nonetheless, CNNs are not restricted

to computer vision tasks, but also successfully used for example in *ContextNet* for speech recognition [15] or in natural language tasks like machine translation as in *ConvS2S* [11]. In this work, CNNs are used to process ECG signals. Firstly, the focus on the following explanation of the concepts behind CNNs lays on visual tasks, since it is the most common use case. Later in this work, instead of a 2D image as an input, a 1-dimensional ECG signal is used as well, but the mathematical principles are the same, which can easily be transfered to a lower dimensional case.

2.2.1 Kernel convolution

A convolutional neural network is based on the kernel convolution. It is a discrete operation where a small matrix W, called kernel or filter, is passed over an 2-dimensional image I and transforms it in dependence of the kernel and image values. The operation is computed as

$$g_{x,y} = W * I_{x,y} = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} W_{dx,dy} \cdot I_{x+dx,y+dy}, \tag{2.8}$$

where a and b indicate the size of the kernel W. The function is calculated for all x and y on the image on which the filtered image g is based on. The symbol $*$ stands for the convolutional operation.

2.2.2 Multi channel convolution

In most images, each pixel is represented by three values which define the color (red, green and blue). The convolutional operation should be able to take these three values (in the following called channels) into account. It is also important that multiple filters can be applied to the image, so that the input image and the output image have multiple channels. A more general description of the kernel convolution with multiple input and output channels is given by

$$g'_{i,x,y} = W'_i * I'_{x,y} = \sum_{j=1}^{N} \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} W'_{i,dx,dy} \cdot I'_{j,x+dx,y+dy}, \tag{2.9}$$

with N as number of channel within the input image (for example $N = 3$ in an rgb image), which is a hyperparameter of a convolutional neural network. Applying

equation 2.9 with every kernel W'_i in $\mathbf{W}' := (W'_1, ..., W'_M)$ gives a multi channel output $\mathbf{g}'_{x,y} := (g'_{1,x,y}, ..., g'_{M,x,y})$ from an input image $\mathbf{I}' \in \mathbb{R}^{C \times H \times W}$, where C are the number of channels, H the height, and W the width of the image.

2.2.3 Valid convolution and same convolution

The equations above don't consider yet what the computation looks like at the edges when $x + dx$ or $y + dy$ exceeds the image boundaries. Two common rules are called *valid* and *same*. *Valid* convolution calculates only values for the respective point (x, y) as far as it lies in the image, while with *same* convolution, zeros are added at the edge of the input image, so that the filtered image has the same width and height as the input. The parameter to implement these rules is called padding. It specifies the thickness of the additional margins added to the image with zeros.

2.2.4 Strided convolution and pooling

In previous equations, the kernel is only shifted with a step size (stride) of one pixel. However, one can increase the step size if $\mathbf{g}$ should have smaller spatial dimensions. Furthermore, as a consequence of a bigger stride, each pixel in $\mathbf{g}$ is dependent of a bigger sub field, the so called *receptive field* on the input image. The receptive field is the region within the input image that a certain feature in the CNN is affected at. To process relations of an image, the size of the receptive field is an important aspect (at least in image classification), as described by Araujo et al. [4] as

> *We observe a logarithmic relationship between classification accuracy and receptive field size, which suggests that large receptive fields are necessary for high level recognition tasks, but with diminishing rewards.*

While *stride* is a hyperparameter of the kernel convolution operation, *max-pooling* is a technique that functions as an independent layer in a neural network. It divides the input (here a 2D-image) into subsets, like 2×2-fields of spatial neighbouring pixels and outputs only the maximal image value of each. This technique reduces the spatial dimension of the input and increases the receptive field.

In this work, convolutional layers are implemented within neural networks of both experiments. The first experiment applies a 1D-convolution to process through electrocardiography signals, while in the second experiment, 2D-convolutions are used

to encode spatial correlations and decode a low dimensional representation of images.

This section showed how convolutional neural networks process through spatial correlations. The terminology *spatial* does not necessarily refer to data points that are spatially related. It is to be understood more generally in the sense that data points, in the form in which they are stored (such as the pixels of an image) also contain information through their position and their neighbors. In contrast to spatial data, temporal data has a direction, where recurrent neural networks are designed to process temporal dynamics in sequences.

2.3 Recurrent neural networks

A prominent domain for recurrent neural networks are sequence-to-sequence problems. It is about training neural networks to convert a sequence to another sequence. A famous task in this domain is machine translation where a sequence of words in one language is transformed to another sequence of words of a different language. Other important tasks are for example speech-recognition where an encoded voice recording has to be converted into a sequence of words, or next-frame-prediction, a task to predict the future frames of a video. In each example, recurrent networks were considered as state-of-the-art, at least for a while [42], [24], [40].

A recurrent neural network is a form of neural network which is, in combination with an iterative update loop of its internal state (memory), able to process through an input sequence with varying length. With every iteration, an internal state, the *hidden state*, which has a fixed dimensionality, is evolving to be dependent on every previous time step. The *hidden state* can be calculated regardless of the length of the input sequence and provides a representation of the sequence.

A characteristic that is generally seen with sequence-to-sequence (seq2seq) problems is that both the input sequence and the output sequence are of variable length. For example, a translation model has to be able to take a sequence with varying length of words to output a translation as a sequence with a varying length as well. Furthermore, a strong translation model should be able to translate single words as well as a long sequences of hundreds of words. In this work, the problem of the experiment from chapter 4 also faces the characteristic of a seq2seq problem, where a recurrent neural network is used as well.

The following equations describe the computations within a very basic recurrent

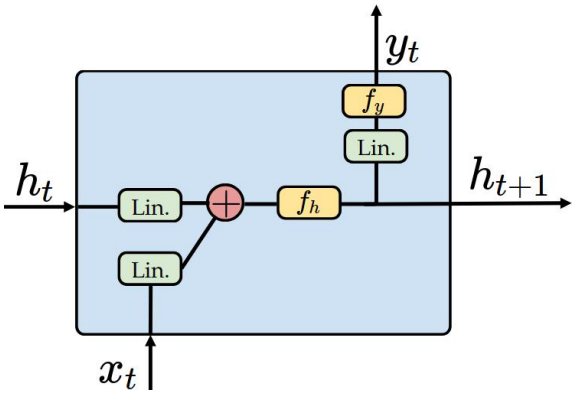

Fig. 2.4: Visualization of the calculations within an Elman network.

neural network which build the so called Elman network [12]. Given a sequence of inputs $(x_1, x_2, ..., x_T)$, the hidden state h_t evolves as

$$\begin{aligned} h_{t+1} &= f_h(\mathbf{W}_x \cdot x_t + \mathbf{W}_h \cdot h_t + b_h), \\ y_t &= f_y(\mathbf{W}_y \cdot h_{t+1} + b_y). \end{aligned} \tag{2.10}$$

Furthermore, it gives an output sequence $(y_0, y_1, ..., .y_T)$ of the same length as the input sequence. The functions f_h and f_y are activation functions like tanh or the logistic function.

2.3.1 Training of recurrent neural networks

The computation of the gradient of recurrent neural networks is not possible by the normal back-propagation algorithm. However, there is a generalization called *back-propagation through time* or *BPTT* [27]. The key idea behind this algorithm is to unfold the network through time into a feed forward network with multiple layers as visualized in figure 2.5. The depth of the unfolded network depends linearly on the length of the input sequence.

During training, the gradient tends to vanish or explode, as firstly discovered by Hochreiter et al. in 1991 [17]. He described it as follows:

With conventional „Back-Propagation Through Time“ [...], error signals

„flowing backwards in time“ tend to either blow up (1) or vanish (2): the temporal evolution of the backpropagated error exponentially depends on the size of the weights. Case (1) may lead to oscillating weights, while in case(2) learning to bridge long time lags takes a prohibitive amount of time, or does not work at all.

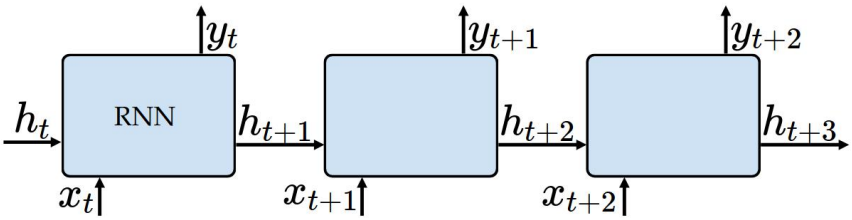

Fig. 2.5: Unfolded recurrent neural network over three time steps.

Within the training algorithm *gradient descent*, which includes back-propagation, these difficulties are theoretically validated with any kind of loss function [8]. Various methods have been proposed to deal with the problem of vanishing and exploding gradient, whereby the most successful is the Long-short-term-memory (LSTM). Hochreiter and Schmidhuber introduced a special kind of RNN in 1997 which is able to prevent the problem of vanishing gradient [16].

2.3.2 Variations and extensions of RNNs

2.3.2.1 Bidirectional RNN

Firstly introduced by M. Schuster et al. in 1997 [33], Bidirectional RNNs combine the input sequence and the inversion through time of the same sequence. It is the implementation of two independent RNNs which compute through the forward and backward direction of the input, whose output is stacked together. With this form of RNNs the output layer can combine information from the past and future.

2.3.2.2 Stacked RNN

A method for stacking RNNs is taking the output sequence (y_1, ..., y_T) or the sequence of hidden states (h_1, ..., h_T) as input for a new RNN. Although it is not

theoretically clear why stacked RNNs perform in certain tasks better than single-layer RNNs, in practice they provide a higher learning capacity [14, p.51].

2.3.3 Long-short-term-memory

LSTMs work by the same principle as RNNs through evolving a hidden state through an input sequence, but they use a different function to compute the hidden state which, furthermore, includes an additional state, the *cell state*. Within the computations in a LSTM it is possible that the gradient of the cell state is unchanged through time. It is possible to process long term dependencies in a sequence more efficiently than with the previously introduced RNN. Furthermore, a LSTM overcomes the problem of a vanishing gradient [16].

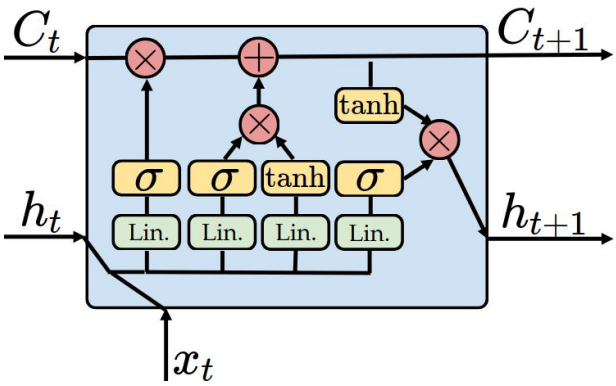

Fig. 2.6: Visualization of the calculations within an Long-short-term-memory (LSTM).

The calculations within a LSTM are given by

$$\begin{aligned} g_t &= \tanh\left(\mathbf{W}_{cx} \cdot x_t + \mathbf{W}_{ch} \cdot h_t + b_g\right), \\ i_t &= \sigma\left(\mathbf{W}_{ix} \cdot x_t + \mathbf{W}_{ih} \cdot h_t + b_i\right), \\ f_t &= \sigma\left(\mathbf{W}_{fx} \cdot x_t + \mathbf{W}_{fh} \cdot h_t + b_f\right), \\ o_t &= \sigma\left(\mathbf{W}_{ox} \cdot x_t + \mathbf{W}_{oh} \cdot h_t + b_o\right), \\ C_{t+1} &= f_t \circ C_t + i_t \circ g_t, \\ h_{t+1} &= o_t \circ \tanh(C_t), \end{aligned} \tag{2.11}$$

as shown graphically in figure 2.6. Here, the symbol ∘ denotes the Hadamard product, also known as pointwise multiplication.

2.3.4 Variations and extensions of LSTMs

2.3.4.1 Convolutional LSTM

At least since the performance of convolutional neural networks (CNNs) in the ImageNet competition [31] for image classification, it is empirically proven that CNNs are more effective in certain computer vision tasks, than fully connected neural networks. In the field of sequential learning, computer vision tasks are also formulated, such as video-classification or video-frame interpolation. The second experiment is also a type of vision task. However, LSTMs as implemented in the equation from 2.11 can be regarded as recurrent extension of a fully connected neural network, which is not as efficient in processing spatial correlations as a CNN.

A LSTM with convolutional structure addresses this problem. It is better performing in certain tasks which include spatio-temporal data, as proven for example by Wu et al. for future video synthesis [43]. This concept is used in the second experiment of this work, which also deals with the processing of spatio-temporal data.

The equations, which build a convolutional LSTM, are shown below:

$$
\begin{aligned}
g_t &= \tanh\left(\mathbf{W}'_{cx} * x_t + \mathbf{W}'_{ch} * h_{t-1} + b_g\right), \\
f_t &= \sigma\left(\mathbf{W}'_{fx} * x_t + \mathbf{W}'_{fh} * h_{t-1} + b_f\right), \\
i_t &= \sigma\left(\mathbf{W}'_{ix} * x_t + \mathbf{W}'_{ih} * h_{t-1} + b_i\right), \\
o_t &= \sigma\left(\mathbf{W}'_{ox} * x_t + \mathbf{W}'_{oh} * h_{t-1} + b_o\right), \\
C_t &= f_t \circ C_{t-1} + i_t \circ g_t, \\
h_t &= o_t \circ \tanh(C_t).
\end{aligned}
\tag{2.12}
$$

As introduced in chapter 2.2.1, the symbol * denotes a discrete convolution while the operators $\mathbf{W}'$ are convolutional kernel. The only difference from a regular LSTM is that the linear operators are replaced by convolutional operators. This version of a LSTM can handle a sequence of spatial data without flattening them to a vector.

2.3.4.2 Spatio-temporal LSTM

As introduced before, a convolutional LSTM is taking into account advantages of convolutions to compute spatial correlation more efficiently. However, in a stacked convolutional LSTM, spatial correlations and temporal dynamics are not equally processed [40], however, these two aspects might be equally important and should be considered equally significant in a machine learning model for spatio-temporal data. To address this problem, Y. Wang et al. introduce in their paper *PredRNN: Recurrent Neural Networks for Predictive Learning using Spatio-temporal LSTMs* [40] a variation of the LSTM and its integration into a stacked recurrent neural network. They do not define how to measure the equality in processing the two aspects (spatial correlations and temporal dynamics) but they formulate it as follows:

> *[...] spatial representations are encoded layer by layer, with hidden states being delivered from bottom to top. However, the memory cells that belong to these [...] layers are mutually independent and updated merely in time domain. Under these circumstances, the bottom layer would totally ignore what had been memorized by the top layer at the previous time step.*

To address this problem they invented a variation of a LSTM, the spatio-temporal LSTM (ST-LSTM), which is defined by:

$$
\begin{aligned}
g_t &= \tanh\left(\mathbf{W}'_{cx} * x_t + \mathbf{W}'_{ch} * h_{t-1} + b_g\right), \\
i_t &= \sigma\left(\mathbf{W}'_{ix} * x_t + \mathbf{W}'_{ih} * h_{t-1} + b_i\right), \\
f_t &= \sigma\left(\mathbf{W}'_{fx} * x_t + \mathbf{W}'_{fh} * h_{t-1} + b_f\right), \\
C_t &= f_t \circ C_{t-1} + i_t \circ g_t, \\
g'_t &= \tanh\left(\mathbf{W}''_{cx} * x_t + \mathbf{W}'_{cm} * M_t^{l-1} + b'_g\right), \\
i'_t &= \sigma\left(\mathbf{W}''_{ix} * x_t + \mathbf{W}'_{im} * M_t^{l-1} + b'_i\right), \\
f'_t &= \sigma\left(\mathbf{W}''_{fx} * x_t + \mathbf{W}'_{fm} * M_t^{l-1} + b'_f\right), \\
M_t^l &= f'_t \circ M_t^{l-1} + i'_t \circ g'_t \\
o_t &= \sigma\left(\mathbf{W}'_{ox} * x_t + \mathbf{W}'_{oh} * h_{t-1} + \mathbf{W}_{om} * M_t^l + b_o\right), \\
h_t &= o_t \circ \tanh(\mathbf{W}_{1\times1} * \left[C_t^l, M_t^l\right]).
\end{aligned}
\tag{2.13}
$$

The difference to a normal LSTM is that here the representations (C, h and additionally M) are not exclusively evolved in time and then transmitted to the next layer. First, the cell state M evolves through the layers of the stacked ST-LSTM, and is then passed to the input of the next time step at the bottom of the network. The idea of this LSTM adaptation is visualized in the figures 2.7 and 2.8. The first figure is a visualization of the computations in the enfolded network, while the second one shows an unfolded stacked ST-LSTM with 4 layers. The orange line is the computational flow of the cell state M in the computational graph.

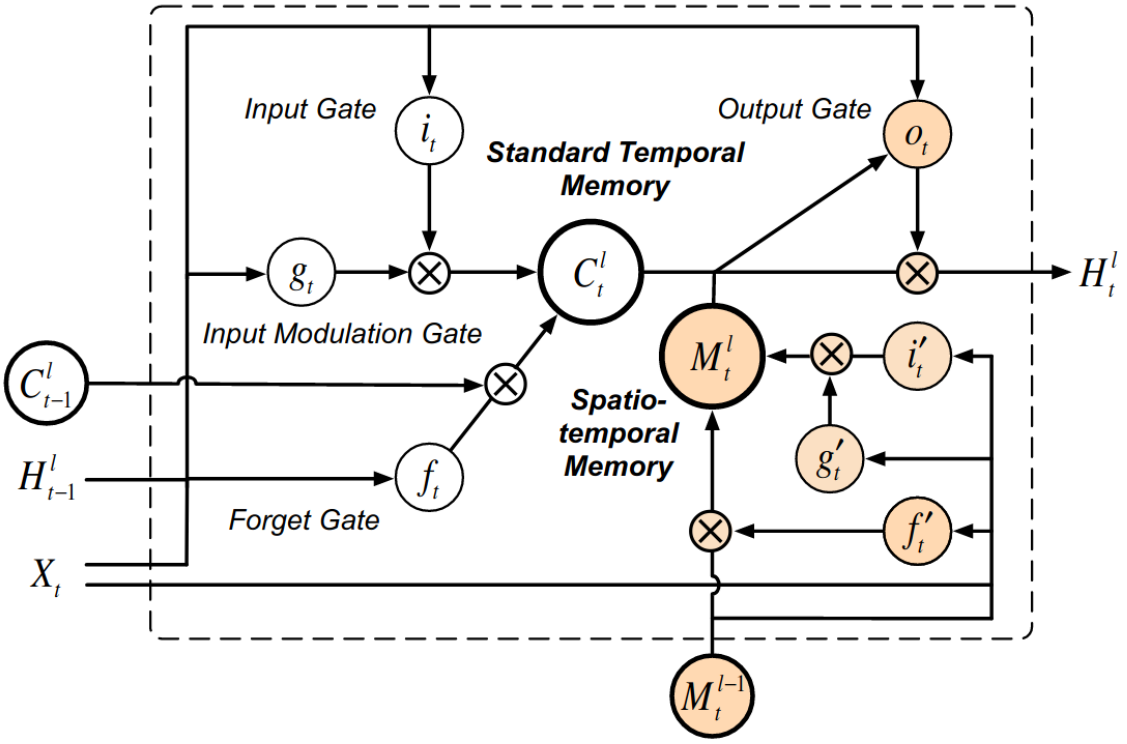

Fig. 2.7: Visualizations of the computations in a spatio-temporal LSTM cell. The orange parts show the additional calculation steps to a LSTM cell [40].

LSTMs are the favored implementation of recurrent neural networks. It is a standard to overcome general problems like vanishing gradient and learning long-term dependencies. Additionally, convolutional LSTMs or spatio-temporal LSTMs are designed to specifically face problems which occur in deep LSTMs where spatial correlation and temporal dynamics are not equally processed.

Although recurrent cells have been mentioned so far, it is not yet clear how exactly this iterative process can be used to face a sequence-to-sequence (seq2seq) problem such as it is posed in the second experiment.

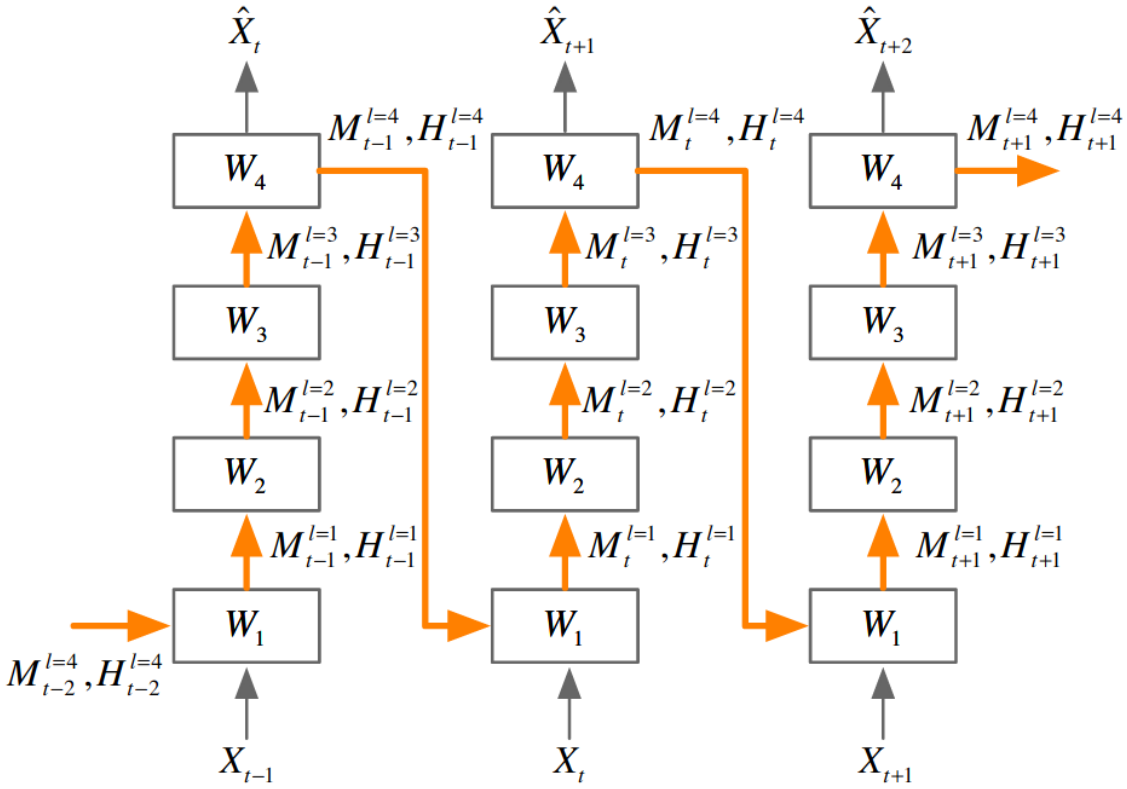

Fig. 2.8: Visualizations of a stacked spatio-temporal LSTM, unfolded through time. The orange arrow marks the computational flow through the layers of the network and time steps of the input sequence [40].

2.4 Seq2Seq-models

A common design for a LSTM-neural network to address a seq2seq problem is an encoder-decoder model. It consists of two LSTMs, one for encoding the sequence to a *thought vector* of fixed dimension, and the other for generating an output sequence with varying length from the thought vector. The thought vector consists of the respective states, which are the hidden state h, and cell state C, with an additional state M, in case of a ST-LSTM. In the example of an encoder-decoder network (figure 2.9), each time step of the input sequence is processed by an encoder independently, such as the output sequence is processed by the decoder (purple parts in the figure). The input sequence of the decoder LSTM (blue parts at the right side of the graphic) is built by a sequence of the same h-value from the thought vector, for every step. Therefore, every iteration in the LSTM takes into account the same representation of the input sequence while evolving another h- and C-value (and probably additionally M) with every iteration.

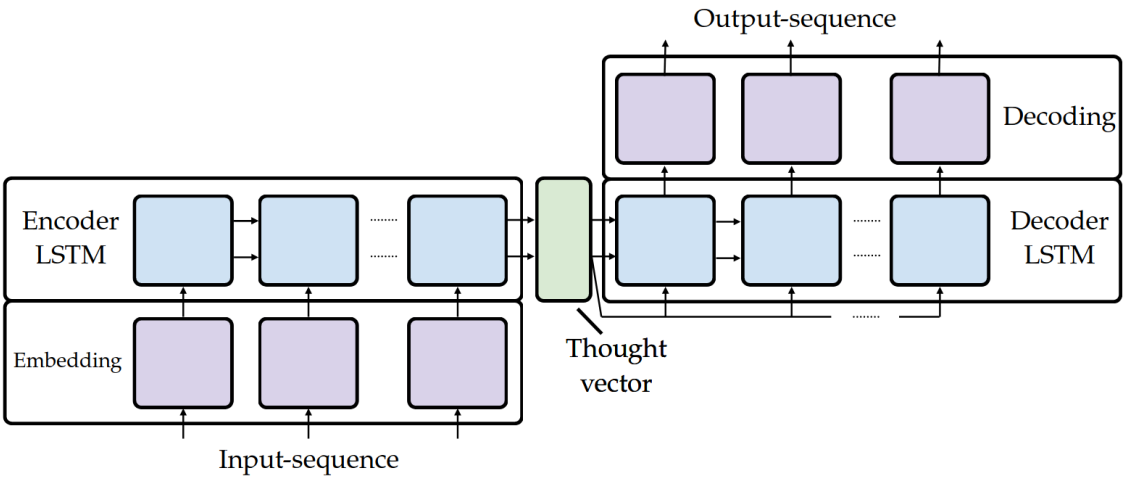

Fig. 2.9: Schematic sequence-to-sequence model (seq2seq) with an encoder-decoder structure.

Within the second experiment (chapter 4), this kind of neural network is used as well. It consists of convolutional neural networks for embedding and decoding and is combined with stacked convolutional LSTMs (and ST-LSTM as well) for seq2seq-processing.

2.5 Tools

This chapter gives a short overview of the programming language Python, which is used to perform the experiments in this work. The neural networks are implemented with help of two of the most famous machine learning libraries: Tensorflow (and its API keras) and Pytorch[2]; which are introduced in this section as well.

2.5.1 Python

Python is a general purpose programming language on a high coding abstraction level [39]. It was released in 1991 and has regular updates to this day. In this work the Python version 3.8 is used, whereby it is backward compatible as long as it concerns versions 3.x. Python is a dynamic language, which means that programs are running through an interpreter who reads one or more source files (or lines of code) and translates them instruction by instruction into machine code, that a computer system can execute them directly. This allows for fast development and debugging

[2]Status in march 1st, 2021; https://paperswithcode.com/trends

of code. The power of Python is mainly given by its large collection of open-source libraries. For example, differential equations in some simulations in this work are solved numerically using the `dolfin` library [21]. Furthermore, Python is the most popular language for machine learning[3]. In this work, the libraries Tensorflow and Pytorch are used. By current status, Pytorch is the most popular framework of its kind. Together with keras (the second most used API for machine learning) around 66% of all papers based on machine learning research have used at least one of these APIs[4]. Both provide a large selection of neural network architectures and will be introduced in more detail in the following.

2.5.2 Tensorflow/Keras

Tensorflow is an open-source library for a number of tasks concerning machine learning, while Keras is a high level API, build on top of tensorflow for implementation and training of neural networks. Keras allows for easy and fast computation on CPU and GPU and is easy to debug, since it is written in Python as well.

The code listing in 2.1 shows the implementation of a convolutional neural network, combined with a feed-forward network, implemented in Keras. The objects `Conv1D`, `BatchNormalization`, `MaxPool1D`, `Flatten` and `Dense` are functions, provided by Keras, which are inserted as layers from the neural network.

The `Model`-object (line 23) combines settings for the training process and the neural network in one class, unlike in `PyTorch`, where the implementation of the neural network does not include anything concerning the training process, but the computational graph. In the function `compile` in line 24, the loss-function (mean-squared-error) and the optimizer (Adam-optimizer) are defined. Furthermore, values for `metrics` can be added (here mean-absolute-error), to define error functions, which evaluate the predictions of the neural network on *unseen* data. This is the standard routine, which is also recommended by the author of keras. If the function `CNN_reg` (from line 4) is called, it returns the `Model`-object which was set to contain the computational graph of the neural network and some hyperparameters. It can be trained in a next step. The training process is explained in detail in section 3.3, where this exact neural network was used to perform the experiment.

[3] Based on GitHub contributions, Gihub announced a list of the most popular programming languages for machine learning, https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning

[4] According to the status in September 2020, https://paperswithcode.com/trends

Listing 2.1: Example of a neural network, implemented with the machine learning API Keras in Python. The specific implementation is part of a neural network which is used in the first experiment in chapter 3.

```
from tensorflow.keras import layers
from tensorflow.keras import models

def CNN_reg(feature_dim, output_dim=3):
    signal_input = layers.Input(shape=feature_dim, name='Input')
    output = signal_input

    # Feature extraction
    for i in range(3):
        output = layers.Conv1D(256, 3, activation='relu')(output)
        output = layers.BatchNormalization(momentum=0.01)(output)
        output = layers.MaxPool1D(2)(output)
        output = layers.Dropout(0.25)(output)
    output = layers.Conv1D(256, 3, activation='relu')(output)
    output = layers.BatchNormalization(momentum=0.01)(output)

    # Regression
    output = layers.Flatten()(output)
    output = layers.Dense(256, activation='relu')(output)
    output = layers.Dense(64, activation='relu')(output)
    output = layers.Dense(output_dim, activation='linear')(output)

    model = models.Model(signal_input, outputs=output)
    model.compile(loss='mse', optimizer='adam', metrics=['mae'])

    return model
```

While Keras is used for neural networks in the experiment in chapter 3, the library PyTorch is for implementing the networks in chapter 4.

2.5.3 PyTorch

Pytorch is an open-source library in python which supports GPU acceleration and automatic differentiation [28] for training neural networks.

The PyTorch library provides a selection of already implemented neural network units, which they call *modules*. They can be used as building blocks for neural networks or as stand-alone neural networks as well. Each module can contain further

modules of the same type, allowing them to build a tree structure for complex neural networks, where each is child of the *nn.Module*-class. Typically, the networks are programmed object-oriented, unlike in Keras. The code below is an example of a convolutional neural network which consists of three convolutional blocks.

Listing 2.2: Example of a neural network, implemented with the machine learning API PyTorch in Python. The specific implementation is part of a neural network which is used in the second experiment in chapter 4.

```
import torch.nn as nn

class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Encoder, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(input_size, hidden_size, 3, padding=(1,1)),
            nn.ReLU(),
            nn.BatchNorm2d(hidden_size))

        self.conv2 = nn.Sequential(
            nn.Conv2d(hidden_size, hidden_size, 3, padding=(1,1)),
            nn.ReLU(),
            nn.BatchNorm2d(hidden_size))

        self.conv3 = nn.Sequential(
            nn.Conv2d(hidden_size, output_size, 3, padding=(1,1),
  stride=(2,2)),
            nn.Sigmoid())

    def forward(self, input):
        output = self.conv1(input)
        output = self.conv2(output)
        output = self.conv3(output)
        return output
```

The parent-class `nn.Module` in the code listing in 2.2 requires the definition of the function `forward`. By its execution it returns an output from the neural network, which is defined in the initialisation-function `__init__`, based on the argument `input` in the function. It is the required function that the network uses for its predictions. Not only the Encoder object is a child of the `nn.Module`-class, but the objects `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d` and `nn.Sigmoid` are also objects of

the same type. Therefore, it is possible to build a tree structure of modules which contain modules themselves to create functional blocks. The example in the code listing in 2.2 is used as an convolutional encoder of a seq2seq-model, which consists of multiple single-layer convolutional blocks with an activation function and batch-normalization. Each block within the encoder could function as an independent neural network.

CHAPTER 3

The inverse problem of electrocardiography

Electrocardiography (ECG) is a widely used diagnostic tool in medicine to identify potential heart diseases. The sinus node forms an electrical excitation that spreads through the heart, triggering the heartbeat. The electrical excitation generates a weak current flow, which can be measured during the ECG with help of electrodes which are placed on the body surface and generate the ECG signals. The correct analysis of these signals is critical in making judgments about possible heart diseases. The terminology ïnverse problem of ECGïs to be understood here in a broad sense as the reconstruction of cardiac dynamics based on ECG signals.

In this experiment, an attempt is made to distinguish a variety of different cardiac excitation patterns using a convolutional neural network (CNN), where the experiment deals with simulated excitations of the heart and its corresponding ECG. In this work the electrodes are not placed as in the classic 12-lead ECG, but lie on two grids which are placed on opposite sides at a certain distance from the heart. The geometries of the electrode placings and the heart are visualized in figure 3.1. In the following, if *ECG signals* are mentioned, it refers to the simulated ECG signals, which represent the measurements of 70 electrodes from the two grids.

In several simulations, different excitation patterns are generated by a periodic external stimulus on a small area on the surface of the heart. A concentric wave emanates from the location of the stimulus and propagates throughout the heart, where the excitation has a periodicity equally to the periodicity of the stimulus. The neural network in this experiment is trained with sequences of ECG signals from many of such simulations, where in each simulation the position of the stimulus is randomly chosen at the heart surface. The network is optimized to predict the location of the stimulus based on these ECG signals. Figure 3.2 is a plot of such

from 8 different electrodes. The values of the sequences lie between 0 and 1 after scaling.

The following description and evaluation of the experiment is divided into 4 parts. First, there is a short overview of the theoretical background concerning the heart and the electrocardiography (section 3.1), which includes a description of the anatomy of the heart, as well as an explanation of the *12-lead ECG*. The process of generating data from the simulation for the experiment in this chapter is the topic of section 3.2. It is followed by a part about the neural network that was trained in this experiment (section 3.3). Afterwards, the predictions of the trained networks are discussed in section 3.4.

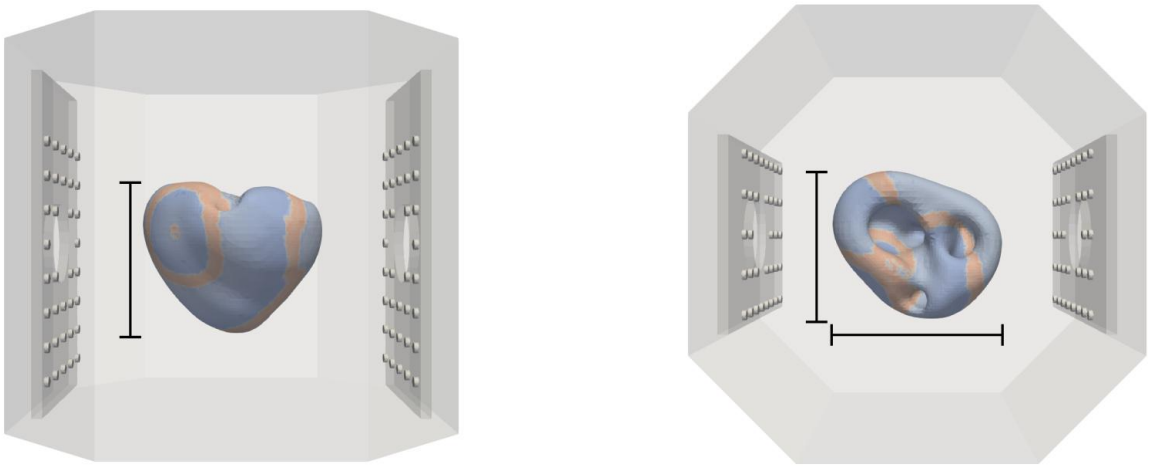

Fig. 3.1: Simulation of a pig heart, visualized with paraview [1]. The panel on the left and right side on each image contains 35 electrodes for the simulated electrocardiographic signals.

3.1 Biological background

The following two sections (3.1.1-3.1.2) deal with the anatomy of the heart and its electrical activity. How this activity is measured with electrodes on the body surface is the subject of section 3.1.3.

3.1.1 Anatomy of the heart

This section is based on chapter 1 of the book *Herz-Kreislauf* by J. Steffel and T. Lüscher [35]. The heart consists of four chambers, the left and the right atrium and

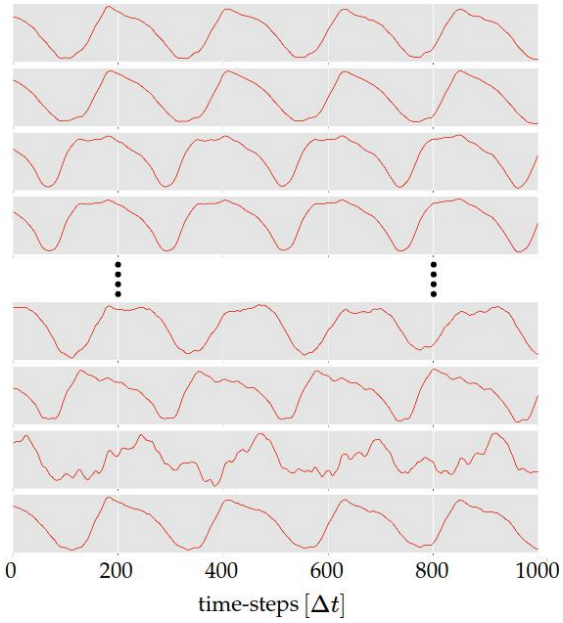

Fig. 3.2: Plot of 8 randomly chosen electrode signals from a grid of 70 electrodes within one of 1024 generated simulations, over a period of 1000 time steps after skipping the first 600 time steps. The process of generating this visualized data is explained in chapter 3.2.

the left and the right ventricle, as visualized in figure 3.3. The right atrium with the right ventricle builds the right half of the heart, which pumps the blood to the lungs for a gas exchange. From there the blood flows towards the left half of the heart (which consists of the left atrium and the left ventricle). The left half pumps the oxygen rich blood towards the individual organs. The heart of an adult beats between 50 and 80 times a minute in normal condition, where the pump frequency is caused by an electrical pulse initiator, the sinus node.

3.1.2 Electrical activity of the heart

The pumping action of the heart is the result of an electrical stimulus, initiated by the sinus node. The electrical stimulus (pulse) is a rapid potential change of the membrane potential induced by moving ions through the cell membrane of cardiac cells. The electrical pulse propagates from the atria to the ventricles via the atrioventricular node where it causes the cells to contract. The propagation is caused by ability of cardiac cells to transmit the signal to neighboring cells.

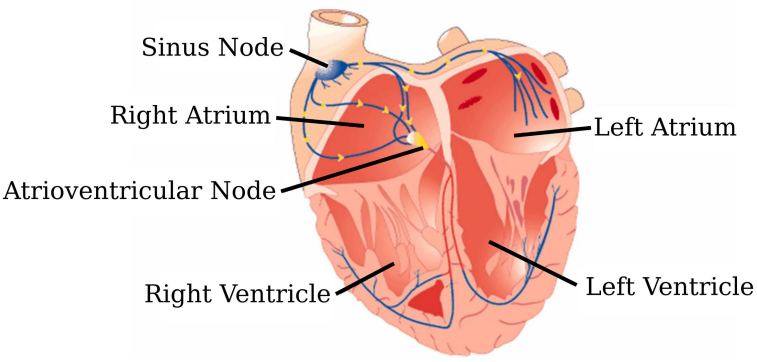

Fig. 3.3: Anatomy of the heart with the four chambers (two atriums and ventricles) including the sinus and atrioventricular node [36].

3.1.3 Electrocardiography

The electrocardiography is a process to gain information about the electrical activity of the heart. It produces a representation of the activity, the electrocardiogram, by measuring potential differences with electrodes on the torso surface of the human body.

The following description is about the so called *12-lead ECG*, which refers to a special form of electrocardiogram, where the activity is represented by 9 electrodes, which are placed in a specific arrangement on the human torso. In this chapter, which is about the first (numerical) experiment, however, the simulated ECG is based on a different spatial arrangement of the electrodes. For distinction, the representation of the electrical activity of the heart is based on 70 electrodes on a grid, whose electrocardiogram is called *70-lead grid ECG* in the following.

A pair of electrodes form a so called lead, by subtracting one signal from the other as a reference. A lead can be partially or fully made up of virtual electrodes, where the term virtual is used, if a measurement is built from the average measurement from multiple electrodes. A virtual electrode is then virtually placed at the spatial average of other electrodes of which it is built of.

A widely known setting is the 12-lead ECG, where 9 electrodes are used to form 12 leads. The positioning of these is visualized in figure 3.4. The leads form vectors of which 6 lie on a horizontal plane (precordial leads) and 6 on a vertical plane (limb leads):

- Precordial leads have in common that one electrode is (ideally) placed virtually inside the heart. The other electrodes (marked on the image as **V1, ..., V6**)

build pairwise leads with the virtual electrode inside the heart, where the spanned vectors are all on a horizontal plane.

- Limb leads are only built from the electrodes **R,L,N** and **F**, and therefore lay on the vertical plane. They are build from a virtual electrode, made of **N** and **F**.

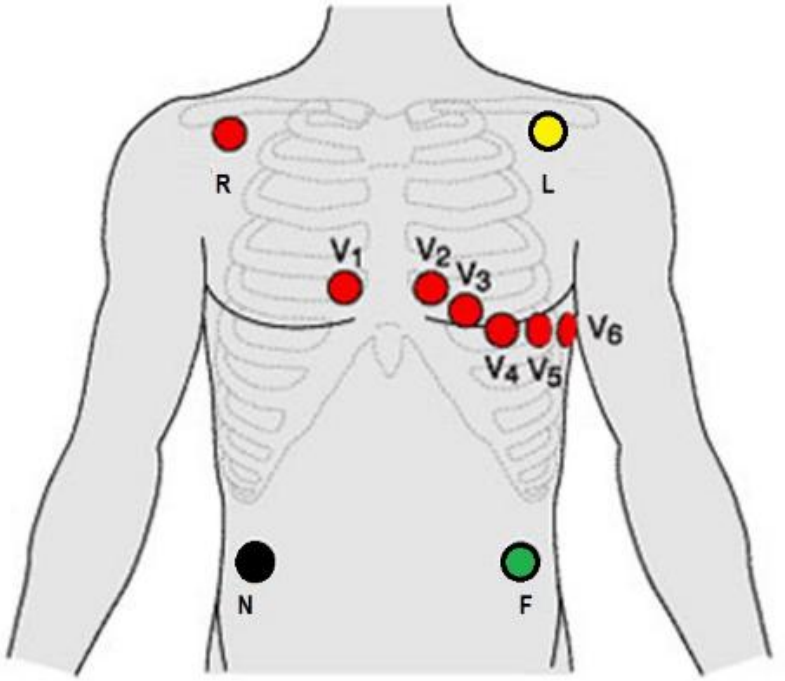

Fig. 3.4: Positioning of the 10 electrodes of a 12-lead ECG positioning [22].

Simulations of the 70-lead grid ECG in this experiment also include the formation of various leads which have the center point in the simulated heart as reference. How the simulation proceeds is one of the issues of the next section.

3.2 Data generation

This chapter is about the process of data generation and preparation for the neural network to work with.

3.2.1 Simulation of ECG signals

The simulation of the ECG signals are based on a monodomain simulation, with the finite element method for spatial discretization and the FitzHugh-Nagumo model

[13] for the local (ionic) dynamics of the heart. It is followed by a diffusion simulation to calculate the electrical potential at a certain distance from the simulated heart. The positions where the potential is calculated, and the geometry of the heart is visualized in figure 3.1.

The FitzHugh-Nagumo model contains an additive variable that represents an external stimulus I_{ext}. This variable is modeled in this experiment as a periodic function over time, which generates excitation waves on the heart surface (see section below).

After the simulation of the FitzHugh-Nagumo model on the heart geometry (a visualization of the geometry can be seen in figure 3.1), the potential is simulated as a diffusion process, and its values are recorded at fixed selections from multiple locations. The locations are built of collections of points which form the electrodes with the shape of a cylinder. Its potential is calculated by the average potentials of the associated points. The 3D model of the heart and the surrounding grid of electrodes is visualized in figure 3.1, where the shape of the heart is based on a computed tomography scan (CT scan) of a pig heart. The cylindrical electrodes are visible at the two grids at the left and right side of the heart.

The simulation framework for simulating the ECG signals and the electrical activity of the heart is provided by Baltasar Rüchardt [29], where the differential equations are solved using the Python library DOLFIN [21], which is the Python interface of `FEniCS` [2], an open-source computing platform for solving partial differential equations.

3.2.2 Concentric wave generation

This experiment is about the differentiation of excitation patterns on the heart surface, where these patterns can be described as a concentric wave, propagating over the heart. To generate these dynamics, different locations on the heart surface are periodically stimulated in several simulations. Each simulation reaches a period of 1600 time steps, where the time discretization Δt is 0.01. The location of the external stimulus is varied with each simulation, where it generates concentric waves, which propagate through the whole heart.

The external stimulus in the simulation is a function $I_{\text{ext}}(t, \mathbf{s})$, which is periodic in time. It is defined as

$$I_{\text{ext}}(t, \mathbf{s}) = \begin{cases} I_0 \cdot \text{Rect}(t) & \text{if } \mathbf{s} \in \Omega \\ 0 & \text{else} \end{cases}, \tag{3.1}$$

with

$$\text{Rect}(t) := \begin{cases} 1 & \text{if } (t \bmod P) < d \\ 0 & \text{else} \end{cases}, \tag{3.2}$$

where P is the duration of a period and d the length of the stimulus and I_0 its strength. The variable $\mathbf{s}$ is element of Ω. A python script defines Ω, which consists of a single to a few finite elements. Due to this local limitation of Ω, a center point of the concentric wave, which is induces by the external stimulus, can be identified.

3.2.3 Data recording from simulations

Each simulation gives an output with 70 time series of length 1600 from 70 electrodes with a time discretization of Δt=0.1. The 3D model of the heart and the arrangement of the electrodes is visualized in figure 3.5. The average of four electrodes, which are marked with the black circles, are used as reference for each of the 70 electrodes.

3.2.4 Preprocessing

The time series of the ECG signals have a dominant period length of in average 223 time steps. Therefore, the length of a single sub sequence includes around 0.45 period lengths. With 1024 simulations, the amount of sub sequences result to $1024 \cdot 4 = 4096$, where 3276 of them are used for training, and 820 for testing. The choice of random starting points comes from the idea, that the neural network should be invariant through time shift in its prediction. The resulting sub sequences are the input of the neural network, to predict the position of the source of the generated concentric wave. The position is represented by a 3-dimensional vector which is not further processed.

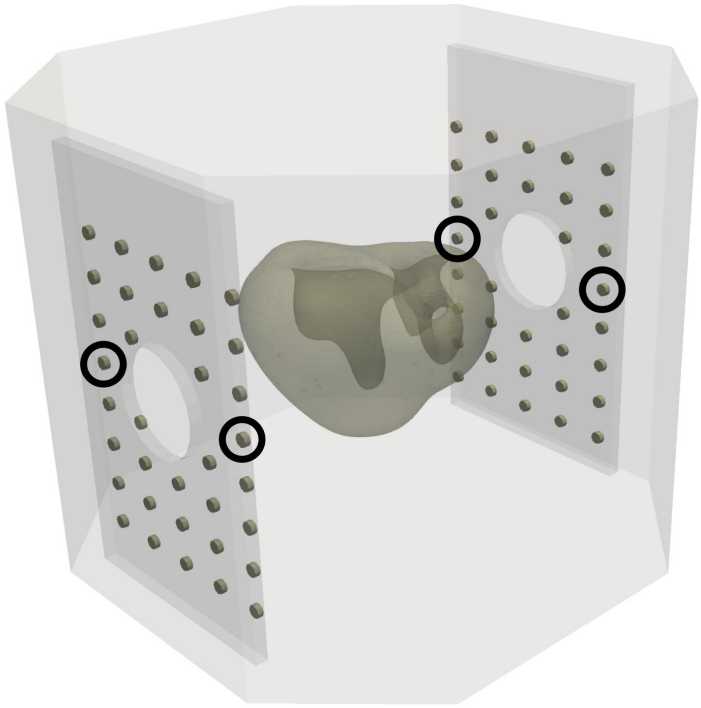

Fig. 3.5: Simulation of a pig heart, visualized with paraview [1]. The panels on the left and right side contain each two electrodes, which are used to build a virtual electrode in the approximate center of the heart.

3.3 Neural network implementation and training process

This section examines the specific design of the neural network (in the following called model) used in the experiment from this chapter and its implementation with help of the keras-API in python. Furthermore, details about the training process and regularizations are explained.

The model is a convolutional neural network (*block* 1), combined with a feed-forward network (*block* 2). The first *block* extracts features of the input sequence (time sequence of length 100 with 70 features) by transforming it into another sequence with different length and number of features. With the network's architecture here it transforms it into a sequence of length 12 with 256 features. This representation of the input is then fed to the second *block*, the feed-forward network. Before the network can process the input from the first *block*, the data has to be flattened

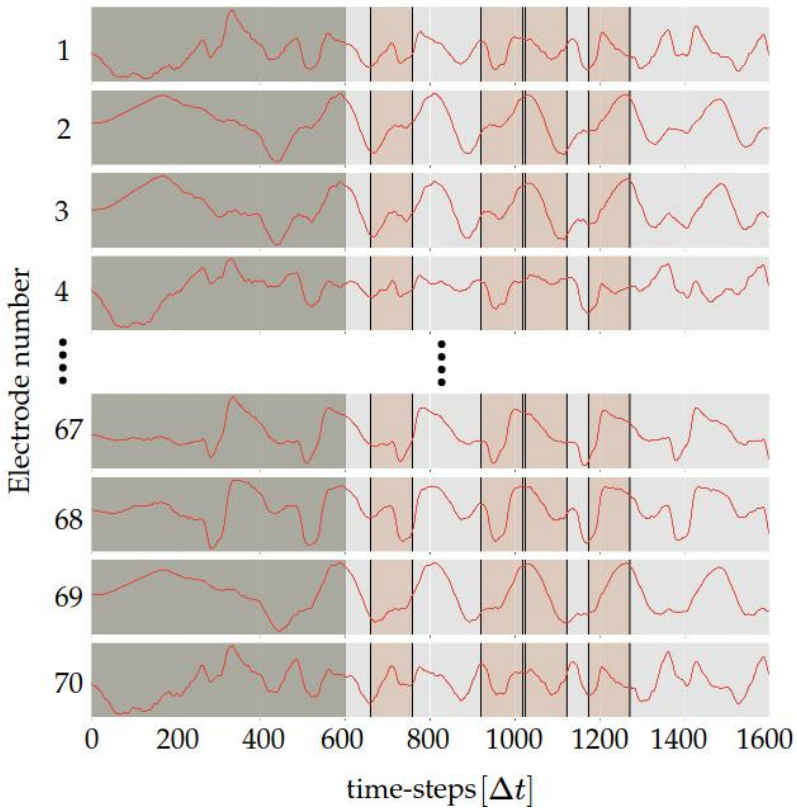

Fig. 3.6: Plot of a few electrode signal sequences over a period of 1600 time steps. The dark section at the first 600 time steps denotes the initial phase which is excluded in the dataset. The red areas mark a section of 100 time steps whose starting points are randomly chosen. All sequences taken together that cover the same area serve as input to the neural network. The ECG signals of each simulation, 4 such areas were randomly selected from which the data set for training and testing the networks is finally generated.

to a vector, which has a length of

$$12 \ \textit{(sequence length)} \cdot 256 \ \textit{(number of features)} = 3072.$$

The feed-forward network consists of three linear layers where the last one has three neurons. Therefore, the output is a 3-dimensional vector. It represents the prediction of the position of the concentric wave source. The architecture is visualized in figure 3.7.

While some hyperparameters are already defined within the model-object, some customized regularizations and the number of epochs and batch size are defined in

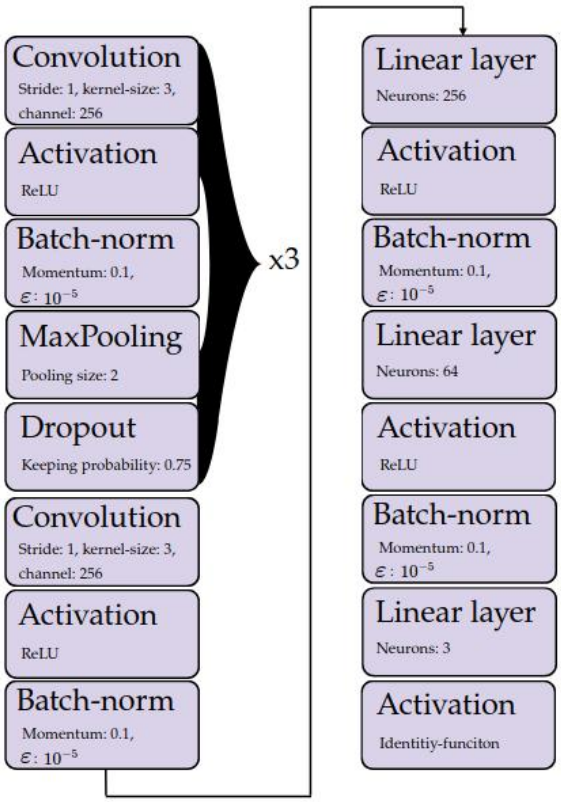

Fig. 3.7: Neural network, consisting of a CNN and a feed-forward network for the regression task of predicting the source of a concentric on the surface of the heart.

the following code 3.1 inside the `fit`-function, which is a function of the `model`-object. All of the hyperparameter are shown in table 3.1.

Listing 3.1: Implementation of the training process with the API Keras in Python.

```
model = CNN_reg(feature_dim=(100,70))
es = EarlyStopping(monitor='val_mae', verbose=1, patience=8)
rp = ReduceLROnPlateau(monitor='val_mae', patience=4)
mc = ModelCheckpoint("Reg_CNN_best", monitor='val_mae')

model.fit(X_train, y_train,
            epochs=1024,
            batch_size=128,
            verbose=1,
            callbacks=[es, mc, rp],
            validation_data=(X_test, y_test))
```

Some regularizations are applied as well, such as *EarlyStopping* (see section 2.1.3.3)

and *ReduceLROnPlateau* (see section 2.1.3.4). With *EarlyStopping* in line 2, the neural network's parameters are saved after each epoch, as long as the loss function has become smaller, with respect to all previous epochs. The number of epochs is set very high. However, the training process is completed after the loss did not get smaller over a period of 8 epochs. Therefore, the amount of epochs practically never reached the maximum number of epochs, which is defined in line 7.

Tab. 3.1: Hyperparameter of the training process.

Parameter	Value
Loss function	MSE
Learning rate	0.01
Batch size	128
Optimizer	Adam

3.4 Results and validation

In the following I present the performance of the convolutional neural network which is trained to predict the source of the stimulated concentric waves. The performance with respect to the computational cost of the training process is discussed afterwards.

3.4.1 Concentric wave localization

With the regularization methods during the training process, the training lasted for 103 epochs, which took around 11 minutes on a GeForce GTX 1660 Ti graphics card. While the heart has an expansion of between around 60 and 80 length steps Δs, the neural network is able to predict the source of the concentric wave from the validation set on average with a displacement of $3.45 \cdot \Delta s$. The displacement is the euclidean distance of the prediction and the true position of the concentric wave source. With determining the approximate size of the pig heart that was simulated between 10cm and 20cm in each direction (x, y, z), the average displacement would be approximately between 0.32cm and 0.5cm.

The comparability faces limitations due to the fact that the simulation produces a system without noise (geometrical noise in time and space, uncertainties on the measurements or positioning of the electrodes etc.) Nevertheless, it shows that the CNN is able to localize the source of concentric waves (tested under perfect con-

ditions) from unknown data, whose displacement is not orders of magnitude worse when comparing with the results from the training dataset.

The figure 3.8 shows the development of the mean displacement after each epoch. The best performance on the training dataset already nearly reached its minimum after around 35 epochs. However, because of the implemented training rule, which aborts the training process after the performance reached a plateau, the training lasted for a few epochs longer.

Tab. 3.2: Mean absolute error (MAE) and mean displacement (displ.) of the predictions from the CNN on the training and test dataset.

Measurement	training dataset	validation dataset
MAE [LU]	1.38	1.99
Mean displacement [LU]	2.39	3.45

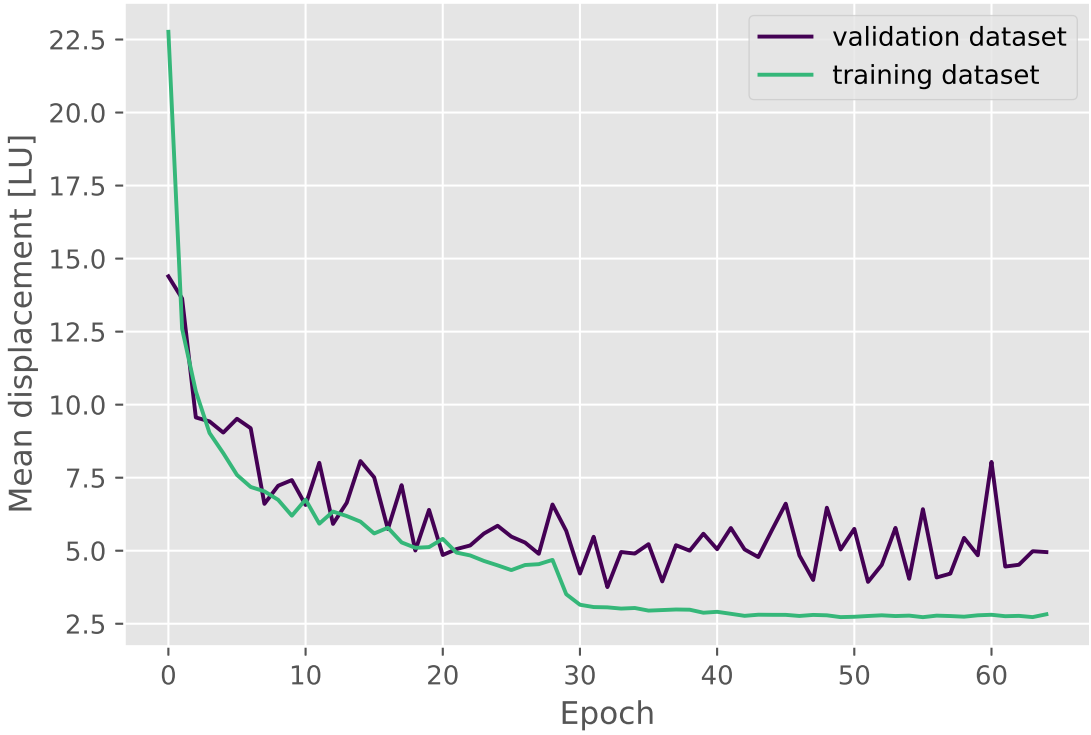

Fig. 3.8: Evolution of the mean displacement on the validation- and training dataset.

The experiment described in this chapter is an attempt to show the principle ability of a CNN to differentiate between a large number of spatio-temporal pattern at the surface a heart, trained with 70 ECG signals. Furthermore, the neural network is able to process temporally randomly selected sub sequence of the ECG signals,

which proves that it recognizes a pattern in the signals regardless of the phase of the concentric wave. However, the simulated heart contains around 10000 elements on the surface, which can represent an approximate source of the concentric wave. With simulating 1024 time series of the 70-lead grid ECG signals, the activity of the heart with unique starting conditions of the concentric wave, around 10% of the available sources are already covered. The properties of a simulation differ only in the location of the pacing events that cause the concentric wave. It is possible that some training and test simulations differ very little from each other if the locations of the pacing events are close to each other. In this case it may not be possible to speak of a truly unknown data set. The average minimal distance from a test simulation to the next training simulation is around 2.47 LU. It shows that the neural network is trained with pacing positions, which are not far distant from the test simulations. Furthermore, the neural networks average displacement of the predictions on the test dataset is bigger than the average distance to the next training simulation.

The same CNN is also trained on the dataset but with additional Gaussian noise with $\mu = 0.0$ and $\sigma = 0.05$. The training with the same conditions as the training described before got displacements as shown in table 3.3. Since the simulated ECG signals are normalized between 0 and 1, and additional noise with σ=0.05 indicates an additional error of 5%.

Tab. 3.3: Average displacement in LU of the prediction from the CNN, with additional Gaussian noise with $\sigma = 0.05$ to the normalized data.

training set	test set
2.69	3.98

The mean displacement on the test dataset increased with 0.4 LU, which is a deterioration of the predictions by about 11%. This indicates that the neural network is not very robust against additional noise.

CHAPTER 4

Reconstruction of hidden 3D excitations

This chapter is about the second experiment. The basis for it is a 3D-simulation of the time evolution of the Barkley model [5]. The question is to what extent machine learning models respectively neural networks can predict a system quantity (here the u-value of the Barkley model), inside the 3D structure of the simulation which is not *visible* from outside. The input to the neural networks and its predictions are values of the same system quantity u, where the input consists of a recording of the u-value on the surface of the 3D structure over a period of time. Here, *not visible* means that the neural network has no explicit information about the system quantity under the surface of the 3D structure that it is supposed to predict. Only on the basis of the surface it is to extrapolate these. In this chapter a study of the limits of machine learning models that face up to the task mentioned above, while having *perfect* surface information is performed. *Perfect* means here, unlike in real experiments, that the data does not have any noise in terms of geometrical uncertainties or measurement errors. Furthermore, with choosing a cube, being a simple geometry, this experiment can be regarded as a base study about the limitations of machine learning models for this kind of problem. In further research, this kind of problem could be transferred to a more complex geometry such as the heart. Two different sequence-to-sequence LSTM architectures on two different regimes, simulated with the Barkley model are tested. The choice of model immanent parameters and initial values determine the dynamics of the regimes. Within the first regime (A), two spiral waves produce a periodic dynamic, while the other regime (B) shows chaotic behavior. Figure 4.2 shows two snapshots of the two regimes, simulated in the cube.

The realization of this experiment can be divided into three parts. The first part is

about generating data for the neural networks. It follows the definition of the neural networks and the training processes with the generated data from simulations of the Barkley model. The predictions are validated in the third part. The process of generating data is done with multiple simulations of the Barkley model (sections 4.1.1 and 4.1.2) in an equilateral cube, taking only into account the u-value of this model for further usage. The simulations generate time sequences of the u-values from all elements in the cube (see section 4.1.5). The input for the neural networks is a time sequence from the surface of the cube at one specific side of the cube. Based on this input, the network is trained to predict the u-value till a certain depth, which the respective simulation has calculated at the first time step of the input sequence. The simulations are producing these input-output pairs with a few preprocessing steps (section 4.1.7). The definition of the neural network model and the training process with the given data is subject of the following section 4.2. These first two parts (sections 4.1-4.2) can be described as methods section, where the third part consists of the validation of the predictions the neural networks made (section 4.3).

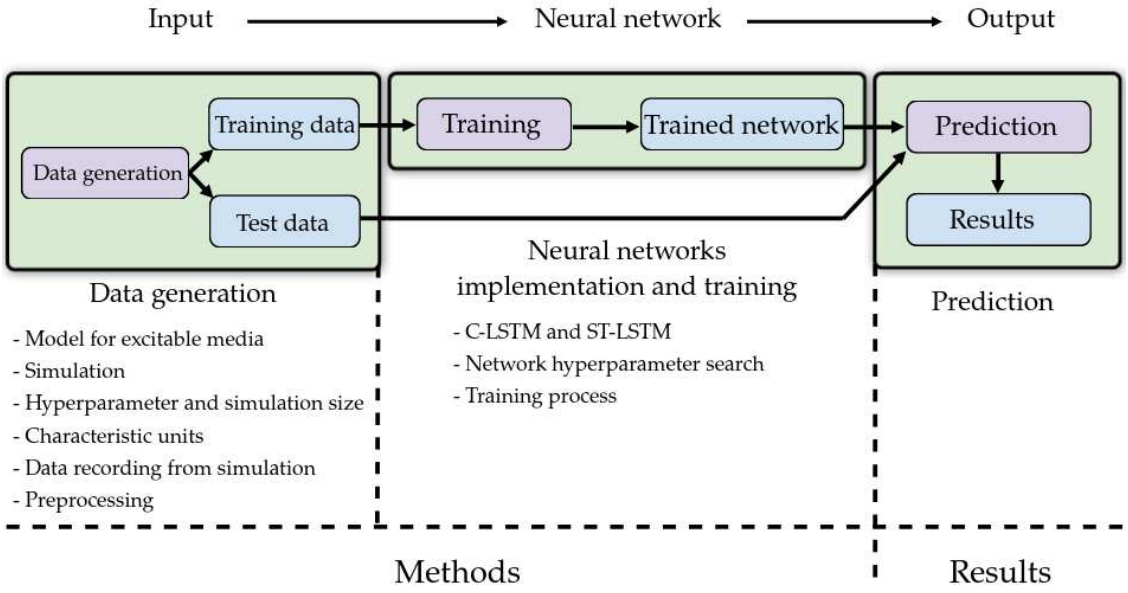

Fig. 4.1: Machine learning pipeline and structure of this chapter.

4.1 Data generation

This section is about the data generation for the second experiment, where firstly the Barkley model is introduced, with which the simulation of excitable media is

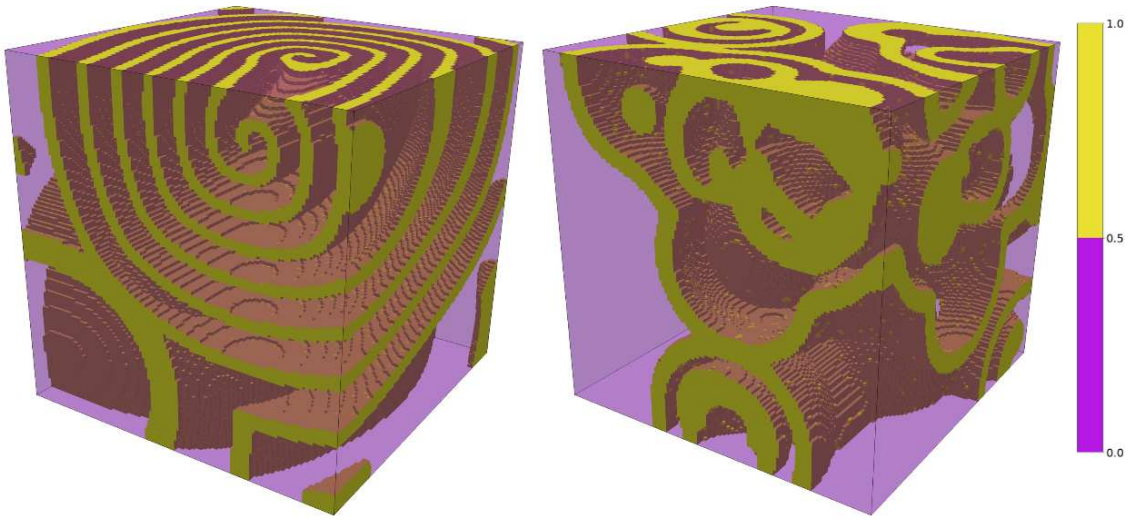

Fig. 4.2: Snapshots of the Barkley-simulation within the cube. The left figure shows regime A, the right one is a visualization of regime B. For a better visibility, the voxels in which the value for u is below 0.5 were made transparent.

realized. Then the simulation method is explained and the hyperparameter choice concerning the Barkley model to generate two different regimes. From the respective hyperparameter configurations, characteristic units are extracted which are determined in the following section. Afterwards, the preprocessing steps, after which the data can be used as input for the neural networks, are explained.

4.1.1 Model for excitable media

There are various models to describe the heart dynamic with differential equations in terms of voltage current. Since the experiments goal is to reconstruct hidden regions under the surface qualitatively, the model for excitable media does not have to provide a realistic approximation of the heart's electrical activity in terms of current voltage. The Barkley model is a model for excitable media and is a system of two coupled differential equations with the variables u and v, which build a reaction diffusion system. It was proposed by Barkley et al. in 1990 [6].

The Barkley model is given by the equations

$$\begin{aligned}\frac{\partial u}{\partial t} &= D \cdot \nabla^2 u + \frac{1}{\varepsilon}(1-u)\left(u - \frac{v+b}{a}\right), \\ \frac{\partial v}{\partial t} &= u^\alpha - v,\end{aligned} \tag{4.1}$$

where u is a fast variable, while the variable v is slower and inhibiting [9]. The parameter a, b and ϵ are positive constants. A bigger value for a increased the excitation duration while an increasing fraction between b and a, $\frac{b}{a}$, results in a larger excitability threshold [5]. In this work the value for α is set to 3, which can cause chaotic behavior. Here, two sets of parameters are tested, where a, b and ϵ are varied. One regime is periodical, while the other shows chaotic behaviour. More details about the regimes are in section 4.1.3, where the values for a, b and ϵ are listed in table 4.1.

4.1.2 Simulation

The 3D-simulation of the Barkley model takes place within a cube with an expansion of 120 discrete units in each dimension. The discretized Laplace operator ∇^2 within the equations from 4.1 is here described through a 7-point stencil such that

$$\left(\nabla_7^2 u\right)_{i,j,k} = \frac{u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - 6u_{i,j,k}}{\Delta s^2}, \tag{4.2}$$

where the indices i, j and k stand for the discrete position in the cube and Δs for the spatial discretization. A simulation step in time is calculated through the explicit Euler method with

$$\begin{aligned}u(t+1) &= u(t) + \Delta t \frac{\partial u(t)}{\partial t}, \\ v(t+1) &= v(t) + \Delta t \frac{\partial v(t)}{\partial t},\end{aligned} \tag{4.3}$$

such that a time-step is described as

$$u(t+1)_{i,j,k} = u(t)_{i,j,k} + \Delta t \cdot \left(D \cdot \left(\nabla_7^2 u \right)_{i,j,k} + \frac{1}{\varepsilon}(1 - u(t)_{i,j,k}) \left(u(t)_{i,j,k} - \frac{v(t)_{i,j,k} + b}{a} \right) \right),$$
$$v(t+1)_{i,j,k} = v(t)_{i,j,k} + \Delta t \cdot \left(u(t)_{i,j,k}^3 - v(t)_{i,j,k} \right). \tag{4.4}$$

The simulation is implemented in Python with orientation on the code for a 2D-simulation of the Barkley model by Roland Zimmermann [44]. The hyperparameters a, b and ε and the starting values for u and v are adjusted for two regimes, on which neural networks are tested regarding the given task, as introduced in the following section.

4.1.3 Hyperparameter and simulation size

From the two regimes, one builds multiple spiral waves and has a periodic behaviour, while the other shows chaotic behaviour. In table 4.1, the values for a, b, ε and α are listed, according to each regime. Table 4.2 contains the discretization size of time and space.

Tab. 4.1: Parameter choices from the Barkley model.

Param.	Regime A	Regime B
a	0.6	0.75
b	0.01	0.06
ϵ	0.02	0.08
α	1	3

Tab. 4.2: Discretization of space and time for the simulations.

α	3
Δt	0.01
Δs	0.1

The choice of parameters leads to the described different behavior of the two regimes. Three characteristic variables are introduced in the following,to see space and time in this context.

4.1.4 Characteristic variables

To have a more intuitive understanding about temporal and spatial sizes, the following characteristic variables are introduced:

- Time T_c: The characteristic time T_c is defined by average period length, which is estimated by identifying peaks within the time series of individual voxels from simulations of regime A. As described in section 4.1.5, a part of the simulated data consists of 2048 recordings from simulations which last, after an initial phase, over a time span of 512 time steps, which are recorded in intervals of 16 time steps. When two peaks are identified within the time series of a voxel, the gap between them is then saved as a single period length. These lengths, calculated from each voxel on each simulation are used to compute the mean period length, which is considered as characteristic time.

- Length λ_c: An other part of the simulation data, as described in section 4.1.5, consists of 2048 recordings from simulations at a certain time step, which include the data till a depth of 32 discrete spatial units. Therefore, the data can be represented by a tensor with the dimensionality of (32,120,120). From this data, the values of the voxels along vectors in x- and y-direction along the axes with length 120 are used. From each individual simulation, 120*32 vectors of length 120 follow, which consist of the respective u-values of the simulations. In these series, the positions of the peaks at which u reaches a local maximum are determined, and then the distance to the next peak is calculated. A histogram of the collection of all distances from each simulation is shown in figure 4.3. The first maximum of this histogram is regarded as characteristic length. The error of this value is estimated with $0.5\Delta s$.

The resulting values for λ_c and T_c are shown in table 4.3. Since in both regimes the discretization of time and space (Δt and Δs) are used, the characteristic variables, scaled with the discretizations are used in the further process. However, due to the chaotic behaviour of regime B, no spiral waves are formed on which such a determination of the characteristic variables would be possible.

Tab. 4.3: Characteristic variables for the Barkley regime A.

λ_c	$(16.2 \pm 0.5) \cdot \Delta s$
T_c	$(74.93 \pm 5.45) \cdot \Delta t$

From the two variables λ_c and T_c, a characteristic velocity can be extracted, which results to

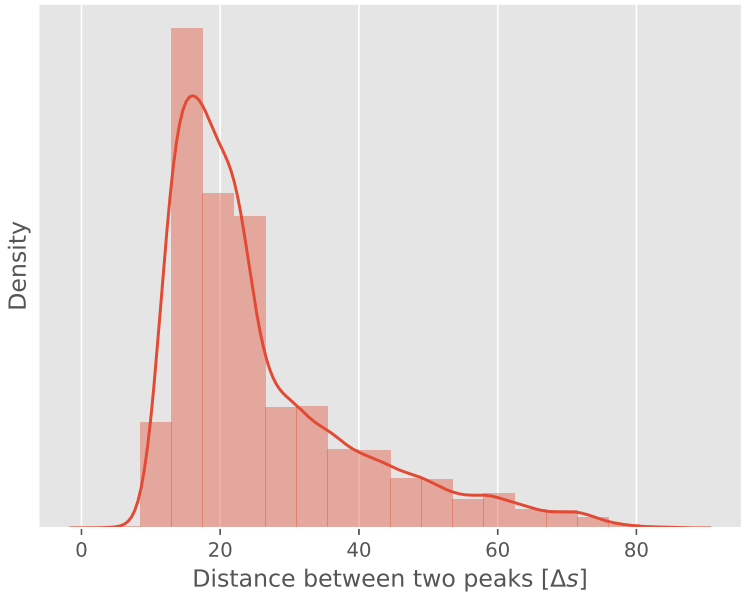

Fig. 4.3: Histogram from the distances between two peaks of the u-value, which are recorded along vectors in x- and y-direction on the simulated cube geometry till a depth of 32.

$$v_c = \frac{\lambda_c}{T_c} = (0.22 \pm 0.02)\frac{\Delta s}{\Delta t}. \tag{4.5}$$

A wave-front which propagates along the normal of the surface in a depth of λ_c, needs approximately 74.93 discrete time steps to be recognisable at the surface, since it is estimated to propagate with a speed of v_c.

The following procedure of data generation and preprocessing is the same for both regimes.

4.1.5 Data recording from simulation

As mentioned before, this experiment focuses on the u-value of the Barkley model for further training of neural networks. Since the cube has an expansion of 120 discretized units in each spatial dimension, a snapshot of the u-value is a tensor of order 3 with the dimensionality of (depth, height, width)=(120,120,120). In a single simulation, each snapshot is recorded after 16 discrete time steps from the

previous snapshot over a span of 512 steps. Therefore, each simulation stores $T = 512/16 = 32$ time steps, which are represented by a tensor with the dimensionality of (32,120,120,120). The recording starts after an initial phase, which last for 4096 time steps on regime A and 2048 time steps on regime B. Basis for the preprocessing steps, which are explained in the following, are 2048 of these simulations for regime A and 256 for regime B. Each simulation in each regime has random starting conditions according to the procedure, described in the following.

4.1.6 Starting conditions

In the following, the procedure of generating unique starting conditions in each new simulation is explained, where the characteristic properties for regime A (periodic dynamics) and regime B (chaotic dynamics) remain the same.

4.1.6.1 Regime A, periodic spiral waves

To initiate spiral waves, the cube in which the simulation takes place, is firstly divided into 8 sub cubes of equal size. The following procedure is the same for two of the 8 sub cubes. In the 6 remaining, the values for u and v are set to 0 at every voxel.

First, the sub cube is *filled* up till a random height (as shown in the left graphic of figure 4.4), where the u-value is set to 1.0 at every voxel inside. The same procedure takes place for the v-value, but rotated by 90 degree, whose values are set to 0.5. Visualized is this procedure in the left figures of 4.4 and 4.5. In the next step, this structure is rotated around all three axes randomly, but remains the same for u- and v-value for one sub cube. As third step, the two sub cubes on which this procedure was performed, are placed diagonally to each other, so that these, with the remaining 6 sub cubes, form the shape of the original cube. This is the initial situation with which the simulation starts.The height till which the sub cubes gets *filled* is random, as well as the rotations. The left plot in figure 4.6 shows an example of a single voxel within the simulation whose u-value is plotted against the time. The periodicity behaviour (after an initial phase) can be seen here.

4.1.6.2 Regime B, chaos

To generate chaos, not only the Barkley parameter a, b, ϵ and α are critical, but also the starting conditions. The cube is firstly divided into equilateral cubes of length

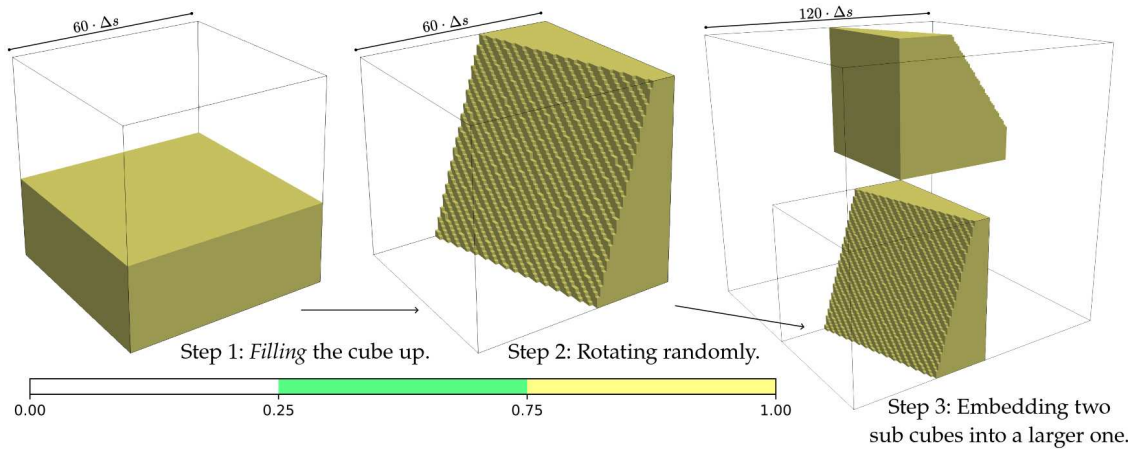

Fig. 4.4: Procedure of generating starting conditions for the u- and v-value. Visualized is the u-value at each step within the procedure.

$6 \cdot \Delta s$ in each dimension, where every voxel in a single sub cube has the same u- and v-value of the Barkley model. The values for u are randomly chosen between 0 and 1 from an equal distribution. If the value exceeds 0.4 in a sub cube, v is set to 0 in the same sub cube. If it is not the case, v is set to 1.0. The right plot in figure 4.6 shows an example of a single voxel within the simulation whose u-value is plotted against the time. It can be seen that there is no periodicity as in the left graphic.

4.1.7 Preprocessing

The following description of preprocessing steps for the training process is identical for both regimes. As mentioned in the previous section, each simulation provides values of u, given by a tensor with the dimensionality of (32, *time steps*;120, *depth*;120, *height*;120, *width*). In the following, X is defined to describe the data the machine learning model gets as input, and the target for the prediction is denoted as y. The experiment is defined in such a way that X is a time series of u-values at the surface. Therefore, X is a tensor with $X \in \mathbb{R}^{32,1,120,120}$. The y-data is the u-value at a certain time step at the first 32 slices from the surface of the cube and is a tensor with $y \in \mathbb{R}^{1,32,120,120}$. Since a cube has 6 sides, each simulation provides 6 (X,y) pairs.

The data within the simulations is handled as float64 (means each single value needs 64 bit of storage, encoded according to the float64 data type). Nonetheless it is possible without a great loss on information to convert the data into one-byte-integers (int8) to save storage. Therefore, X and y are rescaled into the range of int8 (-127 to 128).

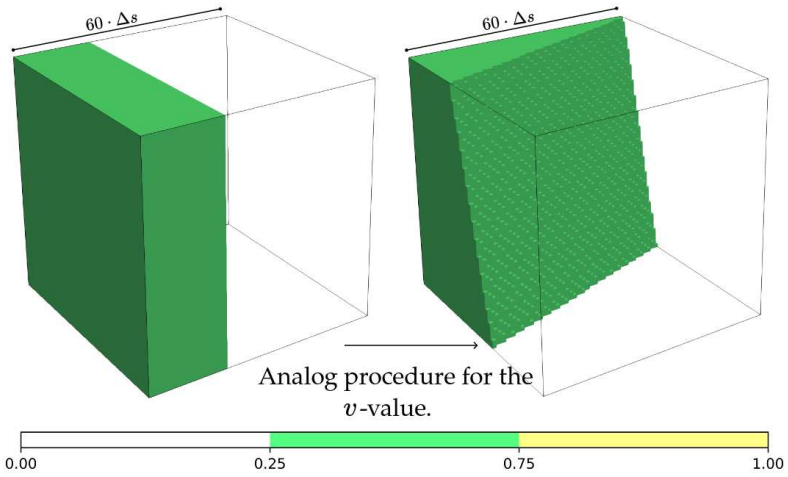

Fig. 4.5: Procedure of generating starting conditions for the u- and v-value. Visualized is the v-value at the first two steps within the procedure.

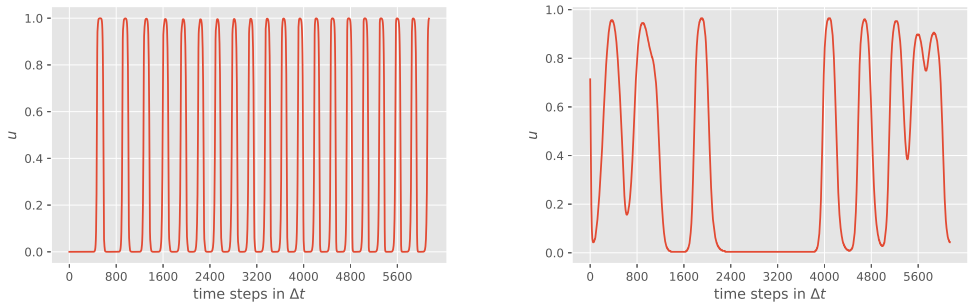

Fig. 4.6: Time development of the u-value of a single voxel from a simulation in the (left) periodic regime A and (right) the chaotic regime B.

Within the training process, the data again has to be rescaled into the range from 0 to 1. This is done with the help of the pytorch `dataloader`-class who scales a batch of data and loads it into the storage of the device, which performs the computation, concerning the training process.

4.2 Neural networks implementation and training

This chapter is about the definition of the neural network architecture (computational graph) and the training process.

4.2.1 C-LSTM and ST-LSTM

In section 2.4, a typical seq2seq model was presented, where in this section, the exact composition of the neural networks, which are based on the seq2seq model are introduced. Two different designs of seq2seq models are tested in this experiment, which only differ in the implementation of their recurrent units. In the following, the neural networks are named *CV* and *ST*. While *CV* consists of a C-LSTM-network (see section 2.3.4.1) as encoder and decoder-LSTM (colored blue in figure 4.7), network *ST* uses a ST-LSTM-network (see section 2.3.4.2). Regardless of the design, there are scalable quantities such as the number of layers in the convolutional encoder and decoder network (colored purple in figure 4.7) or the number of features in the LSTMs, to be defined. These quantities are presented in the following, while they are identical for both networks (*CV* and *ST*).

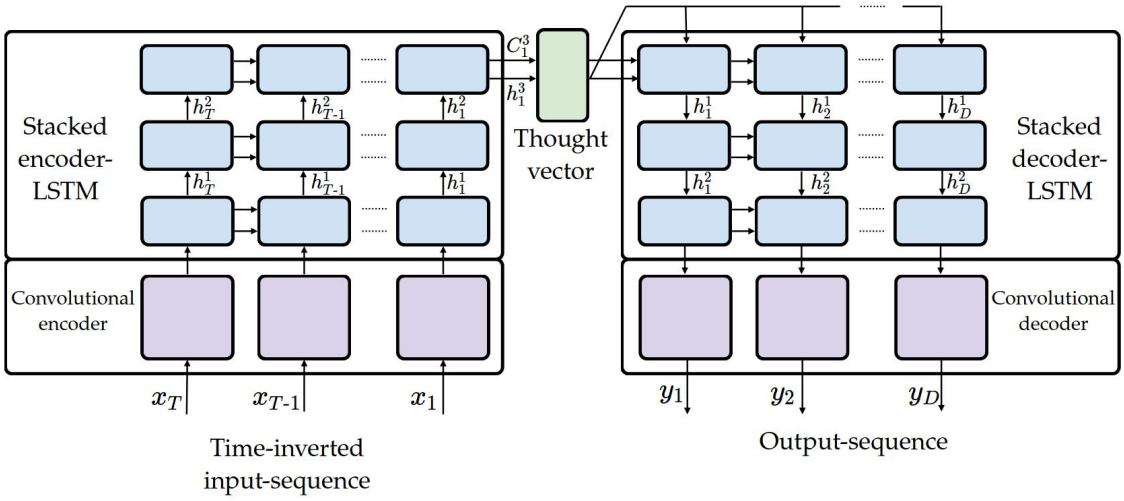

Fig. 4.7: Schematic view of the sequence-to-sequence networks, which are used in this experiment, consisting of a convolutional encoder and decoder with two stacked LSTMs.

Convolutional encoder and decoder

The convolutional encoder and decoder are time distributed, which means that the same network is used separately for each time point of an input sequence. One convolutional layer in the convolutional encoder has a stride parameter of 2, which reduces the spatial dimensionality by the factor of two. Is an input X given with $X \in R^{\text{time-steps} \times \text{depth} \times \text{height} \times \text{width}}$, the encoder reduces it to $x_{\text{out}} \in R^{\text{time-steps} \times \text{features} \times \text{height}/2 \times \text{width}/2}$. Note, that instead of *depth*, the third dimension represents now the number of channels, which is a hyperparameter of a convolutional layer as introduced in section

(2.2.2). The value is always set for all convolutional layers within the encoder and decoder to 64. The design is visualized in the figure 4.8. While the encoder reduces the spatial dimensionality by the factor of 2, the convolutional decoder upscales its input by the factor of two. This is performed by an upsampling layer, as visualized in the figure 4.9.

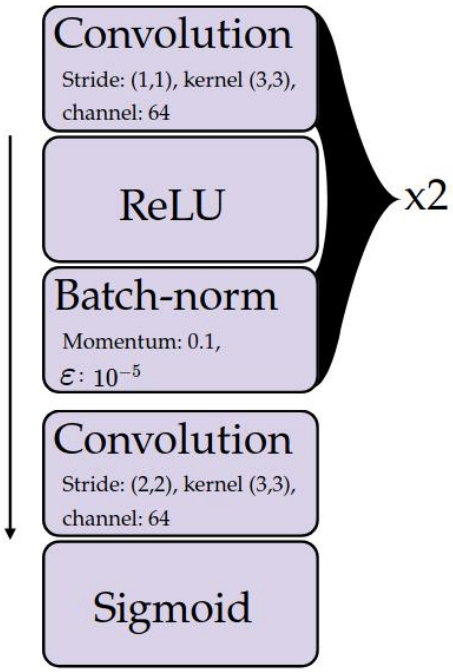

Fig. 4.8: Design of the convolutional encoder network which consists of three convolutional blocks. The last block has a stride value of 2 in both dimensions, which reduces the size of the input by the factor of 2.

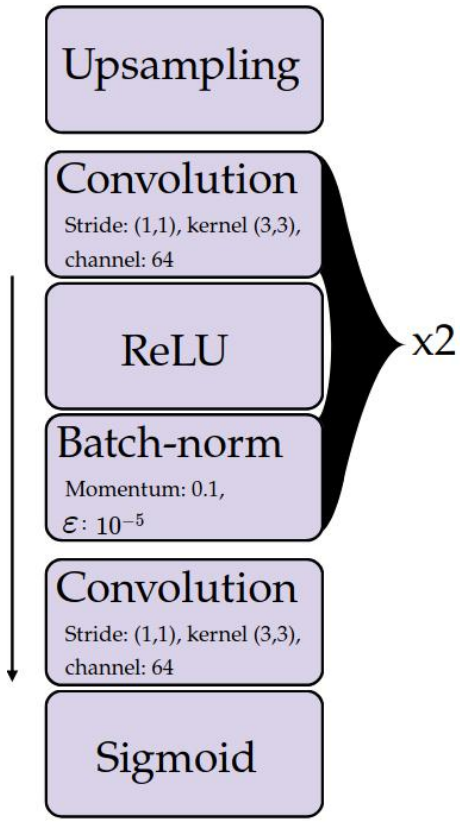

Fig. 4.9: Design of the convolutional decoder network which consists of three convolutional blocks. The first layer *upsampling* upscales the input by the factor of 2.

LSTM-network

In both networks (CV and ST), the encoder-LSTM as well as the decoder-LSTM is stacked with three LSTM-units. Moreover, the input sequence is time inverted, which mean that the states (h, C and in the ST-LSTM additionally M) are evolving backwards in time. The inversion is implemented because the input sequence is a

time span which lasts from the time step 0 to T time steps into the future. The prediction of the Barkley parameter u to a depth of D discrete length units is done for the simulation at the relative time step 0, which is the reason why it is more effective to evolve the states in the LSTM towards to this time step 0 backward in time.

4.2.2 Optimal hyperparameter search

While the architecture of the two neural networks (C-LSTM and ST-LSTM) will be considered as given, the optimal hyperparameter concerning the training process is searched by a grid search. The process only takes into account the performance of the C-LSTM network (CV). The resulting hyperparameters are then also used for the ST model, to compare the performance of the two networks with the exact same properties. The grid search tests the following hyperparameters: The loss function, learning rate, batch size and the type of optimizer.

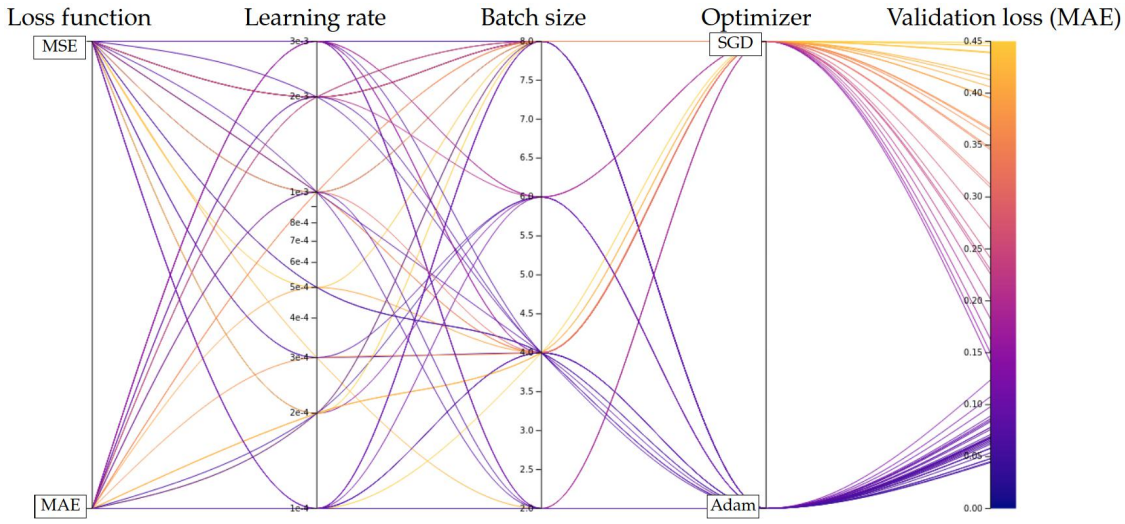

Fig. 4.10: Results of a grid search with 4 varying hyperparameters. The hyperparameters in this graphic are sorted by their influence on the loss from less important (left) to more important (right) (the importance is sorted by random forest feature importance to calculate how much a hyperparameter choice played a role in the result according to the metric). The parameter variations and best combination are in table 4.4. The loss is calculated after one epoch with 1024 samples.

A *parallel coordinates* plot is shown in figure 4.10. The appearance of the hyperparameters is sorted by their influence for a low validation loss. It becomes visible

that the Adam optimizer performs better here than the SGD optimizer in this task: Many lines whose paths pass through the node representing the Adam optimizer end up with lower values for the validation loss than those lines that pass through the node of the SGD optimizer. Furthermore, one can see that a batch size of 4 is the best choice for a low validation loss. The other hyperparameters have a relatively small influence on the validation loss, which is why it is difficult to see graphically which learning rate and loss function is advantageous. Table 4.4 shows the ideal hyperparameter combination with the underlying search space. The grid search, as well as the parallel coordinates plot is done with the experiment tracking API Weights and Biases [10].

Tab. 4.4: Characteristic dimensionless units for the two Barkley configurations.

Parameter	Values	Best value
Loss function	MSE, MAE	MSE
Learning rate	$\{1, 2, 3, 5\} \cdot 10^{-4}$, $\{1, 2, 3\} \cdot 10^{-3}$	10^{-4}
Batch-size	2,4,6,8	4
Optimizer	SGD, Adam	Adam

It becomes visible, that the Adam optimizer is advantageous, when comparing with the SGD optimizer. Furthermore, a batch size of 4 seems to be the best choice. Finally, a learning rate of 10^{-4} provides the best results. Nonetheless, during the grid search, the learning rate was fixed over the whole training process. During training in the experiments, the learning rate is dynamic, and gets smaller by the factor of $\frac{1}{5}$ as soon as the loss function shows no improvement over a certain number of training iterations. The exact properties of the training process are introduced in the following section.

4.2.3 Training process

This section refers to the training process, whereby the techniques of the optimization process will be discussed first. Afterwords, specific settings of the training process are shown.

The training process is done with the Adam optimizer. Two regulations during training are applied: Early-stopping and Reduce-learning-rate-on-plateau. After the loss did not achieve a better result over a period of 192 iterations, the learning rate is dropped by the factor of 5 and till a minimum learning rate of 10^{-7}. The learning rate starts with a value of 10^{-4}. The following is a part of the training loop, coded

in Python with help of the pytorch framework. In line 1 and 8 of the code listing in 4.1, an object `config` is called, which is a dictionary and contains predefined hyperparameter such as the number of epochs or the depth till which the neural network should predict. The object `train_dataloader` is a class in pytorch which groups a list of paired datapoints X,y into batches. Subsequently, the batches can be called iteratively in a `for`-loop (line 2). With the commands in line 6, the data is transferred to the computing device used for the calculation (such as a graphics card, as is also the computing device in this experiment). Especially for a storage consuming training, it can be critical to only load one batch at the same time into the computing device (here a GPU), and delete it from the computing device after the training update.

Listing 4.1: Implementation of the training process with the PyTorch in Python.

```
1    for epoch in range(config['epochs']):
2        for i, (X,y) in enumerate(train_dataloader):
3            model.zero_grad()
4            optimizer.zero_grad()
5
6            X, y = X.to(device), y.to(device)
7
8            y_pred = model(X, max_depth=config['depth'])
9            loss = criterion(y, y_pred)
10           y_pred = y_pred.detach()
11
12           loss.backward()
13           optimizer.step()
```

The calculation of the training process was performed by one respectively two Tesla P100-PCIE-16GB GPU's (depending on the choice of the parameter T and the resulting memory consumption). If two GPU's are used, a batch is divided between these two GPU's, where a training update is performed according to the average of the gradients. The gradients are computed from the loss function on the respective sub sample (batch).

In this experiment, the performance of the neural network is evaluated after including different amounts of input time steps $T \in (1, 2, 4, 8, 16, 20, 25, 28, 30, 32)$, to predict the u-value till a depth of $D = 32$. Since the neural networks (CV and ST) can provide a prediction independently from the input and output sequence length, which is a property of the previously introduced seq2seq-models, the training pro-

cess is identical for every varying T. The two networks are trained on both datasets (regime A and B). This results in 10 (number of different T) $\cdot$ 2 (two regimes) $\cdot$ 2 (two kind of neural networks) $= 40$ training processes. The neural networks are optimized to predict the u-value as accurately as possible at all depths, with being trained from a given amount of input time steps. In the following, the results of the networks are validated, depending on the input sequence length T.

4.3 Results and validation

In the following, the performance of the two neural networks is presented, which are trained to reconstruct the u-value of the Barkley model under the surface at a given time. The evaluation refers to the 4 configurations, consisting of the pairwise combination from the two different neural networks, with C-LSTM and ST-LSTM architecture, which are trained on simulation data from the two Barkley regimes. The following list is a summary of the previously introduced formula symbols and terms for a better understanding of the following evaluation.

- T: Number of time steps included in the input sequence X for the neural networks with $X \in \mathbb{R}^{T\times1\times120\times120}$. The last three dimensions stand for the spatial expansion of the first layer from the cube.
- D: Number of layers of the cube up to which the neural networks make predictions y with $y \in \mathbb{R}^{1\times D\times120\times120}$.
- CV: Seq2seq neural network with convolutional LSTMs (C-LSTM).
- ST: Seq2seq neural network with spatio-temporal LSTMs (ST-LSTM).
- Regime A: Configuration of the Barkley simulation which provides a dynamic with periodic spiral waves.
- Regime B: Configuration of the Barkley simulation, which lead to chaotic behavior.

4.3.1 Predictions

This section shows a few predictions, made from both neural network designs. Here, the focus lays on the influence of the input sequence length T, while the evaluation

considers two aspects regarding the predictions: On the one hand, the principle ability of the neural networks to reconstruct hidden structures is examined. On the other hand, the accuracy of the predictions are validated by the MAE distance measurements. One might describe the two validation aspects as qualitative and quantitative testing of the neural networks predictions: Figure 4.11 shows an example, where the prediction is quantitatively good in terms of pointwise absolute error between the predictions and true values (*good* in comparison to some predictions, made on the chaotic dataset, as it can be seen later). The neural network is, in this example, able to predict the principle behavior of a pattern (marked with the green circles in the figure)in the depth, which expands with increasing depth. Nonetheless, as visible from a depth of 8 (marked with red circles), the prediction does not consider an upcoming pattern. The ability of predicting hidden structures, without regarding a metric for a quantitative error calculation is examined in the following section.

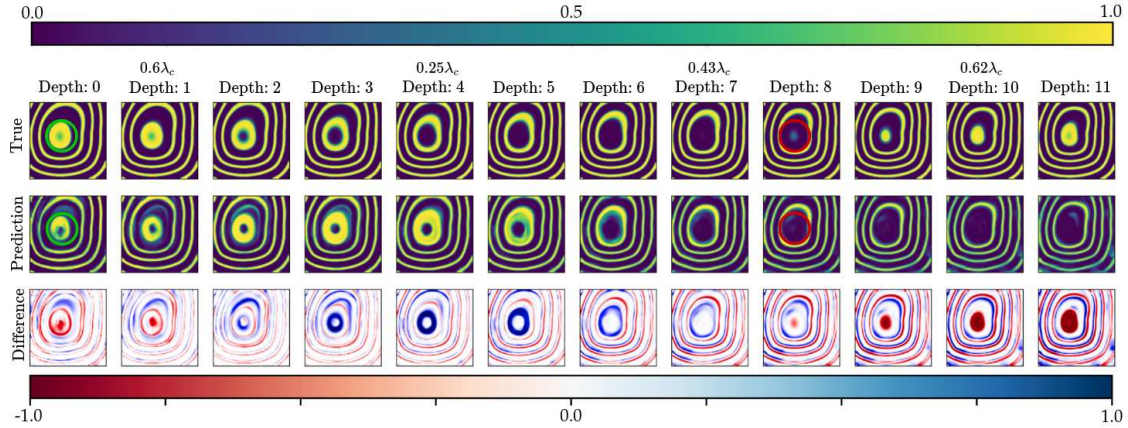

Fig. 4.11: True values and the predictions till a depth of 11 of a *CV* network which was trained with $T = 8$ input time steps on regime A. The last column visualizes the difference between the true values and the predictions. The green circles mark a pattern, whose behaviour in deeper layers is regarded as being predicted well by the neural network. The red circles show where a pattern has not been recognized.

4.3.1.1 Structure reconstruction

In this section, the principle ability of the neural networks to reconstruct pattern is shown. In the following visualizations of the predictions and true values, the u-value is visualized as long as it is over a threshold of 0.5. In figure 4.12, a trend is visible where the predictions get more accurate if more future time steps are included.

In addition, deeper layers can be predicted with more time steps, included in the input. As shown in the figure at $T = 1$, some areas (marked with red circles) are not predicted satisfying. At $T = 8$, the prediction looks better, but although deeper patterns have been predicted in principle, the quality of these is not particularly good. With including 32 time steps, the structure looks qualitatively better. This figure is only for illustration, the quality of the predictions can only be derived to a limited extent with this visualization, since the visualization of the predictions here also depends on the freely selected threshold.

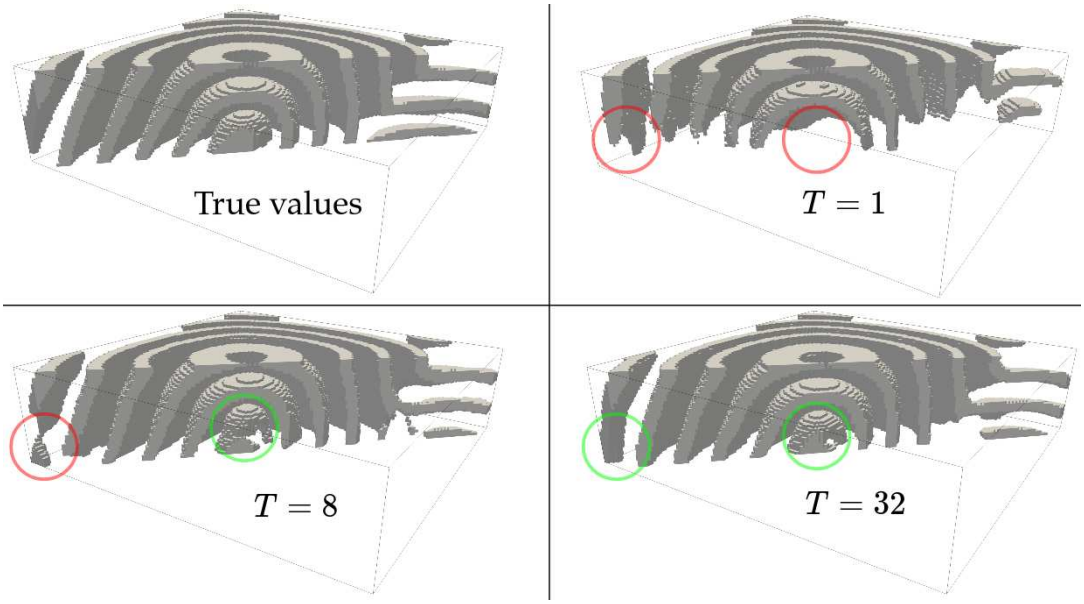

Fig. 4.12: Example of the true values and predictions till a depth of 32 length units of C-LSTM networks, which have been trained with $T = 1, 8$ and 32 time steps. Points are visualized if they are above the threshold of $u > 0.5$. Red circles mark examples of areas where structures under the surface are not predicted satisfying, and green circles are placed, where the quality of the predictions is good. A cross section of the entire prediction is shown.

Another figure 4.13 shows the predictions of three different CV networks trained with different input sequence lengths as well. The values for the u-value are not visualized with respect to a threshold, but color coded. An improvement of the quality of the predictions can be seen by adding more time steps as well. In the figure, red circles mark areas where no emerging structure was detected. The green circles mark areas where it was possible to reconstruct such structures. While no structures were reconstructed below the surface at a depth of 9 in the case of $T = 1$, the reliability of such a reconstruction is improved with increasing T. Therefore, a

correlation between the value of T and the prediction of deeper structures can be seen here. In addition, it can be seen that the predictions improve quantitatively.

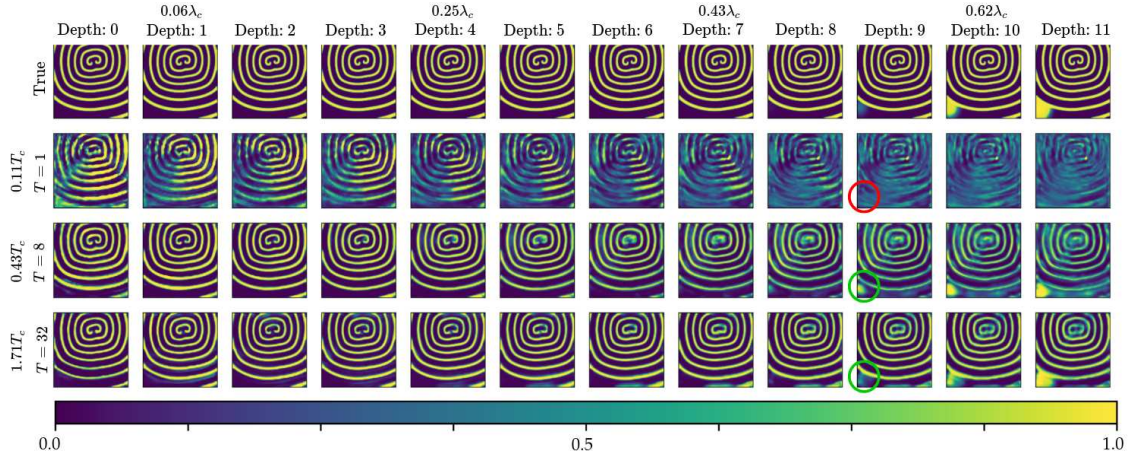

Fig. 4.13: Predictions of multiple C-LSTMs till a depth of 11 on the dataset from regime A, trained with $T \in \{1, 8, 32\}$. The red circle marks an area, on which an upcoming structure with increasing depth is not predicted by the neural network, where the green circle stands for a successful prediction.

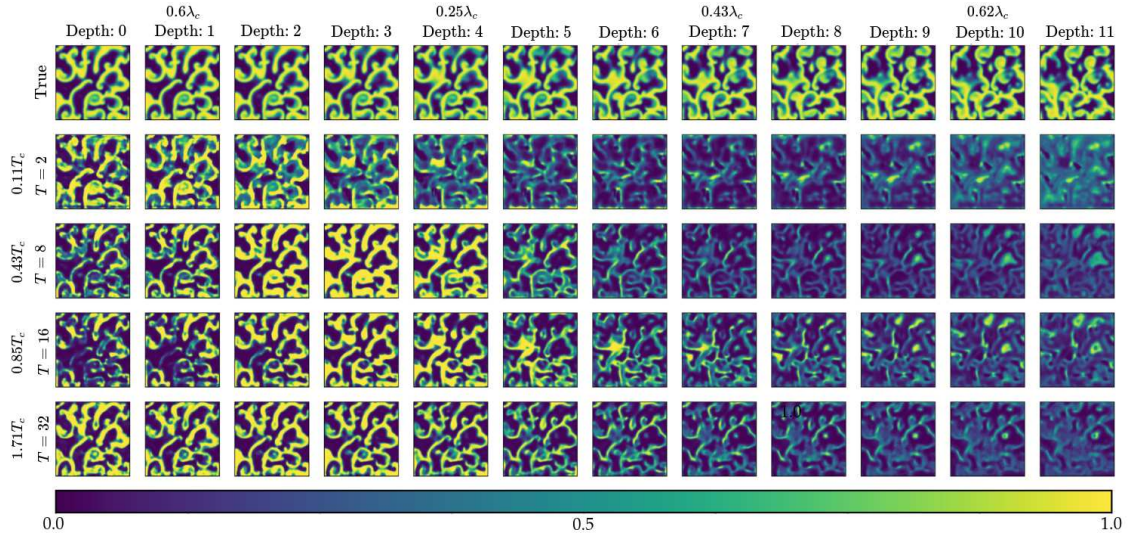

Fig. 4.14: Predictions of multiple C-LSTMs till a depth of 11 on the dataset from regime B, trained with $T \in \{2, 8, 16, 32\}$.

Figure 4.14 shows the kind of visualization of predictions such as the previous one. In this case, the predictions are done by CA-networks as well, but on the dataset from the regime B with chaotic behaviour. It is difficult to identify the ability of

detecting hidden structures such as in figure 4.13. The evaluation of the performance on both regimes is subject of the following (quantitative) evaluation.

4.3.2 Quantitative evaluation

In this section, the predictions of the neural networks are validated the mean absolute error (MAE). Here, the error is calculated per layer of the cube. It means, that if, for example, the neural network predicts the u-value up to a depth of 32, this prediction is divided into the 32 planes, where each lies below the surface of the cube at a certain depth. The error is then calculated for each layer individually. Furthermore, an *average regressor* is applied so that one has a comparison with the error that occurs when a prediction is only the global mean of all u-values from the training dataset. The average regressor has no predictive power, so models whose errors come close to the average regressor at certain depths do not make usable predictions there.

4.3.2.1 Error development in dependence on T

Figure 4.16 shows several curves of the MAE in dependence of the depth. The first column of plots are results from the two neural network designs on the dataset from regime A. A trend is visible, where with increasing T, the MAE is generally smaller, compared to other MAE's at the same depth with smaller T. However, in the CV-network (top left figure), this property faces a saturation at about $T = 25$. For larger values of T above it, no clear improvement is noticeable. The ST-network (top right figure) reaches a saturation already at around $T = 16$. This may be due to the fact that an input sequence with length $T = 16$ already covers nearly one period length ($0.85 \cdot T_c$). Since the regime A is periodical, no more information is provided by covering a longer time span.

The second column show the MAE values on the dataset of regime B. No satisfactory result can be seen here. Furthermore, no neural network (CV or ST) can be identified as better performing machine learning model. The values for MAE fluctuate strongly and are even worse over wide ranges than a regression, which exclusively predicts the global mean of all u-values.

4.3.2.2 Comparison of C-LSTM and ST-LSTM

In the following, a comparison is made between the CV- and ST-network's performance. The idea behind the ST-LSTM design, which is basis for the ST architecture,

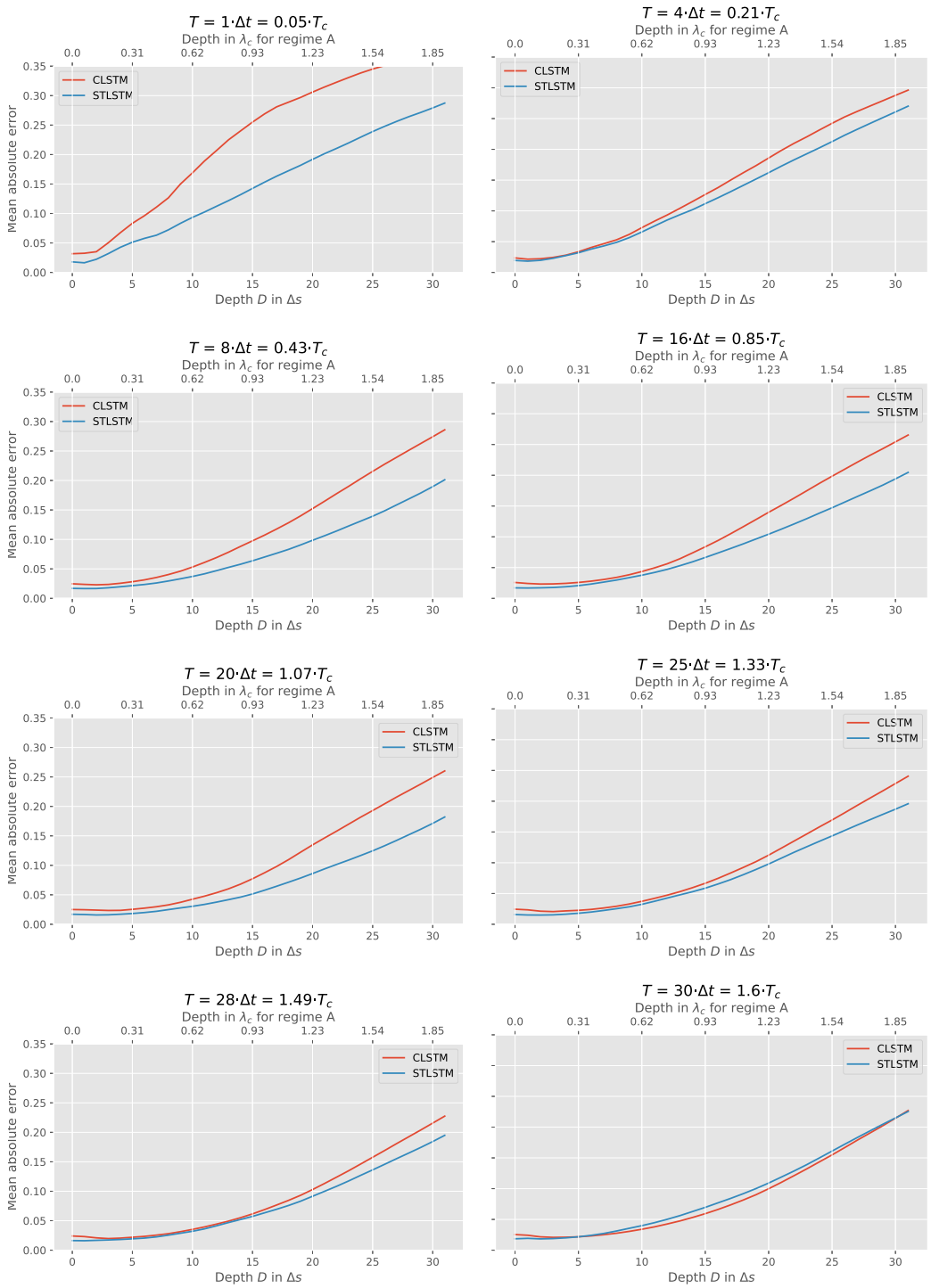

Fig. 4.15: Several mean absolute errors per depth on the dataset of regime A from the two neural network architectures C-LSTM and ST-LSTM.

is to enable the equal processing of spatial correlations and temporal dynamics. The authors of the paper *PredRNN: Recurrent Neural Networks for Predictive Learning*

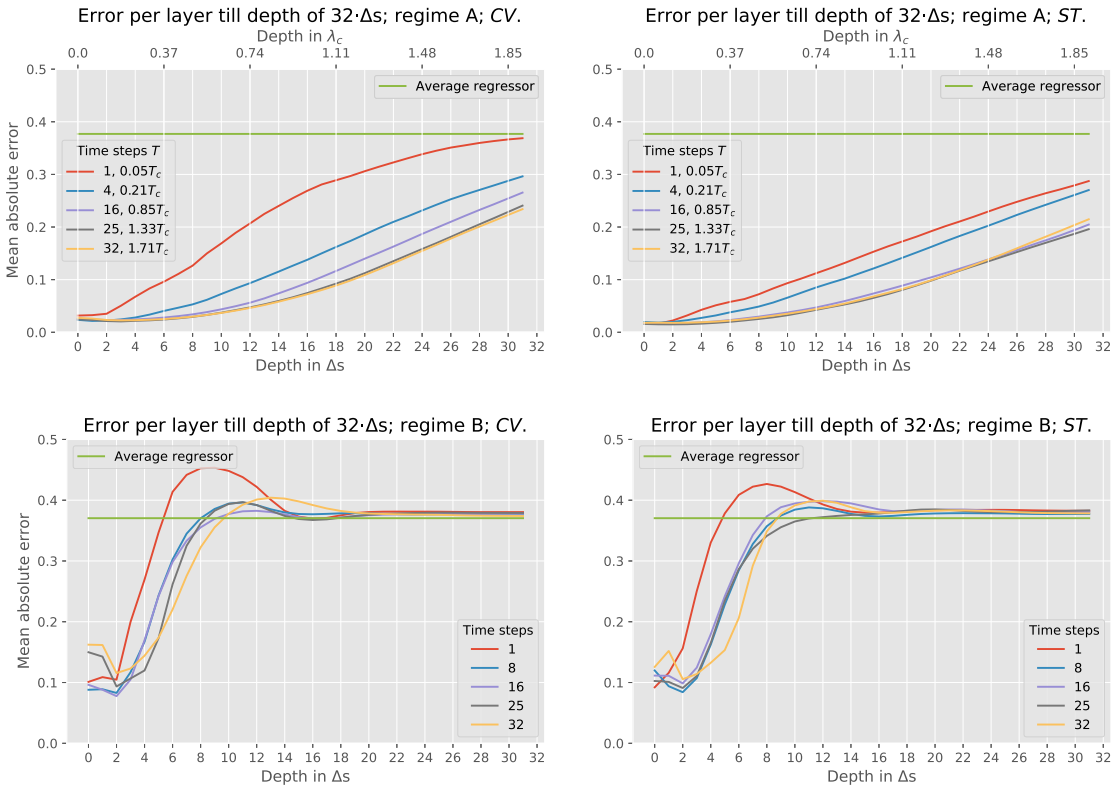

Fig. 4.16: Several mean-average-errors per depth; calculated on the two datasets; trained on the two neural network designs (*CV* and *ST*).

using Spatio-temporal LSTMs [40] argue for the inequality with a missing spatial flow because the representation h is firstly developing over time and then passed to the next layer of the stacked LSTM. The implementation of an additional cell state M which is firstly passed through the layers and then passed to the input of the next time step of at the bottom of the ST-LSTM, is used to reduce this problem. By including only a small amount of input time steps T, the need to process spatial correlations and temporal dynamics more equally does not arise here as much as with larger T. However, the *ST*-network already performs much better at small T compared to the *CV*-network. Reasons for this performance cannot be justified here with the named properties of the *ST* because of the given reasons. It may be due to the higher number of trainable parameters of the ST (3.7M) compared to the CV (1.9M). In each figure in 4.15, the MAE per depth is plotted for *CV*- and *ST*-networks, which are trained on a different amount on input time steps T. It can be seen that the *CV* network has slightly better results than the *ST* network up to $T = 16$. For values of T over 16, the ST-LSTM is clearly better. This may be due

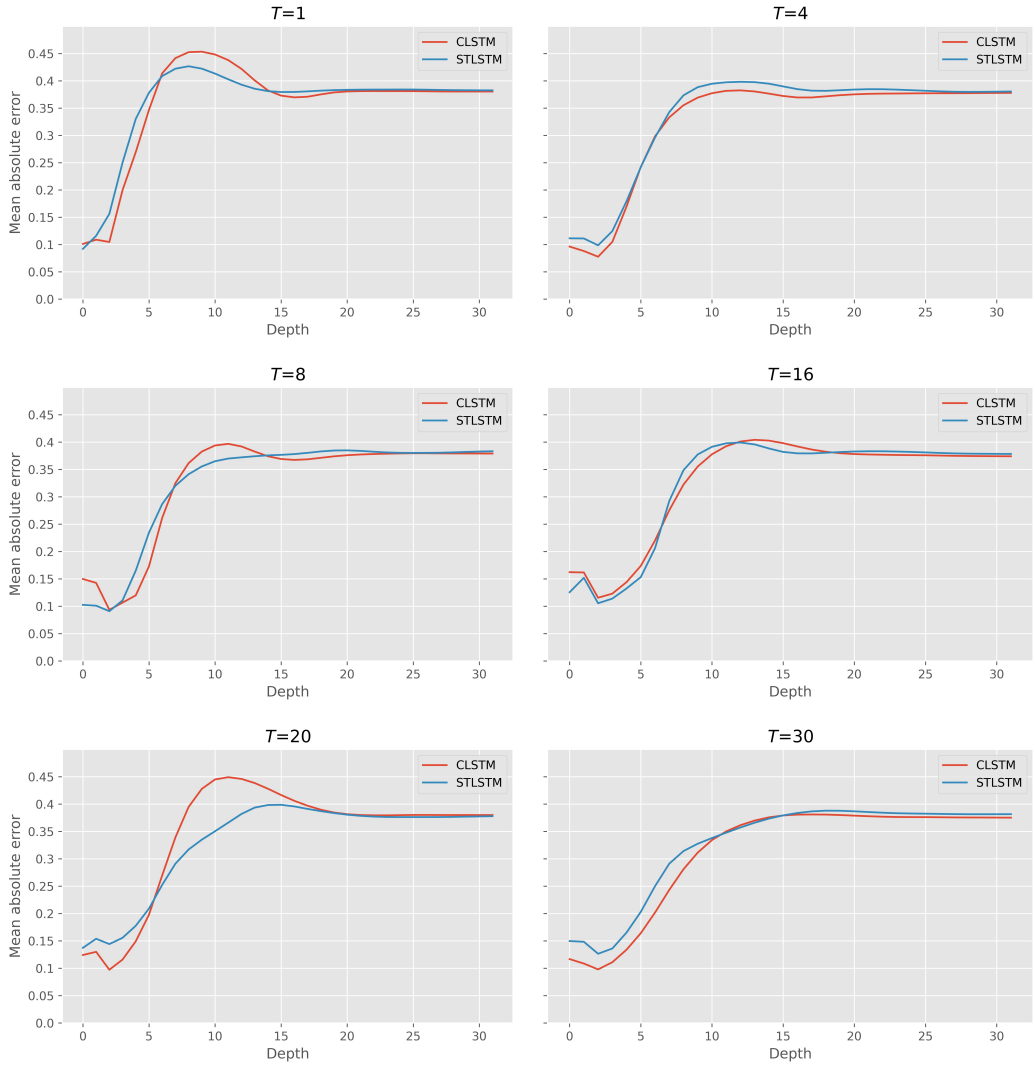

Fig. 4.17: Several mean absolute errors per depth on the dataset of regime B from the two neural network architectures C-LSTM and ST-LSTM.

to the properties of the ST-LSTM, regarding the previous argumentation. However, this property is not visible anymore with using the neural networks on the dataset from regime B, with chaotic behaviour. The errors in dependence of the depth are plotted in the figures in 4.17. In the case of regime A, the already observed better performance of the ST-LSTM for larger T is shown as well, while in regime B no network model clearly behaves better for certain T.

CHAPTER 5

Discussion and outlook

Within this work, two numerical experiments were carried out by using models from the field of machine learning. In both experiments, the aim was to find out more about an excitable medium on the basis of its development over time.

5.1 Inverse problem of ECG

In the first experiment, the ability of reconstructing the dynamics at the heart surface was tested by a specifically posed task. The locations of sources of manually generated concentric waves were predicted. This approach was used to test the feasibility of neural networks in ECG-signal processing. The chosen approach with convolutional neural networks delivered satisfying results. However, it was suspected that the networks tend to overfit due to the similarity of training and test data, and thus no real predictive power can be confirmed. Despite this potential problem, the experiment showed that convolutional neural networks can process ECG-signals well, while being independent of the temporal starting point of the input sequence.

5.2 Reconstruction of 3D excitations

In this experiment, an attempt was made to predict the excitation of the Barkley model in 3D from its u-value within a cube using two different seq2seq models. The networks made predictions for layers below the surface of the cube based on the time evolution of the surface excitation. It was shown that the quality of the prediction depends strongly on the length of the input sequence. It can be said that a longer time span gives a better result but only conditionally: The evaluations of

the predictions have shown that with periodic dynamics, as it was tested here, the results are not necessarily better when the time span exceeds a period length. In the case of regime B, which exhibited chaotic behavior, no useful predictions could be made, since the predictions of the two seq2seq models have not been significantly better or even worse than the *average regressor*.

The experiment is a study in which some aspects are kept simple. These are the model of the excitable media (Barkley-model), which can be regarded as a simple model with its two variables, and the geometry in which the simulation took place (a cube). The results allow to identify two properties of the task in this experiment: Spatio-temporal dynamics can be processed better if the neural network architecture allows it. Such as convolutional neural networks are designed to address spatial relations in images better than feed-forward networks, the ST-LSTM is designed to process through spatial relations and temporal dynamics more equally than a C-LSTM. The ST-LSTM showed in nearly every tested input time span a better performance, compared to the C-LSTM. However, this also may be due to the fact, that the ST-LSTM has more trainable parameters.

However, the neural network approaches, which have been chosen in this experiment, require an assignment of the simulation points, to a certain depth. Moreover, the chosen neural network architectures are only capable of predicting entire layers with uniform depth, while the predictions are spatially arranged on a grid. These two aspects are potentially problematic when trying to apply these neural networks on more complicated geometries, such as that of a heart as in the first experiment, where the geometry is defined through a more complex unstructured grid. Nevertheless, a transfer of the neural networks attempt in this experiment to the geometry of a heart, is the goal of further research.

Bibliography

[1] Ahrens et al. “ParaView: An End-User Tool for Large Data Visualization”. In: *Visualization Handbook, Elsevier* (2005).

[2] Martin Alnæs et al. “The FEniCS Project Version 1.5”. In: *Archive of Numerical Software* Vol 3 (2015). DOI: 10.11588/ANS.2015.100.20553. URL: http://journals.ub.uni-heidelberg.de/index.php/ans/article/view/20553 (visited on 02/16/2021).

[3] Dario Amodei et al. “Deep Speech 2: End-to-End Speech Recognition in English and Mandarin”. In: (2015). eprint: arXiv:1512.02595.

[4] André Araujo, Wade Norris and Jack Sim. “Computing Receptive Fields of Convolutional Neural Networks”. In: *distill* (2019). DOI: 10.23915/distill.00021.

[5] D. Barkley. “Barkley model”. In: *Scholarpedia* 3.11 (2008), p. 1877. DOI: 10.4249/scholarpedia.1877. URL: http://www.scholarpedia.org/article/Barkley_model.

[6] Dwight Barkley, Mark Kness and Laurette S. Tuckerman. “Spiral-wave dynamics in a simple model of excitable media: The transition from simple to compound rotation”. In: *Physical Review A* 42.4 (Aug. 1990), pp. 2489–2492. DOI: 10.1103/PhysRevA.42.2489. URL: https://link.aps.org/doi/10.1103/PhysRevA.42.2489 (visited on 02/18/2021).

[7] Jacques Bélair et al. “Dynamical disease: Identification, temporal aspects and treatment strategies of human illness”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 5.1 (Mar. 1995), pp. 1–7. DOI: 10.1063/1.166069. URL: https://doi.org/10.1063/1.166069.

[8] Y. Bengio, P. Simard and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* (Mar. 1994), pp. 157–166. DOI: 10.1109/72.279181. URL: https://ieeexplore.ieee.org/document/279181/ (visited on 11/16/2020).

[9] Sebastian Berg, Stefan. Luther and Ulrich. Parlitz. "Synchronization based system identification of an extended excitable system". en. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 21.3 (Sept. 2011), p. 033104. DOI: 10.1063/1.3613921. URL: http://aip.scitation.org/doi/10.1063/1.3613921 (visited on 02/18/2021).

[10] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[11] Sergey Edunov et al. "Classical Structured Prediction Losses for Sequence to Sequence Learning". In: (2017). eprint: arXiv:1711.04956.

[12] Jeffrey L. Elman. "Finding Structure in Time". In: *Cognitive Science* (Mar. 1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1. (Visited on 01/20/2021).

[13] Richard FitzHugh. "Impulses and Physiological States in Theoretical Models of Nerve Membrane". In: *Biophysical Journal* (July 1961), pp. 445–466. DOI: 10.1016/S0006-3495(61)86902-6.

[14] Yoav Goldberg. "A Primer on Neural Network Models for Natural Language Processing". In: (2015). eprint: arXiv:1510.00726.

[15] Wei Han et al. "ContextNet: Improving Convolutional Neural Networks for Automatic Speech Recognition with Global Context". In: (2020). eprint: arXiv:2005.03191.

[16] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735 (visited on 10/01/2020).

[17] Sepp Hochreiter et al. "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long Term Dependencies". In: *A Field Guide to Dynamical Recurrent Networks* (2001), pp. 237–243. DOI: 10.1109/9780470544037.ch14.

[18] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (2015). eprint: arXiv:1502.03167.

[19] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). eprint: arXiv:1412.6980.

[20] Xiaodong Liu et al. "Very Deep Transformers for Neural Machine Translation". In: (2020). eprint: arXiv:2008.07772.

[21] Anders Logg and Garth N. Wells. "DOLFIN: Automated Finite Element Computing". In: *ACM Transactions on Mathematical Software* 37.2 (2010). DOI: 10.1145/1731022.1731030.

[22] Jeni Martin and Matt Martin. *Medical*. 2021. URL: https://www.pinterest.de/jenimartin5/medical/ (visited on 02/26/2021).

[23] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* (Dec. 1943), pp. 115–133. DOI: 10.1007/BF02478259. URL: http://link.springer.com/10.1007/BF02478259 (visited on 01/09/2021).

[24] T. Mikolov and G. Zweig. "Context dependent recurrent neural network language model". In: *2012 IEEE Spoken Language Technology Workshop (SLT)*. 2012, pp. 234–239. DOI: 10.1109/SLT.2012.6424228.

[25] Rohit Mohan and Abhinav Valada. "EfficientPS: Efficient Panoptic Segmentation". In: *International Journal of Computer Vision (IJCV), 2021* (2020). eprint: arXiv:2004.02307.

[26] John Moult et al. "A large-scale experiment to assess protein structure prediction methods". In: *Proteins: Structure, Function, and Genetics* (Nov. 1995), pp. 2–4. DOI: 10.1002/prot.340230303. URL: http://doi.wiley.com/10.1002/prot.340230303 (visited on 02/26/2021).

[27] Michael Mozer. "A Focused Backpropagation Algorithm for Temporal Pattern Recognition". In: *Complex Systems* 3 (Jan. 1995).

[28] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA* (2017).

[29] Baltasar Rüchardt. *MonodomainInBath*. https://gitlab.gwdg.de/bruecha/monodomaininbath. 2020.

[30] David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[31] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[32] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: http://www.nature.com/articles/s41586-020-03051-4 (visited on 01/09/2021).

[33] M. Schuster and K.K. Paliwal. "Bidirectional recurrent neural networks". en. In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997), pp. 2673–2681. ISSN: 1053587X. DOI: 10.1109/78.650093. URL: http://ieeexplore.ieee.org/document/650093/ (visited on 12/02/2020).

[34] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[35] Jan Steffel, Thomas F. Lüscher and Corinna Brunckhorst. *Herz-Kreislauf: mit 25 Tab.* de. Springer-Lehrbuch. Heidelberg: Springer Medizin, 2011. ISBN: 978-3-642-16717-1.

[36] Steve S. Ryan. *Atrial Fibrillation: Recources for patients*. 2021. URL: a-fib.com (visited on 02/26/2021).

[37] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *International Conference on Machine Learning, 2019* (2019). eprint: arXiv:1905.11946.

[38] The AlphaFold team. *AlphaFold: a solution to a 50-year-old grand challenge in biology*. 2020. URL: https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology (visited on 12/20/2020).

[39] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[40] Yunbo Wang et al. "PredRNN: Recurrent Neural Networks for Predictive Learning using Spatiotemporal LSTMs". In: 30 (2017). Ed. by I. Guyon et al.

[41] Yanzhao Wu et al. "Demystifying Learning Rate Policies for High Accuracy Training of Deep Neural Networks". In: (2019). eprint: arXiv:1908.06477.

[42] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. eprint: arXiv:1609.08144.

[43] Yue Wu et al. "Future Video Synthesis with Object Motion Prediction". In: (2020). eprint: arXiv:2004.00542.

[44] Roland Zimmermann. *rcp_spatio_temporal*. https://github.com/zimmerrol/rcp_spatio_temporal. 2017.