Deep learning of nonlinear development of unstable flame fronts

by Ludvig Nobel

LUND
UNIVERSITY

Thesis for the degree of Master of Science
Thesis advisors: Assoc. Prof. Rixin Yu

To be presented, with the permission of the Faculty of Engineering of Lund University, for public criticism on the online meeting at the Department of Energy Sciences on Monday, the 14th of November 2022 at 14:30.

This degree project for the degree of Master of Science in Engineering has been conducted at the Department of Energy Sciences, Faculty of Engineering, Lund University.

Supervisor at the Lund University was Rixin Yu. Assoc Prof. at the Department of Energy Sciences, Lund University.

Examiner at Lund University was Professor Johan Revstedt.

ISSN: <0282-1990>
LUTMDN/TMHP-23/5515-SE

Typeset in LaTeX
Lund 2022

Contents

List of Figures

List of Tables

Abstract

The purpose of this study is to investigate Machine Learning methods and their ability to learn the development of nonlinear unstable flame fronts due to diffusive-thermal instabilities. This task is performed by first numerically computing long time-sequences of solutions to the chaotic partial differential equation named Kuramoto-Sivashinsky equation which models such instabilities in a flame front. From the generated solution functions an operator is trained to map the function to a future solution function after a small time-step. The goal is for this operator to be able to accurately map long sequences of solutions through repeated application of the operator. Two networks were trained for this task, a Convolutional Neural Network and A Fourier Neural Operator. The investigation found that the operator were not only able to accurately predict fairly long sequences, but was also able to capture the long-term characteristics of the flame front development. This study also shows that it is possible with specific modifications to a Convolutional Neural Network proposed in the study, a single Neural Network is able to make accurate recurrent predictions for multiple values of a parameter affecting the solution of the partial differential equation considered.

Chapter 1

Introduction

The diffusive-thermal instability considered in this study is an intrinsic flame instability, meaning that it is a combustion instability which can appear in both premixed flames and in diffusion flames. This instability is one of the major reasons that cause a nonlinear development of unstable flame fronts, which can appear as a cellular structure on the ”surface” of a freely propagating flame caused by a combustion of a premixed fuel[1]. The reason that this instability appears is because of the difference in the diffusion coefficient between the fuel and heat transport. Small deviations in these flows can cause this instability to grow so large that they change the features of the flow. Flame propagation due to this diffusive-thermal instability can be modeled by the ”Kuramoto-Sivashinsky” (KS) equation.

The KS equation was first derived by Kuramoto and Tsuzuki but in the context of a reaction diffusion system [2], and later on by Gregory Sivashinsky but this time in the context of flame front propagation[3]. The equation which is a fourth-order nonlinear partial differential equation can be considered to be ”chaotic”. The reason for why it is called chaotic is partly because the distance between nearby solution functions can grow exponentially [4]. Which in turn makes deep learning of the problem very difficult and it is considered close to impossible to make accurate long-term predictions of the time-advancement of the partial differential equation (PDE).

This study is aiming to train a deep neural networks to learn the spatial-temporal evolution of the KS equation. Through training on solution sequences of the PDE generated using the spectral method in combination with with a fourth-order Runge-Kutta to perform the temporal integration. The past decades there has been an exponential increase in interest of the Machine Learning field, and thereby also an increase in new methods being explored and developed such as the PINN method[5], Neural Operator [6], Fourier Neural Operator [7] , different LSTM networks [8] and many more. There are many advantages of using deep learning compared to other methods such as the classical finite element method (FEM) and direct numerical simulation (DNS). For example, direct numerical simulation can be quite slow in terms of computation speed and certain tasks can take days or weeks to simulate, a trained neural network could perhaps compute the same simulation in a fraction of that time. If deep learning methods could be improved so that the performance is comparable to methods like DNS it could make a major difference in fields such as aerospace designing and many other fields where quickly solving partial differential equations is necessary. Machine learning is nowadays also used for many vastly different applications such as image recognition, speech recognition and in search engines to name a few. The networks used for those kind of applications also do not differ very much from the ones used in this study.

Half of this study can be considered as benchmarking of two existing Machine learning methods on the chaotic KS equation for different parameters, these methods which also previously have been tested and proven to be effective when used to learn other PDEs[9]. The methods to be benchmarked are the deep convolution neural network and the Fourier neural operator. While the second half of the study amounts to modify these networks to see if a network can be trained to make accurate prediction for different parameters of the input.

1.1 Background

This study was conducted at the request from Assoc Prof. Rixin Yu at the department of Energy Sciences at Lunds University as a Master Thesis for a master degree in Energy Sciences. The research is done as an extension to the earlier research done by Assoc Prof. Rixin Yu. This research can be found in both the publication [1] about the onset of cellular flame instability in premixed flames and in a publication currently under review, titled: ”Deep learning of nonlinear flame fronts development due to the Darrieus-Landau instability”[9] also done by Assoc Prof. Rixin Yu. The spatial-temporal evolution of these instabilities afflicting a flame front can be modeled through the theoretical partial differential equation named the ”Michelson-Sivashinsky” equation. By introducing some type of noise to the system the solutions were shown to become chaos-like and therefore also very hard for a Neural Network to predict. The research benchmarks three different known Neural Network methods and shows that they are able to learn to accurately predict short-term spatial-temporal evolution of this chaotic-like partial differential equation.

The aim of the research in this thesis is similar to what was described above. With the largest difference being that instead of investigating the flame fronts development due to the Darrieus-Landau instability, this study will instead focus on the spatial-temporal evolution of flame fronts due to diffusive-thermal instabilities. This spatial-temporal evolution can instead be modeled using the ”Kuramoto-Sivashinsky equation”. More on the goals of this project in the next sub-chapter.

1.2 Objective

The objective of this study is to investigate different deep-learning methods ability to learn an operator for the time-advancement of unstable flame fronts. More specifically, to learn the time-advancement of non-linear unstable flame fronts afflicted by diffusive-thermal instabilities. Which can be modeled through the chaotic Kuramoto-Sivashinsky partial differential equation.

The study is focused on investigating two of the more well known Machine Learning methods. These methods are the Fourier neural operator (”FNO”) and deep convolution neural networks (”CNN”). With the aim to see how the accuracy of the networks predictions differ for different implementations of the methods. Also to investigate the ability for the networks to make long-term predictions by altering the training method. Another objective is to investigate how the FNO’s performance is affected by truncating different amounts of Fourier modes in the convolution operator in Fourier space within the Fourier layers.

The final task of the thesis is to modify these methods to see if it is possible to train a network which can accurately learn to make predictions for the time-advancement of a partial differential equation for multiple values of a parameter describing the physical system. In this case the physical system is the channel with the propagating flame subjected to diffusive-thermal instabilities which can be described by the KS equation. The parameter that can be altered is the channel width (”L”).

1.3 Constraints

Due to time constraints, relevance and length aspects a few limitations had to be placed on what this study would include and how thorough the investigation of the problems would be.

The first limitation put upon this study is that only the Kuramoto-Sivashinsky equation will be tested on the neural networks. Furthermore, only the 1-dimensional version of the equation will be considered.

The choice to investigate the CNN and FNO was made because of the large differences in their structure and how they work. No other Neural Network methods were chosen which means that more focus could be placed on investigating the CNN and FNO.

As later will be explained, to use any Neural Network to learn the time advancement of an PDE, the infinite-freedom function has to be discretized to a finite-freedom function. This dicretization can be viewed as a mesh over the continuous function and the size of this mesh determines the resolution. While a coarse mesh is faster to compute it is also less accurate and vice versa. For this study only two mesh sizes were chosen: N=128 and N=256.

Chapter 2

Theory

This chapter briefly explains some of the theory behind the different methods used in the neural networks and some background information about the Kuramoto-Sivashinsky equation. Chapter 2.2 introduces the basics and some history of simple feed-forward neural networks and might be overindulgent if already familiar with the topic of Machine Learning. However, chapter 2.3 and 2.4 describes the Convolutional Neural Network and the Fourier Neural Operator, both of which are used extensively in this study.

2.1 Kuramoto-Sivashinsky equation

A cellular structure can develop on the front of a freely propagating flame caused by combustion of a premixed fuel, this happens mainly due to the Darrieus-Landau instability and the diffusive-thermal instability[1]. This cellular flame structure looks like there is "wrinkling" on the surface of the flame. Since Assoc. Prof. Rinxin Yu already have investigated the deep learning of nonlinear unstable flame fronts due to the Darrieus-Landau instability, this study will instead focus on the diffusive-thermal instability.

This instability can be modeled using the one-dimensional chaotic partial differential equation named the Kuramoto-Sivashinsky equation.

$$u_t + u_{xx} + u_{xxxx} + uu_x = 0, \quad x \in (-L/2, L/2), \quad u = u(x,t) \tag{2.1}$$

The equation was first derived by Kuramoto and Tsuzuki [2] to handle reaction diffusion problems. Later on Sivashinsky derived the same equation but now to model the propagation of a flame front, where u(x,t) is the disturbance of an unstable planar flame front in the direction of propagation [10]. Numerical experiments have shown that the equation becomes chaotic if solved with periodic boundary conditions, a large enough **L** and when time **t** goes towards infinity [2]. Studies done on the the 1-dimensional case of the equation with Neumann boundary conditions led Nicolaenko et al to notice an low modal behaviour and proposed that the number of determining Fourier modes is proportional to the cell size [11] [12]. Furthermore, Hyman and Nicolaenko states that ”there are strong evidence that the infinite dimension solution solution space is spanned by the solutions to a coupled system of ODEs with only a few degrees of freedom”[10]. The authors also state that the KS equations transition to chaos can be analyzed using tools developed for low-dimensional dynamical systems since the KS equation is strictly equivalent to such a low-dimensional dynamical system.

Papageorgiou and Smyrlis states in the paper ”The route to chaos for the Kuramoto-Sivashinsky equation”[12] that since the behaviour of the system is ”low modal” important characteristics such as period-doubling cascades to chaos is not lost by a truncated system. This led them to numerically solve the equation by two different methods. The first method by classical discretization and then performing the time integration of the linear part in Fourier space and the nonlinear part in real space. The second method was through a Galerkin projection onto low-dimensional inertial manifold which yields a coupled nonlinear system of ODEs. For this case the nonlinear part was advanced in time through a fourth-order Runge-Kutta method and the linear term was treated the same way as for the other method.

2.1.1 Fourth order Runge-Kutta

The fourth order Runge-Kutta (RK4) can be used to numerically find the solution to an ordinary differential equation [13]. Consider an initial value problem (IVP) with,

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0 \tag{2.2}$$

where y is an unknown function of time t and the goal is to approximate that function. From 2.2 it is evident that the derivative $\frac{dy}{dt} = f(t, y)$ is a function of t and y. If the function f and t_0, y_0 is given

$$y_{n+1} = y_n + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \tag{2.3}$$

$$K_1 = hf(t_n, y_n) \tag{2.4}$$

$$K_2 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}K_1 h) \tag{2.5}$$

$$K_3 = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}K_2 h) \tag{2.6}$$

$$K_4 = f(t_n + h, y_n + K_3 h) \tag{2.7}$$

where $t_{n+1} = t_n + h$ and h is the step size. The RK4 approximates $y(t_{n+1})$ with y_{n+1}.

2.2 Neural Networks

Both the name ”Neural Network” (referred to as ”NN”) and the fundamental working principle of it, is inspired by how the neurons work in a human brain. Since, as later will be further explained, the ”neurons” of the network operates in a way which is similar to how the biological neurons communicate with each other. There are many different purposes with neural networks and they can be used in vastly different areas. A few of these applications are image recognition, speech recognition, stock market predictions, social media algorithms and predicting the time-advancement of ordinary or partial differential equations. The simplest kind of implementation consists of one layer containing a single neuron. This neuron is fed with one or many input values which are combined through some linear function. This single value is then modified by an arbitrary non-linear operation to produce the output.[14] However, neural networks are only one kind of implementation method within the field of Machine Learning (referred to as ”ML”). Which in turn is only a subset of the academic field Artificial Intelligence (referred to as ”AI”). One can think of the difference between AI and ML like this; a computer uses AI to perform tasks like learning and predicting and generally behave like a human. While ML is how the computer system becomes intelligent and able to perform these tasks [15].

2.2.1 Brief history

The academic fields of machine learning and AI cannot be said to have been recently discovered. Alexander L. Fradkov [16] means that the origin of machine learning is usually associated with the American psychologist Frank Rosenblatt, who had an idea to create a computer system which imitates the way an human brain works. This idea originated from research done by McCulloch and Pitts, in this research they invented the first mathematical model of an artificial neuron back in the 1940s. Almost twenty years later Rosenblatt [17] continued their research on artificial neurons and his goal was to create a system which could recognise the letters of the Alphabet through image recognition. This project resulted in the first modern neural networks which Rosenblatt called the ”perceptron”.

After this invention in the neural network field of the machine learning community the progress continued slowly but steadily until shortly after the new millennium started. This was the occasion when Neural networks got wide-spread attention through the new deep neural networks performance compared to the alternative machine learning methods[16].

2.2.2 Basics

Throughout the years many different types of NNs has emerged with increasingly higher prediction accuracy and to cover different fields and purposes. Today most of the NN that are being employed are so called ”Deep Neural Network” (DNN). The only real criteria for a network to be deep instead of regular is that it contains more than one ”layer”[18], i.e. in the case of the perceptron it would mean that for the network to be considered deep it would have to have two or more consecutive neuron layers. Moreover, each layer usually contains multiple parallel neurons.

A multi-layer ”Feed-forward Neural Network” (referred to as FNN) is one type of DNN[19]. The network consists of many neurons stacked in layers. The neurons in the FNN operates a bit differently from the perceptron neuron, which will be shown in the following sub-chapters.

Another of the most common NNs is the Convolutional Neural Network, the idea was first presented in a network called ”Neocognitron” by Kunihiko Fukushima back in 1979[18]. Since then this idea has been further evolved. Today this method is usually referred to as Convolutional Neural Network (”CNN”). A more detailed account of how the CNN works will be presented in section 2.3.

2.2.3 Architecture

The layout of a NN can vary greatly depending on which model is chosen and for what purpose the NN is created for. The simplest of FNNs contains as earlier stated an amount of layers and within each layer an amount of neurons (also called nodes) are stored. The first layer is usually called the input layer and the last is called output layer, the layers in between these two layers are called hidden layers[19]. The amount of hidden layers deployed in the NN is sometimes referred to as the depth of the network and the amount of neurons in a layer is called the width. In the case when each neuron in each layer is connected to each neuron in the following layer the network is called fully-connected. The structure of the NN can be chosen in a close to infinite amount of ways by choosing different widths, depths and how the layers are connected. For instance, one can choose to create a separate branch for one of the inputs and then merge that sub-branch with the rest.

The best structure to employ depends greatly on the purpose of the NN. For instance, consider a case where the network has four inputs and one of these inputs is completely unrelated to the other three, then one could employ a branching architecture as described above. One example where this could be preferable is if the purpose of the NN is to predict the long-term time-advancement of an ordinary differential equation such as the Lorenz equation. For long-term solutions the time-advancement needs to be split into shorter time-steps where the NN only predicts one step at a time. A common implementation is then to create a NN that is trained with a varying time-step on the input data. This means that the inputs to this network would be (x_1=dt, x_2=x, x_3=y, x_4=z) where ”dt” is the time-step and (x, y, z) are coordinates in euclidean space. The output would be the predicted coordinates (x’,y’,z’) after the specified time-step (dt).

To put it simply, the fundamental principle of a fully-connected FNN is that each input is connected to each neuron in the first hidden layer. Each of these connections has an assigned value called weight. This weight is multiplied with the value of the input and sent to the neuron. Then all the incoming connections to the neuron are summarized together. The neuron contains an activation function which is deployed on this summarized value, if the output of the function is above a stored value within the neuron called bias the neuron is activated. If the neuron is activated the connections to the next layer are opened and so on until the output layer.

One might think that the bigger the network is the better it will be able to make accurate predictions. This is however often not the case. By increasing the width and depth then the amount of trainable parameters for the network grow exponentially. For example, by increasing the width by one in a fully connected FNN. Mean that even if that only adds one extra activation function and one extra trainable bias. It would also add extra trainable weights equal to the amount of neurons in the previous and consecutive layer combined. As later will be proven this leads to heavier computational cost, i.e. longer time to train the network and harder for the network to converge.

2.2.4 Weights and Biases

The FNN learns to make accurate predictions and to recognise patterns with the help of the weights and biases[19]. Both these values are trainable, meaning that through training the network by feeding it with enough input data and then by comparing the output to a desired output these weights and biases in the neurons can be adjusted through penalizing functions. The goal is that this will enable the following predictions to differ less from the desired output.

The perceptron presented earlier is one of the simplest kinds of neurons. However, the general idea is still the same in modern neurons. The perceptron takes the desired amount of binary inputs $[x_1, x_2, ..., x_j]$, multiplies them with a weight $[w_1, w_2, ..., w_j]$

and compares them to the bias value and then outputs a single binary value:

$$Output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq bias \\ 1 & \text{if } \sum_j w_j x_j > bias \end{cases} \tag{2.8}$$

Which can be simplified through instead creating vectors of the inputs and weights and using the dot product of these vectors and moving the bias to the left-hand side:

$$x = [x_1, x_2, ..., x_j], w = [w_1, w_2, ..., w_j] \tag{2.9}$$

$$x \cdot w = \sum_j w_j x_j \tag{2.10}$$

$$Output = \begin{cases} 0 & \text{if}\, x \cdot w + bias \leq 0 \\ 1 & \text{if}\, x \cdot w + bias > 0 \end{cases} \tag{2.11}$$

The (sigmoid) neuron is a more modern variant of the perceptron. One of the major differences between the neurons are that the modern (sigmoid) neurons inputs and outputs does not have to be binary. However, the neuron still uses weights and biases in a similar way as the perceptron does. The output from the neuron is usually determined by the sigmoid activation function ”Logistic”:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.12}$$

$$z = \sum_j w_j x_j + b \tag{2.13}$$

$$Output = \frac{1}{1 + exp(-\sum_j w_j x_j - b)} \tag{2.14}$$

$$Output = \frac{1}{1 + exp(-x \cdot w - b)} \tag{2.15}$$

2.2.5 Activation functions

Today there exists many different activation functions with different properties. The choice of a suitable activation function for the considered task is crucial for the effectiveness of the NN. As Johannes Lederer states in his publication ”Activation Functions in Artifical Neural Networks” [20], ”The activation functions themselves influence the network’s expressivity, that is, the network’s ability to approximate target functions”. In the same publication Johannes Lederer also proves that Networks that only utilizes linear activation functions are always linear, meaning that the network will not be able to predict or learn non-linear functions. Which activation function that is chosen also impacts parameter optimization which in turn affects the computational difficulty. The reason for this is because many of the optimizers that adjust the networks parameters uses gradients of the output to do so. A more detailed explanation of this is given in the chapter about back propagation.

Sigmoid functions

The sigmoid functions are suitable as activation functions due to that they are easily differentiable, non decreasing ($x_1 < x_2 \implies f(x_1) < f(x_2)$), bounded and because they only have one inflection point. In the section below a couple of activation functions from the Sigmoid category will be presented[20]:

$$\text{Logistic: } \frac{1}{1+e^{-z}}, \qquad \mathbb{R} \rightarrow \{0,1\} \tag{2.16}$$

$$\text{Arctan: } arctan(z), \qquad \mathbb{R} \rightarrow \{-\frac{\pi}{2}, \frac{\pi}{2}\} \tag{2.17}$$

$$\text{Tanh: } tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}, \qquad \mathbb{R} \rightarrow \{-1,1\} \tag{2.18}$$

$$\text{Softsign: } \frac{z}{1+|z|}, \qquad \mathbb{R} \rightarrow \{-1,1\} \tag{2.19}$$

The logistic function has been called the ”smooth binary function” which is evident when looking at figure 2.1 and when evaluating the equation 2.18. Since for high positive value of z the output will be one and for high negative the output will be zero.

Piecewise-Linear Functions

The piecewise-linear functions are another category of popular activation functions. The popularity is mainly due to their computational efficiencies when compared to other types of activation functions. For example, both the functions in themselves and their derivatives are easily computed since they contain no exponential or trigonometric functions. They are also not as afflicted by the ”vanishing gradient” problem as the sigmoid functions. The most common piecewise-linear functions are: linear, relu and leakyrelu[20].

$$\text{Linear: } z, \qquad \mathbb{R} \rightarrow \{-\infty,\infty\} \tag{2.20}$$

$$\text{relu: } max\{0,z\}, \qquad \mathbb{R} \rightarrow \{0,\infty\} \tag{2.21}$$

$$\text{leakyrelu: } max\{0,z\} + min\{0,az\}, \qquad \mathbb{R} \rightarrow \{-\infty,\infty\} \tag{2.22}$$

As can be seen in figure 2.2 the linear activation function is purely linear, with the first derivative always equal to 1 and the second derivative equal to 0. Which makes it obvious that this activation function is very efficient in terms of computation expensiveness. Johannes Lederer[20] means that many claim that constant derivatives impacts the

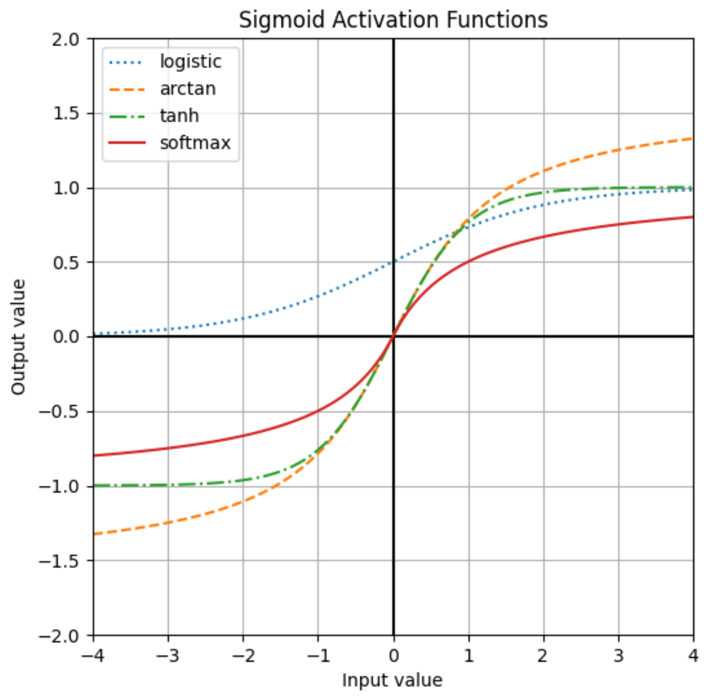

Figure 2.1: Sigmoid activation functions plotted on a input range from (-4,4), the output axis is scaled differently (-2,2)

optimization poorly, but he also states implies that may be inaccurate since the optimization is also affected by the loss function. However, as earlier stated the drawback of using only linear activation function in a NN is that the network would not be able to account for non-linearity. This is why most NN only use the linear activation function in certain layers. The relu function is much similar to the linear in many aspects, however, the function is not linear since all negative inputs returns an output of zero. One of the drawbacks with the relu function is that it is afflicted by a type of vanishing gradient problem called ”dying-relu”. The leakyrelu was developed to counteract the dying-relu phenomena.
To summarize the recent sub-chapters with an example, the input into a neuron from 5 neurons in the previous layer can be viewed as, see figure.2.3:

Where ”a_1” is the output from that specific neuron, also called ”activation”. The output is thereby defined as such:

$$Output = \sigma(a_1 * w_1 + a_2 * w_2 + a_3 * w_3 + a_4 * w_4 + a_5 * w_5 + b) \tag{2.23}$$

Where σ is the activation function.

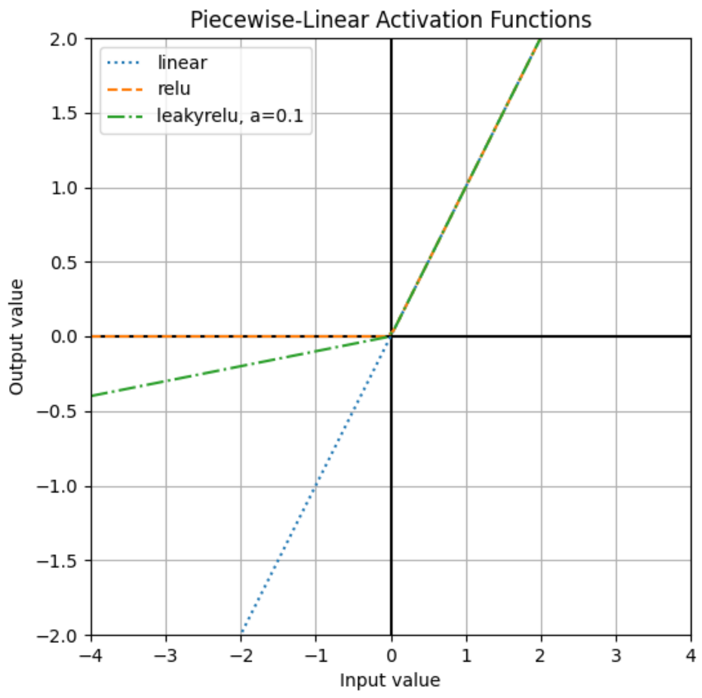

Figure 2.2: Piecewise-linear activation functions plotted on a input range from (-4,4), the output axis is scaled differently (-2,2)

2.2.6 Loss Function

The training process of the NN can be done several ways, the most common is to split the entire training database into smaller ”batches” where each batch contains a certain portion of the entire database. The network then iterates through each batch consecutively until the whole database has been covered which would complete one ”epoch”. Usually the network is trained for a large amount of epochs, which means that it will iterate through the same data multiple times. For instance, consider a case where an network is trained to recognise patterns in images. Each image that is to be used for training needs a corresponding label so that the output of the NN can be compared and then adjusted according to the label or desired output. To train the network efficiently it needs to see many similar samples so that the patterns can be defined in the weights and biases. Say that in this case one has a training database of 10000 samples then in many cases it will be advantageous to split this database in to smaller groups of samples, i.e. ”batch”. The choice of batch size and if it even should be used depends on many different variables such as the size of database, data characteristics, network type and structure, which optimizer that is used and more. In most cases one has to find the correct batch size by trial and error. An oversimplification of how the batch size affects the gradient descent training process can be seen in figure 2.4. In the figure the red arrows represents

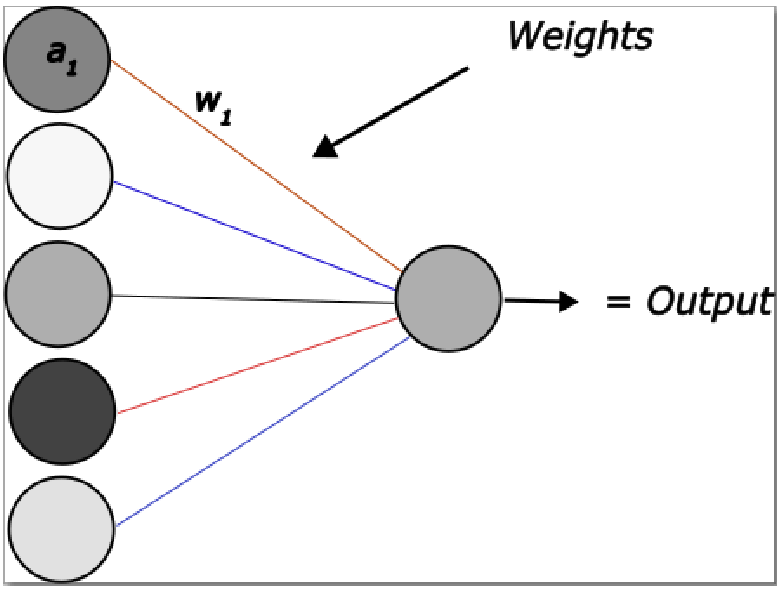

Figure 2.3: An exmaple of the output from a neuron

a training case where batches are not used, blue represents a too small batch size and the black arrows represents the correct batch size for the situation. The figure shows how the batch size affect the gradient descent in the optimization of the network, gradient descent is explained in the following chapter.

By using a very small batch size or running optimization after every sample leads to many more and less generalized adjustment of the parameters but can lead to higher accuracy. Using a large batch size means that the network make predictions for a large amount of samples in parallel and calculates a summarized loss (a value of how far off the prediction is compared to the desired output) based on all the predictions. Then the network is optimized based on this combined loss.

Loss function is the function that determines by how much the network parameters are to be adjusted after each batch. As stated, the loss is an measure of how accurate predictions the network can make, since a low loss means that the prediction is close to the target. Loss functions can also be split into two major sub-categories, regression - and classification loss functions. Regression loss functions predicts output values based on the given input values, examples of this type is mean square error loss (”MSE”) which is categorized under LP loss functions (”LP”)[21]. While classification loss functions produces a vector of probabilities of which categories the input should be classified as. The cross entropy loss function is a common loss function in this sub-category. Classification loss functions are obviously not suited to predict the time advancement of partial differential equations and will therefore not be further studied.

For an input of **n** rows and **p** columns the mean square error would be calculated with

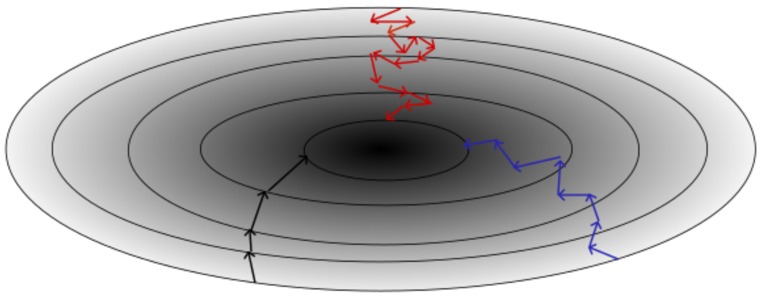

Figure 2.4: Simplification of how batch size can affect gradient descent, i.e. optimization of weights and biases

the following formula:

$$MSE = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}(\hat{x}_{ij} - x_{ij})^2 \tag{2.24}$$

$$L2 = \frac{1}{np}\sum_{i=1}^{n}\sum_{j=1}^{p}(\hat{x}_{ij}^2 - x_{ij}^2)^{1/2} \tag{2.25}$$

Where $\hat{x}$ is the predicted output and x is the desired output value.

2.2.7 Back Propagation Algorithm

Back propagation is a crucial tool for the training and learning of regular neural networks and is performed at the end of each training batch. The main purpose of the task is to, as the name implies, propagate backwards through each layer and each neuron of the network from the output to the input and to adjust each trainable parameter according the loss of the current batch. However, it wouldn't be a very effective process if all the parameters were changed by the same amount. Since in most cases the loss of an entire batch consists of a single real value calculated from a loss function, then a method is needed that can differentiate the importance of each weight in regards to the output and adjust accordingly. Today there exists many such algorithms and many of them are based on a method called ”gradient descent”[22].

If one considers the loss function to have the weights and biases as the input, the training data as the parameters and the loss as the output. Then for a specific batch the goal is to adjust these weights and biases so that the loss function returns the lowest possible value. To make the case easier, one can consider this function to have only one input

(w) and one output ($loss$), which means that $loss = C(w)$. If the function is simple the minimum can be found easily through basic calculus, see figure 2.5 where the x-axis corresponds to the value of the weight and the y-axis to the loss.

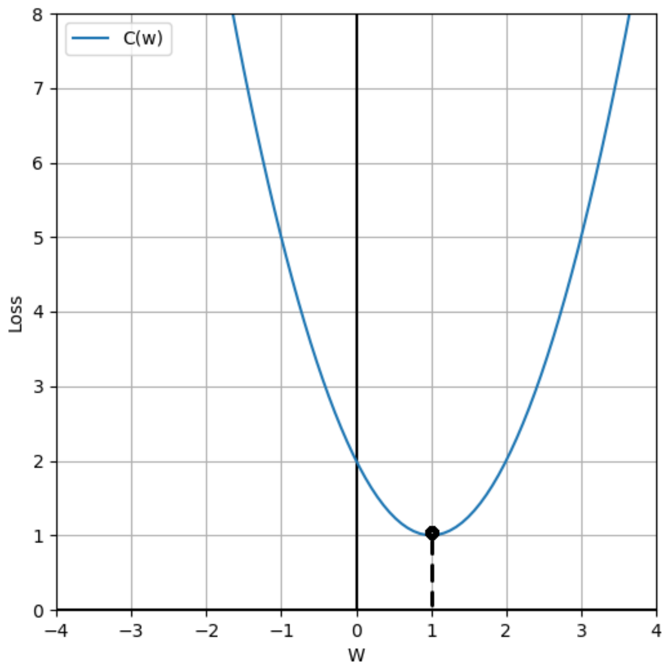

Figure 2.5: Finding minimum of a simple function $loss = C(w)$

Then the minimum is found through:

$$\frac{dC}{dw}(w) = 0 \tag{2.26}$$

However, the case is not as simple as this since the functions are more complicated and especially since the amount of weights in a network usually counts in the thousands. In the case of a more complicated function but still simplified compared to a real case can be seen in fig 2.6, using the same tactic to find the minimum will not suffice. Since this function has multiple extreme values which means that eq.2.26 also has multiple solutions, and in this case one of these solutions will be the local maxima at $w \simeq -0.5$ which obviously is not preferable.

The problem is then to find a method which does not end up in local maximums but instead finds the minimums. One way to do this is by doing an iterative evaluation of the slope at the current position. In other words, by starting at an arbitrary point on the function and then evaluating the slope at this position, if the slope is positive a new evaluation is done a ”step” to the left/backwards of that position and if the slope is

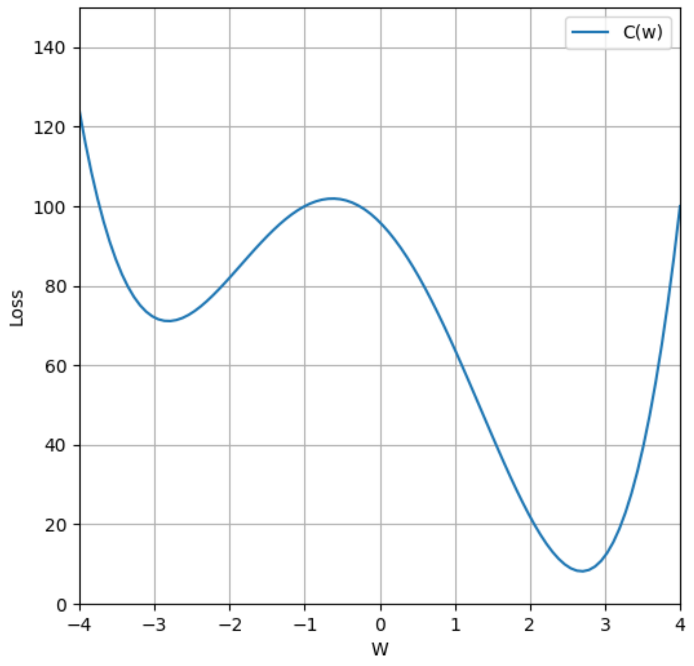

Figure 2.6: Finding minimum of a more complicated function $loss = C(w)$

negative a new evaluation of the slope is done a bit to the right/forward until a minimum is found. See fig.2.7 for clarification.

One way to avoid ”overshooting” the minimum value is to make the size of the step proportionate to the slope at the current position.

However, this method does not take into account if the minimum found is the global- or simply a local minimum of the function. Meaning that depending on the starting point one might as well end up in the first minimum seen in fig.2.7.

If the function instead has two inputs and one output ($loss = C(x, y)$) then one can imagine the input space as the xy-plane and the loss function as an surface graphed above it. Using the same technique as earlier described but now its not the slope that is being evaluated but instead the direction of the step is determined by the gradient. Since the gradient is the direction in which the function increases the most, the negative of that gradient gives us the direction in which the function decreases the most. Another useful property of the gradient is that the length of the gradient is an indication of how steep the slope of that direction is. Which is as described earlier useful for determining the step size.

Today there exists many different back-propagation algorithms that can be used for training a neural networks, this process or algorithm which back-propagates through the

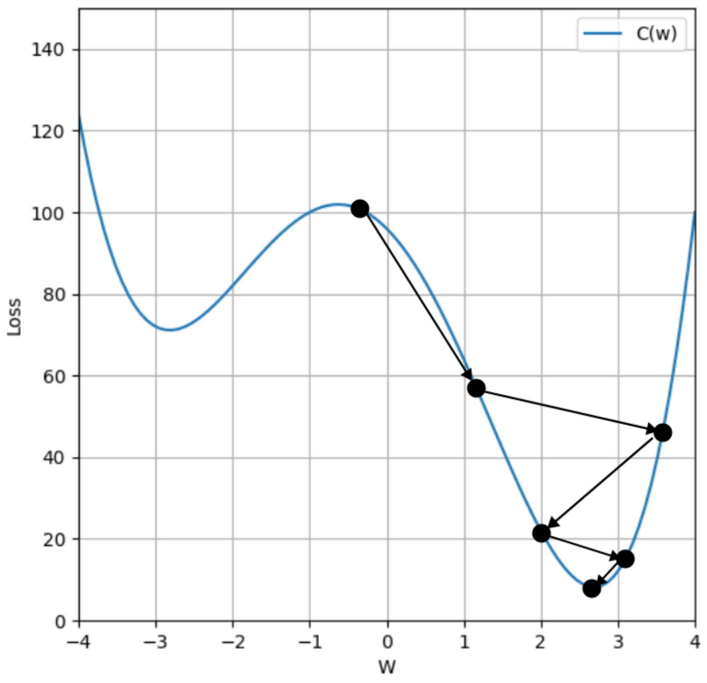

Figure 2.7: Finding minimum of a more complicated function $loss = C(w)$ by stepwise evaluating the slope

network and updates the weights to minimize a certain loss function is usually referred to as the optimizer of the network. Consider a case where one has a database containing the inputs $x_1, ..., x_j$ and outputs $y_1, ..., y_j$ which have been arranged in pairs where $(x_1, y_1), ..., (x_j, y_j)$ and the task is for a neural network (F) to learn this relationship between x and y.

$$y = F(w, x) \tag{2.27}$$

where w are the weights of the network. As earlier stated the loss or cost is calculated through any given loss function C.

$$Loss = C(y, F(x, w)) \tag{2.28}$$

And the task is to minimize this loss function by only altering the weights of the network. One of the most simple optimization techniques is the gradient descent method [23] with a weight update defined as:

$$w_{t+1} = w_t - \eta \nabla_w C(w_t; x, y) \tag{2.29}$$

where η is the learning rate coefficient which determines the size of the weight updates. Note that equation 2.29 has not specified a specific weight in the network to alter. To specify the weight update in a single weight the definition of gradient needs to be considered. The gradient of loss function C in this case is,

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, ..., \frac{\partial C}{\partial x_j}] \tag{2.30}$$

Consider a fully-connected FNN, meaning that the amount of incoming weights to a neuron always equals the amount of neurons in the previous layer. Then for a single weight in layer l and neuron j in that layer and k refers to the neurons in the previous layer then the gradient of this weight can be described as following through the chain rule

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} \tag{2.31}$$

where z^l_j is the output from neuron j in layer l before using the activation function. Then From eq.2.23 it is known that the value z is equal to

$$z^l_j = \sum_{k=1}^{m} w^l_{jk} a^{l-1}_k + b^l_j \tag{2.32}$$

where m is the amount of neurons in the previous layer. By differentiating eq.2.32 with respect to the considered weight w^l_{jk} leads to,

$$\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k \tag{2.33}$$

Which means that by inserting eq.2.33 into eq.2.31 leads to,

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j} a^{l-1}_k \tag{2.34}$$

One of today's most used optimizer in practice is the Adam optimizer[23][22] because of its efficiency compared to other optimization methods. The name Adam stands for Adaptive Moment Estimation and as the name suggests computes adaptive learning rates for each parameter. Meaning that the method stores an average of previous squared gradients (v_t) and an average of past non-squared gradients (m_t) which both decay exponentially over time.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_w C_t \tag{2.35}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla_w C_t^2 \tag{2.36}$$

where β_1 and β_2 are coefficients, common values for these are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

The weights are altered through the Adam update rule [22],

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t \tag{2.37}$$

where ϵ is also an coefficient, a common value for ϵ is 10^{-8}.

Since each update to the weights are affected by past gradients, this gives the update something similar to momentum and thereby also a robustness against sudden unwanted deviations in the loss minimizing. Which in turn makes the gradient descent process smoother, like the black arrows representing gradient descent in fig.2.4.

2.3 Convolutional Neural Network

The Convolutional Neural Network (”CNN”) is in many ways similar to the Feed Forward Neural Network. The basic idea is the same; the network is given an input which is passed along some hidden layers performing some type of operation on the input and an output is returned.

An important difference between the FNN and CNN is that each operation within a convolution layer on a single value in the input matrix is also affected by the neighbouring values unlike for the FNN where every neuron usually is connected to each neuron in the next layer.

Consider a 2-dimensional input matrix of size 3x3, and a kernel matrix of size 2x2 (also called filter). The convolution between these two matrices produces a new matrix where the output values are calculated by sliding the kernel over the input matrix, see figure.2.8

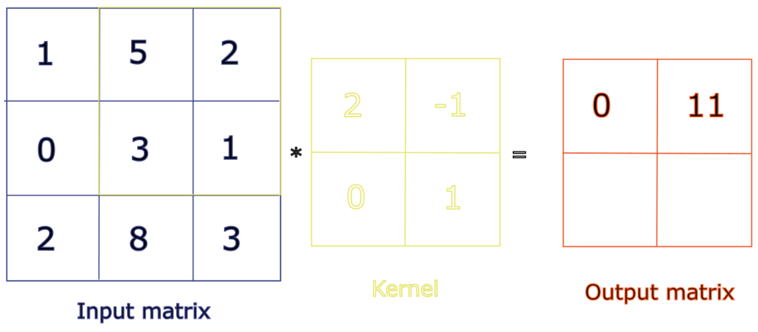

Figure 2.8: By sliding the kernel over the input matrix and performing simple multiplication and additions yields the output matrix

In a standard convolution where the stride is equal to one and padding equal to zero, this case can be seen in figure 2.8. Then the size of the output matrix Y is equal to the size of the input matrix I minus the size of the kernel matrix K plus one $Y = I - K + 1$. The stride of a convolution is what determines how many points on the matrix the kernel should slide after each operation and padding adds an extra layer on the outside of the input matrix without any values stored. By doing this the equation describing the output matrix size becomes $Y = I - K + 3$.

The operation that has been described so far is not actually convolution, this is an operation called cross-correlation[24]. Which is similar to convolution, but for the convolution the kernel matrix is rotated 180 deg. This is denoted as:

$$I * K = I \star rot180(K) \tag{2.38}$$

An important feature to many convolutional neural networks is the ability to shrink the dimension of the matrices while increasing the amount of channels between layers (or the other way around). The way to increase channel size of the input is to in each layer convolute more kernels on the same matrix. These kernels are usually of the same size but storing different values of weights. Meaning that if the input to layer one is one channel with kernel size 3 and the goal is to lift the output channel size to 16 then the amount of kernels will be 16 times 1 each with a size of 3. Now if the next layer lifts these 16 channels to 32 with kernel size 3, means that the convolution takes place on 16 channels with 16 rows times 32 columns of new kernels each storing 3 weights.

The usage of an Autoencoder architecture is common technique for CNNs[25][26]. An autoencoder usually consist of two parts, an encoder and a decoder. The encoder compresses the high dimensional data flow into a lower dimension as described above. The decoder reconstructs this data flow into the original high dimensional input. Many of the Autoencoder CNNs does not use the convolution layers to decrease the dimensionality, but instead uses a pooling layer placed in series with the convolution layer to do this[27]. The max pooling layer reduces the dimensionality of the input through an operation which is similar to convolution. It features an empty kernel which slides over the input matrix and stores the highest value for each step. The reduction in dimensions is dependent on the stride and kernel size of the maxpooling layer. These layers are advantageous because the operation is computationally inexpensive and they introduce small invariance to translation and distortion. The opposite of the max pooling layer is an upsampling layer which can be used to increase the dimension of the data flow.

Consider the case when the input to a convolution layer is a 1-dimensional vector of three values but with a channel size of three, what this actually means is that there is three 1-dimensional vectors all of size three. The desired output size from this layer is two 1-dimensional vectors of size 2, i.e. only two channels. That means that two channels of kernels are needed each containing three 1-dimensional vectors of size two. This operation has been illustrated in fig. 2.9

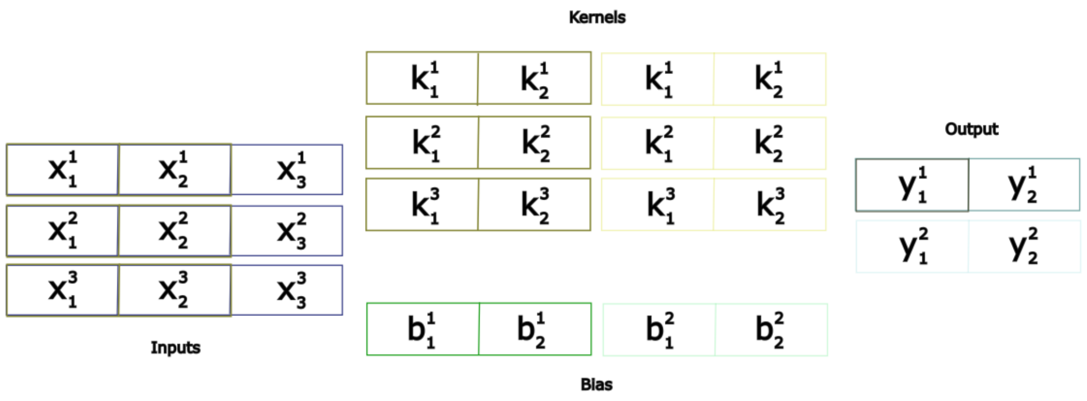
Kernels
Output
Inputs
Bias
x_1^1 x_2^1 x_3^1
x_1^2 x_2^2 x_3^2
x_1^3 x_2^3 x_3^3
k_1^1 k_2^1 k_1^1 k_2^1
k_1^2 k_2^2 k_1^2 k_2^2
k_1^3 k_2^3 k_1^3 k_2^3
b_1^1 b_2^1 b_1^2 b_2^2
y_1^1 y_2^1
y_1^2 y_2^2

Figure 2.9: Convolution with different channel sizes of input/output

Note, the two kernel channels usually does not contain the same weights even though they are denoted as such in the figure. Highlighted vectors means that they are currently involved in the same operation, meaning that y_1^1 is calculated through

$$y_1^1 = k_1^1x_1^1 + k_2^1x_2^1 + k_1^2x_1^2 + k_2^2x_2^2 + k_1^3x_1^3 + k_2^3x_2^3 + b_1^1 \tag{2.39}$$

Which means that each output channel is a linear combination of all the input channels. If the vectors are instead denoted as matrices, where X_1, X_2, X_3 represents each of the channels and $K_{1,1}, K_{1,2}, K_{1,3}, K_{2,1}, K_{2,2}, K_{2,3}$ represents the the kernel vectors in each channel respectively and so on enables eq.2.39 to be developed to,

$$Y_1 = B_1 + X_1 \star K_{1,1} + X_2 \star K_{1,2} + X_3 \star K_{1,3} \tag{2.40}$$

$$Y_2 = B_2 + X_1 \star K_{2,1} + X_2 \star K_{2,2} + X_3 \star K_{2,3} \tag{2.41}$$

$$Y_i = B_i + \sum_{j=1}^{m} X_j \star K_{i,j}, \quad i = 1, ..., d \tag{2.42}$$

where m is the channel size of the input X. Equation 2.42 is basically the forward propagation of a simple convolution layer before the activation function.

The backward propagation for a CNN is very similar to how it works for a FNN, simply by going backwards through the layer and using the chain rule to find the needed partial derivatives. As with the FNN the optimization for a CNN start from the partial derivative of the cost function with respect to the output $\frac{\partial E}{\partial Y_i}$ and the gradients with respect to the weights and biases are what is desired, meaning the partial derivatives $\frac{\partial E}{\partial K_{ij}}$ and $\frac{\partial E}{\partial B_i}$. If the CNN considered is a deep neural network, meaning that it contains multiple convolution layers. Then the partial derivative with respect to the input to the layer also needs to be computed $\frac{\partial E}{\partial X_j}$.

Starting from eq.2.39 but only considering the layer to contain one input channel and output channel, which means that the layer only consists of one input vector of size three, one kernel of size two and so on. Then the equations becomes:

$$y_1 = b_1 + k_1x_1 + k_2x_2 \tag{2.43}$$
$$y_2 = b_2 + k_1x_2 + k_2x_3 \tag{2.44}$$

The goal here is to try to compute the partial derivative of the error from the loss function with respect to the weights

$$\frac{\partial E}{\partial Y} = \frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2} \longrightarrow \frac{\partial E}{\partial k_1}, \frac{\partial E}{\partial k_2} \tag{2.45}$$

Using the chain rule yields:

$$\frac{\partial E}{\partial k_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial k_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial k_1} \tag{2.46}$$

From this last equation, looking at the second division in every term, i.e $\frac{\partial y_1}{\partial k_1}, \frac{\partial y_2}{\partial k_1}$, it is evident that from the forward propagation eq.2.43 that,

$$\frac{\partial y_1}{\partial k_1} = x_1 \tag{2.47}$$

which gives us:

$$\frac{\partial E}{\partial k_1} = \frac{\partial E}{\partial y_1}x_1 + \frac{\partial E}{\partial y_2}x_2 \tag{2.48}$$

This can easily be computed for all kernel points by using cross-correlation,

$$\frac{\partial E}{\partial K} = X \star \frac{\partial E}{\partial Y} \tag{2.49}$$

This is however not really the forward propagation from the case described earlier since this is only for 1 to 1 amount channels, by instead starting from eq.2.42,

$$Y_1 = B_1 + X_1 \star K_{11} + ... + X_n \star K_{in} \tag{2.50}$$
$$Y_2 = B_2 + X_1 \star K_{21} + ... + X_n \star K_{2n} \tag{2.51}$$
$$... \tag{2.52}$$
$$Y_d = B_d + X_1 \star K_{d1} + ... + X_n \star K_{dn} \tag{2.53}$$

The goal is still to find the derivatives of E with respect to K. Note that using the chain rule directly is not possible since the terms are now matrices and not values. Instead by looking at the derivatives separately, for example $\frac{\partial E}{\partial K_{21}}$ and at eq.2.53 it is evident that

$$\frac{\partial E}{\partial K_{21}} = X_1 \star \frac{\partial E}{\partial Y_2} \tag{2.54}$$

since the matrix K_{21} only appears once in the equations. This is also true for all matrices K_{ij}.

Using a similar technique to investigate the gradients for the biases and starting with the simplified version:

$$y_1 = b_1 + k_1 x_1 + k_2 x_2 \tag{2.55}$$

and so on, as before,

$$\frac{\partial E}{\partial Y} = \frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2} \tag{2.56}$$

using the chain rule yields,

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial b_1} \tag{2.57}$$

As for the weights, comparing eq.2.57 to eq.2.43 yields,

$$\frac{\partial y_1}{\partial b_1} = 1 \tag{2.58}$$

$$\frac{\partial y_2}{\partial b_1} = 0 \tag{2.59}$$

which means that,

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y_1} \tag{2.60}$$

$$\frac{\partial E}{\partial b_2} = \frac{\partial E}{\partial y_2} \tag{2.61}$$

$$\longrightarrow \frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y} \tag{2.62}$$

Moving over to the expanded version and looking at the partial derivative of E with respect to B_1

$$\frac{\partial E}{\partial B_1} = \frac{\partial E}{\partial Y_1} \tag{2.63}$$

$$\longrightarrow \frac{\partial E}{\partial B_i} = \frac{\partial E}{\partial Y_i} \tag{2.64}$$

Since B_1 only appears in the first equation.

The last part of the back propagation on the CNN is to look at the partial derivative of E with respect to the input x and again starting by looking at the simplified version. Using the chain rule yields,

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial E}{\partial y_2}\frac{\partial y_2}{\partial x_1} \tag{2.65}$$

By comparing the second division of each term to eq.2.43,

$$\frac{\partial E}{\partial x_{11}} = \frac{\partial E}{\partial y_{11}} k_{11} \tag{2.66}$$

This, however, is not true for all x_i, since for example x_2 appears in both of the equations 2.43. Which means that:

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1} k_1 \tag{2.67}$$

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1} k_2 + \frac{\partial E}{\partial y_2} k_1 \tag{2.68}$$

$$\frac{\partial E}{\partial x_3} = \frac{\partial E}{\partial y_1} k_2 \tag{2.69}$$

If these equations are investigated, one can find that this pattern corresponds to a full correlation where the kernel have been rotated 180 degrees.

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} *_{full}\ rot_{180}(K) \tag{2.70}$$

Which is an convolution not an cross-correlation. Onto the expanded one and starting with eqs.2.53, by again investigating in which of these equations that X_1 appears, which yields:

$$\frac{\partial E}{\partial X_1} = \frac{\partial E}{\partial Y_1} *_{full} (K_{11}) + ... + \frac{\partial E}{\partial Y_d} *_{full} (K_{d1}) \tag{2.71}$$

$$\frac{\partial E}{\partial X_j} = \frac{\partial E}{\partial Y_1} *_{full} (K_{1j}) + ... + \frac{\partial E}{\partial Y_d} *_{full} (K_{dj}) \tag{2.72}$$

$$\implies \frac{\partial E}{\partial X_j} = \sum_{i=1}^{d} \frac{\partial E}{\partial Y_1} *_{full} (K_{1j}), \quad j = 1, .., n \tag{2.73}$$

With this equation derived means that all equations have been derived to be able to perform back propagation on a CNN. The equations needed are:

$$\frac{\partial E}{\partial K_{ij}} = X_j \star \frac{\partial E}{\partial Y_i} \tag{2.74}$$

$$\frac{\partial E}{\partial B_i} = \frac{\partial E}{\partial Y_i} \tag{2.75}$$

$$\frac{\partial E}{\partial X_j} = \sum_{i=1}^{d} \frac{\partial E}{\partial Y_1} *_{full} (K_{1j}), \quad j = 0, ..., n \tag{2.76}$$

Many modern CNNs feature a new type of layer called inception layer [28]. The basic idea behind the layer is to branch the input within the layer to different branches each containing a series of smaller convolution layers and pooling layers. At the end of the inception layer all the branch are concatenated into one output with the desired dimensions. According to Wojna et al. [28] there are several advantages to using inception layers. For example, to avoid representational bottlenecks, high performance vision networks and moderate computation cost in comparison to many other alternative CNN structures.

2.4 Fourier Neural Operator

Fourier Neural Operator is quite a bit different from the classical deep-learning methods such as the CNN or FNN. The idea behind the Fourier Neural Operator [7] is based on the Neural Operator proposed in the scientific paper ”Neural Operator: Graph Kernel Network for Partial Differential Equations” [6]. The Neural Operator can be thought as a structure of several blocks of smaller neural networks, one input neural network ”A” whose task is to lift the input to a higher dimension representation and one output NN ”B” to bring the solution back to the desired dimensional representation. Between these layers is a iterative structure of blocks each performing some linear and nonlinear operation on the solution.

To explain this in more detail, the input PDE solution $v(x)$ is as explained lifted to a higher dimension of representation through a fully connected feed forward neural network. This is done by taking the whole PDE function as one input and returning the solution lifted to the desired dimension by the linear local transformation A, in a way similar to how the convolution layer increases the amount of channels.

$$u(x) = A(v(x)) \tag{2.77}$$

$$A : \mathbb{R}^{d_v} \longrightarrow \mathbb{R}^{d_u} \tag{2.78}$$

where $d_v < d_u$ and $v(x) \in \mathbb{R}^{d_v}$. The function $u(x)$ is iterated T times ($u_1 \mapsto ... \mapsto u_T$ through a block containing a non-local integral operator and a local nonlinear activation function σ such as the RELU explained in chapter 2.2.5. The integral operator contains two branches, one of them is a linear transformation of the input $W : \mathcal{R}^{d_u} \longrightarrow \mathcal{R}^{d_u}$ (typically a convolution layer) and the other branch performs an operation called kernel integral transformation $\mathcal{K}$. Here is where the Fourier Neural Operator differs from the standard Neural Operator. Instead of using a kernel integral transformation, the branch contains a convolution operator defined in Fourier Space. So instead of eq.2.79

$$u_{t+1}(x) = \sigma(W_t u_t(x) + (\mathcal{K}(v;\phi)v_t)(x)) \tag{2.79}$$

One gets the following equation:

$$u_{t+1} = \sigma(W_t u_t(x) + \mathcal{F}^{-1}(\mathcal{R}_t \cdot \mathcal{F}(u_t))) \tag{2.80}$$

where $\mathcal{F}$ is the Fast Fourier Transform (FFT). The linear transformation W_t is the same as in eq.2.79 and does not change the amount of channels/dimensions. The equation is performed by using a kernel of size=1 and also means that each new channel is a linear combination of all previous channels. The last stage of the Fourier Neural Operator is the output neural network B. The task of this network is to reduce the dimensions down to the desired level which is also most often done using a fully-connected FNN. Meaning:

$$v'(x) = B(u_T(x)) \tag{2.81}$$

$$B : \mathbb{R}^{d_u} \longrightarrow \mathbb{R}^{d_v} \tag{2.82}$$

See figure 2.10 for clarification of the FNO architecture.

The convolution operator in Fourier space is the part that is similar to how the spectral method works. From eq.2.80 one can see that this operation corresponds to $\mathcal{F}^{-1}(\mathcal{R}_t \cdot$

$\mathcal{F}(u_t))$. The operator first transforms each of the channels of the input function $u_t(x)$ using the FFT ($\mathcal{F}$). The FFT can be used in this study since the input function is a PDE solution discretized on a uniform mesh with periodic boundary conditions. After the transformation one yields the same amount of Fourier modes κ as the mesh size N. By truncating the highest frequency modes the computational speed can be increased while only losing little to none information (depends on how many modes that are truncated and the modal behavior of the equation). With the remaining Fourier modes κ_{max} a multiplication is done between the modes and a matrix of complex trainable parameters $\mathcal{R} \in \mathbb{C}^{\kappa_{max} \times d_u \times d_u}$ in frequency space.

$$\mathcal{F}^{-1}(\mathcal{R}_t \cdot \mathcal{F}(u_t))_{\kappa,j} = \sum_{j=1}^{d_u} \mathcal{R}_{\kappa,j}(\mathcal{F} v_t)_{\kappa,j}, \quad \kappa = 1, ..., \kappa_{max}, \quad j = 1, ..., d_u \tag{2.83}$$

Truncating $N - \kappa_{max}$ amount of Fourier modes increases computational speed of the operator since it also removes a large portion of the operations performed in the convolution operator in Fourier space. This also means that there are fewer trainable weights in the matrix $\mathcal{R}$ since the size of this matrix depends on κ, $\mathcal{R} \in \mathbb{C}^{\kappa_{max} \times d_u \times d_u}$. The FNO can make use of skip connections in a similar way as for the CNN, the difference is that within each Fourier layer the input data is copied and this copy bypasses the Fourier layer and then added to the output of the layer[9], see figure 2.10.

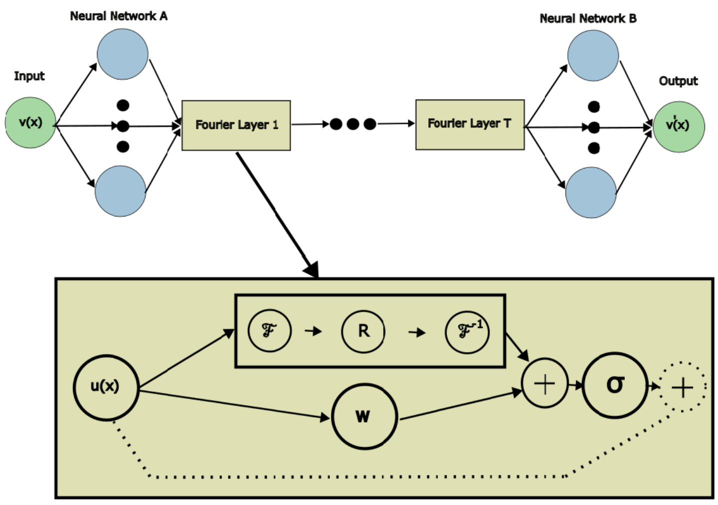

Figure 2.10: Architecture of a standard Fourier Neural Operator with width T, dashed lines representing skip connections

Chapter 3

Method

Since it does not exist a computer software which is coded to perform the objectives of this study, several scripts had to be created from scratch for these purposes. The decision was made to conduct the entire study in the programming language python using the integrated development environment pycharm. Even though there does not exist a fully developed computer software for this there are open source python packages which helps with some parts of the coding. See table 6.1 in appendix A for all packages used. All code for this study will also be made available in the github library linked to in the appendix. To speed up the generation of training data and the training of the networks a GPU cluster was employed. This resource is provided by The Swedish National Infrastructure for Computing (”SNIC”) and hosted at Chalmers University. The specific clusters used is called ”ALVIS” and is a national SNIC resource dedicated for research within Artificial Intelligence and Machine Learning. Since parts of this study are based on research[9] done by Assoc Prof Rixin Yu some parts of the code for that project could be reused in this project.

3.1 Problem setup

The problem to be solved in this thesis is as explained to train an operator to learn the time-advancement of a non-linear unstable flame front propagating through a channel, by using the KS-PDE to model the flame front subjected to diffusive-thermal instabilities. In other words, to train an operator to map the 1d flame front from one point in time to a future point after a short fixed time interval, by repeating this process yields a sequence of predicted flame fronts. For a neural network to learn this it needs to be trained using many samples of propagating flame fronts. Which means that before the training of the networks can start a database of samples has to be generated. There are several ways to solving/approximating the solution of a PDE, such as finite element and difference methods, integrating factor, Runge-Kutta and many more. Using the fourier spectral method is another way of solving the problem.

If one considers the task at hand, the problem consists of a physical system with the flame front which can be described by the PDE. The entire system can be viewed as many separate functions where the solution at each time-step corresponds to one function, then the goal is to create an operator $\mathcal{G}$ which can map one function $v(x)$ to another $v'(x')$ by approximating the real map $\hat{\mathcal{G}}$ between the functions[9].

$$\hat{\mathcal{G}} : v(x) \mapsto v'(x') \tag{3.1}$$

The input function:

$$v : \mathcal{D} \to \mathbb{R}^{d_v}; x \mapsto v(x) \tag{3.2}$$

where v is a part of the function space $\mathcal{V}(\mathcal{D}; \mathbb{R}^{d_v})$ where $\mathcal{D} \subset \mathbb{R}^d$ and the codomain $\mathbb{R}^{d_v}$.

By making the assumption that the two functions share the same domain and codomain, yields the following output function:

$$v' : \mathcal{D} \to \mathbb{R}^{d_v}; x' \mapsto v'(x') \tag{3.3}$$

To further simplify the problem the assumption was made that $\mathcal{D}$ does not vary in time and has periodic boundary conditions. The map $\hat{\mathcal{G}}$ for this problem is to be an operator mapping one function to another after a specific time step Δ_t, which means:

$$\hat{\mathcal{G}} : u(x;t) \mapsto u(x;t+\Delta_t) \tag{3.4}$$

where $v(x)$ is replaced by the PDE solution $u(x;t)$ with time parameter t and $v'(x)$ with $u(x;t+\Delta_t)$. The task of the neural network is then to find the approximated map $\mathcal{G}$ so that

$$\mathcal{G} : \mathcal{V} \times \Theta \to \mathcal{V}' \quad \text{or equivalently,} \tag{3.5}$$

$$\mathcal{G}_\theta : \mathcal{V} \to \mathcal{V}', \theta \in \Theta \tag{3.6}$$

where Θ is a finite-dimensioned space with all the trainable weights of the neural network.

3.2 Data generation

To generate the large database containing all the sequences of solutions to the KS equation some initial setup was required in terms of coding and preparation. For a case where a flame propagates in a channel of width L with x being a normalized coordinate with direction normal to the channel wall. Assuming periodic boundary conditions and that all coordinates along the flow can be described by a function $u(x,t)$. Then,

$$u_t + u_{xx} + u_{xxxx} + uu_x = 0, \quad x \in (-L/2, L/2), \quad t > 0 \tag{3.7}$$

$$u(x,t) = u(x + 2\pi L, t), \quad u_x(x,t) = u_x(x + 2\pi L, t), \quad t \geq 0 \tag{3.8}$$

$$u(x,0) = u_0(x), \quad x \in (-L/2, L/2) \tag{3.9}$$

By using the Fourier Spectral method in combination with the Runge-Kutta method, eq.3.7 could be numerically solved. But before this can be done the initial conditions $u_0(x)$ had to be generated. Since the goal is to learn to make predictions for any initial condition means that all initial conditions used are randomly generated on a mesh by two different methods. Half of the database is generated by an initial condition made to imitate white-noise. The initial condition is generated on the mesh of the channel with N points with the following formula:

$$u(x_i) = 0.1 * \alpha \tag{3.10}$$

After evaluating the initial condition and the solution after one time-step $u(x, \Delta_t)$ the change between the function was deemed too big to be learnable for a neural network, see figure.3.1. This led to that the first solution was removed for all sequences generated with the white-noise initial condition, meaning that $u(x, \Delta_t)$ was instead considered the first solution in all of these sequences.

The other half of the database was instead generated as spectral-noise[9] meaning smoother waves more similar to a sinusoidal wave than the previous white-noise. This initial condition was generated by:

$$u_0(x) = 0.031\gamma N \sum_{\kappa=2}^{8} (\beta_\kappa + 1) cos(\kappa x + 2\pi\alpha\kappa) \tag{3.11}$$

where γ is a randomly generated value between 0 and 1, α_κ and β_κ are arrays of randomly generated values also between 0 and 1. In figure 3.2 one sample of the initial condition using spectral noise can be seen and the solution after one time-step. The change between these two functions were deemed satisfactory and the sequence was not altered.

The initial condition $u_0(x)$ is already discretized on the mesh of size N which means that the solution is already of finite-freedom and can be transformed with the Fast Fourier Transform FFT. The spectral method is used to generate the sequence of solutions from the initial condition by first translating eq.3.7 into ordinary differential equations (ODE) with $N/2 + 1$ number of complex Fourier modes.

$$F_\kappa(u(x_j; t)), \quad \kappa = 0, ..., N/2 \tag{3.12}$$

Then the time-step is done by temporal integration of these ODEs by using a fourth-order

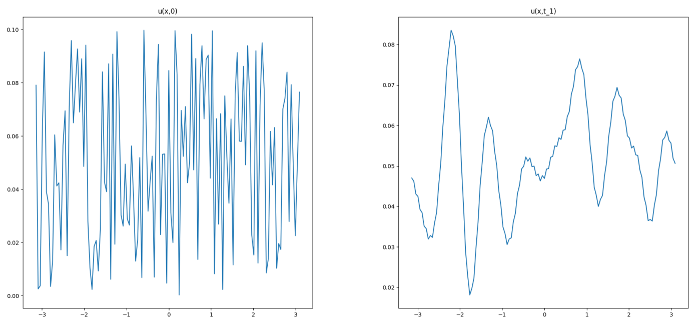

Figure 3.1: Comparison between one sample of $u_0(x)$ and $u(x, \Delta_t)$ for a white-noise initial condition

Runge-Kutta update.

The choice of time-step size (Δ_t) is another important decision to consider. Since a too large time-step would lead to too large changes between each solution and a too small step size would lead to an almost stationary system with very little changes. Both of those options are undesirable and therefore a reasonable step-size had to be found through trial and error and by comparing to similar cases. In the end a step size of $\Delta_t = 0.15$ was found to work well for this study.

The solutions generated have been verified with both a matlab script also using the spectral method in combination with a temporal integration using fourth-order Runge-Kutta and another python script using spectral method in combination with a semi-implicit third-order Runge-Kutta method.

The database generated contains different lengths of the KS-solution sequences to better cover distinct features during different time scales of the evolution of the flame front. In table 3.1 all the generated solution sequences used in this study can be found.

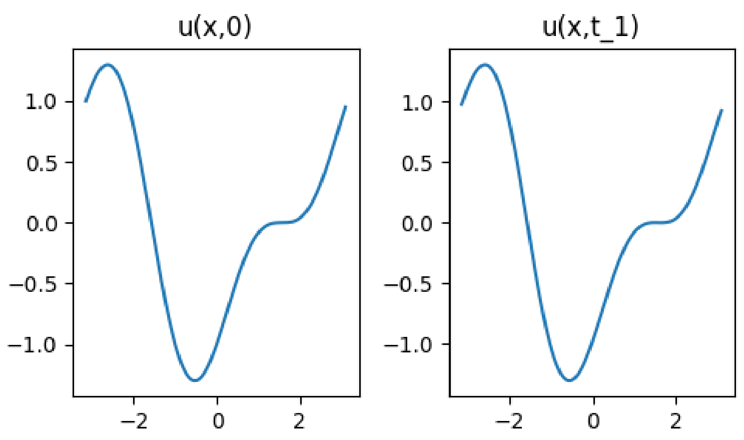

Figure 3.2: Comparison between one sample of $u_0(x)$ and $u(x, \Delta_t)$ for a spectral-noise initial condition

Amount of sequences	Length	N	L	Initial condition	Δ_t
200	501	128	2π	white-noise	0.15
200	501	128	2π	spectral-noise	0.15
100	201	128	2π	white-noise	0.15
100	201	128	2π	spectral-noise	0.15
50	2001	128	2π	white-noise	0.15
50	2001	128	2π	spectral-noise	0.15
200	501	256	2-,5-,8π	white-noise	0.15
200	501	256	2-,5-,8π	spectral-noise	0.15
100	201	256	2-,5-,8π	white-noise	0.15
100	201	256	2-,5-,8π	spectral-noise	0.15
50	2001	256	2-,5-,8π	white-noise	0.15
50	2001	256	2-,5-,8π	spectral-noise	0.15

Table 3.1: Table of all solution sequences to the KS equation

3.3 Deep-learning methods

The networks are created for the purpose of making a long sequence of predictions evolving in time starting from one single PDE solution. Two different approaches for training the neural networks for this were adopted in this study. The main difference between the methods is that one training methods forces the network to make multiple consecutive predictions before calculating the loss function and performing back propagation. These setups are called one-to-one training and one-to-many training. As the name implies one-to-one (1-to-1) setup involves training the network using a magnitude of one-to-one data pairs, meaning that the entire database of PDE solutions are divided into pairs of input function and the target output function which is when the input function have evolved one time-step into the future (as in eq.3.4). Simply put, the NN is given the input function and then the output predicted function is compared to the target solution function, the goal is to alter the parameters θ in the map G_θ to minimize the loss of the following equation.

$$\textit{1-to-1 pair:} \quad (v, \hat{G}v) \tag{3.13}$$

$$\underset{\theta\in\Theta}{min}\mathbb{E}[C((\hat{G}v),(G_\theta v))] \tag{3.14}$$

where C is the loss function. Respectively, the one-to-many (1-to-n) setup is as described to train the network using 1-to-n pairs of data. Meaning that one initial solution is paired with n amount of consecutive solutions each evolved by a time-step Δ_t from the previous solution, creating a sequence PDE solutions of the flame front propagating in time. The input to the network is only the initial solution then from the output prediction the network makes a new prediction and so on until n consecutive predictions has been done ($\hat{G}^n = \underbrace{\hat{G} \circ ... \circ \hat{G}}_{n}$). These n predictions are then compared to the n PDE solutions from the 1-to-n pair using a loss function and first then is the backpropagation initialized.

$$1-to-npair: \qquad (v, \hat{G}^1 v, ..., \hat{G}^n v) \tag{3.15}$$

$$\underset{\theta\in\Theta}{min}\mathbb{E}[C((\hat{G}^1 v, ..., \hat{G}^n v),(G_\theta^1 v, ..., G_\theta^n v))] \tag{3.16}$$

where n is a natural number.

The **FNO** investigated in this study is the same as described earlier and almost the same as proposed by Zhongyi Li, et al in the paper "Fourier Neural Operator for Parametric Partial Differential Equations" [7]. Meaning that the FNO tested consisted of four Fourier layers using the convolution operator in Fourier space as one of the two branches in the Fourier layer. The network also uses skip connections within each layer. The FNN at the beginning of the network starts by lifting the initial input function $v(x)$ to 20 channels. The other branch within the Fourier layer consists of a single layer of a standard convolution operation. The stride of the kernel in this convolution operation is set to one with zero padding which means that the dimensions will not be altered in this operation. After each Fourier layer the RELU activation function is applied to the output solution and the FNN at the end brings the solution back to one channel through a shallow fully connected architecture.

The standard **CNN** studied is of an autoencoder structure and the encoder and decoder features seven layers respectively. The shrinking of the dimensions in the encoder is performed using maxpooling layers and the decoder uses upsampling layers to increase the dimensions. Each layer uses the RELU as activation function after the convolution operation, the structure of the network can be seen in figure 3.3 where the dimensions and channels of the data in each layer is also given. Note that the figure is illustrated in a way that makes it seem like the data is two-dimensional which is not true for this study. The reason for why it was illustrated this way is because it makes the layers easier to visualize. The model also features skip connection on all layers, meaning that the output from each layer in the encoder is stored in a list and then added to its corresponding layer in the decoder as extra channels.

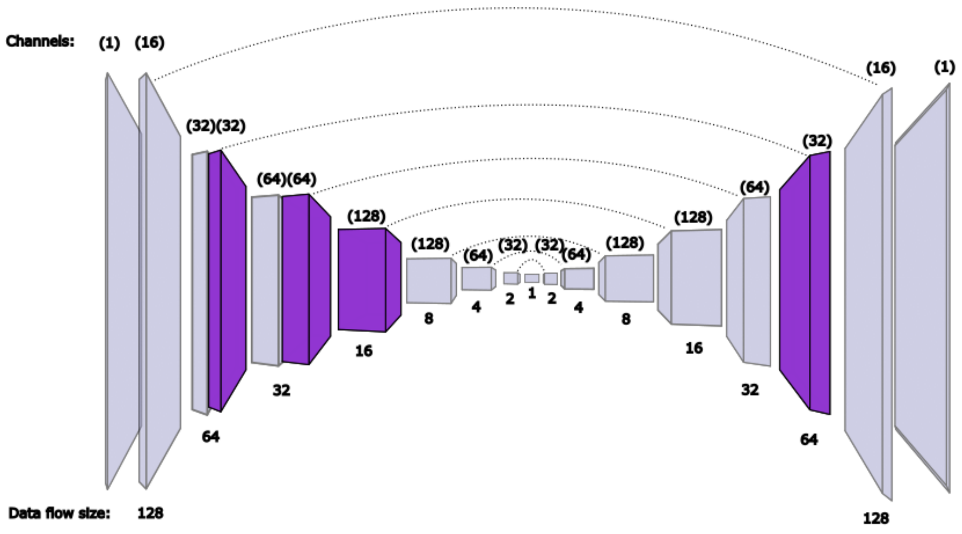

Figure 3.3: Architecture of a Parametric Fourier Neural Operator with width T, dashed lines representing skip connections

3.3.1 Modifications for varying parameters

The parametric version of the FNO is the same in every way as the standard FNO except for an extra multiplication in all the Fourier layers convolution operator in Fourier space. Consider the figure 3.4, the entire FNO now have an extra dimension on the input, namely the channel width L which is a parameter of the function $v(x)$. Since, if given the same initial condition the solution function will not be the same if the initial condition is placed on different meshes based on different channel widths. For example,

$$x_{1_j} \neq x_{2_j}, x_1 \in (-2\pi/2, -2\pi/2), x_2 \in (-8\pi/2, -8\pi/2), j = 1, ..., N \tag{3.17}$$

When creating the architecture of these networks one of the ideas to improve the performance were to keep the translational symmetries of the input to output function throughout the networks operations. Both the Fourier Neural Network and the Convolutional Neural Network have been proven to be able to use translational symmetries to improve results

[29][30] and are therefore already employed in the versions described earlier. Why these network methods have translational symmetry will not be dwelt upon in this study, but can instead be read about in section 4.2 and 5.1 in the paper ”Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges” by Michael M. Bronstein et al[30]. The proposed methods below for the modified networks are believed to keep the translational symmetry from the standard versions of the methods. Since the modifications only adds a few scalars that affect the data on a few instances in the networks, this will be further explained below.

This modified version of the FNO first splits the input to its two components, the KS solution function $v(x)$ and the channel width L that corresponds to the considered solution $v(x)$ and which is a single value. This L is modified by a linear transformation $M : \mathcal{R}^v \to \mathcal{R}^L$ to output κ_{max} amount of values.

$$A_\kappa = M(L), \quad \kappa = 1, ..., \kappa_{max} \tag{3.18}$$

while the function $v(x)$ is treated in the same way as described in chapter 2.4 about the FNO until the convolution operator in the Fourier layer. This operation is modified so that each value in in the array A scales its respective Fourier mode as a multiplication of scalar on a matrix in Fourier space. Exactly the same modification is done on all of the Fourier layers using the same array A.

$$\mathcal{F}^{-1}(\mathcal{R}_t \cdot \mathcal{F}(u_t))_{\kappa,j} = \sum_{j=1}^{d_v} R_{\kappa,j}(\mathcal{F}v_t)_{k,j}M(L)_\kappa, \quad \kappa = 1, ..., \kappa_{max}, \quad j = 1, ..., d_u \tag{3.19}$$

The idea behind the Parametric CNN (modified version of CNN) is similar to the modified FNO described above. But the idea could not be implemented in exactly the same way due to the different structures and operations in the networks. Instead the shallow FNN outputs one value for each of the extra convolution layers added to the structure, see figure 3.5. This value is used to scale the output from the extra layer before being merged back into the encoder. The version where the extra convolution is placed in series (figure 3.5) with the original convolution layer can mathematically be explained as,

$$u(x) = \sigma(W_l v(x) + (W_l'(W_l v(x)))P_l(L)) \tag{3.20}$$

The version where the extra convolution is placed in parallel (figure 3.6) with the original convolution layer can mathematically be explained as,

$$u(x) = \sigma(W_l v(x) + (W_l' v(x))P_l(L)) \tag{3.21}$$

If the convolution layer it is placed in parallel with contains a max pooling layer then this layer must do so as well to retain the same dimension size as the main branch.

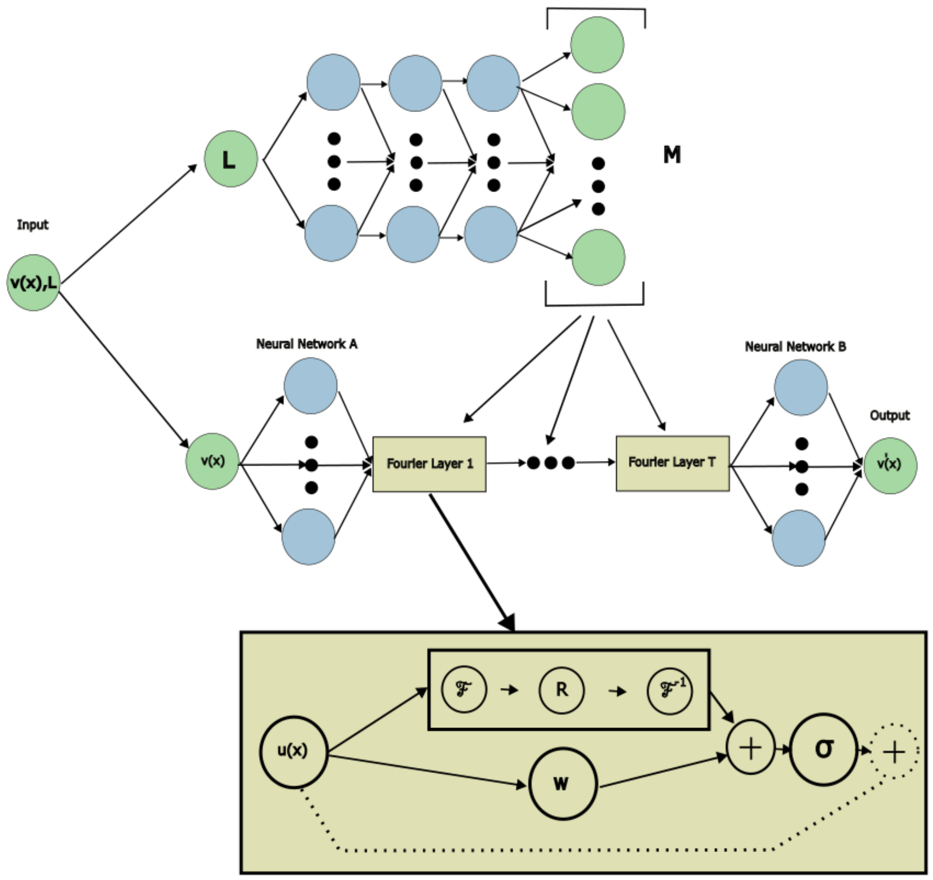

Figure 3.4: Architecture of a Parametric Fourier Neural Operator with width T, dashed lines representing skip connections

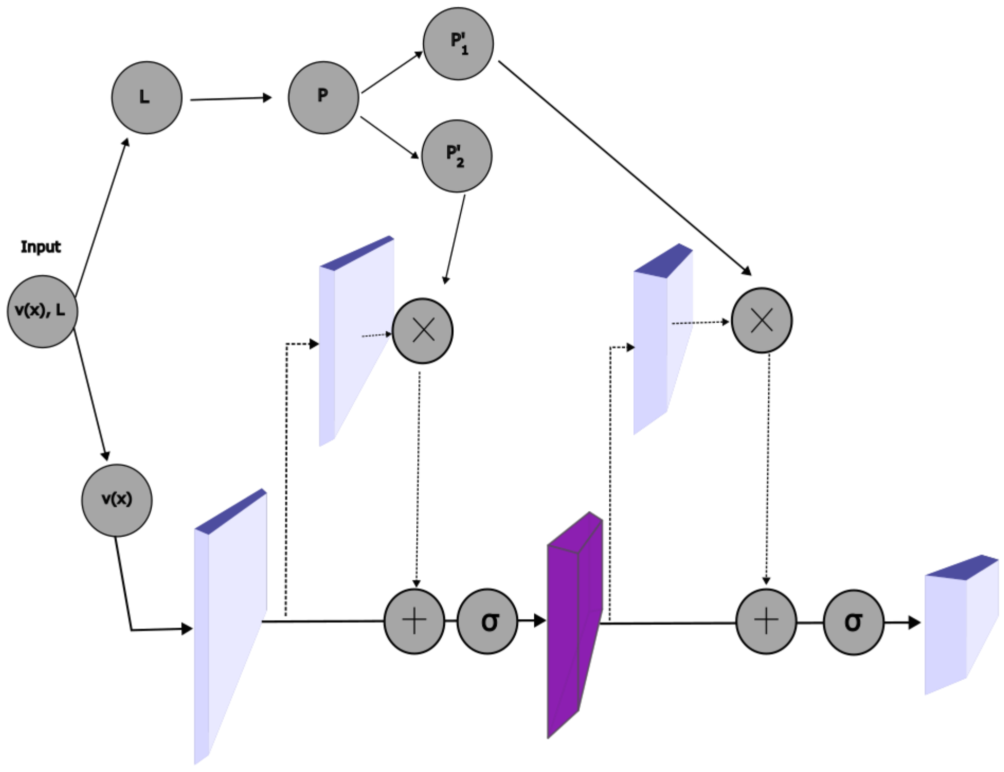

Figure 3.5: Architecture of modified layers of the Parametric CNN, with a series structure

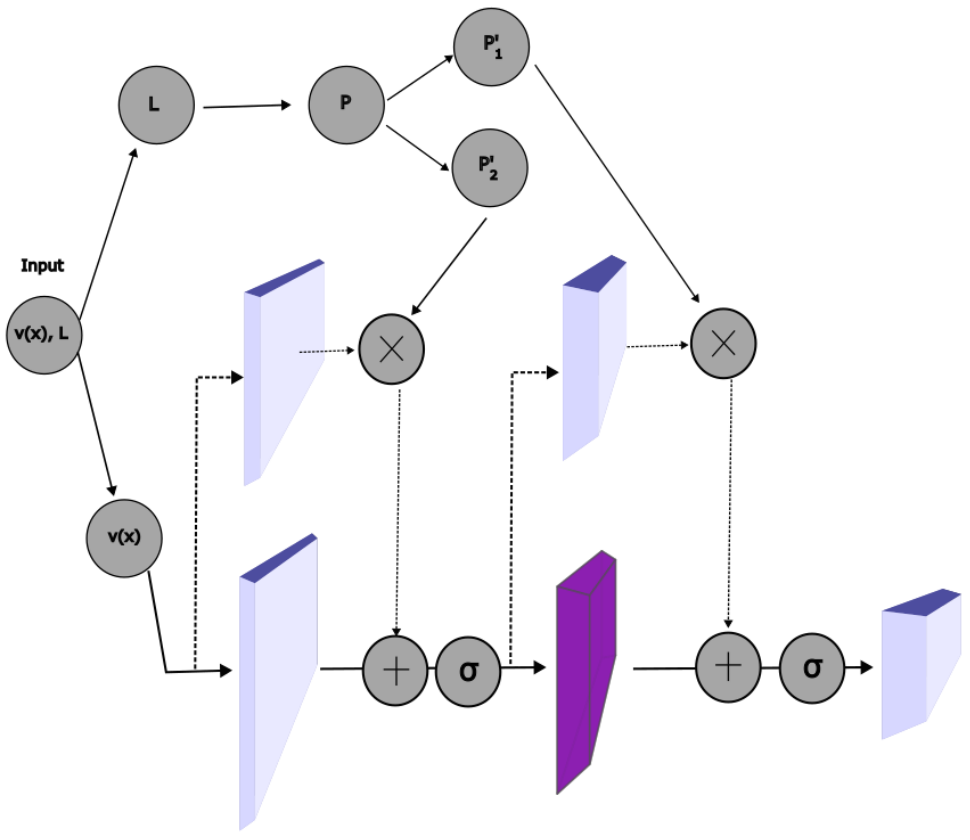

Figure 3.6: Architecture of modified layers of the Parametric CNN, with a parallel structure

Chapter 4

Numerical Results

The entire chapter is purely dedicated to presenting the results of the trained networks. All networks are benchmarked through visual representation in figures comparing the predicted solution to the reference case. Another indicator of the result of the training is the loss calculated from the loss function in the last batch and epoch of the training process. Different neural network can be compared to each other when trained under the same conditions. The amount of learnable weights and training time per epoch are also considered, since they give an indication of the complexity and effectiveness of the model. To simplify the denotation of the networks the training methods will be included when referring to a network, meaning that for a standard CNN and FNO trained with 1-to-20 pairs will henceforth be denoted to as CNN_{20} and FNO_{20}. Three different types of plots are used to benchmark the networks, the first type shows long-term predictions for a network compared to the reference (real) solution. The second type is similar to the first type but instead shows short-term predictions. The third type corresponds to contour plots of the spatial derivative of the flame front slope ($\partial_x u(x,t)$) evolving in time.

4.1 Benchmarking the FNO and CNN

All networks presented under this sub-chapter are the standard versions of the models, meaning they are precisely as presented in the previous chapter. Furthermore, the solutions are defined on a discretized equispaced mesh with length $L = 2\pi$ and of size $N = 128$. The amount of truncated Fourier modes κ_{max} for the FNO networks equals to $N/2 - \kappa_{max} = 32$

Network type	1-to-n	L	Trainable parameters	Time per epoch[s]	Loss
FNO	1-to-1	2π	28877	2.7	0.00173
FNO	1-to-10	2π	28877	4.3	0.0059
FNO	1-to-20	2π	28877	4.8	0.0124
CNN	1-to-1	2π	339733	12.5	0.0037
CNN	1-to-10	2π	339733	21.9	0.0265
CNN	1-to-20	2π	339733	26.7	0.0224

Table 4.1: Table of trained Networks using the standard architectures, with respective parameters and training results

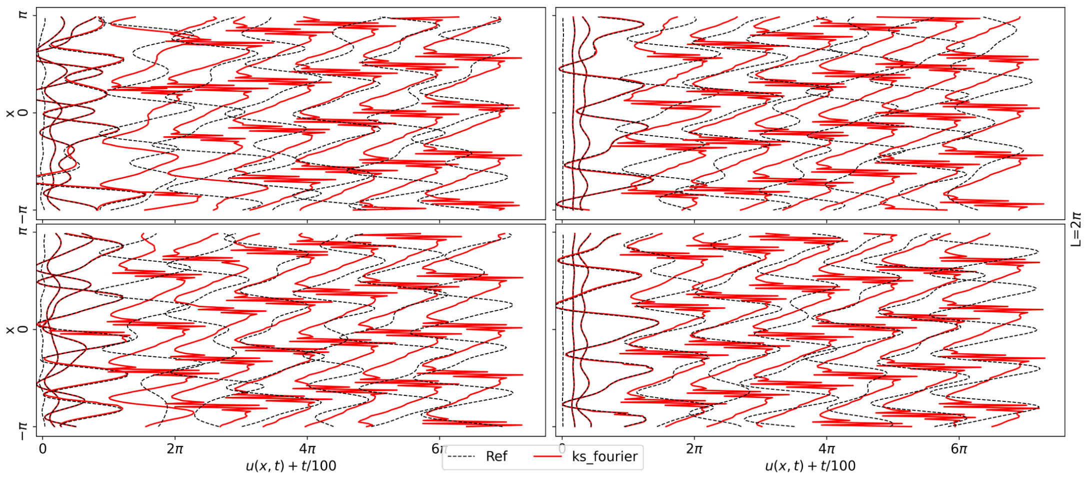

Figure 4.1: Four samples of recurrent long-term predictions $u(x,t)$ made by Network FNO_1 (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$ (the two initial conditions to the left generated as spectral noise and the other two as white-noise). The flame fronts visualized in the figure are samples taken from these time evolution sequences at t = Δ_t[0, 50, 100, 200, 500, 800, 1100, 1400, 1700, 2000] with Δ_t = 0.15. Each consecutive flame front is shifted with a value of (t/100) to the right in the figure

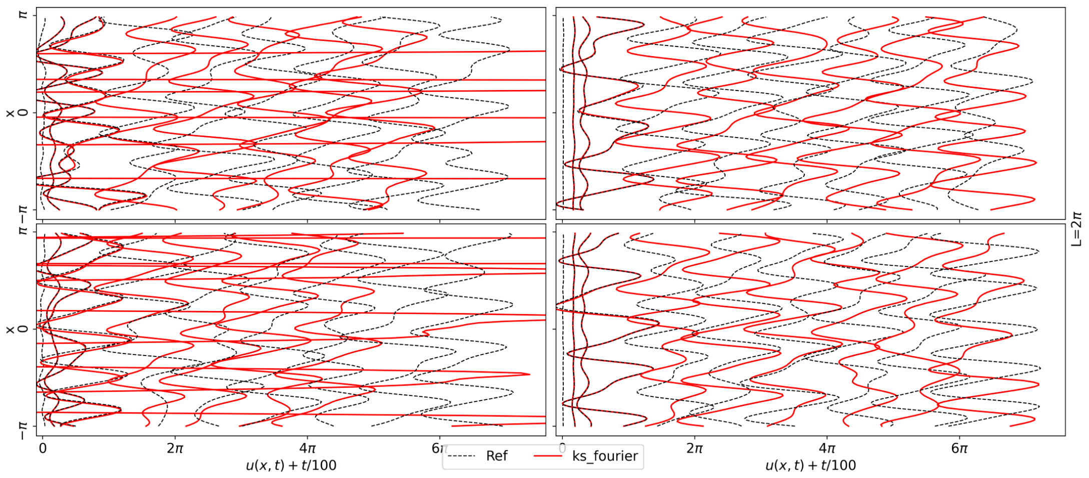

Figure 4.2: Four samples of recurrent long-term predictions $u(x,t)$ made by Network FNO_{10} (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details

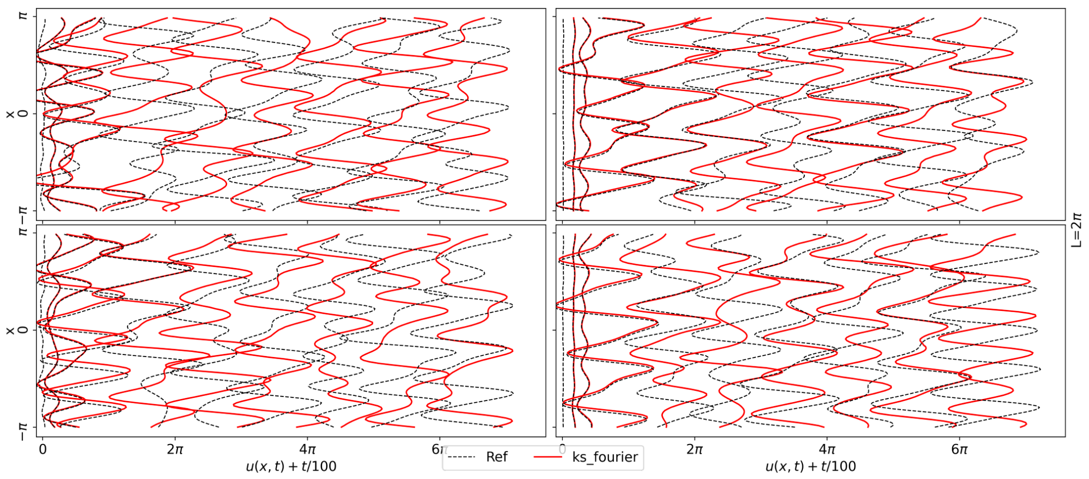

Figure 4.3: Four samples of recurrent long-term predictions $u(x,t)$ made by Network FNO_{20} (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details

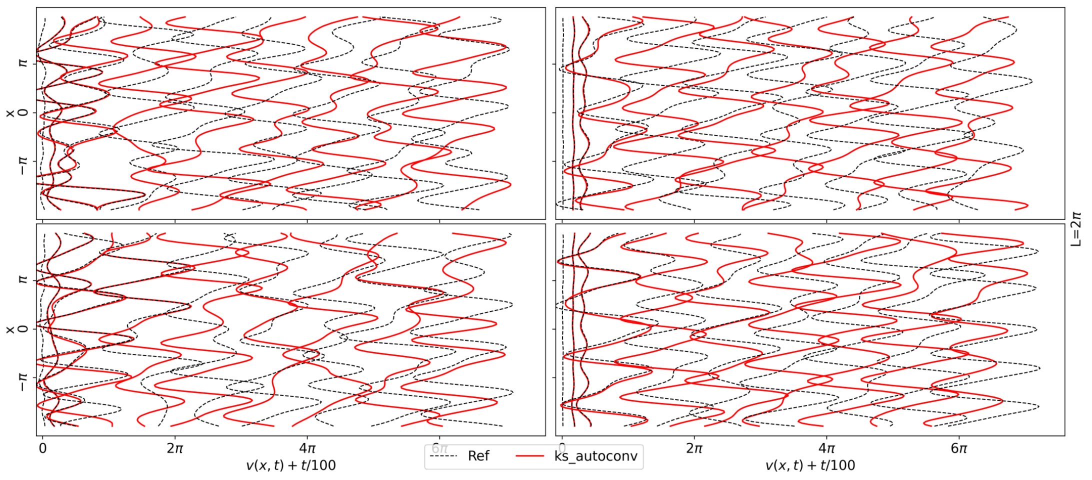

Figure 4.4: Four samples of recurrent long-term predictions $u(x,t)$ made by Network CNN_1 (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details

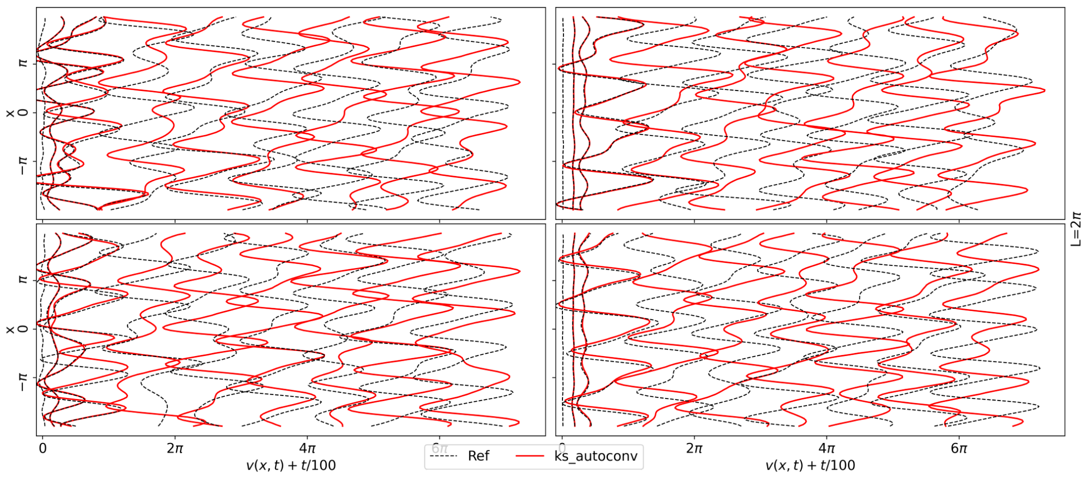

Figure 4.5: Four samples of recurrent long-term predictions $u(x,t)$ made by Network CNN_{10} (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details

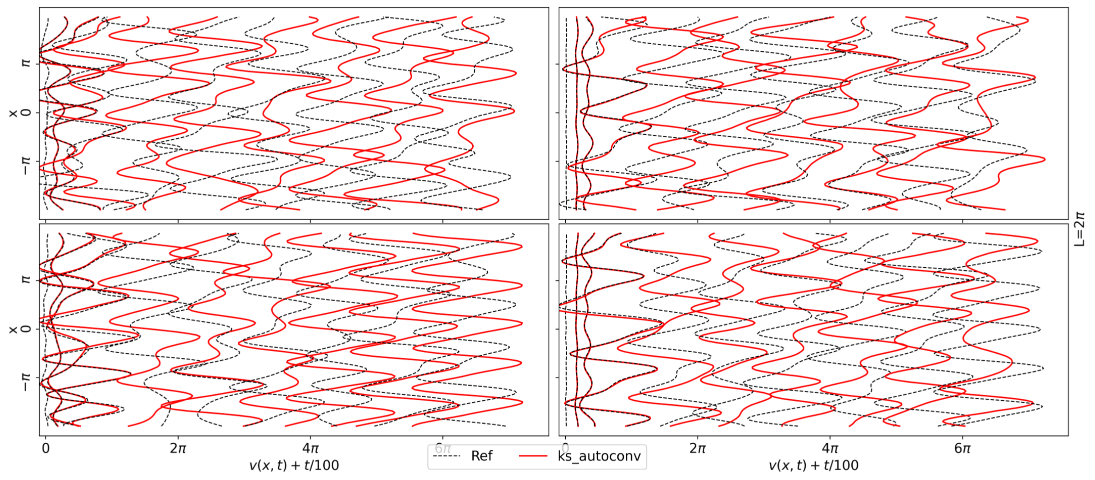

Figure 4.6: Four samples of recurrent long-term predictions $u(x,t)$ made by Network CNN_{20} (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details

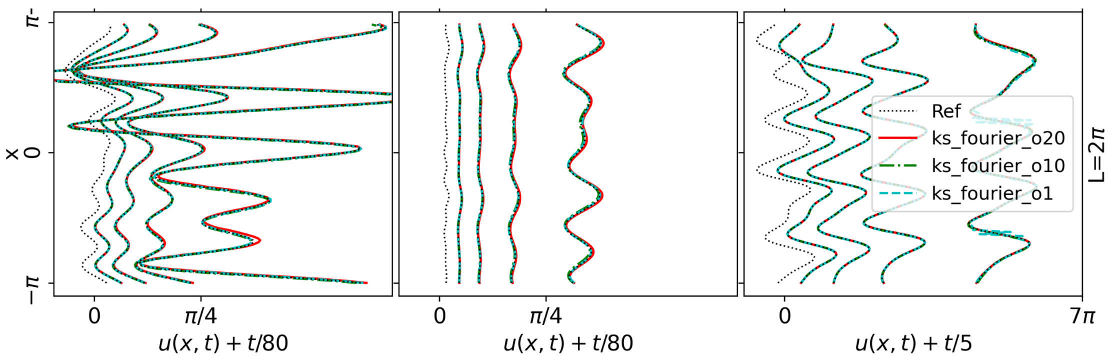

Figure 4.7: Three samples of three recurrent short-term predictions $u(x,t)$ made by the Networks $FNO_1, FNO_{10}, FNO_{20}$ (see plot legend for line colours/style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from two randomized initial conditions $u_0(x,0)$ (spectral-noise to the left and white-noise in the center) and the last one starts from the end of an long-term solution. The flame fronts visualized in the figure are samples taken from these time evolution sequences at t = Δ_t[0, 8, 20, 40, 80] with Δ_t = 0.15. Each consecutive flame front is shifted with the value (specified below each plot) to the right in the figure

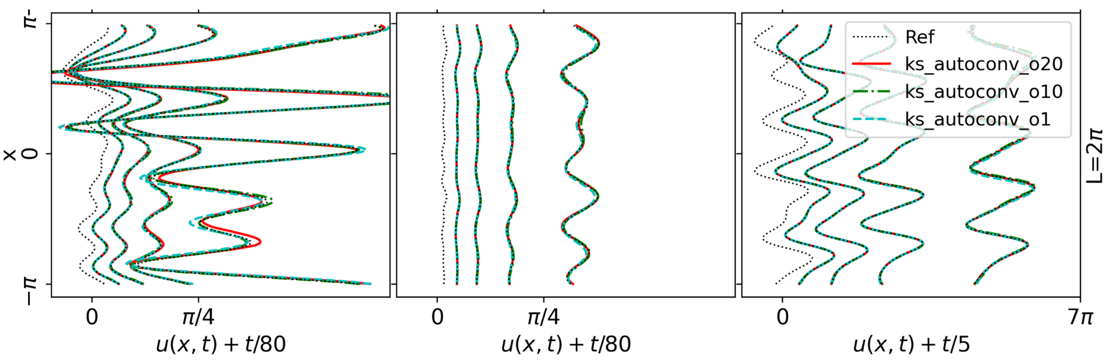

Figure 4.8: Three samples of three recurrent short-term predictions $u(x,t)$ made by the Networks $CNN_1, CNN_{10}, CNN_{20}$ (see plot legend for line colours/style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

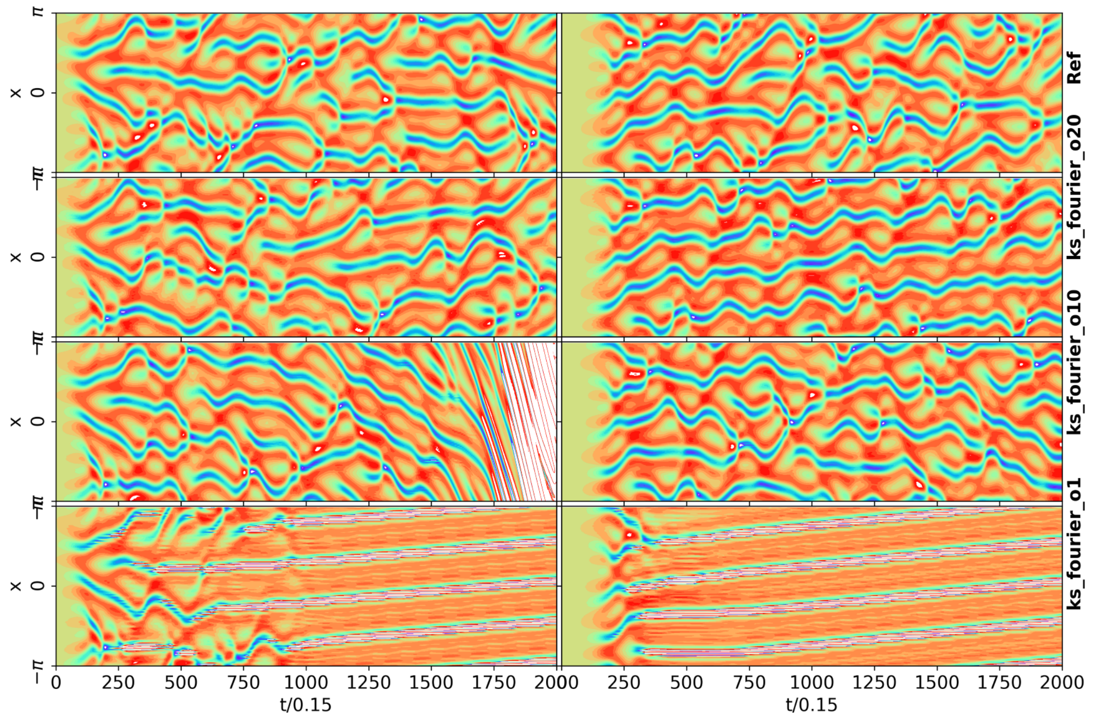

Figure 4.9: Four sets of flame front slope derivatives $(\partial_x u(x,t))$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other three corresponds to predictions by $FNO_{20}, FNO_{10}, FNO1$ from top to bottom. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

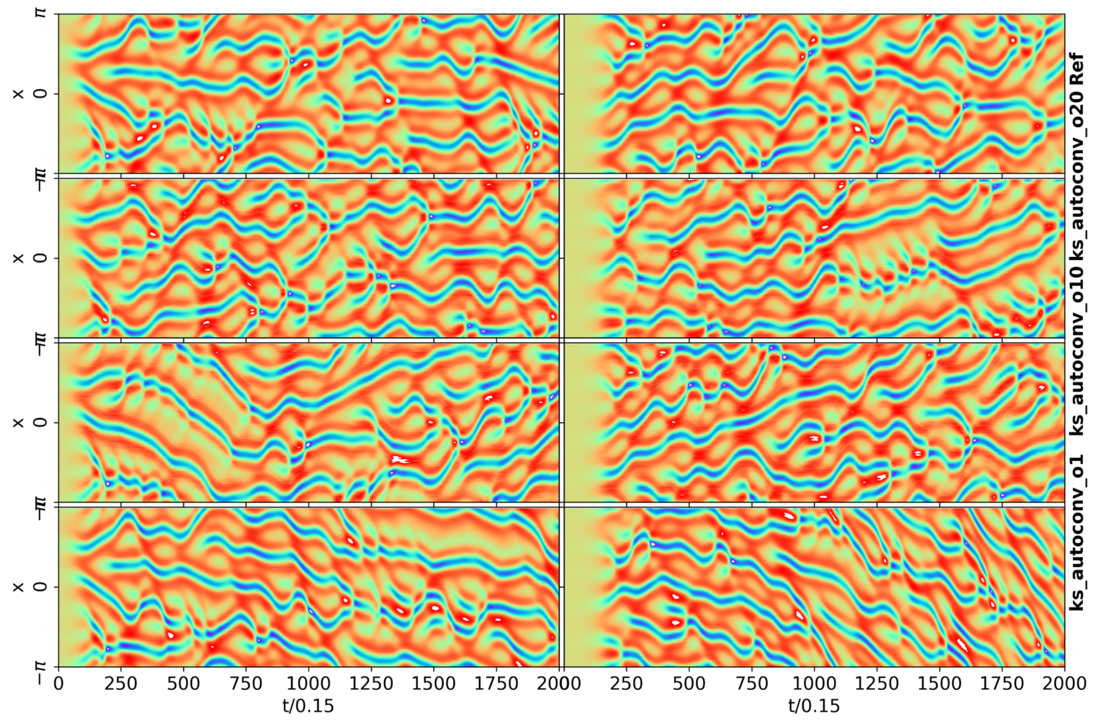

Figure 4.10: Four sets of flame front slope derivatives $(\partial_x u(x,t))$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other three corresponds to predictions by $CNN_{20}, CNN_{10}, CNN1$ from top to bottom. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

4.2 Truncation of Fourier modes in the FNO

As explained in chapter 2.4 regarding the FNO, there is a controllable parameter $M = N/2 - \kappa_{max}$ which determines how many of the Fourier modes that are to be truncated κ_{max} in the convolution operator in Fourier space of the FNO. This sub-chapter presents how this truncation affects the networks ability to make accurate prediction. Since each Network is trained with different amounts of Fourier modes truncated an extra parameter will be added to the denotation of the network $FNO_{n,M}$, where n still refers to the training method (1-to-n). The number of points on the mesh or the "resolution" equals to $N = 256$.

Network type	1-to-n	L	κ_{max}	Trainable parameters	Time per epoch[s]	Loss
FNO	1-to-1	8π	64	54477	5.4	0.0016
FNO	1-to-1	8π	32	28877	4.3	0.0035
FNO	1-to-1	8π	16	16077	3.6	0.0169
FNO	1-to-20	8π	128	105677	11.3	0.0138
FNO	1-to-20	8π	64	54477	8.7	0.0108
FNO	1-to-20	8π	32	28877	7.6	0.0309

Table 4.2: Table of FNOs with different amounts of truncated modes

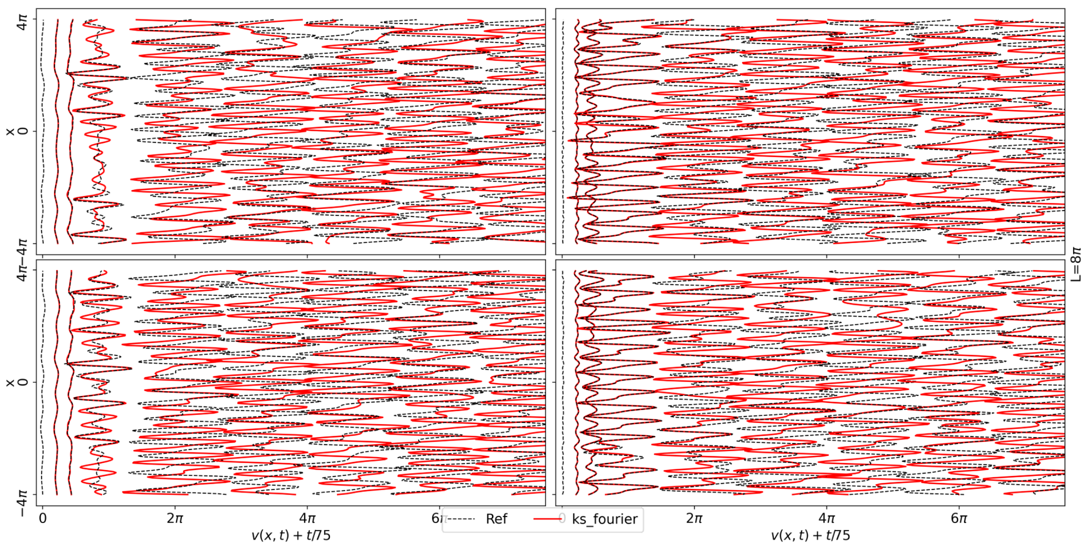

Figure 4.11: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{1,64}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details. Note difference in time shift of each sample (t/75)

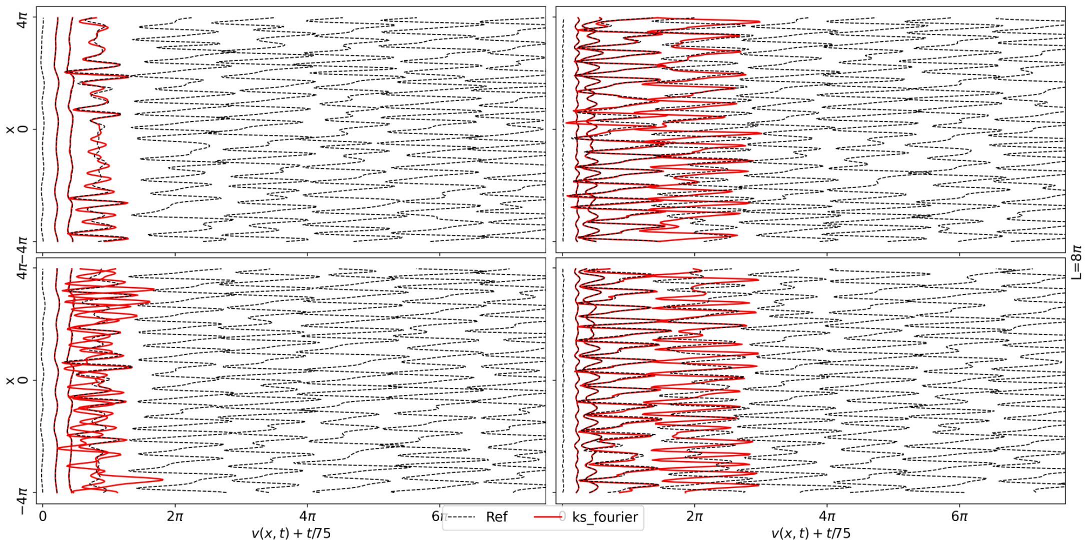

Figure 4.12: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{1,32}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

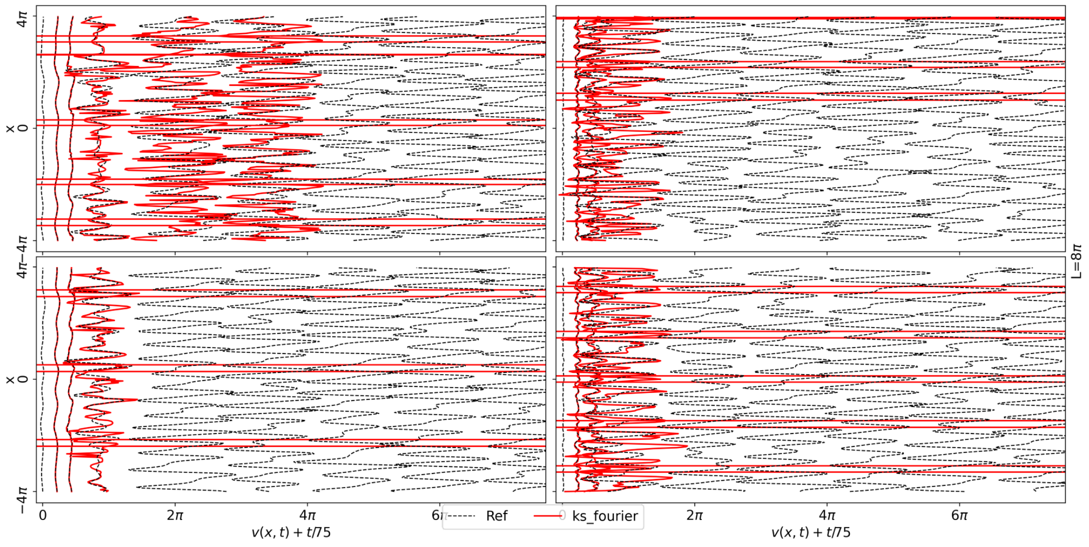

Figure 4.13: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{1,16}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

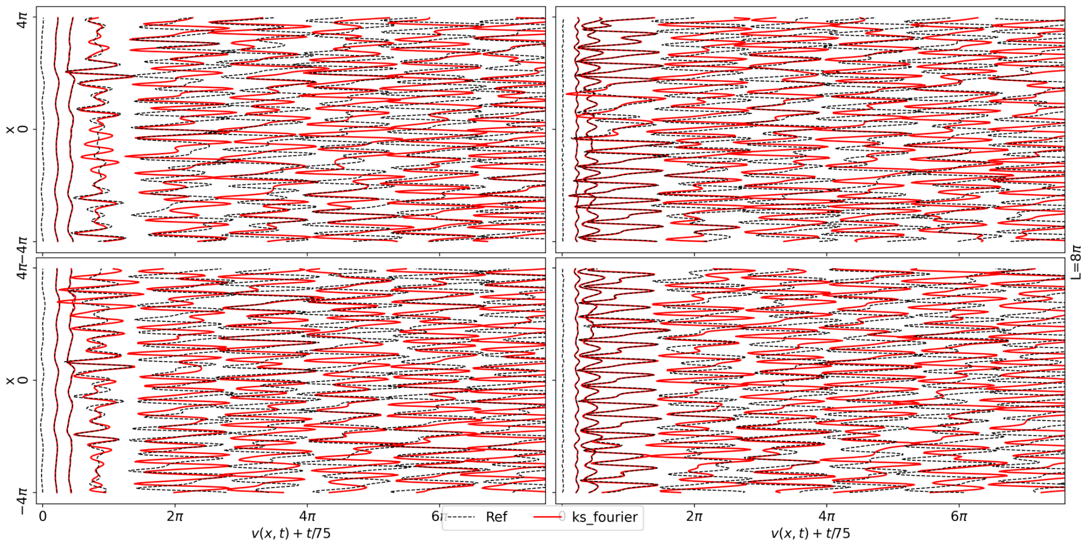

Figure 4.14: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{20,128}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

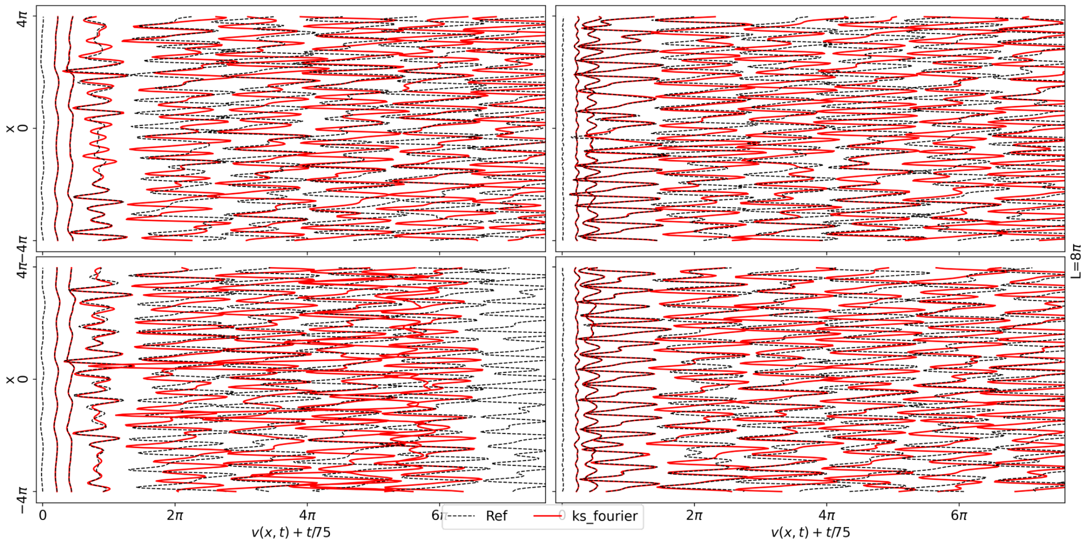

Figure 4.15: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{20,64}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

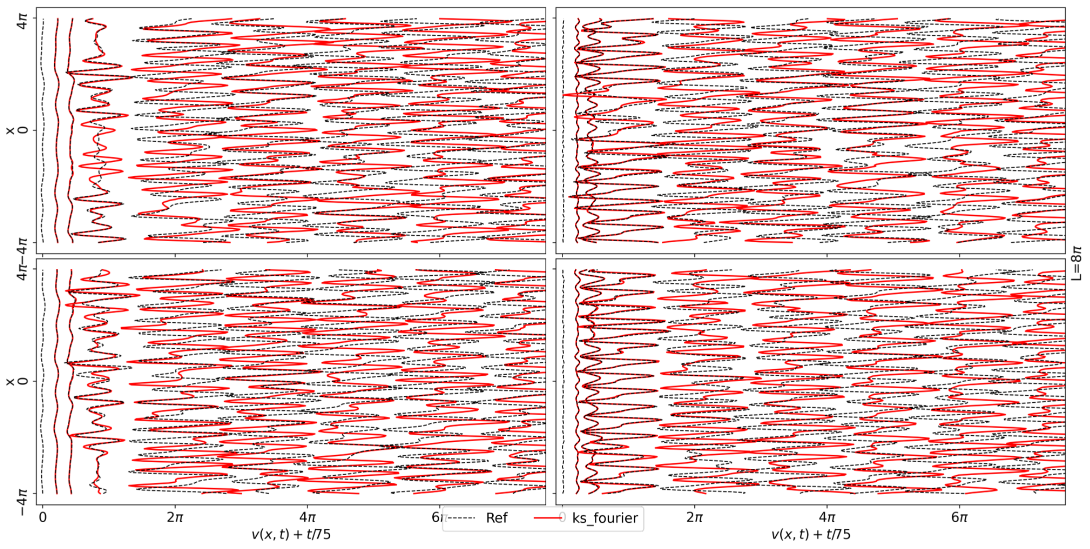

Figure 4.16: Four samples of recurrent long-term predictions $u(x,t)$ made by Network $FNO_{20,32}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

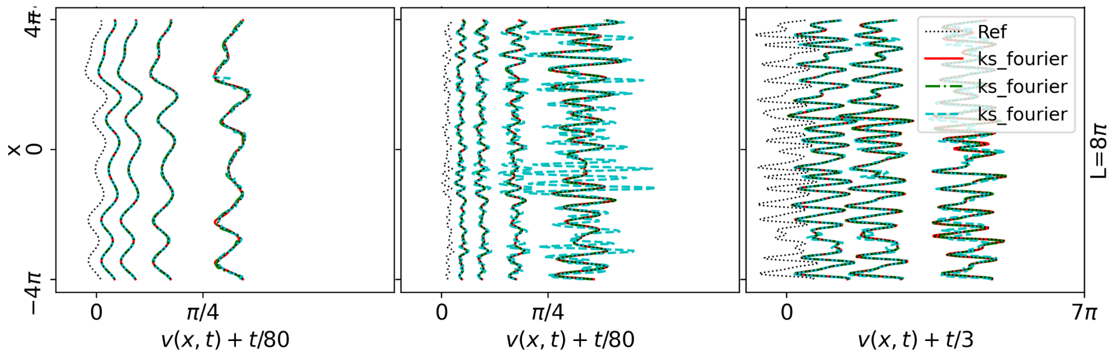

Figure 4.17: Three samples of three recurrent short-term predictions $u(x,t)$ made by the Networks $FNO_{1,64}$, $FNO_{1,32}$, $FNO_{1,16}$ (teal dashed line represents $FNO_{1,16}$, green dot-dashed $FNO_{1,32}$ and red $FNO_{1,64}$) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

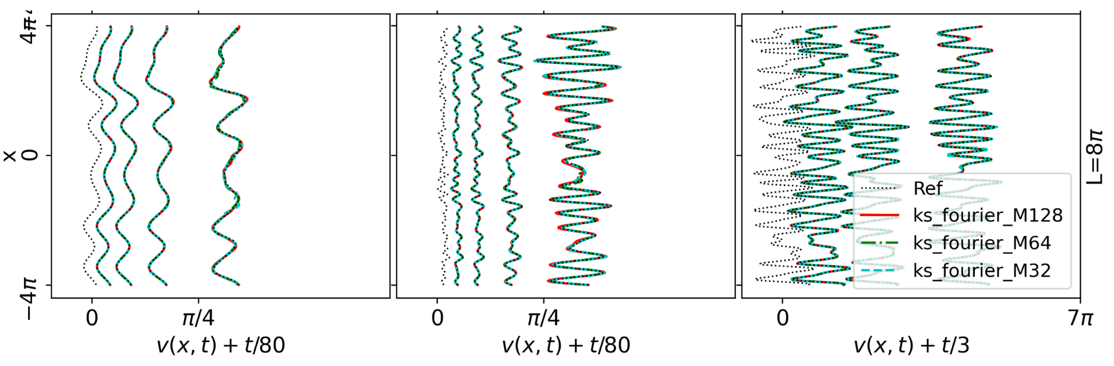

Figure 4.18: Three samples of three recurrent short-term predictions $u(x, t)$ made by the Networks $FNO_{20,128}, FNO_{20,64}, FNO_{20,32}$ (see plot legend for line colours/line-style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

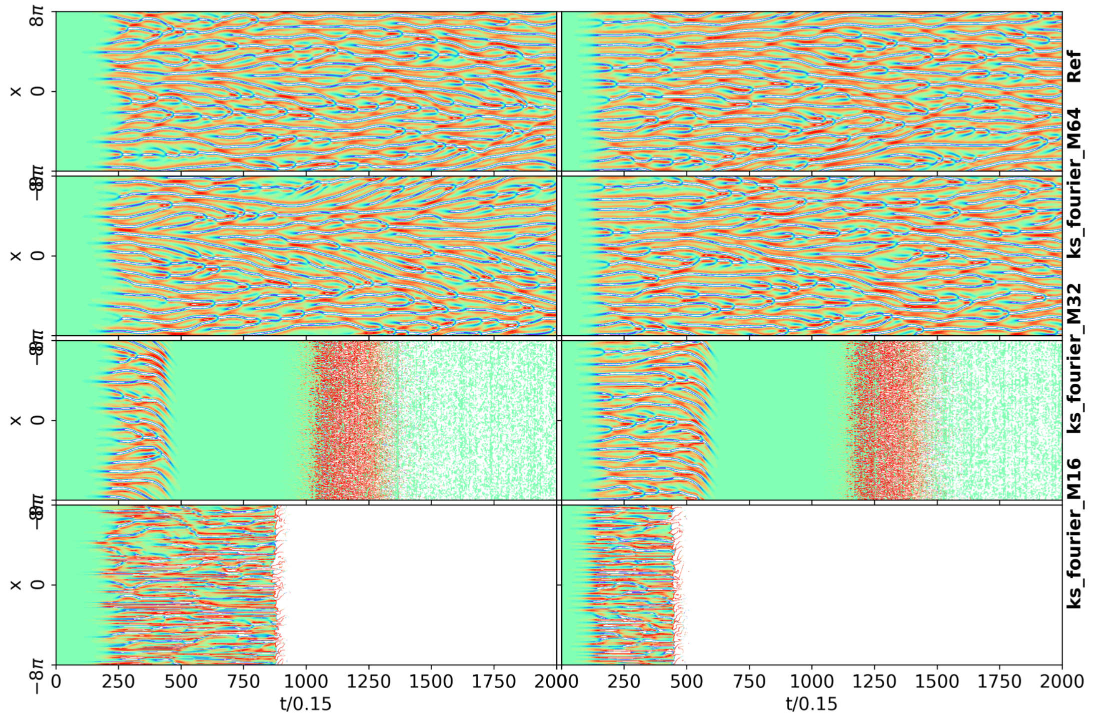

Figure 4.19: Four sets of flame front slope derivatives $(\partial_x u(x, t))$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other three corresponds to predictions by $FNO_{1,64}, FNO_{1,32}, FNO1, 16$ from top to bottom. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t, 0)$ (spectral-noise to the left and white-noise to the right)

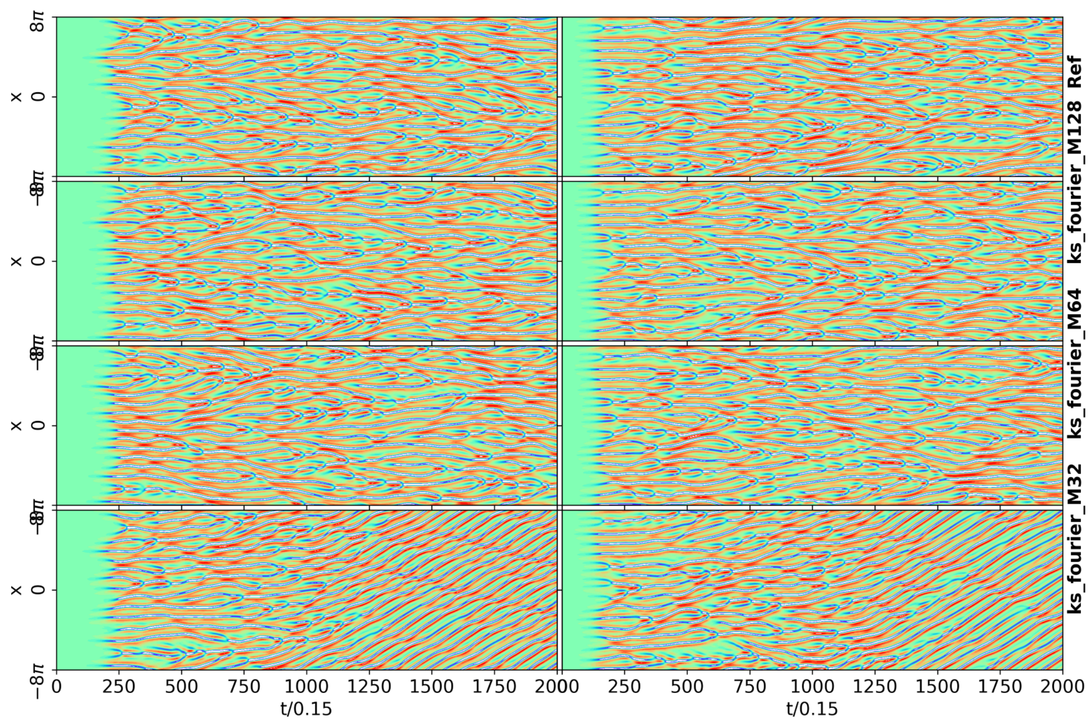

Figure 4.20: Four sets of flame front slope derivatives ($\partial_x u(x,t)$) visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other three corresponds to predictions by $FNO_{20,128}, FNO_{20,64}, FNO20, 32$ from top to bottom. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

4.3 Parametric Neural Networks

Only two of the Parametric NNs trained are included in this report, the best one from the FNO and CNN. Both networks were trained for both 1-to-1 and 1-to-10 pairs separately. The Parametric CNN for which the extra layers were placed in series are not included in the report due to the superior performance when the extra layer is placed in parallel. Each of the networks were trained with a varying channel width of $2-, 5-$ and 8π simultaneously with a constant mesh size of $N = 256$. The extra layers are only placed on the two first layers of the encoder.

Network type	1-to-n	Type	Trainable weights	Time/epoch[s]	Loss
Parametric FNO (P-FNO)	1-to-1	None	55311	12.4	0.0388
Parametric FNO (P-FNO)	1-to-10	None	54477	12.4	0.0361
Parametric CNN (P-CNN)	1-to-1	Parallel	347736	19.5	0.0072
Parametric CNN (P-CNN)	1-to-10	Parallel	347736	30.1	0.0254

Table 4.3: Table of Parametric Neural Networks

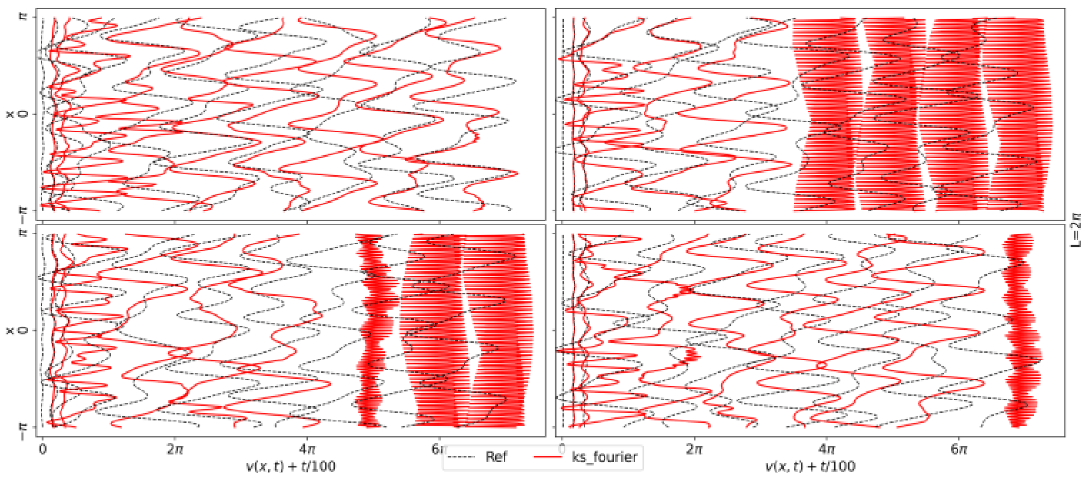

Figure 4.21: Four samples of recurrent long-term predictions $u(x, t)$ with channel width $L = 2\pi$ made by Network $P - FNO_{10}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x, 0)$. See fig.4.1 for more details.

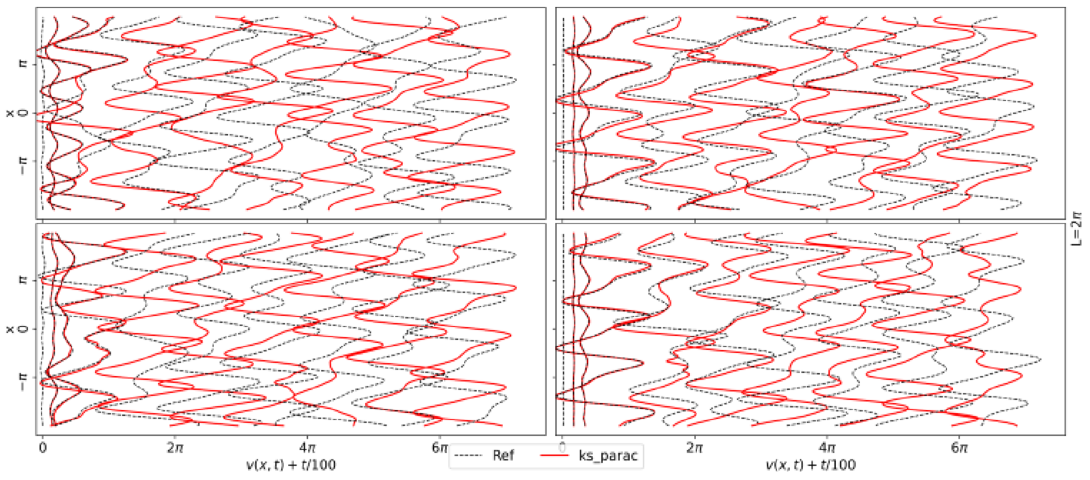

Figure 4.22: Four samples of recurrent long-term predictions $u(x,t)$ with channel width $L = 2\pi$ made by Network $P-CNN_1$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

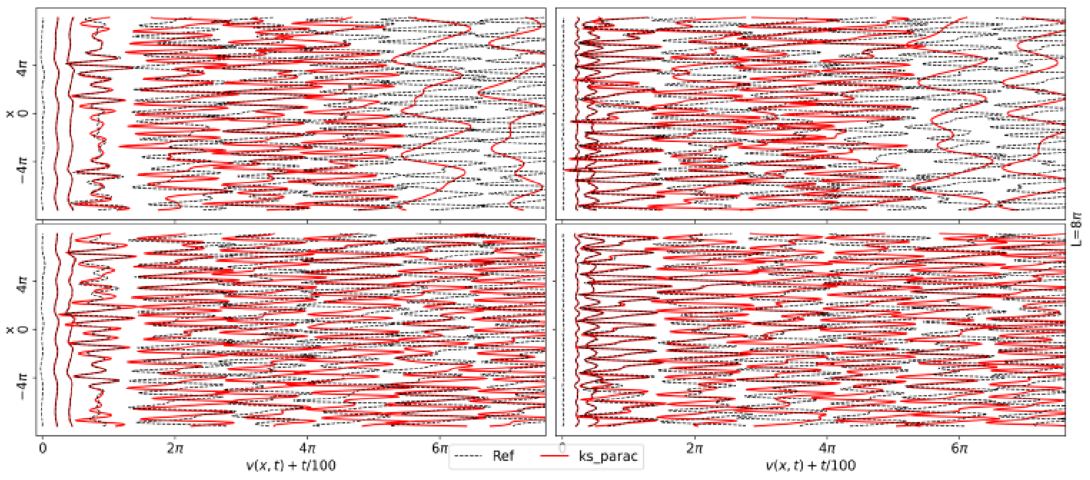

Figure 4.23: Four samples of recurrent long-term predictions $u(x,t)$ with channel width $L = 8\pi$ made by Network $P-CNN_1$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

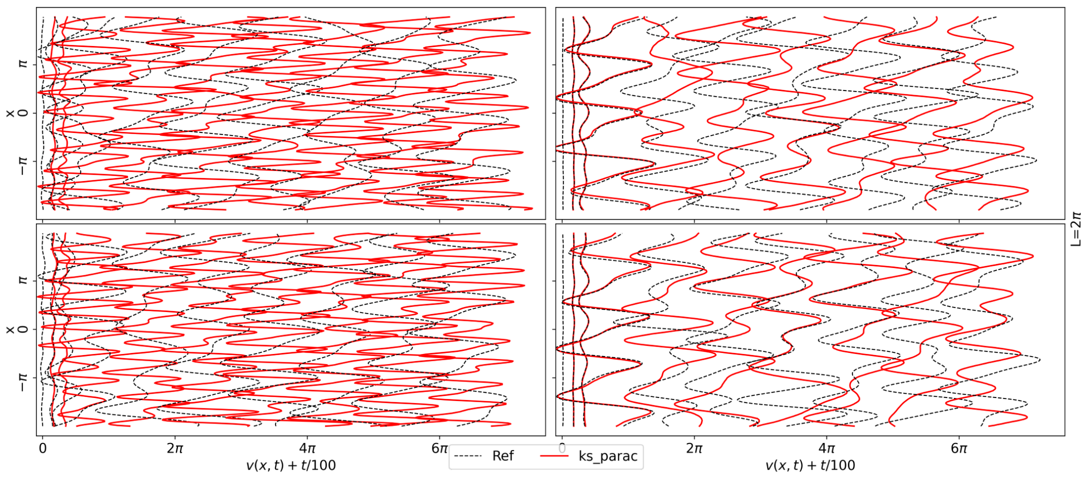

Figure 4.24: Four samples of recurrent long-term predictions $u(x,t)$ with channel width $L = 2\pi$ made by Network $P - CNN_{10}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

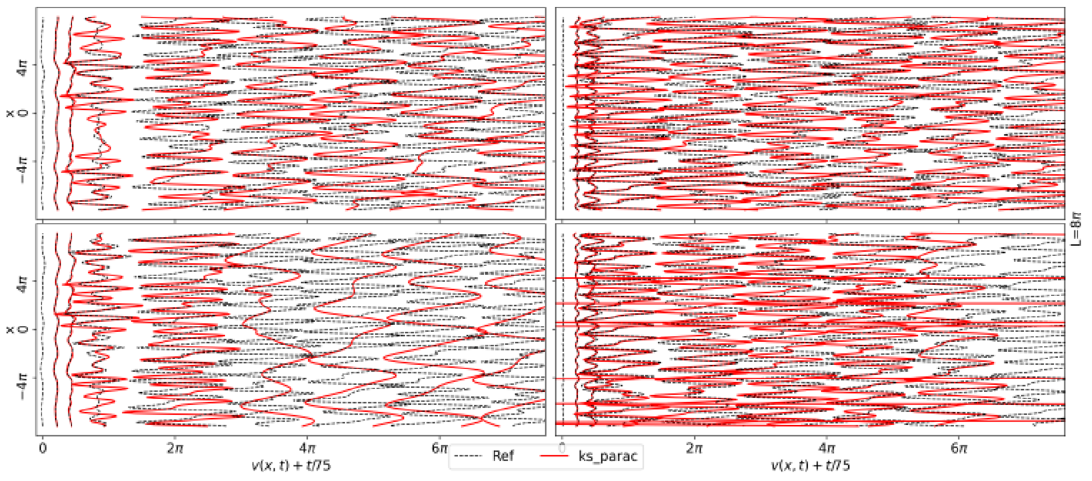

Figure 4.25: Four samples of recurrent long-term predictions $u(x,t)$ with channel width $L = 8\pi$ made by Network $P - CNN_{10}$ (red lines) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation when starting from four randomized initial conditions $u_0(x,0)$. See fig.4.1 for more details.

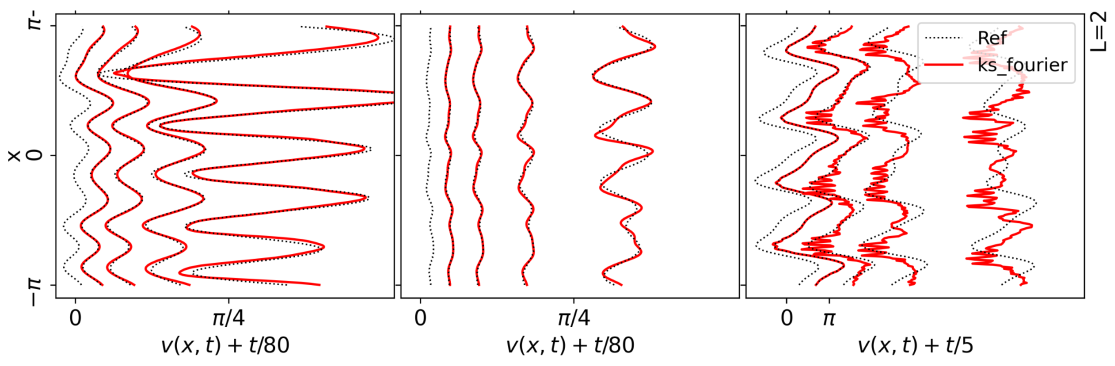

Figure 4.26: Three recurrent short-term predictions $u(x, t)$ when $L = 2\pi$ made by the Network $P - FNO_1$ (see plot legend for line colours/line-style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

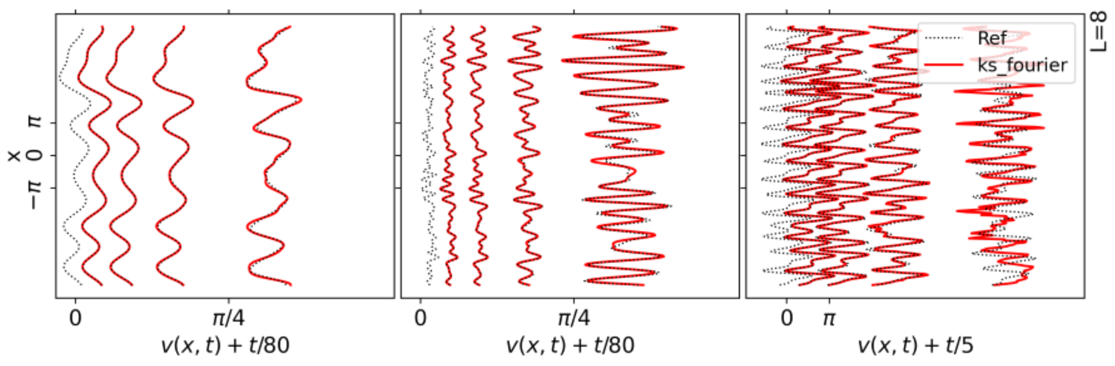

Figure 4.27: Three recurrent short-term predictions $u(x, t)$ when $L = 8\pi$ made by the Network $P - FNO_1$ (see plot legend for line colours/line-style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

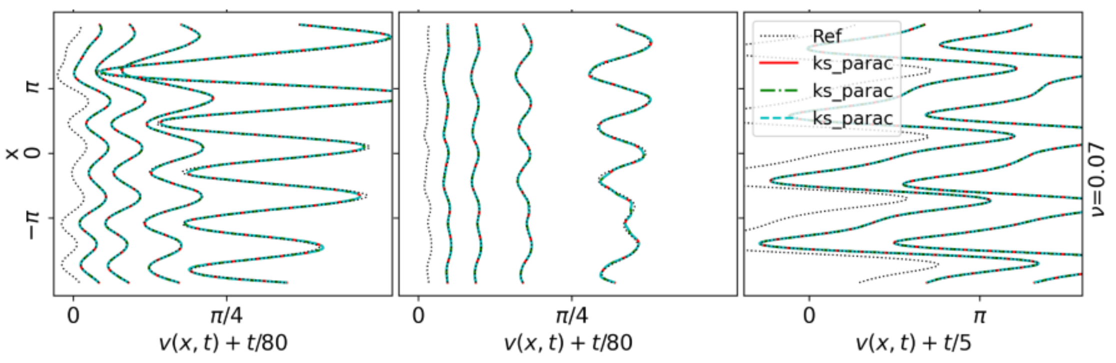

Figure 4.28: Three recurrent short-term predictions $u(x,t)$ when $L = 2\pi$ made by the Network $P - CNN_1$ (see plot legend for line colours/line-style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

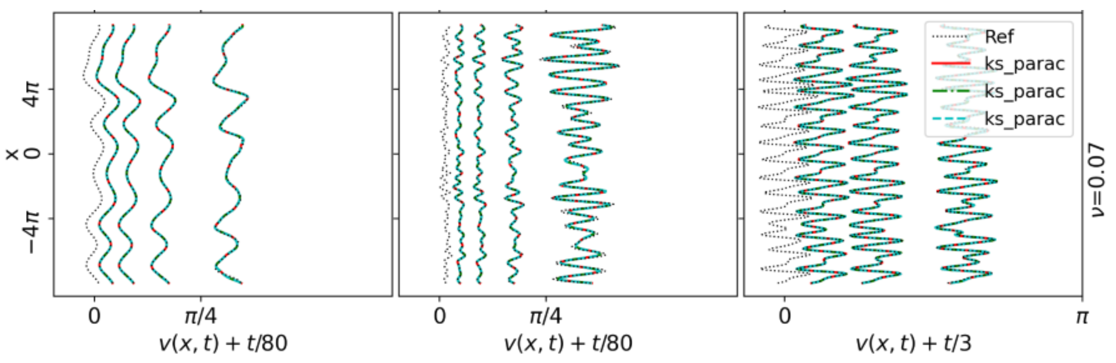

Figure 4.29: Three recurrent short-term predictions $u(x,t)$ when $L = 8\pi$ made by the Network $P - CNN_1$ (see plot legend for line colours/line-style) and the corresponding reference case (black, dotted line) which are the numerical solutions to the KS equation. See fig.4.7 for more details

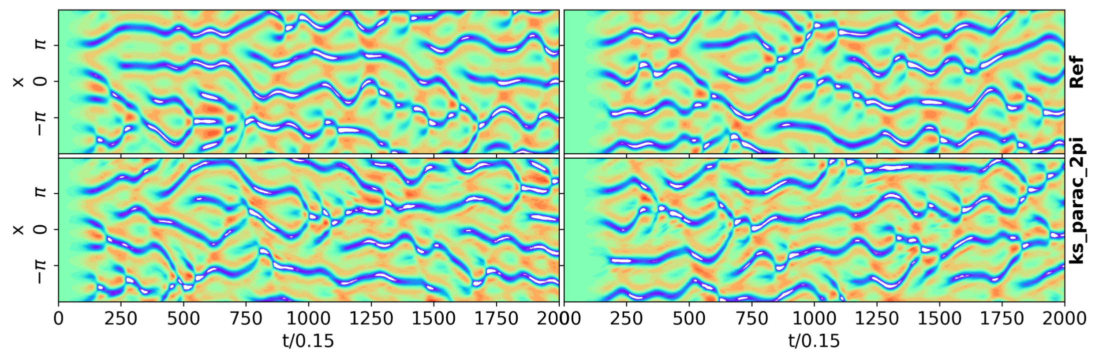

Figure 4.30: Two sets of flame front slope derivatives ($(\partial_x u(x,t))$) when $L = 2\pi$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other corresponds to predictions by $P - CNN_1$. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

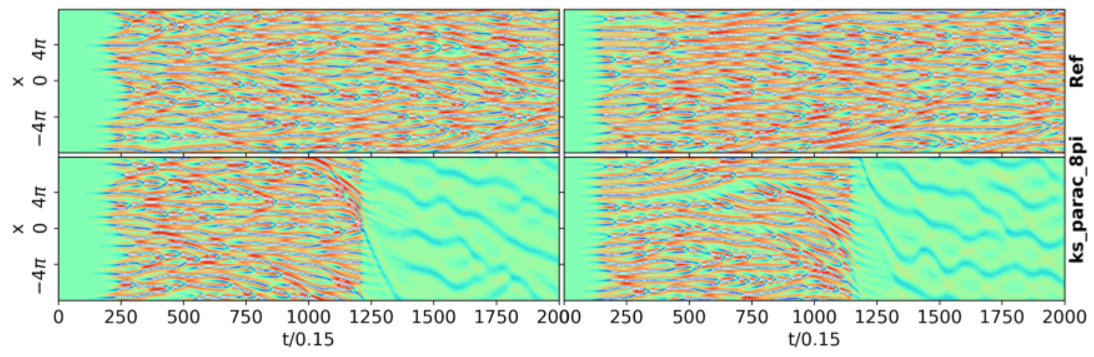

Figure 4.31: Two sets of flame front slope derivatives ($(\partial_x u(x,t))$) when $L = 8\pi$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other corresponds to predictions by $P - CNN_1$. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

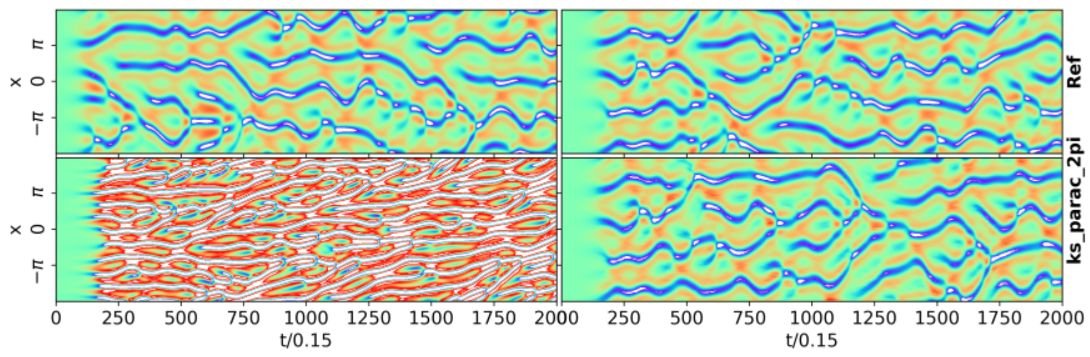

Figure 4.32: Two sets of flame front slope derivatives ($\partial_x u(x,t)$) when $L = 2\pi$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other corresponds to predictions by $P - CNN_{10}$. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

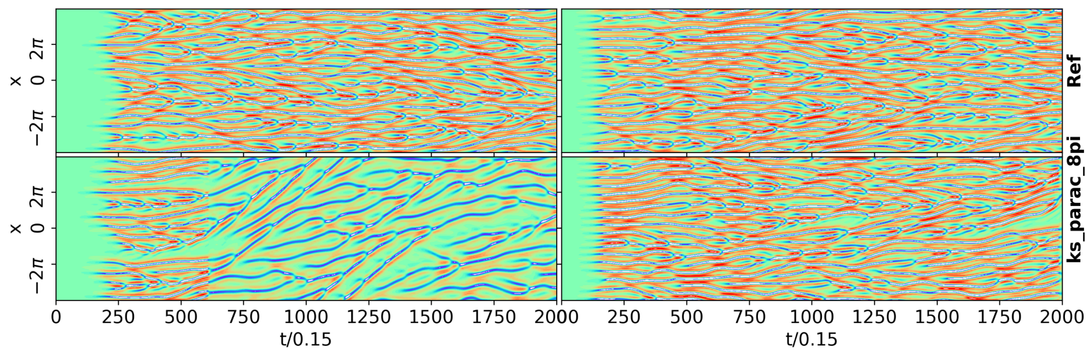

Figure 4.33: Two sets of flame front slope derivatives ($\partial_x u(x,t)$) when $L = 8\pi$ visualized from t=0 to t=2000 (T=2000*0.15=300). Above most set corresponds to the reference case and the other corresponds to predictions by $P - CNN_{10}$. Each set contains a row of two solutions each starting from a differently randomized initial condition $u_0(t,0)$ (spectral-noise to the left and white-noise to the right)

Chapter 5

Conclusion

The main goal of this thesis was to investigate different deep learning methods to predict the spatial-temporal evolution of an unstable nonlinear flame front due to diffusive-thermal instabilities. This evolution can be modeled through a 1-dimensional chaotic fourth-order partial differential equation named the Kuramoto-Sivashinsky equation.

This task was carried out by training an operator to recurrently predict the development of the flame front from one point in time to future states through a small constant time-step. Chaotic PDEs like this one are known to be very difficult to predict for long time sequences. Which is why the initial expectation before conducting this investigation was not that the chosen methods would be able to make accurate long-term recurrent predictions of the solution to the PDE.

By investigating all the figures featuring the recurrent short-term predictions it can be seen that all presented networks are able to make relatively accurate predictions for at least 80 time-step except for two of the models. These two networks are the $P - FNO_1$ and $FNO1, 16$, see figures 4.26 and 4.17. In the case of $FNO1, 16$ the reason for the poor performance for both long term and short term prediction is most likely due to that too much information is lost when truncating the Fourier modes down to an amount of 16, but it might also be because of how much fewer trainable weights the network contains compared to all the other. The $FNO1, 16$ only contains 16077 trainable weights which is quite few in comparison to the 347736 the $P - CNN_1$ have. The poor results generated from the $P - FNO_1$ could be due to that the model structure has too little freedom, perhaps a better structure would be for the FNN to output $4xM$ values, meaning that every Fourier mode not truncated would be scaled by a separate value.

From the long-term prediction and flame front slope derivative plots it is evident that several of the Networks are not only able to make relatively accurate long-term predictions but are also to capture the long-term characteristics of the flame front development. Meaning that even if the prediction deviates from the reference case the characteristics features of the flame front development is still evident. Examples of this are figures.

4.3,4.22,4.5,4.6,4.11,4.14 and4.16. From only inspecting the figures from the standard cases one could draw the conclusion that the CNN performs better than the FNO, especially when considering the flame front slopes and long term time sequences. An important detail to note though is that the CNN contains more than 10 times more trainable weights and the time per epoch about 5 times longer. Which makes one think that it would probably have been good to test extending the width and the depth of the Fourier layers.

Considering the result of the parametric neural networks in chapter 4.3, I think that it is safe to conclude that the implementation of the P-FNO was not very effective. While it is clear that the network is learning to make predictions for different values of L (fig.4.27) the solutions tends to ”explode” as in in fig.4.21. On the other hand the results for the P-CNN are much better than expected, the short term predictions for different values of L are all close to perfectly predicted, see figures. 4.28 and 4.29. The results for the flame front slope derivatives and long-term predictions are also quite good. These results were far above the expectations since the only actual difference to the standard CNN is adding an extra convolution layer in parallel to each of the first two layers in the encoder and combining the outputs through a linear combination determined by a shallow FNN.

One of the bigger problems with the implementation of the parametric NNs is how to ensure that the network actually takes into account for which channel width L it is supposed to make an prediction for. The generation of the initial function $u_0(x, 0)$ was for this study not in any way dependent on the channel width. The only solution thought of was to make the channel width an additional input alongside the solution function $u(x, t)$ to the Network. This leads to the next problem to consider, which is how to use this L within the network structure. The simplest idea tested was if it could work to only use the value of L as a scalar that scales the input function into different length ranges depending on the magnitude of L.

One of the most promising of the ideas was to branch the encoder part of a CNN with an autoencoder structure into two or more branches which merges at the throat before the decoder. By structuring the data preparation so that every sample within a batch have an assigned value of L, then by sending the value of L into a shallow FNN that returns a number f between one and zero. With this value f the input function $u(x, t)$ could be split into two functions by $v_1(x, t) = u(x, t)f$ and $v_2(x, t) = u(x, t)(1 - f)$. Each of which is sent into a separate branch of the encoder. The idea was to use skip connections from each encoder layer to the corresponding decoder layer and possibly having a function to do some linear combination of the two functions when merging at the throat of the network. The idea was never tested due to time limitations but remains as an idea for the future.

Bibliography

[1] J. Yu, R. Yu, X. Fan, M. Christensen, A. Konnov and X.-S. Bai, “Onset of cellular flame instability in adiabatic ch4/o-2/co2 and ch4/air laminar premixed flames stabilized on a flat-flame burner”, English, *Combustion and Flame*, vol. 160, no. 7, pp. 1276–1286, 2013, ISSN: 0010-2180. DOI: `10.1016/j.combustflame.2013.02.011`.

[2] D. Michelson, “Steady solutions of the kuramoto-sivashinsky equation”, *Physica D: Nonlinear Phenomena*, vol. 19, no. 1, pp. 89–111, 1986, ISSN: 0167-2789. DOI: `https://doi.org/10.1016/0167-2789(86)90055-2`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0167278986900552`.

[3] G. SIV ASHINSKY, “Nonlinear analysis of hydrodynamic instability in laminar flames—i. derivation of basic equations”, in *Dynamics of Curved Fronts*, P. Pelcé, Ed., San Diego: Academic Press, 1988, pp. 459–488, ISBN: 978-0-12-550355-6. DOI: `https://doi.org/10.1016/B978-0-08-092523-3.50048-4`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/B9780080925233500484`.

[4] J. C. Baez, S. Huntsman and C. Weis, “The kuramoto-sivashinsky equation”, 2022. DOI: `10.48550/ARXIV.2210.01711`. [Online]. Available: `https://arxiv.org/abs/2210.01711`.

[5] S. Cai, Z. Wang, S. Wang, P. Perdikaris and G. E. Karniadakis, “Physics-Informed Neural Networks for Heat Transfer Problems”, *Journal of Heat Transfer*, vol. 143, no. 6, Apr. 2021, 060801, ISSN: 0022-1481. DOI: `10.1115/1.4050542`. eprint: `https://asmedigitalcollection.asme.org/heattransfer/article-pdf/143/6/060801/6688635/ht\_143\_06\_060801.pdf`. [Online]. Available: `https://doi.org/10.1115/1.4050542`.

[6] Z. Li, N. Kovachki, K. Azizzadenesheli *et al.*, *Neural operator: Graph kernel network for partial differential equations*, 2020. DOI: `10.48550/ARXIV.2003.03485`. [Online]. Available: `https://arxiv.org/abs/2003.03485`.

[7] Z. Li, N. Kovachki, K. Azizzadenesheli *et al.*, *Fourier neural operator for parametric partial differential equations*, 2020. DOI: `10.48550/ARXIV.2010.08895`. [Online]. Available: `https://arxiv.org/abs/2010.08895`.

[8] T. M. Breuel, *Benchmarking of lstm networks*, 2015. DOI: `10.48550/ARXIV.1508.02774`. [Online]. Available: `https://arxiv.org/abs/1508.02774`.

[9] R.Yu, "Deep learning of nonlinear flame fronts development due to darrieus-landau instability", *Submitted for journal publication*, 2022.

[10] J. M. Hyman and B. Nicolaenko, "The kuramoto-sivashinsky equation: A bridge between pde's and dynamical systems", *Physica D: Nonlinear Phenomena*, vol. 18, no. 1, pp. 113–126, 1986, ISSN: 0167-2789. DOI: https://doi.org/10.1016/0167-2789(86)90166-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167278986901661.

[11] B Nicolaenko, B Scheurer and R Temam, "Attractors for the kuramoto-sivashinsky equations", Jan. 1985. [Online]. Available: https://www.osti.gov/biblio/5696580.

[12] D. Papageorgiou and Y. Smyrlis, "The route to chaos for the kuramoto-sivashinsky equation", *Theoretical and Computational Fluid Dynamics*, vol. 3, pp. 15–42, Sep. 1991. DOI: 10.1007/BF00271514.

[13] R. England, "Error estimates for Runge-Kutta type solutions to systems of ordinary differential equations", *The Computer Journal*, vol. 12, no. 2, pp. 166–170, Jan. 1969, ISSN: 0010-4620. DOI: 10.1093/comjnl/12.2.166. eprint: https://academic.oup.com/comjnl/article-pdf/12/2/166/1021770/120166.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/12.2.166.

[14] I. C. Education, *Neural networks*, 2020. [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks#toc-how-do-neu-vMq6OP-P (visited on 26/10/2022).

[15] A. Microsoft, *Artificial intelligence (ai) vs. machine learning (ml)*. [Online]. Available: https://azure.microsoft.com/en-us/solutions/ai/artificial-intelligence-vs-machine-learning/#process (visited on 25/10/2022).

[16] A. L. Fradkov, "Early history of machine learning", *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 1385–1390, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.1888. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320325027.

[17] J.-C. B. Loiseau, *Rosenblatt's perceptron, the first modern neural network*, 2019. [Online]. Available: https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a (visited on 25/10/2022).

[18] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural Networks*, vol. 61, pp. 85–117, 2015, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[19] D. Svozil, V. Kvasnicka and J. Pospichal, “Introduction to multi-layer feed-forward neural networks”, *Chemometrics and Intelligent Laboratory Systems*, vol. 39, no. 1, pp. 43–62, 1997, ISSN: 0169-7439. DOI: https://doi.org/10.1016/S0169-7439(97)00061-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743997000610.

[20] J. Lederer, *Activation functions in artificial neural networks: A systematic overview*, 2021. DOI: 10.48550/ARXIV.2101.09957. [Online]. Available: https://arxiv.org/abs/2101.09957.

[21] K. Janocha and W. M. Czarnecki, *On loss functions for deep neural networks in classification*, 2017. DOI: 10.48550/ARXIV.1702.05659. [Online]. Available: https://arxiv.org/abs/1702.05659.

[22] S. Ruder, *An overview of gradient descent optimization algorithms*, 2016. DOI: 10.48550/ARXIV.1609.04747. [Online]. Available: https://arxiv.org/abs/1609.04747.

[23] S. Bock and M. Weiß, “A proof of local convergence for the adam optimizer”, in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852239.

[24] T. Homma, L. Atlas and R. II, “An artificial neural network for spatio-temporal bipolar patterns: Application to phoneme classification.”, Jan. 1987, pp. 31–40.

[25] J. Xu and K. Duraisamy, “Multi-level convolutional autoencoder networks for parametric prediction of spatio-temporal dynamics”, *Computer Methods in Applied Mechanics and Engineering*, vol. 372, p. 113 379, 2020, ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2020.113379. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782520305648.

[26] V. Badrinarayanan, A. Kendall and R. Cipolla, “Segnet: A deep convolutional encoder-decoder architecture for image segmentation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017. DOI: 10.1109/TPAMI.2016.2644615.

[27] J. Masci, U. Meier, D. Ciresan, J. Schmidhuber and G. Fricout, “Steel defect classification with max-pooling convolutional neural networks”, in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, 2012, pp. 1–6. DOI: 10.1109/IJCNN.2012.6252468.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, “Rethinking the inception architecture for computer vision”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308.

[29] I. Higgins, D. Amos, D. Pfau *et al.*, *Towards a definition of disentangled representations*, 2018. DOI: 10.48550/ARXIV.1812.02230. [Online]. Available: https://arxiv.org/abs/1812.02230.

[30] M. M. Bronstein, J. Bruna, T. Cohen and P. Veličković, *Geometric deep learning: Grids, groups, graphs, geodesics, and gauges*, 2021. DOI: `10.48550/ARXIV.2104.13478`. [Online]. Available: `https://arxiv.org/abs/2104.13478`.

Chapter 6

Appendix A - Programming setup

All the computational experiments performed in this study were conducted in the Integrated development environment Pycharm, except for a. Python (version:3.9) was chosen as the pro The following table contains all python packages used for this study. All scripts used will be made available online on my Github account: LudvigNobel in

Table 6.1: Table of Python packages

Package	Version
Pytorch	1.9.0
Scipy	1.7.3
Matplotlib	3.5.1
Pandas	1.4.2
numpy	1.21.2
functools	-
math	-
pickle	-
timeit	-

the repository deep-learning-KS-eq-master-thesis.