Inclusion of Physics Constraints in Neural Network Surrogate Models for Fusion Simulation

Master Thesis

Philipp Horn BSc

May 5, 2020

Industrial and Applied Mathematics

and

Simulation Technology

Dutch Institute for Fundamental Energy Research

and

Center for Analysis, Scientific Computing and Applications

First Supervisor

Prof. Dr. Barry Koren

TU Eindhoven, CASA

Second Supervisor

Dr. Jonathan Citrin

DIFFER

Third Supervisor

Prof. Dr. Bernard Haasdonk

University of Stuttgart

Fourth Supervisor

Dr. Martijn Anthonissen

TU Eindhoven, CASA

Fifth Supervisor

Ir. Karel van de Plassche

DIFFER

Abstract

High-fidelity numerical calculation of turbulent transport for fusion plasmas is complex and computationally expensive. To obtain real time predictions, necessary for operational optimization and control-oriented applications, surrogate models in the form of neural networks (NNs) are used. These NNs reproduce the reduced-order quasilinear gyrokinetic model QuaLiKiz. This is achieved by supervised learning based on a large dataset of $3 \cdot 10^8$ simulation results. In order to obey known physical constraints an adjusted cost function has to be used. In this work we compare the regression quality of these standard fully-connected feed-forward NNs with networks that directly impose the physical structure in the mapping using a special topology. Through visual inspection it shows that the output of the specialized networks is very smooth and still accurate. This is important for future implementation in implicit transport solvers.

Contents

List of abbreviations

API	Application programming interface
CASA	Center for Analysis, Scientific computing and Applications
CGM	Critical gradient model
CGMnet	Critical gradient model neural network
DAVIDE	Development of an Added Value Infrastructure Designed in Europe
DIFFER	Dutch Institute for Fundamental Energy Research
ETG	Electron temperature gradient
HPC	High performance computing
ITG	Ion temperature gradient
JET	Joint European Torus
JINTRAC	An integrated modelling suite [20]
MLP	Multilayer perceptron
MSE	Mean square error
NN	Neural network
QLKNN	Current state-of-the-art neural networks
QuaLiKiz	A quasilinear gyrokinetic code [3, 11, www.qualikiz.com]
RAPTOR	A transport code [21]
RMS	Root mean square
TEM	Trapped electron mode
TF	TensorFlow

1 Introduction

Reducing the emission of greenhouse gases is one of the biggest global challenges. The largest fraction of the greenhouse gases produced by humans is carbon dioxide. This in turn is mainly due to the energy sector [1]. One of the possible ways to produce energy without emitting carbon dioxide in the future is nuclear fusion. The main advantage of fusion energy over renewable energy sources like wind, water and solar is its availability. While renewable energy often depends on the weather or cannot be located everywhere, fusion energy could be generated everywhere and at any time. Nevertheless, for fusion reactors to generate net energy efficiently there are still some obstacles that have to be overcome.

For fusion to be possible in a reactor, extremely high temperatures around 10^8 K are needed. Because no material can withstand these high temperatures, there are steep temperature gradients in the fusion plasma. This way the plasma is hot enough in its core for fusion to happen but cold enough on its surface to be surrounded by the reactor. These gradients cause turbulence inside the plasma and this is a drive for transport of heat and particles from the core of the plasma to the surface [2]. For a reactor to run efficiently this transport needs to be restricted. Hence, turbulence in a fusion reactor has to be better understood and precisely predicted. However, simulating the turbulent transport in a fusion plasma is very complicated and computationally expensive. Direct numerical simulations with nonlinear gyrokinetic codes would require 10^4 - 10^6 hours of computation on massively parallel high performance computing (HPC) systems for one turbulent flux at a single radial location.

For applications like reactor optimization or control of a fusion reactor, fast calculations of the turbulent transport are needed. It would be optimal if these predictions were computed in real time. Even reduced order codes are not fast enough to achieve this, although they are six orders of magnitude faster than the previously mentioned nonlinear gyrokinetics codes. However, they are accurate and fast enough to create a dataset for surrogate models. These surrogate models in turn are able to predict turbulent transport in real time. Surrogate models are used to approximate the input-output mapping of a more complex mathematical model. For the surrogate model it is irrelevant how the input-output mapping is performed by the complex model. Before the surrogate model can be built, a dataset of input-output pairs of the original model has to be computed. The datapoints for this dataset are chosen in an area where it is relevant for the surrogate model to be accurate. The surrogate model is then optimized to reproduce these data pairs. However, the quality of the surrogate model is measured by its ability to generalize from the seen dataset onto the whole input-output mapping. Since the surrogate model is supposed to predict outputs for new inputs instead of only reproducing the outputs for already known input-output pairs.

For this work a dataset produced by the quasilinear gyrokinetic code QuaLiKiz is used [3, 4]. There is a multitude of different types of surrogate models. For this purpose neural networks produce good results [5]. They predict new outputs over five orders of magnitude faster than the original quasilinear code while also computing the derivatives with respect to the inputs. These derivatives are important for the integration of the surrogate model in transport codes. The benefit of neural networks over algorithms like an interpolation is that their optimization is much faster in higher dimensions for a given dataset. Furthermore, no explicit function, like a polynomial that is optimized to approximate the data, has to be given.

The biggest drawback of surrogate models is the loss of all physical properties that are in the original model. Physical constraints that directly work on the input-output level may be reintroduced into the surrogate model though. The focus of this master thesis is to construct neural networks that incorporate one such physical constraint directly in their topology. This physical constraint is described by the so called *critical gradient model*. These neural networks are then compared to the current state-of-the-art neural networks for turbulent flux predictions in fusion reactors [6]. The main advantage of incorporating the critical gradient model in the topology of the neural network is an output with fewer fluctuations. This hopefully leads to a faster convergence of implicit solvers in transport codes that use these predicted turbulent fluxes and need the derivatives with respect to the inputs. In summary the following research question arises:

> *Is a neural network with a specialized topology capturing physical features better suited to predict turbulent fluxes in a fusion plasma than the current state-of-the-art neural networks?*

In this context better suited means that it can predict the fluxes at a similar speed with an increased accuracy and a smoother output.

This thesis is divided into six more chapters after this short motivation. It is structured in a way that all relevant background information is given to be able to understand the choices that were made in the construction of the final neural networks. In Chapter 2 a very brief introduction into the involved physics is given. After that in Chapter 3 the basic aspects and terminology of neural networks is explained. This is followed by the design of the specialized neural networks in Chapter 4. A few remarks on the implementation of the neural networks are given in Chapter 5 and the choices of the hyperparameters in the neural networks are explained in Chapter 6. The results and findings of this work are presented in Chapter 7. Finally, a short outlook is given in Chapter 8.

2 Theory of the involved physics

Nuclear fusion describes the process in which two atoms, or to be more precise their nuclei, fuse together into a different nucleus. For light nuclei there is a huge amount of energy per nucleus released during this reaction. For example all the energy emitted by the sun is due to nuclear fusion in its core. For two nuclei to fuse there is a lot of energy needed beforehand since these two nuclei repel each other due to their electrical charge and the corresponding Coulomb force. Inside the sun nuclear fusion is possible because of the enormous pressure in its core. However, under the conditions on earth the most promising concept for fusion power generation is thermonuclear fusion of deuterium and tritium. For this reaction the nuclei need to overcome the Coulomb barrier of around 200 keV. Luckily at an energy level of around 10 keV the quantum mechanical phenomenon called quantum tunnelling already enables the nuclei to fuse. At such a high energy which is equivalent to over 100 million Kelvin the nucleus and the surrounding electrons separate and a plasma develops. This conversion from temperature T to energy E can be done through the formula

$$E = \frac{3}{2} k_B T. \tag{2.1}$$

With k_B being the Boltzmann constant. This hot plasma needs to be confined. In a thermonuclear fusion reactor the plasma is confined through strong magnetic fields in a toroidal geometry. The furthest developed concept for this task is the tokamak [7]. In a tokamak the magnetic field is generated through external field coils and the plasma itself. The resulting magnetic field has helical field lines and confines the plasma through the Lorentz force. These helical field lines can be seen as black lines in the draft of a tokamak in Figure 2.1.

Nuclear fusion research has been an active field of research for nearly 80 years. A detailed introduction would therefore go beyond the scope of this work. Nonetheless, the following sections briefly explain the relevant topics for the rest of this thesis.

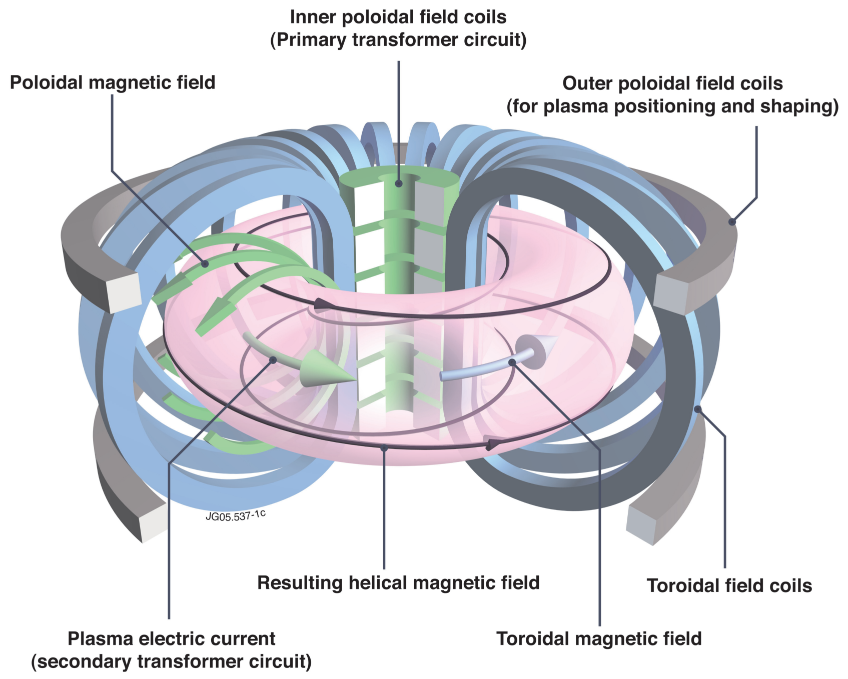
Inner poloidal field coils
(Primary transformer circuit)
Poloidal magnetic field
Outer poloidal field coils
(for plasma positioning and shaping)
JG05.537-1c
Resulting helical magnetic field
Toroidal field coils
Plasma electric current
(secondary transformer circuit)
Toroidal magnetic field

Figure 2.1: Layout of a tokamak, its field coils and the resulting magnetic field. (Source: EUROfusion)

2.1 Turbulence in a tokamak

Inside the plasma of a tokamak there is a multitude of different instabilities and turbulence. The instabilities of relevance in this work are microinstabilities in the core of the plasma that are triggers for turbulent transport. These instabilities have been one of the major obstacles in tokamak research during the last 35 years. Without these microinstabilities fusion reactors like the Joint European Torus (JET) would already have been able to reach breakeven and even ignition. Breakeven describes the case in which the energy released through fusion rises above the externally applied heating power. This was almost achieved in JET. Ignition is the name for the point from which on the heating from the fusion power is enough to sustain the fusion conditions and no externally applied heating power is needed.

The fusion power P_{fusion} in a tokamak scales at the relevant temperate around 10 keV like

$$P_{\text{fusion}} \propto n^2 T^2. \tag{2.2}$$

Therefore, in order to build a tokamak that reaches breakeven, a high temperature T and density n of the plasma are essential. In a reactor the density and temperature vary over time and depending on the radial position in the tokamak. The highest temperature and pressure can be reached in the core of the plasma. The previously introduced helical field lines form so called flux surfaces. Along the magnetic field lines, transport is unhindered and therefore temperature and density are constant on a flux surface. Transport across the flux surfaces on the other hand is considerably smaller. Turbulence is one of the main reasons for transport of heat and particles across flux surfaces. Through this turbulent transport large amounts of energy are lost from the core of the plasma and more external heating is needed.

In order to compute radial temperature and density profiles of the plasma in a tokamak the turbulent heat and particle fluxes of the electrons and ions are needed. Those can be obtained from kinetic theory. In kinetic theory the plasma is described through the distribution function of its particles

$$f(\boldsymbol{x}, \boldsymbol{v}, t), \quad \text{with: } \boldsymbol{x} \in \mathbb{R}^3, \boldsymbol{v} \in \mathbb{R}^3, t \in \mathbb{R} \tag{2.3}$$

in the six dimensional phase space $(\boldsymbol{x}, \boldsymbol{v})$. One such function f_j is needed per species j of particles. The particle species in a plasma are the electrons and the different types of ions. In the dataset used in this work there are two different types of ions and therefore three different species of particles. Alternatively a plasma can be described through magnetohydrodynamics. The use of a particle distribution function is however more accurate in the case of a low number of collisions of the particles. The time evolution of a particle distribution function is given through Liouville's theorem by the Fokker-Planck equation [8].

$$\frac{\partial f_j}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla}_{\boldsymbol{x}} f_j + \frac{q_j}{m_j}(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) \cdot \boldsymbol{\nabla}_{\boldsymbol{v}} f_j = \left(\frac{\partial f_j}{\partial t}\right)_{\text{coll}}, \tag{2.4}$$

with

q_j = electric charge of particles from species j,

m_j = mass of particles from species j,

E = electric field,

B = magnetic flux density.

The right hand side of this equation is a collision operator. In fusion often the Fokker-Plank operator is used, hence the name of the equation. The left hand side is the total derivative of the distribution function with respect to time. The acceleration is substituted with the Lorentz force divided by the mass. These equations, one per particle species, have to be coupled with Maxwell's equations. In these Maxwell equations the current and charge density are given through moments of the particle distribution function, because in a plasma the particles are charged. Therefore, they produce their own electric and magnetic fields:

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \cdot (\epsilon_0 \boldsymbol{E}) = \sum_j q_j \iiint f_j \, d^3v, \tag{2.5}$$

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \times \boldsymbol{B} = \mu_0 \sum_j q_j \iiint \boldsymbol{v} f_j \, d^3v + \frac{1}{c^2}\frac{\partial \boldsymbol{E}}{\partial t}, \tag{2.6}$$

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \cdot \boldsymbol{B} = 0, \tag{2.7}$$

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}, \tag{2.8}$$

with

ϵ_0	= vacuum dielectric permittivity,
μ_0	= vacuum magnetic permeability,
c	= speed of light in a vacuum.

Solving this large system of partial differential equations numerically may be too time-consuming. So approximations may have to be applied. Since these partial differential equations are not solved in this form, no boundary conditions are given here. The boundary conditions depend on the approximations one makes before solving the equations.

2.2 Gyrokinetics

The most common approximation of the previously introduced system of equations is the so called *gyrokinetic approximation.* The electrons and ions in the plasma perform a helical motion around a magnetic field line. The circular motion around the magnetic field line while following its direction is called gyromotion. The time for one orbit is much smaller than the timescale of the turbulence in a fusion plasma. The gyrokinetic approximation separates the guiding center motion from the gyromotion by averaging over the gyro orbit [9]. The resulting equations have one dimension less and are therefore five dimensional. Also, due to the elimination of the small timescale of the gyromotion, numerical methods become more efficient. While it is possible to run single simulations with this approximation, additional approximations are needed in order to generate a whole dataset of input-output mappings for a surrogate model. Since the disturbance of the particle distribution function due to the instabilities in the core is rather small percentage-wise, the distribution function can be expressed as the sum of a static part f_0 and a small perturbation δf.

Through gyrokinetic simulations it is possible to explain the turbulent transport in the core of the plasma of reactors like JET. The found microinstabilities that are the cause of the turbulence are divided into instabilities of the ions and instabilities of the electrons. Furthermore, they are also classified based on their wavelengths [10]. For this work there are three turbulence modes that are of interest: Ion Temperature Gradient (ITG), Trapped Electron Mode (TEM) and Electron Temperature Gradient (ETG). ITG and TEM turbulence can drive heat and particle fluxes of ions and electrons. ETG turbulence however has very short wavelengths and therefore it leaves the ions unaffected. Hence, it only produces a heat flux of the electrons. This property of ETG turbulence simplifies the design of a neural network surrogate model for this type of turbulence.

2.2.1 QuaLiKiz dataset

While the previously introduced δf simulation codes already are orders of magnitude faster than full f codes, they still take around 50 000 CPUh to compute the fluxes for one radial point [11]. To create a large dataset of gyrokinetic simulations, a reduced order model with additional approximations has to be used. For this work a dataset of around 290 million flux computations by the quasilinear gyrokinetic code QuaLiKiz [3, 11, www.qualikiz.com] is used. The dataset has been created during a previous master thesis [5]. This dataset consists of a hypercube of nine input parameters for QuaLiKiz and the corresponding fluxes. The definition of the input parameters for QuaLiKiz that vary in the hypercube for the dataset can be found in Table 2.1. Out of this huge dataset also a smaller dataset is extracted by fixing two inputs as $Z_{\text{eff}} = 1$ and $\log(\nu^*) = -3$. Therefore, the extracted dataset can be reduced to a seven-dimensional dataset, since two of the nine input parameters are fixed in this dataset. The smaller dataset is created since it allows a 20 times faster optimization of the neural networks. This smaller dataset is used to find good hyperparameters of the surrogate model. Previous work showed that neural networks with these hyperparameters also produce good results on the large dataset [6]. The full dataset is filtered to remove flux predictions that are much higher than any fluxes found in experiments. Additionally, input-output pairs with indications

Table 2.1: Nine inputs for QuaLiKiz in the dataset

Name	Description	Formula
Ate	Inverse normalized electron temperature gradient scale length	$\frac{R_0}{L_{T_e}} = -\frac{R_0}{T_e}\frac{dT_e}{dr}$
Ati	Inverse normalized ion temperature gradient scale length	$\frac{R_0}{L_{T_i}} = -\frac{R_0}{T_i}\frac{dT_i}{dr}$
An	Inverse normalized density gradient scale length	$\frac{R_0}{L_n} = -\frac{R_0}{n}\frac{dn}{dr}$
q	Safety factor	q
smag	Magnetic shear	$\hat{s} = r\frac{q'}{q}$
x	Normalized averaged minor radius of the flux surface	$x = \frac{r}{a}$
Ti_Te	Quotient of ion and electron temperature	$\frac{T_i}{T_e}$
Zeff	Effective ion charge	$Z_{\text{eff}} = \frac{\sum_j n_j Z_j^2}{n_e}$
logNustar	Logarithmic collisionality	$\log(\nu^*)$

of problems during a numerical integration inside QuaLiKiz are filtered out [6].

QuaLiKiz takes only 5 CPUmin for one radial profile [11]. Hence, for the creation of the dataset approximately 1.25 million CPUh were needed. For QuaLiKiz to be this fast, major approximations have to be used. Hence, the speedup comes at the cost of accuracy. However, the turbulent fluxes computed by QuaLiKiz are validated against higher fidelity gyrokinetic codes and the approximations proved to be for a wide tokamak core parameter space. For a discussion of the approximations and assumptions in QuaLiKiz see [12]. QuaLiKiz takes local plasma parameters at a radial position and computes the turbulent heat and particle fluxes. Around 20 of these local simulations are needed for one radial temperature and density profile.

2.3 Critical gradient model

ITG, TEM and ETG turbulence all follow a critical gradient model [13]. This means that the microinstabilities only occur if a critical temperature gradient is exceeded. This behavior is observed in experiments as well as in simulations. It can also be seen in Figure 2.2, which shows typical data from QuaLiKiz. Capturing this behaviour also in the surrogate model is critical since most experiments happen in regimes around this critical gradient. The type of temperature gradient of relevance is the temperature gradient that drives the turbulence. Hence, for TEM and ETG turbulence the critical gradient is an electron temperature gradient and for ITG turbulence it is an ion temperature

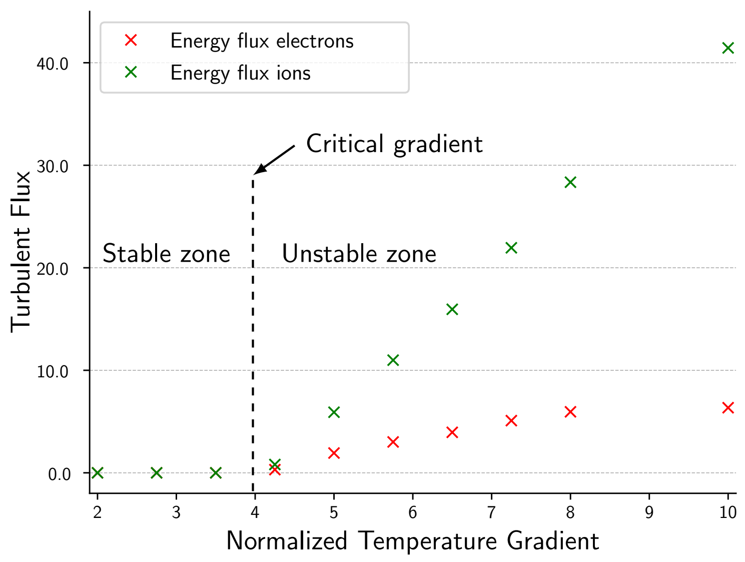

Figure 2.2: Typical slice of data from QuaLiKiz for ITG turbulence. The normalized temperature gradient is the ion temperature gradient in this case.

gradient. The critical gradient is the same for all transport channels of one turbulence type, meaning for all turbulent fluxes, i.e. ion heat flux, electron heat flux, and particle flux. The growth of the fluxes in the unstable region can be modeled differently. For this work it is modeled by a signomial in one dimension and with only one term. To write the critical gradient model (CGM) in a formula, the input parameters can be divided into a special input x_s which is the temperature gradient that drives the turbulence and the remaining general inputs $\boldsymbol{x}_g$. A turbulent flux that follows the CGM is then described by

$$f(\boldsymbol{x}_g, x_s) = \begin{cases} 0, & \text{for } x_s \leq \left(-\frac{R_0}{T}\frac{dT}{dr}\right)_{\text{crit}} = c_1 \\ c_3(x_s - c_1)^{c_2'}, & \text{else} \end{cases} \tag{2.9}$$

$$= c_3 H(x_s - c_1)(|x_s - c_1|)^{c_2'} \tag{2.10}$$

$$= c_3 R(x_s - c_1)(|x_s - c_1|)^{c_2}, \tag{2.11}$$

with:

$$c_2' = c_2 + 1. \tag{2.12}$$

The critical gradient c_1 and the other constants c_2, c_3 depend on $\boldsymbol{x}_g$. With $H(\,\cdot\,)$ the Heaviside function and $R(\,\cdot\,)$ the ramp function or rectifier:

$$R(x) = \max(0, x). \tag{2.13}$$

3 Introduction into neural networks

This chapter gives a brief introduction into neural networks (NNs). For a more extensive yet still basic introduction see [14].

For this thesis only so called feedforward neural networks were used. In these networks information only flows in one direction. An NN consists of several layers. Each layer in turn consists of neurons. Neural networks have at least two layers. One layer for the input with one neuron per input dimension and one layer for the output which consists of one neuron per output dimension. In the case of two layers the NN would be called a perceptron. Additionally, a neural network can have any number of layers between the input and output layer called hidden layers. It is named a multilayer perceptron then. Such a multilayer perceptron (MLP) is illustrated in Figure 3.1. When using MLPs each

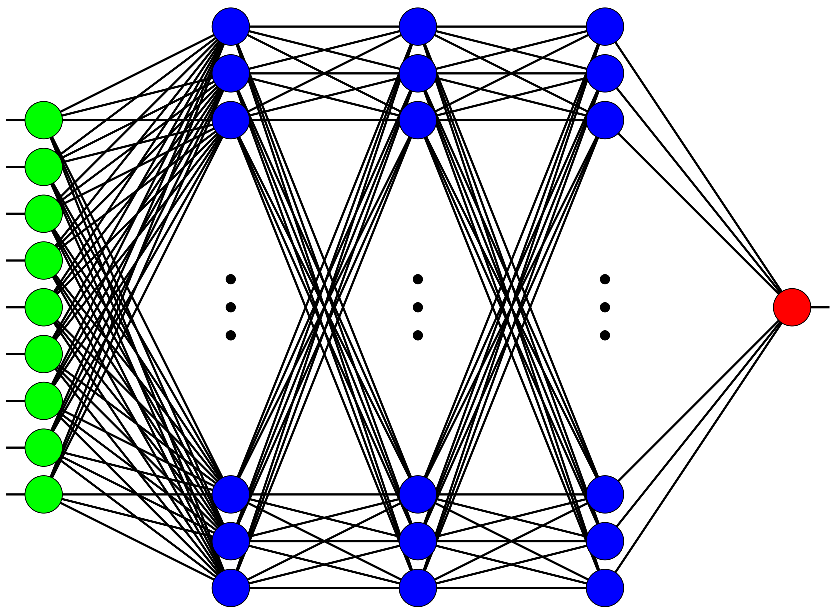

Figure 3.1: Layout of a multilayer perceptron. The neurons of the input layer are colored green, the neurons of the hidden layers blue and the neuron of the output layer red. Information flows from left to right.

neuron in a layer is connected to all neurons in the previous layer and to each neuron of the following layer. But since all perceptrons are feedforward networks the neurons of one layer are not connected between themselves. Also, the output of one neuron only influences the input of the neurons in the next layer and not the ones in the previous layer. That is the realization of the information flow in one direction only.

A neuron j calculates its input x_j as a weighted sum of the previous outputs

$$x_j = \sum_{i=1}^{N} w_{ji} y_i + \beta_j, \tag{3.1}$$

with

N	= number of neurons in the previous layer,
w_{ji}	= weight of the j-th neuron in this layer for the output of the i-th neuron of the previous layer,
y_i	= output of the i-th neuron of the previous layer,
β_j	= bias value of the j-th neuron in this layer.

The weights and bias values of the neurons also vary across the layers. This is not captured in this formula with an additional index for better readability. In order to calculate its own output a neuron uses an activation function φ. In principle this can be any bounded and nonconstant function. The functions that are used in practice are inspired by the biological neuron. A biological neuron releases an electrical impulse if the combined impulses it receives are above a certain level. So the obvious choice for φ would be the Heaviside step function. However, this function is not continuous, which can be important for the theoretical analysis of the capabilities of an NN, as introduced later in this chapter. Furthermore, since a differentiable activation functions is preferable for the optimization of the weights and bias values, it is better to smoothly approximate the Heaviside function. Popular examples are:

$$\varphi(x) = \mathrm{sig}(x) = \frac{1}{1 + \exp(-x)},$$

or

$$\varphi(x) = \tanh(x) = 1 - \frac{2}{1 + \exp(2x)}.$$

In this work only the second activation function is used since it produced good results in previous works [5, 15]. In summary the output $\boldsymbol{y_l}$ of layer l is calculated as

$$\boldsymbol{y}_l = \varphi(\boldsymbol{x}_l) = \tanh(\boldsymbol{x}_l) = \tanh(\boldsymbol{W}_l\, \boldsymbol{y}_{l-1} + \boldsymbol{\beta}),$$

where all the bold symbols are the respective values in the according layer arranged in a vector. $\boldsymbol{W}$ is the matrix of all the weights in that layer and tanh is evaluated element wise. The input layer of the NN has no activation function and is only needed to feed the input to the following layer without weighting. The output layer normally uses a linear activation function in order to be able to produce any output in $\mathbb{R}$ per neuron.

In conclusion the neural network is a nonlinear mapping

$$f_{\text{NN}} : \mathbb{R}^{N_{\text{input}}} \longrightarrow \mathbb{R}^{N_{\text{output}}},$$

with:

$$\begin{aligned} N_{\text{input}} &= \text{number of neurons in the input layer,} \\ N_{\text{output}} &= \text{number of neurons in the output layer.} \end{aligned}$$

One of the main motivations for the use of NNs as surrogate models is the existence of Universal Approximation Theorems. One such theorem given in [16, Theorem 3] states that for any continuous function f on any compact $\mathcal{X} \subset \mathbb{R}^{N_{\text{input}}}$ and any given ϵ there exists an MLP with only one hidden layer such that

$$\sup_{x \in \mathcal{X}} ||f(x) - f_{\text{NN}}(x)||_2 < \epsilon. \tag{3.2}$$

For this theorem the activation function can be chosen to be any bounded, nonconstant and continuous function. The theorem however does not give the number of neurons that are needed in the single layer. Nor does it provide an algorithm for the calculation of the weights and bias values. Experiments show that, often, adding a second or third layer helps to increase the approximation quality faster than adding lots of neurons in one layer. More than three layers in an NN are typically only used for more complex tasks. The number of layers and the numbers of neurons in these layers form the topology of an NN. Adjusting the topology is usually done through large hyperparameter studies, since an a-priori analytical optimization of the numbers of layers and neurons would be very difficult.

In order to optimize the weights and bias values, supervised learning is used. In supervised learning the NN is fed with an input value for which the desired output is known. The output of the NN is then compared to the desired output. The discrepancy of these two values is used in a goodness metric for the NN. In the machine learning field this metric is called the loss function. An example for such a loss function is given in equation (3.3).

$$E = \frac{1}{n} \sum_{k=1}^{n} ||\boldsymbol{o}_k - \boldsymbol{p}_k||_2^2 + \lambda \sum_{l=1}^{N} ||\boldsymbol{W}_l||_F^2, \tag{3.3}$$

with:

n	= number of training points,
$\boldsymbol{o}_k$	= desired output for the k-th input,
$\boldsymbol{p}_k$	= predicted output of the neural network for the k-th input,
λ	= coefficient for the L_2 regularization,
N	= number of layers,
$\|\|\boldsymbol{W}_l\|\|_F^2$	= squared Frobenius norm of the weight matrix in layer l .

This particular loss function uses a mean square error (MSE) to measure the error the NN makes in predicting the output. In this loss function also a second term is incorporated. This term is an L_2 regularization of the weights of the NN. A regularization like this is used in order to prevent overfitting. Overfitting describes the behaviour of an NN or any surrogate model in which it perfectly fits the data, but in order to do so it sacrifices the ability to predict new outputs. Normally this happens because the behaviour of the NN becomes very erratic between the datapoints it is trained on. More details on this topic are given in Chapter 6.

The NN learns the input-output mapping by minimizing this loss function. This is called training. The minimization is carried out by an optimization algorithms. There are many optimization algorithms suited for this task. Since the NN is an analytical function for which we can calculate all derivatives, the optimization algorithms range from Gradient Descent to higher order algorithms. All of the algorithms are iterative methods and each iteration is called an epoch in NN training. Most of the algorithms are specifically designed for the training of NNs or they are at least adjusted to it. For example the adjusted Gradient Descent for NNs is one of the most widely used algorithms for NN training. For this work the Adam algorithm is used [17]. Previous work shows that this algorithm is the most robust and efficient of the tested ones [5]. There exists a theorem in [17, Corollary 4.2] that guarantees the convergence of the Adam algorithm for convex functions. However, since the loss function of an NN is not necessarily convex, it can not be applied in our case.

One of the main reasons why optimization algorithms are specifically developed for the minimization of the loss function of NNs is the use of batches. Calculating the loss of an NN for the whole dataset is very time consuming, especially for large datasets. This is a problem for optimization algorithms that rely on a fast evaluation of the loss function. A commonly used method to speed up the training is to divide the dataset in smaller batches and evaluate the loss function separately for the batches. The weights and bias values of the NN are then updated according to the optimization algorithm but with the loss on only one batch. During one epoch the loss is still calculated for all batches but the parameters of the NN are adjusted after every batch. Hence, the weights and bias values of the NN are updated more often during one epoch if batches are used, which leads to the mentioned speedup.

Since the objective function is only evaluated for random samples the minimization of it becomes a stochastic optimization problem instead of a deterministic one. This leads to the already mentioned adjusted optimization algorithms and new challenges. Two of the challenges are to avoid that the NN learns the order of the batches or which datapoints are grouped in which batches. In order to prevent this learning the dataset is shuffled after each epoch and divided into new batches. For a dataset containing many samples this can take up even more time than the actual epoch. The speedup due to the use of batches is however considerably larger. When choosing the number of batches, there are several points to be considered. Small batches lead to more variance in the calculated losses and derivatives. This leads to larger fluctuations in the loss over the epochs and problems finding a minimum of the total loss. Furthermore, changing the weights and bias values takes up a considerable amount of time and the optimization can be parallelized over the datapoints in one batch. Therefore, smaller batches increase the time needed for one epoch but also the number of updates during one epoch. This leads to the conclusion that there is an optimal number of batches when training on a given dataset.

Independent of the method used for the training of the NN not all data is used for the optimization of the weights and bias values. The data is divided into training, validation and test data before the training starts. The validation data is used during the training to check how good the NN can predict data it has never seen. As soon as the loss on the validation data stops to drop, the training is finished. The validation loss is also used to tune all hyperparameters of the NN. The test data is used to finally quantify how good an NN performs.

4 Topology of neural networks to fit QuaLiKiz data

There are several approaches of how an NN can be used to approximate the input-output mapping of QuaLiKiz. The simplest method is to use a basic MLP with one output per turbulent flux or to train one MLP per turbulent flux with only one output. This approach is shortly discussed in Section 4.1. When using basic MLPs, an adjusted loss function has to be used in order to obey known physical constraints. A more elegant approach is presented in Section 4.2. In this approach the known physical constraints — the critical gradient model — are directly imposed on the mapping of the NN by the use of a special topology. Again there are several possibilities in the choice of the topology within this approach. In the end a final topology of the further used NNs is given.

4.1 Multilayer Perceptrons

Using an MLP as a surrogate model for QuaLiKiz is the current state-of-the-art approach [6]. In theory an MLP should be enough to approximate the input-output mapping of QuaLiKiz up to any desired accuracy due to the universal approximation theorem. In practice the weights and bias values of the NN have to be chosen by an optimization algorithm. The only information that goes into this algorithm are single datapoints of the mapping. Therefore, there is a limit on the accuracy because of the partial information of the original mapping. This is always the case in machine learning applications and one of its big limitations. Furthermore, the outputs of the NN do not necessarily obey any known physical constraints. The only way physics is incorporated in the NN as a model, is through the training on datapoints that themselves follow physical constraints. Increasing the number of datapoints that are used during the training increases the chance of the NN obeying physical constraints and leads to a more accurate mapping. However, this is not always a feasible solution. As discussed in previous chapters, the calculation of these datapoints is very time consuming. Also, the necessary time for the training of the NN increases with a higher number of datapoints. So from a theoretical point of view the universal approximation theorem is a very important achievement but from a practical point of view this does not guarantee good approximation properties of an NN.

Another method of leading the NN to obey physical constraints is to adjust the loss function. So far, the introduced loss function consists of a mean square error term and a regularization term. Additional terms can be introduced that produce a higher loss if the NN breaks any of the known physical constraints. For this approach to work it has to be known how the physical constraints influence the output for a given input, because input and output are the only physical variables we can measure in the NN as a surrogate model. For example, energy conservation can not be enforced this way as long as it is not known how it influences the input-output mapping. The only physical constraint we want to implement in this work is the CGM. Since the CGM directly constrains the shape of the output for one input dimension it is possible to include terms in the loss function that increase if the output differs from this shape. For more details see [6].

4.2 Critical gradient model neural networks

In this section a completely different topology for the NN compared to the current state-of-the-art NNs is presented. The topology is specifically designed for the use as a surrogate model for QuaLiKiz. A first proof of concept of the topology has been given in [15]. With this topology the CGM is directly enforced. The basic idea is to build an NN that calculates the three constants in the CGM and afterwards combines these to the desired output instead of an NN that directly calculates the output. This approach integrates the physical structure in the topology of the network. Hence, no modification of the basic loss function (3.3) is needed. Also, the optimization of the additional parameters in the loss function for the MLP can be omitted. Since these NNs are naturally very regularized it is questionable if the regularization term in the loss function is really necessary. This is further investigated in Chapter 6.

In the CGM one input dimension is treated differently from the others. This dimension is the gradient that drives the turbulence. For ETG and TEM mode turbulence this special dimension x_s is the electron temperature gradient and in the ITG mode the ion temperature gradient drives the instabilities. This resulted in equation (2.11) for the calculation of a turbulent flux f. This equation is repeated here for better readability:

$$f(\boldsymbol{x}_g, x_s) = c_3 R(x_s - c_1)(|x_s - c_1|)^{c_2}.$$

The constants c_1, c_2 and c_3 depend on the other general input dimensions $\boldsymbol{x}_g$ and are independent of the special input. Therefore, the special input also has to be treated differently from the general input by the NN. In Figures 4.1 and 4.3, layouts of the possible topologies can be seen. The important part for the physical structure is the

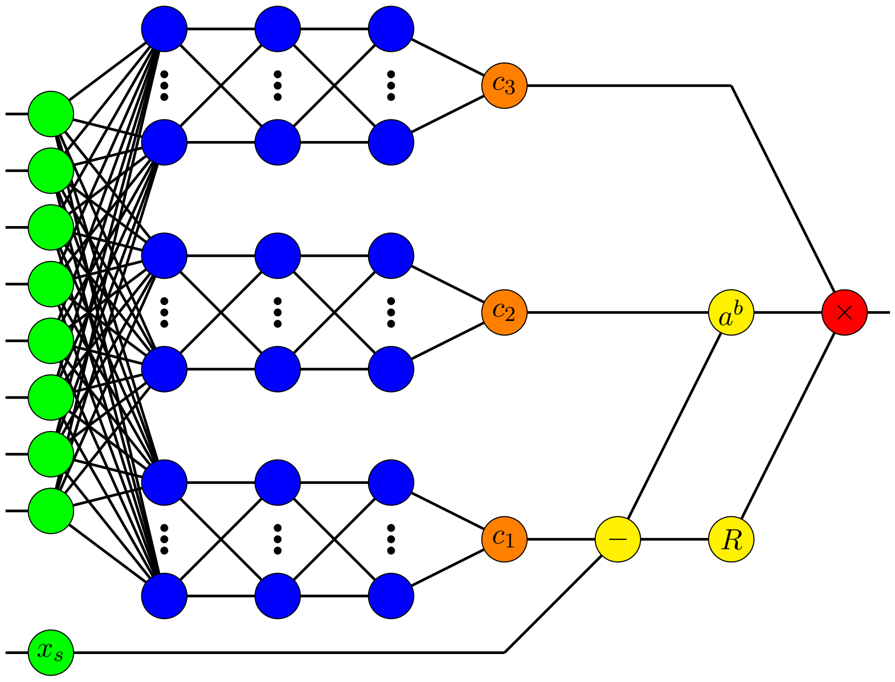

Figure 4.1: Layout of a Critical Gradient Model NN (CGMnet) with three independent neural networks for the calculation of the constants. The neurons are colored as in Figure 3.1. Additionally, the neurons that output the constants are orange. The neurons performing mathematical operations are colored yellow.

later part (right in the figures). The difference in the beginning of the topologies in Figures 4.1 and 4.3 is explained in the following two sections.

One can see that the independence of the constants from the special input is achieved by inserting x_s to the NN only after the constants have been calculated, as can be seen in Figure 4.1. In contrast to the layout of the MLP in the present NNs, more types of layers than input, hidden and final output layers are needed. An intermediate layer for the calculation of the constants is needed. The neurons in this layer are built in the same way as neurons in an output layer. Except that the user of the NN does not see the output of these neurons, it is processed further. In the next layer, the special input is inserted into the NN. This layer computes the difference between the temperature gradient and c_1 that acts as critical gradient or threshold. This difference is next fed to the following layer consisting of two neurons. The first is an exponent neuron. It calculates the difference to the power of c_2. The second one is a neuron with all weights equal to one and a bias value of zero. As an activation function it uses the rectifier

$$\varphi(x) = \max(0, x).$$

The actual output layer consists of one neuron that multiplies the two outputs of the previous layer and c_3. This together leads to an output that has the desired shape of the CGM. All the just described neurons, except for the rectifier neuron, are no standard neurons. They do not have weights, bias values and an activation function. Instead, they perform mathematical operations.

Since the optimization algorithm for the calculation of the weights relies on derivatives one has to be careful. The rectifier is not differentiable for $x_s = c_1$. But since the derivatives near that point are still bounded this does not cause uncontrolled large jumps of the weights and bias values during the training. The power neuron however can lead to problems. If c_2 is negative, the derivatives of the final output of the NN diverge at $x_s = c_1$. Also, a negative c_2 would imply a sub-linear growth of the turbulent flux with respect to the gradient that drives it, which is not physical. Therefore, it is reasonable to also use a rectifier as activation function inside the neuron that calculates c_2. Thus, the overall exponent in the CGM is larger or equal to one and the derivatives no longer diverge. A comparison of the evolution of the loss with and without a constrained c_2 during the training can be seen in Figure 4.2.

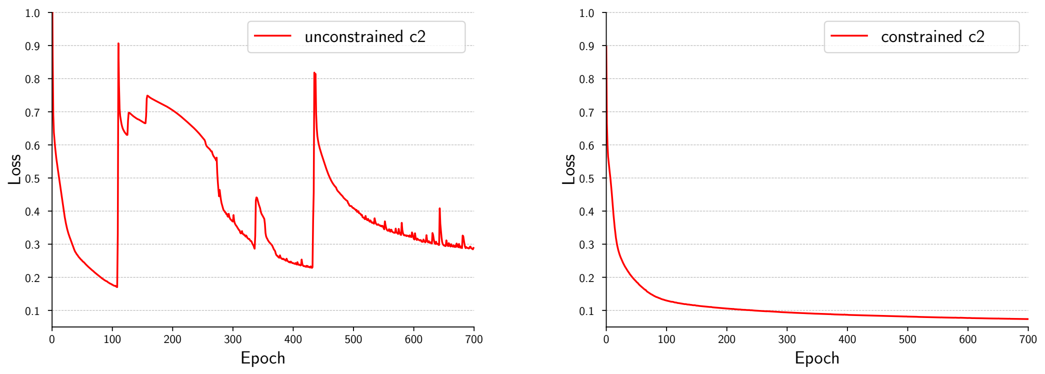

Figure 4.2: Loss function during the training of an NN with an unconstrained c_2 (left) and a constrained one (right). The optimization algorithm does not necessarily converge with an unconstrained c_2.

The following two sections focus on two different approaches of how the topology in the part of the NN that calculates the constants can be chosen. A quick comparison is given and afterwards a hybrid topology is developed. In the last section the final topology is presented. The NN has to be extended to be able to produce more than one output.

4.2.1 Independent neural networks

This first choice of a topology for the calculation of the constants is a generalization of what was used in the previously mentioned proof of concept [15]. For this topology, several completely disconnected MLPs are used, one MLP per constant of the CGM. This topology is visualized in Figure 4.1. In theory, the number of layers and neurons per layer can now be chosen independently for every constant. This allows for an individual adjustment in order to achieve the desired accuracy for all constants. In practice only the same topologies for the three NNs have been tested. This was done for two reasons. Only the error of the final output is measured. Therefore, figuring out which constant should be improved in order to achieve the biggest decrease in the overall error is not possible. Also, the number of combinations for hyperparameter studies is very high. So increasing the number of neurons and layers for all constants until a desired error is achieved is all that was done.

The disadvantages when using several NNs for the constants are the time the NN needs for the training and the time it needs to calculate an output after it was trained. These times increase with the number of weights and bias values in the whole NN and therefore with the number of neurons. One MLP should be enough to model the mapping from the eight general inputs to all three constants. This follows directly from the universal approximation theorem. Hence, from a simple mathematical point of view there is no need for three independent NNs. At the same time it is not guaranteed that one NN is faster at calculating the constants because it might have to be larger than the three independent ones in order to achieve the same accuracy.

4.2.2 One fully-connected neural network

As hinted at the end of the previous section, it is also possible to calculate the constants in the CGM with only one fully-connected NN. The topology of an NN using that approach can be seen in Figure 4.3.

If there is any kind of correlation between the three constants, a single NN should be able to predict the constants with a lower total number of neurons than needed for the

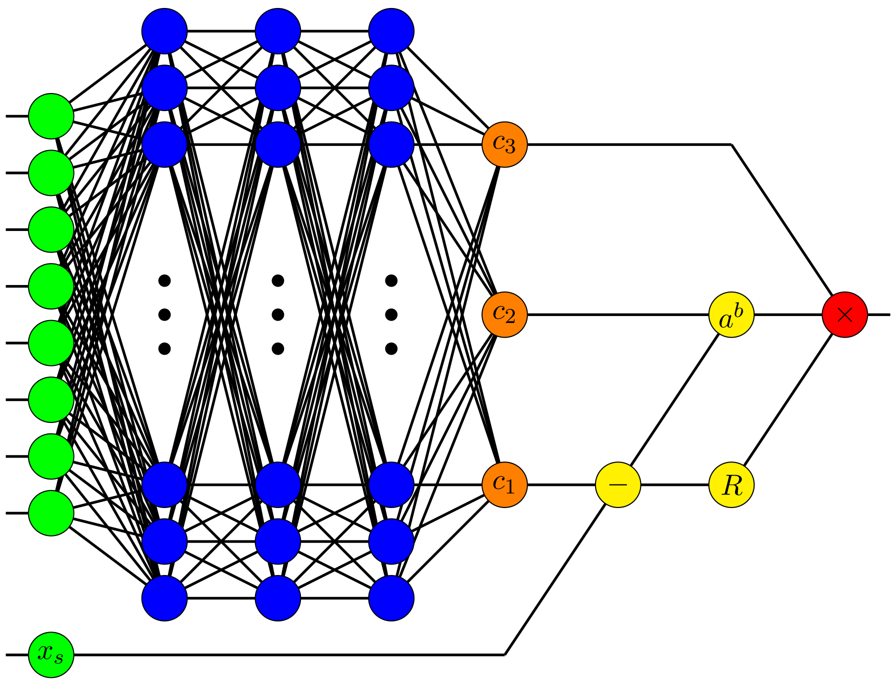

Figure 4.3: Layout of a CGMnet with a single neural network for the calculation of the constants. The neurons are colored as in Figure 4.1.

three NNs. A low number of neurons is always a desirable goal, since it reduces the time it takes the NN to predict turbulent fluxes. This is crucial for the integration of the NN in transport solvers. Theoretically, each mapping by the three NNs can be reproduced by one big NN with the same number of layers and three outputs. A guaranteed way to achieve this, is to choose the number of neurons in a layer of the big NN as the sum of the neurons in this layer of the three smaller NNs. The big NN can reproduce the smaller NNs by forming groups of neurons with the size of the layers of the smaller NNs and with the same weights and bias values. All weights between these groups have to be zero. This works in theory but in practice an NN that big would need more time to calculate one output than the three independent ones. Assuming the number of neurons in all layers is the same, one can calculate how big the single NN should be maximally to be as fast as the independent NNs. If the one NN needs to be larger than that to achieve the same accuracy as the independent NNs, the use of independent NNs should be preferred. The number of weights between the layers of the NN that calculate the constants is the same for both approaches if the following is fulfilled:

$$(L-1)N_c^2 = 3(L-1)N_d^2 \quad \Longrightarrow \quad N_c = \sqrt{3}N_d, \tag{4.1}$$

with:

L = number of hidden layers,

N_c = number of neurons in the hidden layers of the fully-connected NN,

N_d = number of neurons in the hidden layers of the disconnected NNs.

The part of the NN that calculates the constants is the most time consuming for NNs with more neurons in the hidden layers than input or output neurons. Hence, if condition (4.1) is fulfilled both approaches should need the same time to compute an output. If the second approach needs fewer neurons than calculated with (4.1), it is faster. In practice this is not true. Due to algorithms that are optimized for large NNs and parallelization on a GPU, the single NN can be larger than the criterion (4.1) allows and still be faster than the independent NNs.

For a surrogate model with one output, as used for the ETG turbulence type, both approaches have the same accuracy. This is true even if the big NN has only the same size as one of the disconnected NNs. This means a speedup of a factor three is achieved with the second approach. For more than one output this is no longer true. This problem will be discussed in more detail in the following section.

4.2.3 Neural network topologies for ITG and TEM

For ITG and TEM turbulence, more than one transport channel exists and therefore the surrogate model has to predict different fluxes. It is possible to achieve this by the use of independent surrogate models in the form of NNs, which is the state-of-the-art approach. However, if the fluxes are predicted by independent surrogate models the thresholds for the fluxes can be different. In [6] this is avoided by defining one leading flux, dividing the other fluxes by the leading flux and then training the NNs for these other fluxes on the fractions of the leading flux. After the training the predicted fractions are multiplied with the predicted leading flux. This way all fluxes are zero as long as the leading flux is zero and the thresholds are the same. Since the threshold is directly calculated in the CGMnets in the form of c_1 this trick is no longer needed if one surrogate model is used for all fluxes. Per output, one c_2 and one c_3 have to be calculated. The threshold c_1 is shared between all outputs.

The reason for the split into independent surrogate models and therefore independent MLPs in [6] was cross-talk between the outputs. Cross-talk describes the effect, that the predicted outputs influence each other. The same effect can be seen when a CGMnet with one big NN for all constants is used. Because the NN is fully-connected, a change of one weight in order to adjust one of the constants affects all other constants. This leads to problems if the fluxes strongly differ in magnitude. Then the predictions for all fluxes are biased. However, this does not happen if independent NNs are used for the calculation of the constants, as can be seen in Figure 4.4. The CGMnet predicting the turbulent fluxes for the left graph has one NN for all constants. Therefore, the outputs all influence each other and are shifted towards another. The CGMnet with one NN per constant in the right graph is able to predict the data more accurately, since the calculations of the constants are independent. With the goal to benefit from both

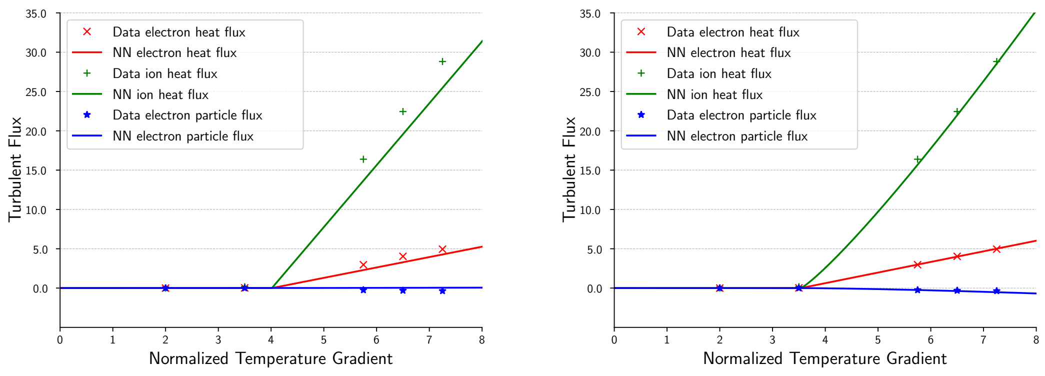

Figure 4.4: Comparison of a CGMnet with one NN for all seven constants (left) and with one NN per constant (right). The three colors refer to the three fluxes predicted by one surrogate model.

approaches one can create a hybrid topology.

The idea behind this hybrid approach is to have one or more fully-connected layers after the input layer. The last fully-connected layer then splits into one independent NN per constant that itself can consist of one or more layers of arbitrary size. This way there is a number of shared neurons and therefore some cross-talk. But also, there are neurons with weights that only influence one constant. The number of shared layers, independent layers and the number of neurons in them can now be chosen with the goal to optimize the speed for a given accuracy. Apart from the cross-talk a second reason for the use of at least one independent layer before the constants are calculated is motivated in the next paragraph.

For some combinations of the general input data the particle flux for ITG and TEM turbulence does not behave according to the CGM. This behaviour can be seen in Figure 4.5. The particle flux is still zero up to the critical gradient but afterwards it does not

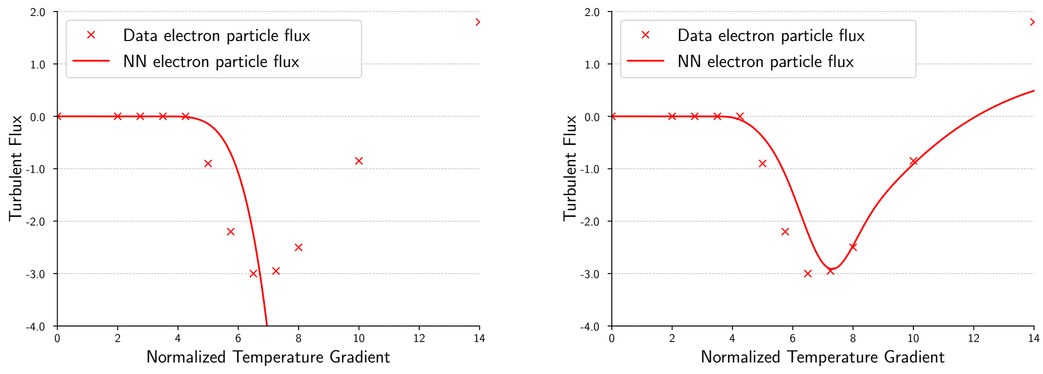

Figure 4.5: Particle flux does not behave as the CGM suggests. This behaviour for example appears for the general input: ${}^{R_0}/_{L_n} = 4, {}^{R_0}/_{L_{T_e}} = 8, {}^{T_i}/_{T_e} = 1, Z_{\text{eff}} = 1, \log(\nu^\star) = -3, q = 3, \hat{s} = 1, x = 0.69$. The NN in the left plot is forced to follow a CGM and therefore can not reproduce the data. The right one is not forced to follow a CGM.

follow the simple power structure. While this is expected from the underlying physics, the output of the NN, with the topology introduced so far, can only follow this power structure. Hence, the topology for the part of the NN that outputs the particle flux has to be adapted. One solution would be to find a generalization of the CGM. This is investigated with the following model for the particle flux f:

$$f(\boldsymbol{x}_g, x_s) = c_3 R(x_s - c_1)(|x_s - c_1|)^{c_2}(x_s - c_4). \tag{4.2}$$

The results however are still not promising and the possibilities for alternative models are many. A further investigation would therefore be very time consuming.

The best solution for this problem is found by going back to the state-of-the-art approach and using the direct output f_d of an NN to compute the particle flux. In order to still have the same threshold as for all other fluxes, the output of the NN is also multiplied with the rectifier in the output layer:

$$f(\boldsymbol{x}_g, x_s) = f_d(\boldsymbol{x}_g, x_s) R(x_s - c_1(\boldsymbol{x}_g)). \tag{4.3}$$

As a consequence, the loss in the particle flux can still influence the choice of the critical gradient. In the topology of the NN, this is realized as follows. For the particle flux no constants are needed. Instead, we want to multiply the output of an NN directly with the rectifier. This direct output of an NN cannot be calculated by the same fully-connected

NN as the constants, since it dependents on the special input. This is the second reason for the use of the hybrid topology. With this topology, the special input can be fed into the NN after the split of the common layers into independent layers. This way only the new f_d depends on the gradient that drives the turbulence and the other outputs of the NN are still constants in that dimension.

This final topology can be seen in Figure 4.6. The total number of three layers for

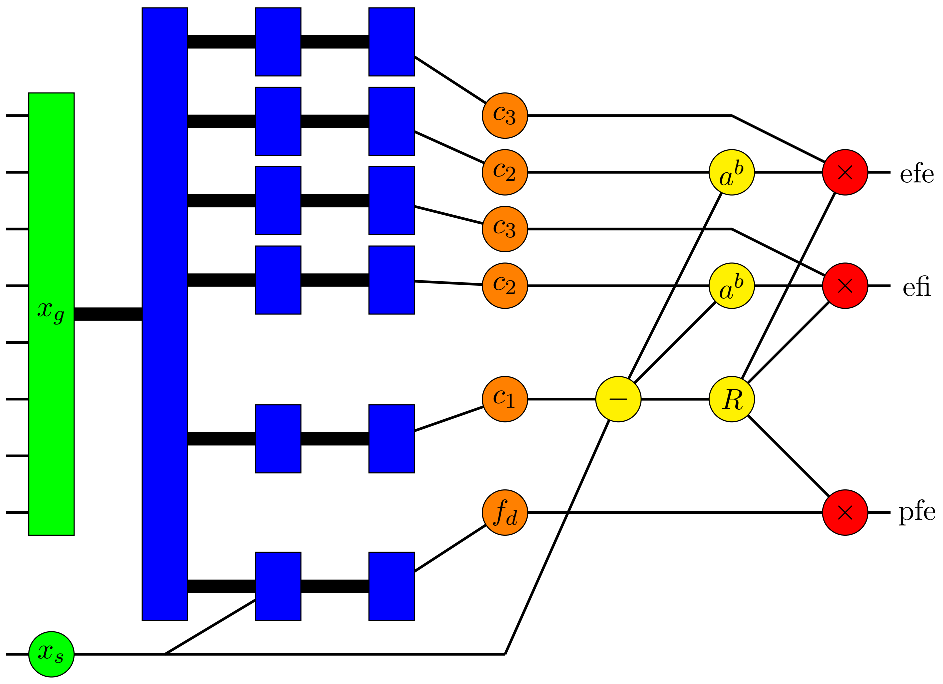

Figure 4.6: Final layout of the CGMnet for ITG and TEM. Rectangles symbolize whole layers where the height is not necessarily linked to the size of the layer. Thick black lines between layers indicate that these two layers are fully-connected. The outputs of this NN are the heat flux of the electrons (efe), the heat flux of the ions (efi) and the particle flux of the electrons (pfe).

the calculation of the constants is motivated through early small hyperparameter studies and due to the use of three layers in the current MLPs to approximate QuaLiKiz data [5]. Four layers lead to too slow progress during the learning and an NN with three layers has a considerably lower loss than an NN with only two layers, while still learning reasonably fast. In the hybrid topology one common layer and two individual layers are used since with only one individual layer the predicted particle flux does not have the desired accuracy. Results of detailed hyperparameter studies of the number of neurons in these layers and training times are presented in Chapter 6.

5 Implementation

All topologies introduced in the previous chapter are implemented in Python using the open source machine learning platform TensorFlow (TF) [18]. This was done with the high level API (application programming interface) Keras. With Keras first results for rather simple NNs can be achieved with only a few lines of code. However, implementing the more complex topologies of this work pushes Keras to its limits. Since Keras is directly integrated in TensorFlow and it is the recommended way to use TF, a similar performance as a direct implementation in TF is expected. Keras however is optimized for large NNs with few datapoints. The state-of-the-art NNs are directly implemented in TF. This leads to training times a factor 2.5 larger in the Keras implementation, even for the same NNs.

Keras manages all computing resources. Without any additional code the training is parallelized over all available cores of the CPU. If a GPU is available then the training is mostly executed on the GPU. The use of a GPU is very efficient since most of the calculations are matrix-vector multiplications. Parallelization over several GPUs can also be achieved. Then the training data is split up over the different GPUs. This needs additional code and has to be treated carefully since it is a rather new feature in Keras. Therefore, it has not been implemented in this work even though the NNs have been trained on systems with more than one GPU.

The training on the small seven-dimensional (7D) input dataset has been performed on a PC at DIFFER dedicated solely to this purpose. The hardware in this PC is specified in Table 5.1. The NNs for the full nine-dimensional (9D) dataset have been trained on the DAVIDE (Development of an Added Value Infrastructure Designed in Europe) supercomputer at CINECA in Italy. The components on each node of DAVIDE can be found in Table 5.1.

In both cases the NN has been trained on one GPU. For the use of one GPU on DAVIDE one fourth of the node has been charged. Therefore, also four CPU cores and one fourth of the memory have been reserved for the training of one NN. For the large dataset with nine input dimensions some optimizations had to be done. Else the 64 GB

Table 5.1: Components of the training PCs.

	PC at DIFFER	One node of DAVIDE
CPU	Intel Xeon E3-1220 v5 (4 x 3.5 GHz)	2 x IBM Power8 (2 x 8 x 4 GHz)
GPU	NVIDIA Quadro K620 (2 GB)	4 x NVIDIA Tesla P100 (4 x 16 GB)
Memory	16 GB	256 GB

of memory would not have been enough. The dataset takes up around 30 GB if it is loaded entirely into memory as a NumPy array. So any operation on it is not allowed to take up much more than twice that space. Shuffling it for example takes up twice the space. As already mentioned in Chapter 3 shuffling also accounts for a large portion of the wall clock time during training. When training on the smaller dataset, 9% of the time during training is spent on shuffling the dataset. On the full 9D dataset already 50% of the time one epoch takes is due to shuffling. It has been tried to avoid shuffling completely, but while the loss on the training data still drops each epoch, the validation loss rises. This is a clear sign of overfitting. Options like a shuffle frequency lower than each epoch or only partial shuffling in order to gain a speedup during the training have been investigated for the state-of-the-art NNs [19]. However, it has to be tested if the results can be transferred to the NNs developed in this work.

In Table 5.2 training times and evaluation times on the different computing resources are compared. They are also given for the small and large dataset. It can be seen that

Table 5.2: Training and evaluation (prediction) times of the TF implementation of an NN for TEM turbulence with 64 neurons in all layers and 1000 batches during training. The prediction of all three turbulent fluxes is counted as one prediction.

	Time for training	Time for predictions
CPU (4 cores) DAVIDE 7D	130 s / epoch	60.10 ms / 1000 predictions
CPU (4 cores) DAVIDE 9D	1525 s / epoch	59.62 ms / 1000 predictions
GPU DAVIDE 7D	15 s / epoch	92.61 ms / 1000 predictions
GPU DAVIDE 9D	300 s / epoch	92.67 ms / 1000 predictions
GPU DIFFER PC 7D	21 s / epoch	-

training an NN with a GPU is 5 to 8 times faster depending on the dataset. The speedup on the 9D dataset is smaller due to the shuffling. The use of the more powerful GPU on DAVIDE however does not affect the training times so much. Maybe the difference is larger for a bigger NN or for the full dataset. The memory on the PC at DIFFER however is not big enough to load the large dataset fully into memory. Therefore, a fair comparison between the GPUs on the large dataset is not possible. Training on the dataset with nine input dimensions takes 12 to 20 times as long. The larger dataset has a factor 40 more training points. So even a 20 times slower training is not too bad. The evaluation times of the Python implementation with TensorFlow given here are not that important. While it is crucial that the NN can predict fluxes very fast, the NNs will be implemented in FORTRAN for the use in transport solvers. Hence, the evaluation times of the FORTRAN implementation will be more important.

6 Hyperparameters

For the training of one NN, between 15 and 25 parameters have to be chosen. Most of these parameters are either never changed and adopted from [5], or small hyperparameter studies for one parameter are carried out while keeping the other parameters constant. This way, a suitable value for this single parameter is found. Mostly for the topology and the regularization of the NN, larger hyperparameter studies have been performed. For these studies the small dataset has been used. Training on this dataset is 20 times faster than on the large dataset. Also, the smaller dataset takes up considerably less memory. Hyperparameters from the NNs trained on the 7D dataset can be used most of the time for the NNs trained on the 9D dataset. While there exist some rules of thumb on hyperparameters like the number of layers and the number of neurons in those layers, it was found that those rules are not applicable for our use case. This is due to the fact that those rules were developed for deep NNs that are trained on fewer samples and in higher dimensions.

In the next sections, the choice of four hyperparameters is discussed in more detail. One obstacle that makes choosing the perfect hyperparameter difficult, is the randomness in the choice of the weights and bias values. This is due to the random initialization of the NN, the shuffling of the data and also due to the random division into training, validation and test data. Since the optimization algorithm normally does not find the global minimum, but only a local minimum, it is hard to make a general statement about which NN has the better hyperparameters. A possibility to circumvent this problem, is to train multiple NNs with the same hyperparameters and analyze the distribution of their losses. However, this increases the number of NNs that have to be trained highly.

6.1 Data standardization

Normally the training input and output for an NN is standardized or normalized in some way. This leads to better performance of the NN in most cases. For the previous works, both input and output are scaled to a standard deviation of one and shifted to a zero mean:

$$x_{\text{new}} = \frac{x_{\text{old}} - \bar{x}_{\text{old}}}{\sigma}. \tag{6.1}$$

This is the standard approach, it is also known as Gaussian normalization. However, shifting the output is problematic for the CGMnets. These NNs can only predict fluxes that are zero in the stable region. After shifting the data, it is no longer zero in this region. Several solutions exist for this problem. In order to still be able to use the Gaussian normalization one may add a bias value to the output neurons. With this bias value, the NN can shift its output and is therefore able to produce any output in the stable region. The other two solutions are only scaling the output or using no standardization of the output at all. All three possibilities have been implemented and compared. Training on unscaled output data leads to a slightly slower learning in the beginning. Both other approaches behave similarly in the MSE. No clearly better one exists. This learning behaviour can be seen in Figure 6.1 on the left side. For this work the output is only scaled. This guarantees an exactly zero output in the stable region, which is helpful if the output is processed further.

Since the optimization algorithm minimizes the sum of the mean square errors of the different outputs, it is also of interest how the outputs are scaled relative to each other. If they are all scaled by their individual standard deviations, their MSEs are scaled by their squared standard deviations. After the training, the scaled individual MSEs are typically of the same order. If big differences exist in the standard deviations, big differences also occur in the absolute mean square errors after rescaling them. Indeed, scaling the outputs by the mean of their standard deviations leads to better results in the total unscaled MSE. A comparison of the MSE during the training with those two approaches is displayed in Figure 6.1 on the right. We decide however that a good approximation of the behaviour near the threshold for all fluxes is more important than a low absolute error. Therefore, all outputs are scaled down independently.

6.2 Batches

The batch hyperparameter has already been introduced in Chapter 3. It has also been motivated why it is important to optimize it. The number of batches has no big influence on the quality of the output of a trained NN since it does not change the capabilities of an NN. It has however the largest impact of all investigated hyperparameters on the time necessary for the training of the NN. The results of a small hyperparameter study can be seen in Figure 6.2. One can see the dramatic increase in the speed with which

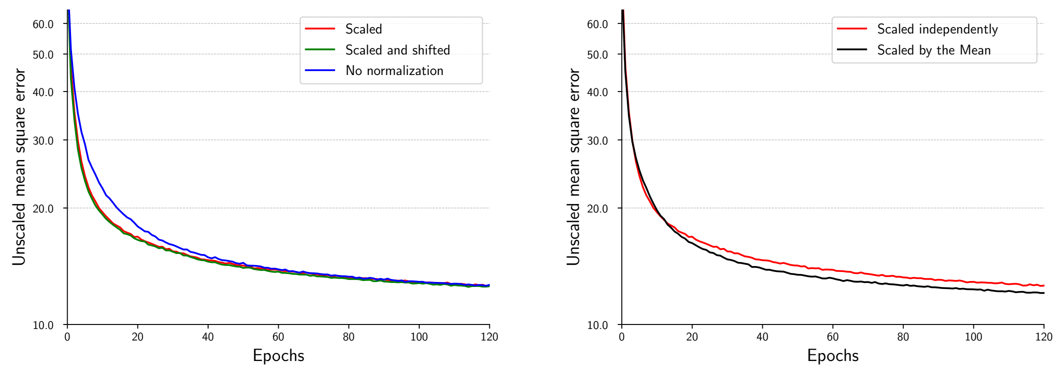

Figure 6.1: Mean square error during the training with differently standardized data.

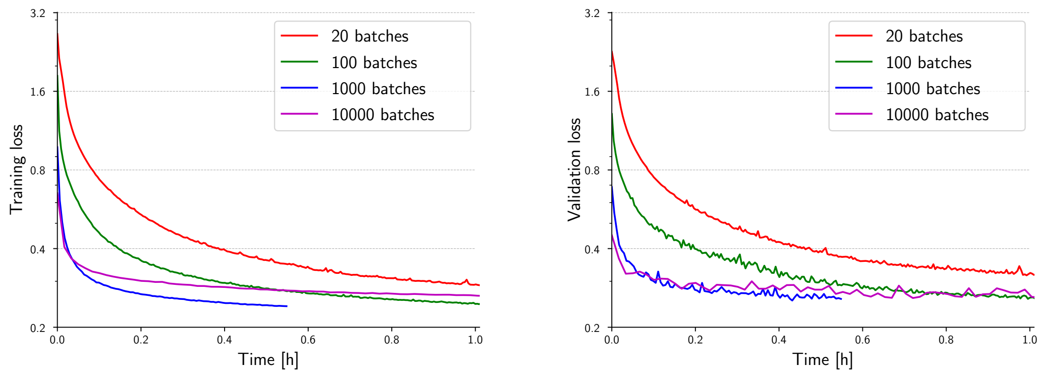

Figure 6.2: Progress of the loss over wall clock time during the training for different number of batches. Left: loss on training data, right: loss on validation data

the NN learns with more batches. The NN that is trained with only 20 batches reaches in the end the same loss as the one that is trained with 1000 batches. It does however take at least ten times longer for that to happen. It can also be seen that if 10000 batches are used the validation loss starts to become more erratic. Due to the lower number of samples in each batch, the optimization algorithm does no longer perform as well as with fewer batches. This leads to a higher final loss. Also, due to the higher number of batches, each epoch starts to take up considerably more time. The time spent per epoch is listed in Table 6.1. With 1000 batches each batch still consists of nearly 6000 samples. This number is two orders of magnitudes larger compared to other standard applications of NNs like handwritten-digit recognition. There, often batch sizes of about only 30 samples are recommended. This shows how important it is to adjust

Table 6.1: Training times per epoch on 7D data.

Number of batches	20	100	1000	10000
Time in seconds	15.72	16.2	21	60.28

the hyperparameters of an NN to the specific use case and the data.

6.3 Number of neurons

In contrast to the hyperparameter investigated in the previous section, the number of neurons in each layer has a major influence on the *approximation capabilities* of the NN. The number of neurons also affects the training time and in contrast to the previous hyperparameter also the evaluation time of the NN for future predictions. Therefore, an optimal number of neurons in each layer has to be found. This number should not be too large in order to produce fast predictions, but still large enough for the predictions to be accurate. As mentioned at the end of Chapter 4, one common layer and two individual layers for each constant are used in the final topology. The number of neurons in the individual layers are all chosen to be the same. There is no obvious reason to vary them and testing different combinations would take too much time. The number of neurons in the common layer however is chosen separately from the others. A hyperparameter study for the two parameters "neurons in the common layer" and "neurons in the individual layers" is performed. Both parameters are varied from 8 to 128. They are changed by doubling them until the maximum of 128 is reached. For all 25 different combinations one NN is trained. The validation loss of six of these NNs can be seen in Figure 6.3.

This figure shows that the minimum configuration of only eight neurons in all layers is not enough. While it is still able to learn something from the data, it has about twice the validation loss of the best NN. If one prefers a very fast NN, this one can still produce reasonable indications of the turbulent fluxes. But especially the prediction quality of the complex particle flux is poor in most cases. The other fluxes are often underestimated. The NNs, with their loss plotted in green and blue, are to compare a doubling of the neurons in the common layer and in the individual layers. As expected, a doubling of the number of neurons in the individual layers leads to a higher drop in the validation loss than a doubling of the number of neurons in the common layer does. This is because there are 12 individual layers in the final layout. Hence, by doubling the number of neurons in these layers the total number of degrees of freedom in the NN

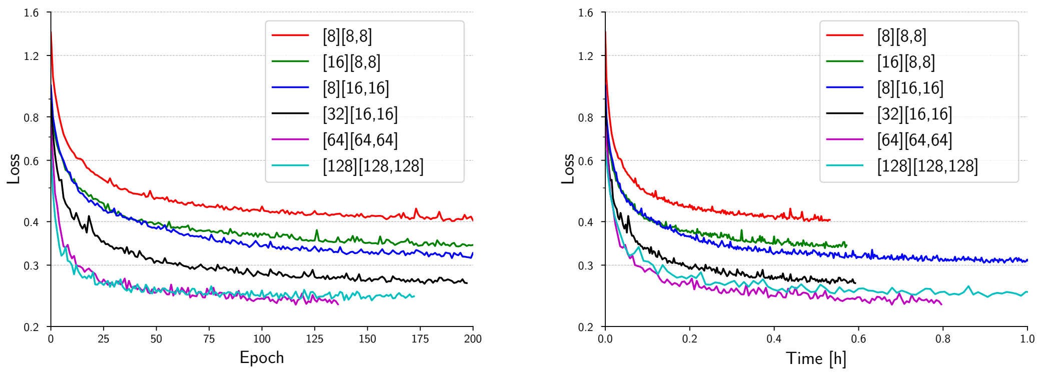

Figure 6.3: Progress of the validation loss over epochs (left) and wall clock time (right) during the training of different topologies, for the calculation of the constants. The first number in the legend is the number of neurons in the common layer. The other two are the neurons in the individual layers. The NNs stop their training according to an early stopping criterion. This is explained in the next section.

rises a lot more than by doubling the number of neurons in the single common layer.

Together with the number of degrees of freedom, the training time for the NN rises. While doubling the number of neurons in the common layer leads to a rise of 1.2% wall clock time per epoch, a doubling of the neurons in the individual layers results in 12.5% more time spent in each epoch. This suggests that for a fast NN with good approximation properties there should be more neurons in the common layer than in the other layers. The last three NNs in the legend perform very similarly in terms of validation loss. The last one is the largest investigated NN in this work. While it seems to outperform the NN with half as many neurons in all layers in the beginning, the smaller NN converges to a lower minimum. Also, if the validation loss is plotted over the actual time instead of over the epochs, it is clear that the large NN is too big. The smaller NN achieves a lower validation loss at all times. The NN with 32 neurons in the common layer and only 16 neurons in the other layers appears to have a higher validation loss than the two bigger NNs. While this is true, the right plot shows how small this difference really is. This NN only takes more epochs to converge. Since it only needs half the time per epoch this is not a problem though.

The NN with 64 neurons in all layers is the one with the lowest validation loss of the NNs that have been trained for this hyperparameter study. This topology was chosen for the training on the full dataset. All NNs smaller than the one with 32 neurons in the common layer and 16 in the individual layers performed noticeably less well, while all bigger NNs did converge to a very similar validation loss. For a better comparison of these bigger NNs, more NNs with the same topology would have to be trained to quantify the statistical variations due to the random initialization and shuffling during the training. If the evaluation of the NN with 64 neurons in all layers proves to be too slow for the use in transport solvers, an NN with 32 neurons in the common layer and 16 neurons in the individual layers could be a good alternative.

6.4 Regularization

Regularization is used in the field of artificial neural networks in order to produce smoother outputs and prevent the NN from overfitting. There are several approaches on how an NN can be regularized. One of the most basic ones is the L_2 regularization. It is the only regularization technique used in this work. In Chapter 3 it was already briefly introduced together with the regularization coefficient in equation (3.3). This coefficient is the hyperparameter that is optimized.

Early stopping is often also considered as regularization of the NN. Early stopping is a technique where the training of the NN is stopped before the maximal number of training epochs is reached. This is done because it is assumed that the optimal state of the NN is already reached. There are different criteria on the basis of which the training can be stopped early. Here, only a stop in improvement of the validation loss over a given number of epochs is used. This number of epochs is also considered to be a hyperparameter. Since the choice of this parameter does not affect any other hyperparameter, it was adjusted during the other hyperparameter studies. A value of 20 was found to produce good results. This value was found by lowering it from 1000 until the NN stopped its training at a reasonable point. It is as all the other hyperparameters very problem dependent. In this work the early stopping only once stopped the training of an NN due to a rising validation loss and therefore prevented overfitting. It was primarily used to determine when the optimization algorithm converged.

The NNs with the specialized topology are naturally very regularized. In the dimension of the temperature gradient that drives the turbulence, the NN cannot overfit because of the CGM. However, in other dimensions the NN can overfit. For example the constants the NN calculates could become very sensitive to small changes in the general input. It is therefore not clear if a regularization is even needed and if it is, how strong it should be. The particle flux however is directly calculated from the output of an NN and therefore not regularized at all. Hence, the coefficient of the L_2 regularization for the part of the NN with a built-in CGM is chosen independent of the coefficient for the part with a direct NN output.

The coefficient of the regularization in the total loss has to be very small since also the mean square error term becomes small rather fast. For the part of the NN that predicts the particle flux a regularization coefficient of 10^{-4} is found to produce good results, as can be seen in Figure 6.4. With a higher coefficient, the NN no longer captures all the

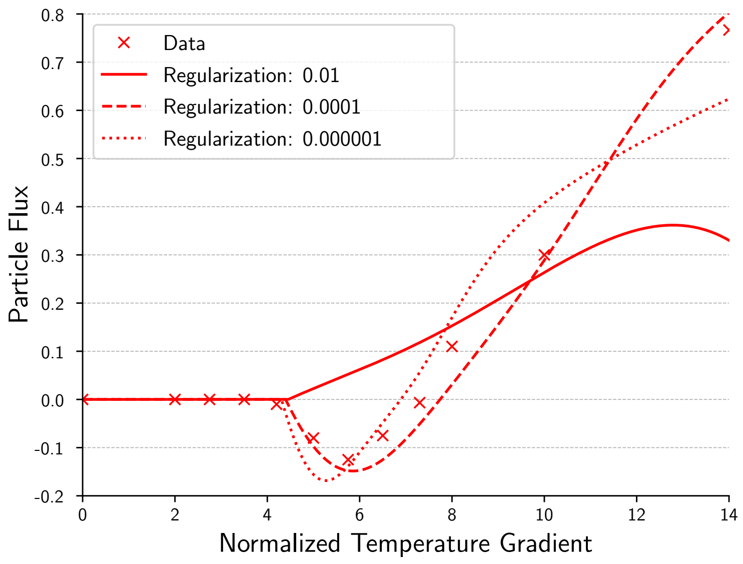

Figure 6.4: Particle flux predicted by NNs with different regularization coefficients during the training compared to the actual data.

relevant features in the output but instead focuses too much on minimizing the weights. With a smaller coefficient the predicted particle flux becomes too erratic and the NN overfits the data.

Choosing the regularization coefficient for the rest of the NN was more complicated. While a coefficient of 10^{-6} worked well on the smaller dataset with seven inputs, it led to problems on the large dataset. On the 9D dataset a higher fraction of datapoints is in the stable region of the turbulence. Therefore, a higher share of the outputs are zero. These can easily be predicted accurately by the CGMnets. This leads to a lower mean square error on the large dataset. Hence, with the same coefficient for the regularization the optimization algorithm focuses too much on keeping the weights of the NN small. Even though this is not desired, it leads to an interesting discovery. Instead of smaller weights for all constants, the algorithm reduces all weights in the layers for the calculation of the exponents (c_2) for all fluxes to zero. This can be seen in Figure 6.5. In this figure the progress of the distribution of the weights in one of the hidden layers for the calculation of one c_2 is displayed. This distribution is plotted as color coded and normalized histograms per epoch. It can be seen that the uniform distribution collapses to zero after a few epochs if an L_2 regularization coefficient of 10^{-6} is used. The same behaviour can be seen in all hidden layers for the calculation of all exponents. In the lower two plots the calculated exponents are displayed in the same way. Here the effect of the collapsed weights can be seen.

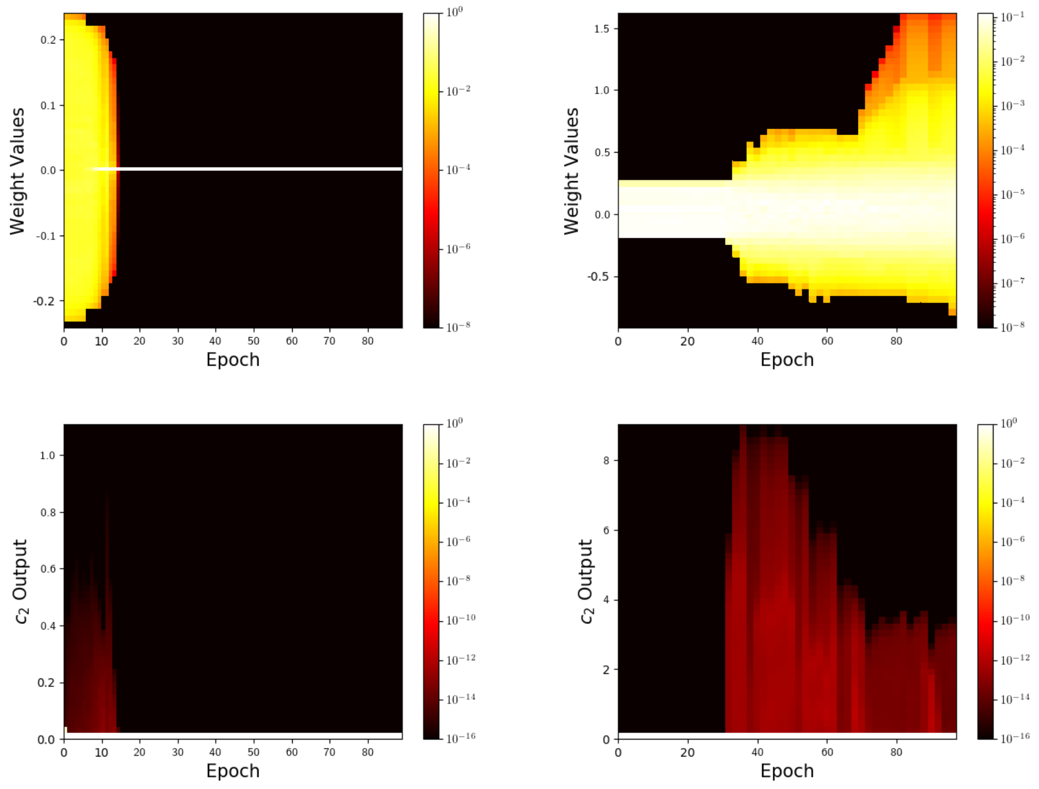

Figure 6.5: Top: Distribution of the weights in the second hidden layer for c_2 with an L_2 regularization coefficient of 10^{-6} (left) and a regularization coefficient of 10^{-11} (right). Bottom: Distribution of c_2 for the same regularization coefficients as the ones above.

These zero weights together with small bias values lead to a value for c_2 that is constantly zero. Therefore, the predicted fluxes only grow linearly in the unstable region. This indicates that the exponent is the constant with the least effect on the prediction quality of the NN. One could set the exponent to be one in a more reduced model of the CGM. However, for some inputs, an NN that predicts a flux that can bend, is better at reproducing the data and especially predicts a more accurate threshold. An example is given in Figure 6.6. Additional testing shows that even NNs which are trained completely

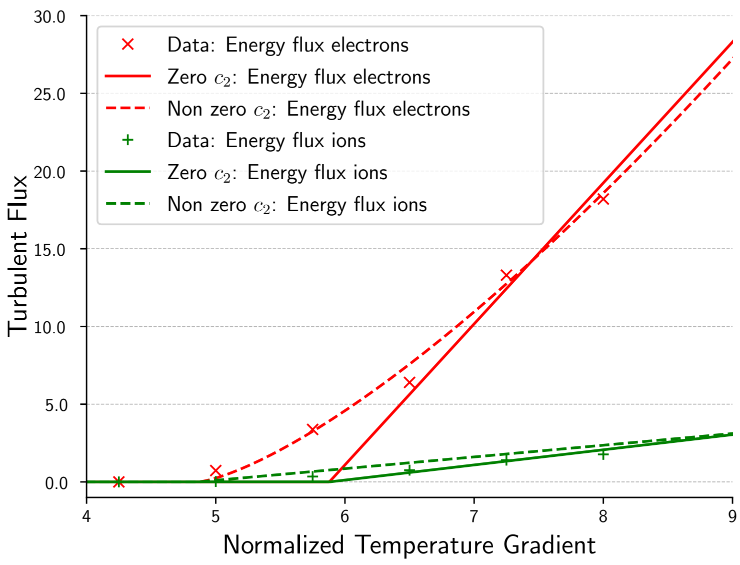

Figure 6.6: Ion and electron heat fluxes predicted by an NN with a c_2 that is forced to zero because of the regularization and an NN that can have a c_2 different from zero.

without an L_2 regularization in the layers for the calculation of the constants for the CGM do not overfit. Also, outside the dimension of the special input the predicted fluxes are as smooth as the fluxes computed by the current state-of-the-art NNs.

7 Results

While the previous chapter already presents first results, in this chapter a short summary of the performance of the neural networks created during this work is given. They are compared against the current state-of-the-art NNs. Furthermore, the predicted critical gradient behavior is compared to the literature.

7.1 Comparison between fully-connected NNs and CGMnets

It is difficult to compare the quality of the predicted turbulent fluxes of the CGMnets with the turbulent fluxes predicted by the current state-of-the-art NNs (QLKNN). For the current state-of-the-art NNs measures of merit are defined to assess their performance [6]. Four of those six measures are not necessary for the CGMnets. The CGMnets perform perfectly regarding those measures because of their topology. The measures "no threshold fraction" and "threshold misprediction" can be used. "No threshold fraction" describes the percentage of general input combinations for which QuaLiKiz predicts a critical gradient within the range of special inputs but the neural network does not. The measure "threshold misprediction" is the mean difference between the critical gradients calculated by QuaLiKiz and the NN. Additionally, the NNs can still be compared by the root mean square (RMS) error of the predictions on the known dataset. In Table 7.1

Table 7.1: Comparison of the CGMnets and the current state-of-the-art NNs using measures of merit for the leading flux.

Measures	ITG: q_i		TEM: q_e		ETG: q_e	
	QLKNN	CGMnet	QLKNN	CGMnet	QLKNN	CGMnet
RMS error	2.3	3.6	1.8	2.0	2.0	1.2
No thresh frac [%]	4.2	11.8	14.3	23.7	3.3	6.9
Thresh mispred [%]	-0.26	-0.12	-0.31	0.32	-0.38	-0.33

those measures are listed for the latest CGMnets for ITG, TEM and ETG turbulence.

They are calculated for the leading heat flux and compared to the measures of the QLKNNs given in [6]. The values in Table 7.1 were not calculated using the whole dataset. Instead, 5% of the different combinations for the general inputs were used. It is interesting to see that the RMS error of the CGMnets is higher than the one of the QLKNNs for ITG and TEM turbulence, but it is lower for ETG turbulence. One possible explanation for those values is the fact that the CGMnets for ITG and TEM predict three fluxes. This in combination with the decision introduced in Section 6.1 to scale all fluxes with their individual standard deviations before training, causes the leading flux to have the highest RMS error. The values for the measure "no threshold fraction" is higher for all CGMnets. A visual inspection of the cases where QuaLiKiz predicts a critical gradient and the CGMnet does not, revealed that often the critical gradient predicted by QuaLiKiz is near the upper boundary of the range of temperature gradients. The last measure shows that for ITG turbulence the critical gradients predicted by the CGMnet are more accurate than the ones of the current QLKNN. For TEM turbulence the critical gradients of the CGMnets are on average too high while the ones of the QLKNN are too low. They are however on average off by the same amount. For ETG turbulence the "threshold misprediction" has nearly the same value.

One of the measures of merit for which the CGMnets perform perfectly is the "unstable zone smoothness". It tries to capture how strong the predictions fluctuate in the unstable zone. This is important for the integration in an implicit transport solver. The solver can take longer to converge due to fluctuations of the predicted turbulent fluxes. Since the CGMnets do not fluctuate in the unstable zone, a faster convergence of implicit transport solvers is expected.

A visual comparison by producing lots of plots and comparing the predictions of both types of networks is also possible but can be quite subjective. In a first visual inspection, through slices in the dimension of the critical gradient, both approaches seem to perform comparable. For some slices the CGMnets outperform the fully-connected ones as can be seen in Figure 7.1. For most turbulent fluxes however there is no large difference between the approaches, but for the particle flux in regimes with a second zero crossing the CGMnets are often more accurate and less erratic. If the turbulent fluxes are plotted with a different x-axis than the temperature gradient that drives the turbulence, both NNs predict similar fluxes and it is difficult to tell which one is better. Also, the fluxes of the CGMnets do not seem to be wobbly in those dimensions even though the regularization in those dimensions is lower than the one of the state-of-the-art NNs.

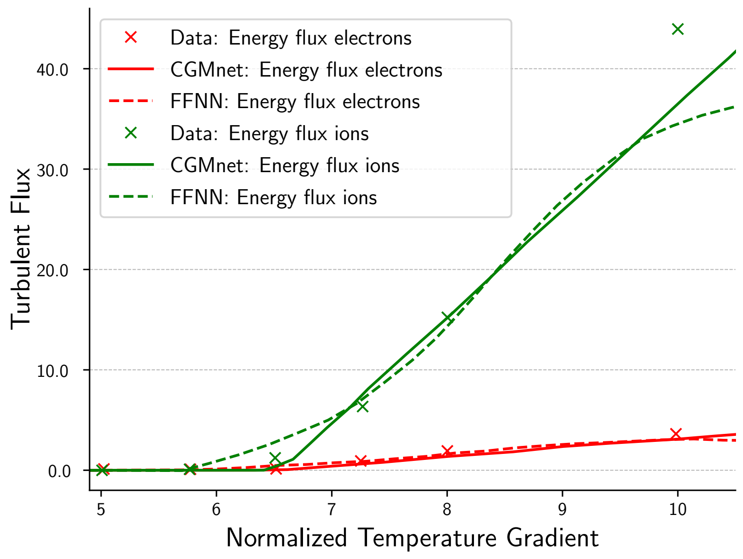

Figure 7.1: Comparison of CGMnet and QLKNN. The CGMnet has sharper threshold because no L_2 regularization is needed. Also, the predictions of the CGMnet fluctuate less.

7.2 Comparison of the threshold of the CGMnet with the literature

In [13] a formula for the critical gradient for ETG turbulence is derived. This is done through the analysis of a large amount of linear gyrokinetic simulations. The resulting formula written in the previously introduced parameters is:

$$\left(\frac{R_0}{L_{T_e}}\right)_{\text{crit}} = \max\left\{\left(1 + Z_{\text{eff}}\frac{T_e}{T_i}\right)\left(1.33 + 1.91\frac{\hat{s}}{q}\right)\left(1 - 1.5\frac{R_{\text{min}}}{R_0}x\right),\ 0.8\frac{R_0}{L_n}\right\}. \quad (7.1)$$

Since the critical gradient is directly calculated within the CGMnet it can be extracted and compared to this formula. The results of this comparison can be seen in Figure 7.2. The limits of the x-axis are chosen such that they are in the regime for which equation (7.1) was derived and also within the input interval of the data the NN is trained on. It can be seen that the predicted critical gradient of the NN is constantly higher than the one calculated by the formula. This is due to the different gyrokinetic codes that were used to create the dataset in both projects. Apart from this shift both models predict very similar critical gradients. The trend of both predicted critical gradients is the same. This is true for all input dimensions. The shift in the critical gradient is going to be the

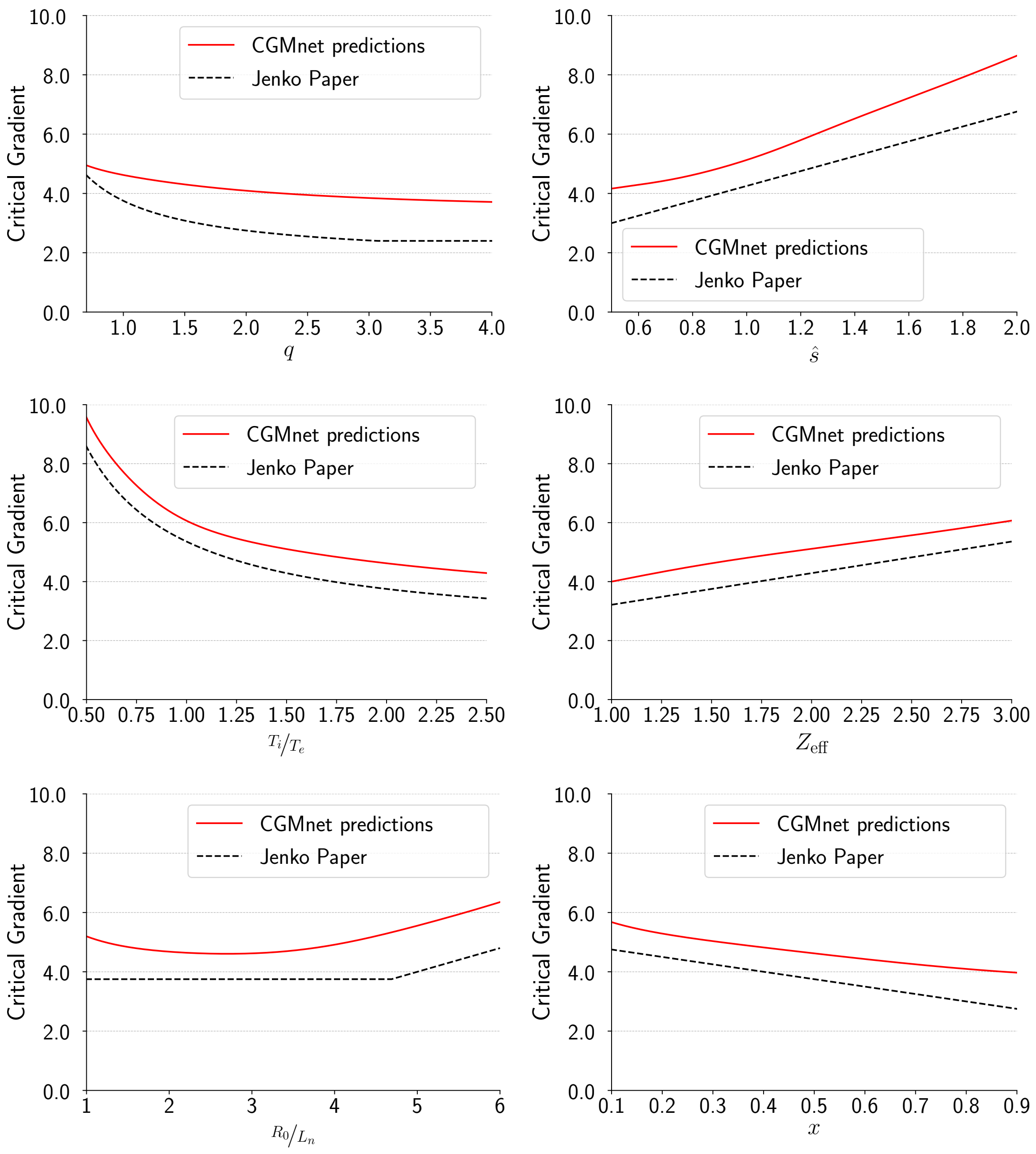

Figure 7.2: Comparison of the critical gradient predicted by the CGMnet and the critical gradient calculated with equation (7.1). The general input parameters are fixed at the following values: $R_0/L_n = 3, R_0/L_{T_i} = 2, T_i/T_e = 2, Z_{\text{eff}} = 1.5, \log(\nu^\star) = -5, q = 1, \hat{s} = 0.8, x = 0.5$

subject of further investigations outside the scope of this work. The fact that the trends are so similar further validates the turbulent fluxes predicted by the CGMnets.

8 Outlook

For a perfect comparison of the newly developed CGMnets during this work and the state-of-the-art NNs, the new CGMnets need to be implemented in the integrated modelling suite JINTRAC [20] or the transport code RAPTOR [21]. This implementation is done but not tested. With the CGMnets integrated in the transport codes the prediction quality could be compared with actual experiments in tokamaks. Also, a fair comparison of the prediction speed of the different NNs would be possible. Furthermore, the convergence of the transport solvers could be analyzed. Since RAPTOR uses an implicit transport solver, it would be interesting to see if it converges faster with the smoother flux predictions of the CGMnets.

In future work it could also be analyzed if the NNs can be trained faster when they are initialized differently. The distribution of the weights in a trained NN could be used as distribution for the randomly initialized weights of a new NN. With a speed up in the training more thorough hyperparameter studies can be performed leading to better suited NN topologies.

Since the prediction quality around the threshold is most important for the performance in integrated modelling, the focus of the NN during the training can be shifted towards that region through instance weighting of the training data. Instance weighting describes a process in which every input-output data point is assigned a weight according to its importance. In the calculation of the loss function the error made at this data point is then multiplied with the weight. Hence, the optimization algorithm focuses more on reducing the error for datapoints with large weights. With the use of instance weighting, the high fluxes could be left in the training dataset instead of filtering them out. They have been excluded to achieve more accurate predictions with the QLKNN around the critical gradient. This is no longer necessary if instance weighting is used. For the CGMnets it may not even be necessary to filter out high fluxes without instance weighting. This is going to be the topic of future research.

For most general inputs it is possible to estimate the critical gradient using the data. It can be set between the highest stable temperature gradient and the lowest unstable temperature gradient for a fixed general input. Using this estimate also the measure "threshold misprediction" was calculated. A further improvement in the training pipeline could be achieved by directly including the estimated critical gradient in the data and training the NN on this additional data. Using this approach, the "threshold misprediction" could be decreased further.

Bibliography

[1] IPCC. *Climate Change 2014: Mitigation of Climate Change.* Ed. by O. Edenhofer, R. Pichs-Madruga, Y. Sokona, E. Farahani, S. Kadner, K. Seyboth, A. Adler, I. Baum, S. Brunner, P. Eickemeier, B. Kriemann, J. Savolainen, S. Schlömer, C. von Stechow, T. Zwickel, and J. Minx. Contribution of Working Group III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change. Cambridge University Press, 2014.

[2] W. Horton. "Drift waves and transport". In: *Rev. Mod. Phys.* 71 (Apr. 1999), pp. 735–778. DOI: `10.1103/RevModPhys.71.735`.

[3] J. Citrin, C. Bourdelle, F. J. Casson, C. Angioni, N. Bonanomi, Y. Camenen, X. Garbet, L. Garzotti, T. Görler, O. Gürcan, F. Koechl, F. Imbeaux, O. Linder, K. van de Plassche, P. Strand, G. Szepesi, and J. Contributors. "Tractable flux-driven temperature, density, and rotation profile evolution with the quasilinear gyrokinetic transport model QuaLiKiz". In: *Plasma Physics and Controlled Fusion* 59.12 (Nov. 2017), p. 124005. DOI: `10.1088/1361-6587/aa8aeb`.

[4] K. L. van de Plassche and J. Citrin. *QLKNN10D training set.* Version 1.0. [Dataset]. Oct. 2019. URL: `https://doi.org/10.5281/zenodo.3497066`.

[5] K. L. van de Plassche. "Realtime capable turbulent transport modeling using neural networks". Master Thesis. DIFFER, Aug. 2017.

[6] K. L. van de Plassche, J. Citrin, C. Bourdelle, Y. Camenen, F. Casson, V. Dagnelie, F. Felici, A. Ho, S. Mulders, and J. Contributors. "Fast modelling of turbulent transport in fusion plasmas using neural networks". In: *Physics of Plasmas* 27.2 (2020), p. 022310. DOI: `10.1063/1.5134126`. eprint: `https://doi.org/10.1063/1.5134126`. URL: `https://doi.org/10.1063/1.5134126`.

[7] J. Wesson. *Tokamaks.* Oxford University Press, 2004.

[8] R. J. Goldston and P. H. Rutherford. *Introduction to Plasma Physics.* Taylor & Francis Group, 1995.

[9] A. J. Brizard and T. S. Hahm. "Foundations of nonlinear gyrokinetic theory". In: *Rev. Mod. Phys.* 79 (Apr. 2007), pp. 421–468. DOI: `10.1103/RevModPhys.79.421`.

[10] X. Garbet, Y. Idomura, L. Villard, and T. H. Watanabe. "Gyrokinetic simulations of turbulent transport". In: *Nuclear Fusion* 50.4 (Mar. 2010), p. 043002. DOI: `10.1088/0029-5515/50/4/043002`.

[11] C. Bourdelle, J. Citrin, B. Baiocchi, A. Casati, P. Cottier, X. Garbet, F. Imbeaux, and J. Contributors. “Core turbulent transport in tokamak plasmas: bridging theory and experiment with QuaLiKiz”. In: *Plasma Physics and Controlled Fusion* 58.1 (Dec. 2015), p. 014036. DOI: `10.1088/0741-3335/58/1/014036`.

[12] C. Bourdelle. “Turbulent Transport in Tokamak Plasmas: bridging theory and experiment”. Habilitation à diriger des recherches. Aix Marseille Université, Jan. 2015. URL: `https://tel.archives-ouvertes.fr/tel-01113299`.

[13] F. Jenko, W. Dorland, and G. W. Hammett. “Critical gradient formula for toroidal electron temperature gradient modes”. In: *Physics of Plasmas* 8.9 (2001), pp. 4096–4104. DOI: `10.1063/1.1391261`.

[14] M. A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015. URL: `http://neuralnetworksanddeeplearning.com` (visited on 02/01/2020).

[15] D. Schaefer. “Hybrid Neural Networks in Nuclear Fusion Transport Modelling”. Master Thesis. IPP, 2019.

[16] K. Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: `10.1016/0893-6080(91)90009-T`.

[17] D. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (Dec. 2014).

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[19] R. J. M. Winkeler. “Hyperparameter optimization for the QuaLiKiz neural network training pipeline”. Internship report. DIFFER, Dec. 2019.

[20] M. Romanelli, G. Corrigan, V. Parail, S. Wiesen, R. Ambrosino, P. Belo, L. Garzotti, Harting, F. Köchl, T. Koskela, L. Lauro-Taroni, C. Marchetto, M. Mattei, E. Militello-Asp, M. Nave, S. Pamela, A. Salmi, P. Strand, and G. Szepesi. “JINTRAC: A system of codes for integrated simulation of Tokamak scenarios”. In: *Plasma and Fusion Research* 9 (Jan. 2014). DOI: `10.1585/pfr.9.3403023`.

[21] F. Felici and O. Sauter. “Non-linear model-based optimization of actuator trajectories for tokamak plasma profile control”. In: *Plasma Physics and Controlled Fusion* 54.2 (Jan. 2012), p. 025002. DOI: `10.1088/0741-3335/54/2/025002`.