Minimum levels of high energy particle bombardment on fusion reactor vessels:

Towards a computational multi-parameter scan for automated data reduction ("big data") applications

Christof Backhaus

February 2020

Contents

1 Introduction **3**
1.1 Fusion Devices . . . 3
1.2 PROCESS Systems Code . . . 4
1.3 Reduced Model Approaches . . . 5

2 Model **7**
2.1 EIRENE . . . 7
2.2 Plasma Profiles . . . 8
2.2.1 Density Profile . . . 10
2.3 Choice of sampling set . . . 11

3 Methods **13**
3.1 Neural Networks . . . 14
3.1.1 General Introduction To Neural Networks . . . 14
3.1.2 Functionality . . . 15
3.1.3 Training . . . 17
3.1.4 Hyperparameters . . . 24

4 Results **31**
4.1 Neural Networks . . . 31
4.1.1 Architecture . . . 31
4.1.2 Training Set Size . . . 34
4.1.3 Derivatives . . . 35

Abstract

Chapter 1

Introduction

The design process for a facility like a fusion power plant takes into account a manifold of aspects. Thereunder a cost analysis for the fusion device. To make an estimate of the cost analysis one has to consider the lifetime of machine parts. Most prominently the divertor and first wall suffer from shortened life spans due to erosion. Which is partly due to neutral particle induced sputtering.
To include considerations like these in the design of a power plant one uses so called systems codes like PROCESS [14]. These codes focus on optimizing design parameters of large scale systems like power plants, which consist of many smaller subsystems. Due to the amount of subsystems the need arises to simplify models in order to achieve reasonable run times for systems codes. The following work is concerned with deducing a fast surrogate in place of a simulation for the sputtering rate of a fusion device component.
The following chapter gives a brief overview of the motivating applications while also introducing the concept of reduced model approaches via machine learning algorithms. Furthermore it considers which methods are most applicable in the given situation.

1.1 Fusion Devices

Fusion technology has been an ongoing field of research for almost one century. The earliest records of fusion research go back to the 1920s when Francis William Aston discovered the potential energy gain of combining hydrogen atoms into helium atoms. Later in the 1920s Arthur Stanley Eddington proposed the proton-proton chain reaction as the primary working mechanism of the sun. add citation

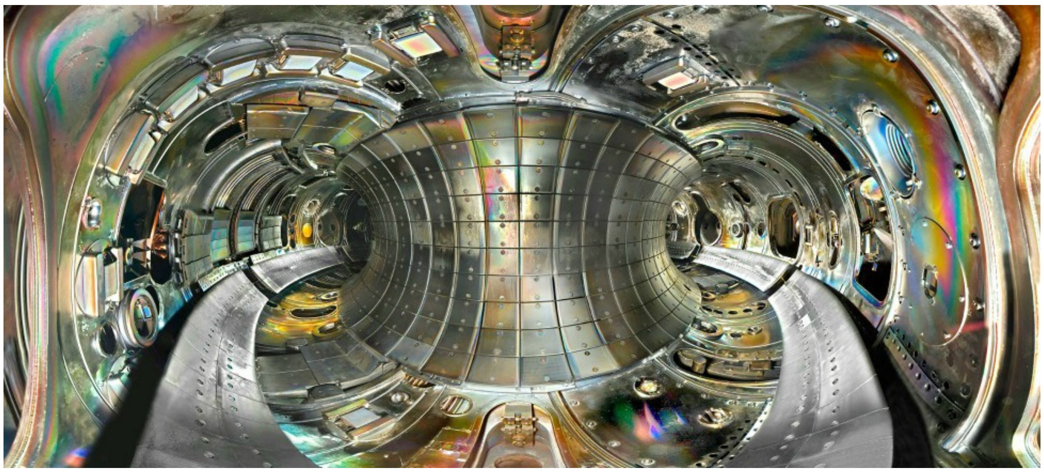


Figure 1.1: A picture of the TEXTOR Tokamak reactor. Depicting the inside with a wide angle camera shot.

The basic idea of fusion power generation is exploiting a difference in binding energy of different elements. A basic calculation as in 1.1 shows that by combining the hydrogen isotopes deuterium and tritium into helium a neutron with 14.1MeV is released which can be used to extract energy as heat, thus allowing fusion to be used as a means of generating electric power.

$$^{2}_{1}\mathrm{D} + ^{3}_{1}\mathrm{T} \Rightarrow ^{4}_{2}\mathrm{He}(3.5\,\mathrm{MeV}) + \mathrm{n}(14.1\,\mathrm{MeV}) \tag{1.1}$$

add citation

The containment principle for fusion plasma is based on the magnetic bottle/mirror phenomenon. This allows to encase charged particles inside a magnetic field. Neutral particles will not be confined by the magnetic field and quickly exit the fusion plasma towards the first wall of any fusion device. The impact of neutral particles will also lead to erosion of the wall. The erosion can be estimated by using monte carlo simulations via the EIRENE [18] code.
To counter the erosion of integral machine parts a device called blanket is used as a replaceable first layer. The operating life of the blanket needs to be taken into consideration for maintenance cycles and operating cost of a fusion power plant.

1.2 PROCESS Systems Code

The systems code PROCESS [14] is concerned with the combination of physics, engineering and economical simulation and evaluation for a fusion power plant scenario[1]. It is based on TETRA (Tokamak Engineering Test Reactor Analysis) [1] and has been used for the Power Plant Conceptual Study [17].

PROCESS can be operated in two different modes, namely optimization mode and non-optimization mode. In non-optimization mode PROCESS will find a single set of parameters that form a viable fusion device within the given constraints, while the more commonly used optimization mode finds a set of parameters that minimize or maximize a chosen figure of merit. The list of figures of merit includes capital cost, cost of electricity or more physical/engineering related quantities such as neutron wall load.[2]
There are several hundred input parameters which can be chosen as iteration variable for a run in optimization mode. Studies of a given design for a fusion device might want to use the optimization mode to scan a range in multiple input parameters, easily resulting in a high number of runs and an accordingly high total run time. Hence underlying physical models have to be sufficiently simplified in an attempt to balance required runtime against a higher potential for error.

This work is concerned with finding a surrogate model replacing the monte carlo simulation of EIRENE for contexts like the PROCESS systems code. Whereas this work is concerned with a simple model, see chapter 2, the methods used are examined for further use in full scale 3D models.

1.3 Reduced Model Approaches

Simulations based on complex models face the barrier of having long run times, which is often unsuited for studying general systematic behaviours. Reducing run time of numerical simulations can be achieved via model order reduction methods like dimensionality reductions.[3] Given that in many cases a complete analytical model can not be given there are approaches to model order reduction using machine learning algorithms that approximate the simulated system via surrogate functions. The resulting surrogates do not give further insight into working mechanisms of the system, but allow for

[1]So far most published work has been on ITER and DEMO Tokamak like scenarios.

[2]Further information on the details of PROCESS code operation can be found in the publication of M. Kovari et. al [14].

[3]Ideally one would apply control theory to the given system with proper orthogonal decomposition and reduced basis methods. Though this method relies on an analytical model description.

much faster computation at reasonable approximation errors. Depending on the desired capabilities of the surrogate a machine learning methods should be chosen accordingly. The methods chosen in this work are Deep Neural Networks (NN) and Gaussian Processing (GP). Both methods feature high flexibility and NN also provide excellent scalability. Furthermore combining both methods allows to reduce downsides of a single method. For example GP achieves accurate prediction when interpolating but has much greater error margins when extrapolating. A more in depth discussion can be found in chapter 3.

Chapter 2

Model

2.1 EIRENE

EIRENE is a Monte Carlo code used for simulating fusion plasma particles. Its first version was written in the scope of the PhD thesis by D. Reiter. It has been used on a variety of studies[1] in the fusion community. The following section contains a brief introduction to the capabilities of EIRENE, followed by the specific example case it was used to study in this work.

EIRENE simulates transport and plasma interaction of neutral particles in a fusion device given a plasma background. It is often used in connection with the EMC3 or SOLPS[2] codes to study simulations of fusion plasma.

Add citation

The Wang Chang-Uhlenbeck (WCU) equation [23], a multi species set of "Boltzmann"-type equations, is solved by EIRENE as described in [18] chapter 1. This allows to calculate trajectories and mean free path length that can be used to simulate interactions in a standard Monte Carlo procedure. The interactions of neutral particles can form source terms for plasma codes. To this end the Monte Carlo codes uses probabilities by considering effective cross sections and plasma chemistry reaction rates.
Furthermore EIRENE is capable of calculating radiation transport and simulating charged particle movement.[3] All of these features are available in

this is also a direct quote from EIRENE manual, reword or quote??

More explanation necessary?

[1]For Example a B2-EIRENE and SOLPS numerical study of Iter divertors [13]

[2]SOLPS stands for Scrape-Off Layer Plasma Simulation, see [10] for detailed information.

[3]Though since it is often used in conjunction with plasma codes this feature is less prominently known and hence used.

arbitrary 3-dimensional geometries.[4]

In this work EIRENE is used in a one dimensional geometry with two surfaces. One perfectly reflective, simulating the outer wall, and one perfectly absorbing, simulating the core plasma. When particles are reflected they can cause atoms of the wall to be released as ions, which are redirected onto the wall by the surrounding magnetic field. The rate at which this happens is called a sputter rate.
The geometry is chosen to be as simple as possible to minimize run time and thus allow for a production of big data sets. Furthermore the amount of input parameters has been reduced by assuming that ions and electrons have the same temperature $T_I = T_e$. This is a reasonable assumption depicting the source plasma in a thermal equilibrium.
The work of Mitja Beckers [2] used the very same model to investigate the erosion of the first wall. A more in depth introduction into the plasma physics background can be found in chapter 3 of his work and is not repeated here. Most of the concepts are rather elaborate and not suited to be summarized in a short manner. The following section will only highlight necessary parameter origins to give an insight as to the physical context of this work.

2.2 Plasma Profiles

For the inputs of EIRENE plasma profiles are needed, that can be dynamically provided by other algorithms like SOLPS or EMC3. Central piece of this work is to investigate if a substitute function can be found for the full range of possible plasma profiles by using big data methods. One can ascertain the physical limits of the parameters constituting the plasma profiles from tables 2.3 and 2.4. These limits are based on different phenomenons in plasma physics, explained by Mitja Beckers in his PhD thesis [2] and an example limitation is visualized in one of his graphics, which is depicted in 2.2 .

Look into source 55 from Becker PhD for english picture and source

[4]Please refer to the geometry section of the EIRENE manual [18] for further information.

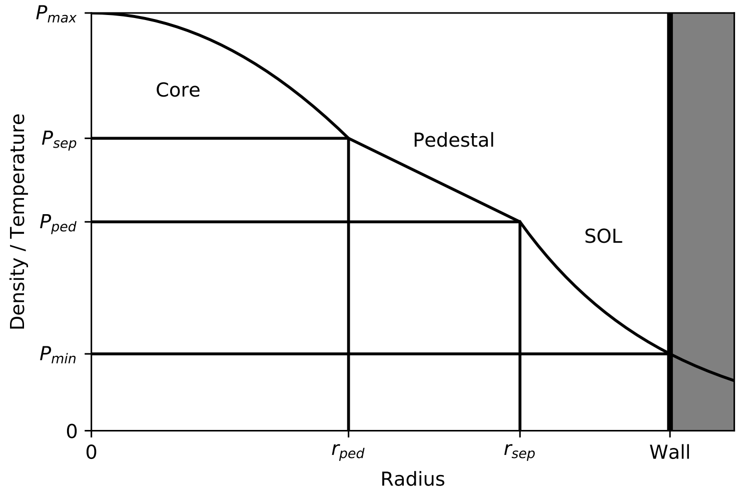


Figure 2.1: Illustration of functional dependencies of temperature and density profiles according to equations 2.1 and 2.2. Note that this schematic is not proportioned realistically, but has elongated pedestal and SOL regions for better visibility. Realistic values for the parameters can be found in tables 2.3 and 2.4.

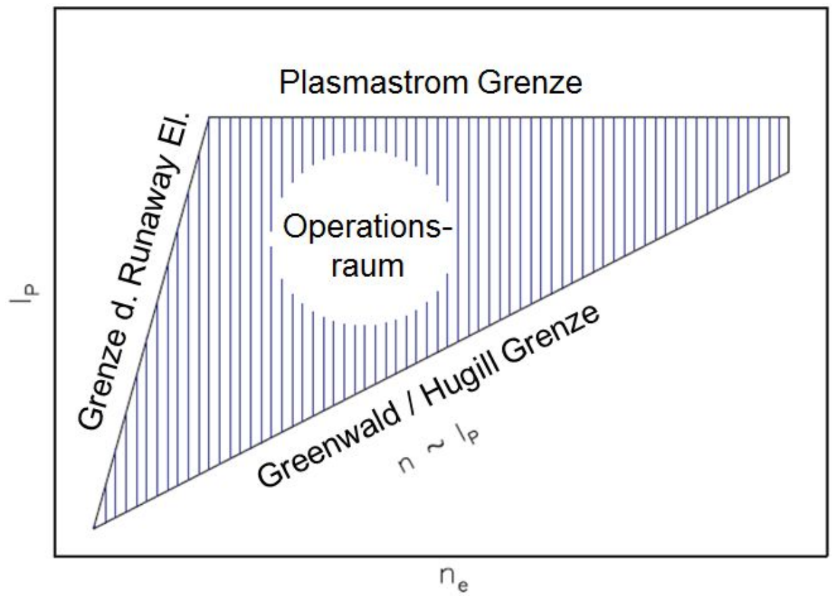


Figure 2.2: Density and current limits for toroidal plasma devices, i.e. Tokamak reactors. The shaded area is the parameter space in which a plasma can be stably operated. Image taken from [2]

Table 2.3 shows the breakdown of temperture profile parameters used in equation 2.1, ordered according to the input into the EIRENE code.

$$T(r) = \begin{cases} T_{\text{ped}} + (T_0 - T_{\text{ped}}) \left(1 - \left(\frac{r}{r_{\text{ped}}}\right)^2\right)^{\alpha_T} & \text{if } r \leq r_{\text{ped}} \\ mr + b & \text{if } r_{\text{ped}} < r \leq a \\ \max\left(T_{\text{sed}} \cdot e^{\left(\frac{-(r-a)}{\lambda_{T_1}}\right)}, T_{min}\right) & \text{if } r > a \text{ and } T > T_{\text{min}} \\ T_{\text{min}} \cdot e^{\left(\frac{-(r-a)}{\lambda_{T_2}}\right)} & \text{other cases} \end{cases} \tag{2.1}$$

with

$$m = \frac{T_{\text{sed}} - T_{\text{ped}}}{\Delta_{\text{ped}}}, \quad b = T_{\text{sed}} - ma$$

2.2.1 Density Profile

Analogue to the parameters of temperature profile table 2.4 shows the breakdown of parameters used in equation 2.2 in order of the EIRENE input file.

$$n(r) = \begin{cases} n_{\text{ped}} + (n_0 - n_{\text{ped}}) \left(1 - \left(\frac{r}{r_{\text{ped}}}\right)^2\right)^{\alpha_n} & \text{if } r \leq r_{\text{ped}} \\ mr + b & \text{if } r_{\text{ped}} < r \leq a \\ n_{\text{sed}} \cdot e^{\left(\frac{-(r-a)}{\lambda_n}\right)} & \text{other cases} \end{cases} \tag{2.2}$$

Figure 2.3: Temperature Profile Parameter Overview

Name	Unit	Min	Max	Note
T_0	eV	$2 \cdot 10^4$	$3 \cdot 10^4$	Core value
ρ_{ped}	1	0.9	0.96	Relative Distance to Pedestal region
α_T	q	0.5	2	Core profile Shape Parameter
R_{sep}	cm	264.3	264.3	Small Radius
T_{ped}	eV	$0.3 \cdot 10^4$	$0.8 \cdot 10^4$	Upper Pedestal Value
T_{sep}	eV	$0.1 \cdot 10^4$	$0.5 \cdot 10^4$	Lower Pedestal Value = Upper Seperatrix Value
λ_T	cm	0.1	1.0	Penetration depth at $\rho \geq 1$
T_{min}	eV	5	20	Ceiling in SOL
λ_{T_2}	cm			Penetration depth At low temperatures in SOL

Figure 2.4: Overview of density profile parameters

Name	Unit	Min	Max	Note
n_0	$10^{14}/cm^3$	1	1.4	central value
ρ_{ped}	1	0.9	0.96	Relative Distance to pedestal region
α_n	1	0.3	0.7	Core Profile Shape Parameter
R_{sep}	cm	264.3	264.3	Small Radius
n_{ped}	$10^{14}/cm^3$	$0.56n_0$	$0.78n_0$	Upper Pedestal value
n_{sep}	$10^{14}/cm^3$	$0.17n_{ped}$	$0.24n_{ped}$	Lower Pedestal value = Upper Seperatrix value
λ_n	cm	1.0	20.0	Penetration depth at $\rho \geq 1$
n_{min}	$10^{14}/cm^3$	0	0	Ceiling in SOL

with

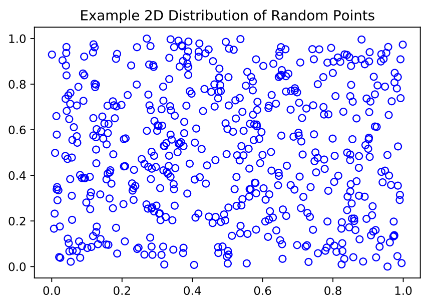
$$m = \frac{n_{\text{sed}} - n_{\text{ped}}}{\Delta_{\text{ped}}}, \quad b = n_{\text{sed}} - ma$$

The input parameters for EIRENE that define the background plasma and geometry will be used as inputs and the sputter rate will be used as the label of the data sets used in this work. The following section talks about how to make intelligent choices on sampling from the parameter space to improve coverage of high dimensional spaces with sparse data. This might not seem necessary for the relatively basic model of this work, but is a conceptual improvement for further projects with more complicated geometries and additional parameters.

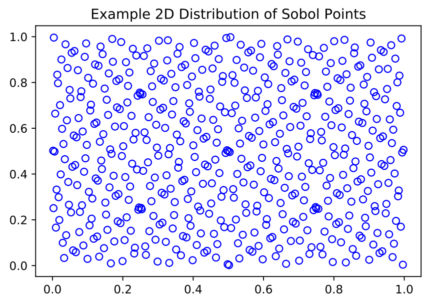
2.3 Choice of sampling set

Since the parameter space is high dimensional, the training points have not been selected randomly. Randomly sampled points might form clusters, which could skew the training towards a subsection of the parameter space. To avoid this a low discrepancy sequence, namely the Sobol sequence, is used to sample training data.[5] It is a sequence in base 2 that form successively finer uniform partitions of the given space. Due to this attribute observations on the influence of the size of training data set is formed on base 2 bench marks instead of intuitive base ten orders of magnitude.
The following figure 2.5 illustrates the clustering of randomly drawn data points in comparison to the Sobol sequence: Especially in high dimensional

[5]For a more detailed explanation of the Sobol sequence and its use cases please refer to [19] chapter 5, section 4: 'Better Random Numbers'.



(a) 512 2D randomly generated data points.



(b) First 512 Sobol points.

Figure 2.5: Comparison of data point distribution to illustrate clustering of randomly drawn samples.

spaces where data is often sparse it is important to make sure that the data is sufficiently spaced in order to adequately represent the parameter space.

The validation and test data were sampled from a random distribution[6] to ensure systematics from the training data do not influence network performance evaluation.

[6]Using pseudo random generator from numpy standart library, based on Mersenne-Twister generator.

Chapter 3

Methods

This chapter is concerned with introducing the methods used to investigate the data reduction of the previously introduced model. The focus of this work will be on artificial neural networks (ANN in short) and Gaussian processes. The following list will provide a brief overview of other machine learning techniques that are frequently employed in machine learning approaches:

- Support Vector Machines (SVM) [15]:
  SVMs are used to classify data similar to ANNs. In contrast to ANNs SVMs are build up from theory and contain little Hyperparameters[1] making them easier to analyse and less prone to overfitting. Generally speaking a SVM tries to separate data by calculating a hyperplane using given training data. The separation has a margin[2] that is maximized. Classification of SVMs are based on which side of the separation the data point lies. For non-linear classification the kernel trick[3] can be used to create a high dimensional feature space. Better suited for classification tasks. Could be used in future endeavours to assess the viability of a certain configuration by classifying input toward a threshold value.
  There are variations of regression SVMs which are difficult to optimize for performance since SVMs rely on analytically calculating the separating hyperplane.

- Random Forests :
  Random forest are ensembles of decision trees, hence the name forest. Each individual tree provides a classification prediction. The class with

[1]Please refer to section 3.1.4 for further information.
[2]Area around separation plane that contains no data.
[3]A detailed explanation can be found at [4]

most votes is prediction of the random forest.
A forest with many uncorrelated trees outperforms highly correlated forest. Random forests have good predictive performance, but slower prediction time which makes them unsuited for system codes.[4]

- Adaptive Boosting:
  Not unlike random forests AdaBoost works with an ensemble of decision trees, though in contrast the decision trees used are single split trees called stumps. When training an AdaBoost algorithm the algorithm boosts weights of individual stumps based on their contribution to difficult to classify instances.[5]

3.1 Neural Networks

The following section is concerned with discussing neural networks as a means of investigating functional dependencies.[6]

3.1.1 General Introduction To Neural Networks

An artificial neural network (ANN) in the following called neural network, abbreviated to NN, is "a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs." [8] First concepts of learning processes based on neural plasticity have been introduced in the late 1940s by D. O. Hebb[7]. In 1975 backpropagation became possible via an algorithm by Paul J. Werbos [22], this led to an increase in machine learning popularity. During the 1980s other methods like support vector machines and linear classifiers became the preferred and/or dominating machine learning approach. With the rise in computational power in recent years neural networks have gained back a lot of popularity.
The concept idea of neural networks is to replicate the ability of the human brain to learn and to adapt to new information. The structure and naming convention reflect this origin.
A neural network is made up of small processing units called neurons. These are grouped together into so called layers. Every network needs at least two

[4]A more detailed introductory text can be found at [24]

[5]A more detailed introductory text can be found at [11]

[6]To aid with understanding the terminology used there is a glossary in the appendix section **??**.

[7]Hebbian theory from the neuroscientific field [6].

layers, the input layer and the output layer. If a network has intermediary layers between input and output, they are called hidden layers. A network with at least two hidden layers is called a deep neural network (DNN). The amount of layers in a network is called the depth of the network. While the amount of neurons in a layer is called the layers width. In a typical NN information stored in neurons is transferred into the next layer by a weighted sum. The connected neuron of the following layer then applies a non-linear function, called activation function, to calculate it's final value. This process in repeated until the output layer is reached. The activation function as well as the amount and interconnectivity of connections can vary in between layers. The system according to which a network is designed is called a network architecture. The most important architectures in the following work will be *dense deep feed forward* and *autoencoder*. To give insight into the basic working principle an example neural network is depicted and described in the following section 3.1.2.

Neural networks are usually used in two ways, optimization or classification. Well known examples are handwriting recognition as classification and least mean squares (LMS) optimization.

3.1.2 Functionality

The working principle is to form a weighted sum $\sum_{k=1}^{N} w_{j,k} \cdot x_k$ over the values from neurons of the previous layer $\vec{x}$ weighted by the connecting weight, elements of the weight matrix $\mathbf{w}$. The weighted sum is then evaluated by the activation function[8] $\sigma(x_k, w_{j,k}, b_j)$, with bias b_j such that the new value $x_j = \sigma(\sum_{k=1}^{N} w_{j,k} \cdot x_k + b_j)$. Here k refers to the index of the neuron in the previous layer and j to the index of the neuron in the current layer.[9]
Since forming a weighted sum is a linear operation the activation function must be non-linear to enable the network to learn non-linear behaviour. The most common activation functions are the sigmoid function and more recently the rectified linear unit (ReLU) and exponential linear unit (ELU) shown in figure 3.2.

[8]An example activation function is the rectified linear unit function which is depicted in figure 3.2a and discussed greater detail in section 3.1.4.1

[9]The order of indices becomes more intuitive when talking about backpropagation and its matrix notation in section 3.1.3.1

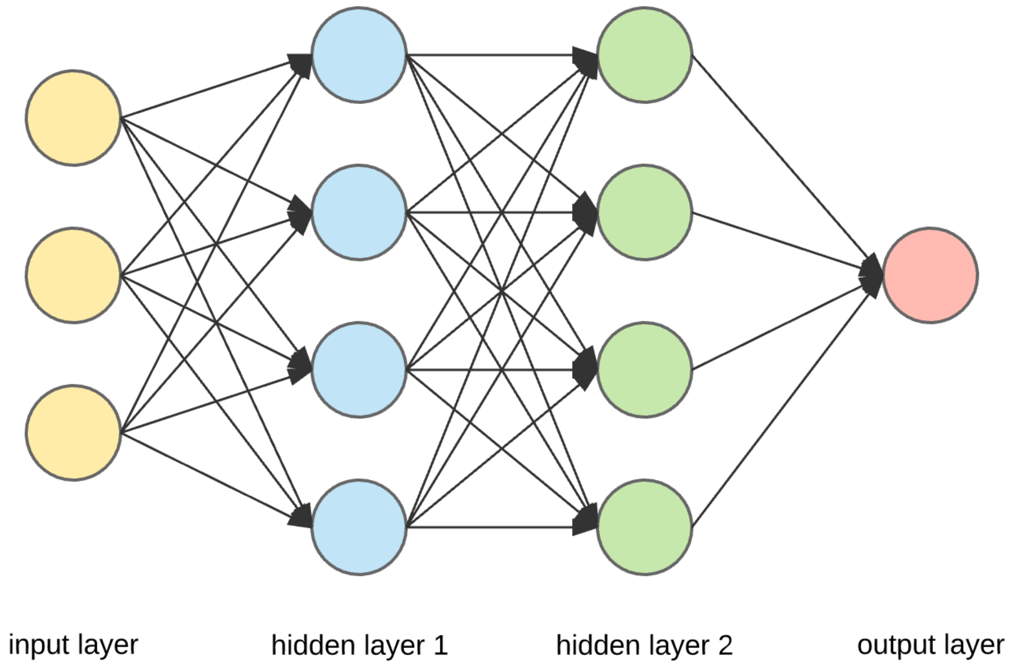


Figure 3.1: Schematic structure of the most basic fully connected deep neural network. Indicated are the input (yellow), output (red) and hidden layers (blue and green). Each neuron outputs to all neurons in the following layer, but there are no interconnection between neurons of the same layer. Note that while the network has the minimum depth (2 hidden layers) to qualify for a deep neural network, the width could be smaller.

3.1.3 Training

Before a neural network can be put to work it needs to be trained. To train a NN a set of training and test data has to be generated. This work uses the afore mentioned Monte Carlo simulations from the EIRENE code. The training data consists of input e.g. temperature and density of the plasma and output e.g. the sputtering rate of the first wall. The EIRENE input data is used as input of the network and the sputtering rate is compared to the output of the network via a cost function. Afterwards the weights of the network are adjusted by using backpropagation, which is a method that calculates partial weight derivatives of the output. A more detailed explanation can be found in section 3.1.3.1.

3.1.3.1 Backpropagation

To talk about backpropagation it is necessary to first set down a notation. In the following $\mathbf{w}$ dontes the weight tensor, that contains all weight matrices $\mathbf{w}^l$ of connections between individual layers l. The elements of the individual weight matrices $w^l_{j,k}$ will denote the weight of the connection from neuron k in layer $l-1$ to neuron j in layer l. Similarly the matrix $\mathbf{b}$ contains all bias vectors $\vec{b}^l$ of layer l which in turn contain the biases b^l_j to each individual neuron j of layer l.
The activation $a^l_j = \sigma(z_j)$ denotes the weighted input $z_j = \left(\sum_k w^l_{j,k} \cdot a^{l-1}_k + b^l_j\right)$ evaluated by the activation function $\sigma(\ )$. Finally a cost function C has to be defined to evaluate the output of the network $\vec{a}^L$ and compare it to the true label $\vec{y}$[10] assigned to the data point $\vec{x}$ by the EIRENE simulation. In the context of training a network the label and inputs are given constants, whereas the weights and biases are adjustable. Therefore the cost function is considered to be a function of all weights and biases $C = C(\vec{a}^L) = C(\mathbf{w}, \mathbf{b})$. Shortening the weights notation of $w^l_{j,k} \cdot a^{l-1}_k$ to $\mathbf{w}^l \cdot a^{l-1}$ with Einstein convention for index k applied.[11]
This leads to the vector notation:

$$\vec{a}^l = \sigma(\vec{z}^l) = \sigma(\mathbf{w}^l \vec{a}^{l-1} + \vec{b}_l) \tag{3.1}$$

The backpropagation algorithm aims to provide a computational fast way of calculating the partial derivatives 3.2 and 3.3.

$$\frac{\partial C}{\partial w^l_{j,k}} \tag{3.2}$$

[10]If the network has only one output y is a scalar value.
[11]This description closely follows chapter 2 of [16]

$$\frac{\partial C}{\partial b_j^l} \tag{3.3}$$

Before looking at the partial derivatives of the cost function it is necessary to make two assumptions about the cost function C.

Assumptions of the cost function
The following two assumptions have to be made.

1. The cost function can be written as an average of cost functions for individual training examples.

$$C = \frac{1}{n}\sum_x C_x \tag{3.4}$$

2. The cost function can be written as a function of the outputs $\vec{a}^L$ of the network:

$$C = C(\vec{a}^L) \tag{3.5}$$

   Where L is the number of layers in a network such that the activation $\vec{a}^L$ is the output of the network.

A good example for a cost function that fulfils these requirements is the quadratic cost function

$$C(\vec{a}^L) = \frac{1}{2}\left\|\vec{y} - \vec{a}^L\right\|^2 = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2 \tag{3.6}$$

Furthermore the partial derivative $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$ is known and easily evaluated.

Backpropagation algorithm A few more intermediate steps are necessary:

1. Define the error $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$.

2. Start with the error of the output layer[12]:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'\left(z_j^L\right) \tag{3.7}$$

[12]σ' denotes the derivative of the activation function $\frac{\partial \sigma}{\partial z}$

3. Express $\vec{\delta}^L$ in a matrix equation[13]

$$\vec{\delta}^L = \nabla_a C \odot \sigma' \left(\vec{z}^L\right) \tag{3.8}$$

4. Express δ^l as a function of δ^{l+1}:

$$\delta^l = \left(\left(\mathbf{w}^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma' \left(\vec{z}^l\right) \tag{3.9}$$

Now the partial derivative 3.2 can be expressed as:

$$\frac{\partial C}{\partial w^l_{j,k}} = a^{l-1}_k \delta^l_j \tag{3.10}$$

And the partial derivative 3.3 can be expressed as:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{3.11}$$

Equations 3.8 and 3.9 provide a fast functionality once the components are known. Luckily all $\sigma'(\ )$ and $\frac{\partial C}{\partial a^L_j}$ are known before start of training and all weight matrices w^{l+1} are calculated during forward pass of each training point. Lastly the error δ^l can be deduced from the following error δ^{l+1}. Hence, as the name suggests, the algorithm works from layer L backward to the first layer $l = 1$.
Therefore the main computational cost of backpropagation is to apply the matrix multiplication of $\left(w^{l+1}\right)^T$ to δ^{l+1}. This can be done in a computational efficient manner over multiple training samples called mini-batches.[14]

3.1.3.2 Choice of optimizer

With the partial derivative of the cost function in respect to any given weight and bias it is possible to adjust them. Additionally a learning rate η that depends on the type of optimizer used and architecture of the network has to be chosen. A basic optimizer is the first order Gradient Descend. It applies the following formula to update parameters θ:

$$\theta_{t+1} = \theta_t - \eta \nabla C(\theta_t) \tag{3.12}$$

Where θ can be any $w^l_{j,k}$ or b^l_j from the previous section 3.1.3.1.
There are multiple optimization techniques that improve upon gradient descend. A few concepts are briefly mentioned here, but for a more detailed explanation please refer to [21]:

[13]Hadamard prodcut of two vectors $\vec{x}$ and $\vec{y}$ is given by $\vec{x} \odot \vec{y} = x_j \cdot y_j$

[14]For a more detailed explanation and short proofs of equations 3.8 to 3.11 please refer to [16]

Mini Batches Applying a parameter update after each training example will cause fluctuations that can be helpful in finding minima, but also slow the convergence once close to them. On the other hand applying only one update per training set slows the learning rate immensely. It might not even be possible for training sets that are too large to fit in memory at once. To compromise one uses a mini batch system where subsets of training data are accumulated for update steps. Typical mini batch sizes range from 50 to 256 training examples.

Momentum Using the gradient descend optimizer it is easy to see that moving along a slope one can imagine a ball rolling down a slope collecting momentum along the way. This is realized by adjusting the weight update with an additional term from the previous update.

$$\theta_{t+1} = \theta_t - V(t) \tag{3.13}$$
$$V(t) = \gamma V(t-1) + \eta \nabla C(\theta_t) \tag{3.14}$$

Where γ is a simple numerical factor to control the size of the momentum. A typical value for γ is around 0.9.[15]
While this method speeds up learning it can also lead to overshooting a minimum. To negate the negative effect a method called Nesterov Accelerated Gradient (NAG)[16] is used, in which a predictive term slows down momentum if the slope changes signs.

$$V_{NAG}(t) = \gamma V(t-1) + \eta \nabla C(\theta_t - \gamma V(t-1)) \tag{3.15}$$

Adaptive learning rates Some neurons activate more seldom than others and therefore it makes sense to put more emphasis on updates of infrequently activated neurons by adjusting learning rates of neurons individually. To do so manually is not feasible, but there are methods like the AdaDelta optimizer that utilise a running average $E[g_t^2]$ of past updates to decrease the learning rate of neurons.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g_t^2] + \epsilon}} g_t \tag{3.16}$$
$$E[g_t^2] = \gamma E[g_{t-1}^2] + (1-\gamma) g_t^2 \tag{3.17}$$

Here $g_t = \nabla C(\theta_t)$ is a shorthand for the gradient and g_t^2 the square not laplacian. ϵ is a small positiv number usually on the order of 10^{-8}.

[15]This factor can be thought of as how many previous time steps influence the current update. See [7]ef for more information.

[16]More details on NAGs can be found at [3].

The Adaptive Moment Estimation (Adam) optimizer combines adaptive learning rates, momentum and batch application in one optimizer. It is well suited for sparse problems and has been shown to yield fast convergence. It is therefore the optimizer of choice in the following work.

$$\hat{m_t} = \frac{m_t}{1-\alpha_t} \tag{3.18}$$

$$\hat{v_t} = \frac{v_t}{1-\beta_t} \tag{3.19}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}}+\epsilon}\hat{m_t} \tag{3.20}$$

Here $m_{t+1} = \alpha\, m_t + (1-\alpha)g_t$ denotes the mean of the gradient and $v_{t+1} = \beta v_t + (1-\beta)g_t^2$ the variance. α is typically as large as γ from AdaDelta, around 0.9. β is close to 1 with a default value of 0.999.

3.1.3.3 Regularization

From the description above it should be clear that the number of parameters in a neural network can easily exceed the one or even ten thousand, some state-of-the-art neural networks even exceed 40 million parameters[17] . With that many parameters a model can fit to nearly any set of data reasonably well. John von Neumann famously said:"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."[18]

Search for further references

The aim of a network is to give accurate predictions on data it has never seen before. Therefore it is critical to ensure the learning gains generalize well to unknown data. Any method that aims to increase prediction accuracy on the test set at a disregard to the accuracy of the training predictions is called a regularization method. Inversely the increase in training accuracy with stagnating or even degrading of test prediction accuracy is called overtraining or overfitting.

In the following we will shortly introduce the most commonly used regularization methods.

Hold out In this work training and test sets have been mentioned before. This is the appropriate point to go into a little more depth at why the distinction is necessary. Furthermore a third set called validation set is introduced. The naming of the three sets is already indicative of what their purpose is.

[17]See [5] page 149

[18]See [9]

Training Set Set of examples used during the training process. The network iterates on these points to adjust weights and biases to minimize the cost function.

Validation Set Set of examples used after training to evaluate the performance of the network. After which hyper parameters like architecture or learning rate are reassessed.

Test set Set of examples used after hyper parameters have been tuned. Used to judge final performance of the network.

At first glance validation and test set seem to fulfil a similar role in that they are used to validate the learning process of the network. Since overtraining is a major concern, it is also important to consider overfitting the hyper parameters. Considering the tuning of hyper parameters as an optimization task to improve test accuracy shows that the validation set really is more similar to the training set on a higher meta-level. Therefore it is necessary to split potential test data into a validation and test set. This allows to have a data set that the network does not see over the training and validation process.

L1 Regularization Adding an additional term to the cost function that is dependent on the weights forces the network to use small weights. For the L1 Regularization this term is $\frac{\lambda}{n}\sum_w |w|$ such that the new cost function becomes:

$$C = C_0 + \frac{\lambda}{n}\sum_w |w| \tag{3.21}$$

Here C_0 denotes the original cost function and λ is a hyper parameter called the regularization parameter. The effect of this regularization becomes apparent when considering its partial derivative $\frac{\partial C}{\partial w} = \frac{\lambda}{n}\mathrm{sgn}(w)$ that is used to adjust the weights via backpropagation.
The new update rule becomes:

$$w_{t+1} = w_t - \frac{\eta\lambda}{n}\mathrm{sgn}(w) - \eta\frac{\partial C_0}{\partial w_t} \tag{3.22}$$

Subtracting a constant amount drives the weights towards 0. This causes the network to focus on a few high importance neurons which can be an advantage but is not generally a wanted quality. The following L2 Regularization improves upon this idea.

L2 Regularization From the name and the previous L1 regularization it can be inferred that the L2 regularization adds the following term to the cost function:

$$C = C_0 + \frac{\lambda}{n} \sum_w \|w^2\| \tag{3.23}$$

Which leads to the following update rule:

$$w_{t+1} = w_t \left(1 - \frac{\eta\lambda}{n}\right) - \eta \frac{\partial C_0}{\partial w_t} \tag{3.24}$$

Where the L1 regularization shrank each weight by the same amount, the L2 regularization rescales the weights while penalising large weight terms harsher. This leads to many small weights contributing to the network performance. Reducing the size of weights is appealing because large weights can be used to learn single features like particularities of the training data.

Dropout Before discussing neural networks there was a brief mention of alternative machine learning methods. Many of which made use of an ensemble of weak classifiers to work together as a strong classifier. Dropout manages to achieve much the same in that it deactivates a portion of neurons randomly selected in each training phase. The remaining neurons form a subnetwork which is tasked with learning the same task as the full size network. Each configuration of neurons or subnetwork can be seen as a weak classifier that when dropout is deactivated for validation perform together as a strong classifier. Furthermore dropout regularizes the network by averaging over the results of the subnetworks. Therefore for a overfitting a feature it has to be learned by multiple subnetworks.

Data variation As explained overfitting happens when a model has more parameters than data points to fit. Hence procuring more training data would remedy this problem. Data variation provides a method to expand the pool of training examples without the need for new data.
It is more easily explained when thinking of images to classify. A common example is recognition of handwritten numbers. Here a training example is an image of a single digit, which can be streched, rotated or otherwise be transformed and still be recognizable as the same digit. Therefore applying a transformation to the training data allows to multiply the number of available training examples by the amount of transformations applied.
For the task at hand a data transformation can be applied and seen as measurement inaccuracy for the inputs and stochastic inaccuracy of the Monte

Carlo data for the output.[19]
A neat side effect of data variation is that the model becomes more robust to exactly the transformations applied. Again this can be more easily understood in terms of image recognition. For example face recognition software might be able to better recognize reflections if a flip transformation (mirror effect) has been applied to the training examples.

3.1.4 Hyperparameters

3.1.4.1 Activation Functions

When designing a neural network it is important to consider which activation function to use. There are requirements of a suited activation as well as varying advantage of using one or another.
In the past a major problem of neural networks has been vanishing of gradients.[20]
To avoid vanishing gradients a better choice than the sigmoid can be found. The most common activation function for neural networks is the rectified linear unit (ReLU), shown in figure 3.2a.

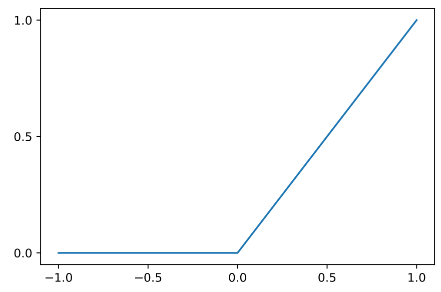
$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{3.25}$$

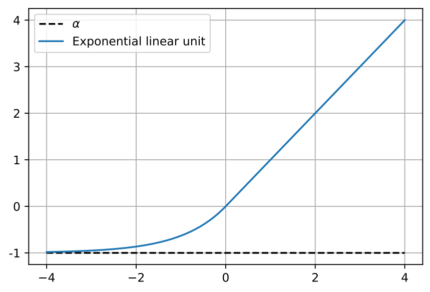
The derivatives of this function is easily computed to 0 for $x < 0$ and 1 for $x \geq 0$. While an activation like the sigmoid function has two sided saturation[21]. The ReLU activation saturates only for negative values, which can be interpreted as neurons that work like switches specialising in detecting certain features, see [12]. In some networks this is a wanted quality of the ReLU activation.
The downside of saturation, and it's vanishing gradient, whether one or two

[19]This assumes that the functionality of the underlying model is smooth, which can be seen as given due to the network trying to learn such a smooth function anyways.

[20]As explained in section 3.1.3.1 in order to adjust the weights it is necessary to calculate the partial derivative of the cost function in respect to each weight. Backpropagation can be done without needing to apply the chain rule to calculate the partial derivatives, but using the chain rule obviously has to yield the same result, if our algorithm is working correctly.
Choosing a sigmoid $f(x) = \frac{1}{1+e^-x}$ as activation function leads to a derivative $f(x) = \frac{e^-x}{(e^-x+1)^2}$ with vanishing values at the fringes. For each layer between the current and the output applying the chain rule will result in a partial derivative factor with values between 0 and 1. Hence in a network with realistic depth e.g. 50, the partial derivative calculated for adjusting the weight will be almost always negligible for the beginning layers.

[21]Values at either end of the spectrum have small derivatives.


(a) Rectified Linear Unit



(b) Exponential Linear Unit

Figure 3.2: Example activation functions rectified linear unit (a) and exponential linear unit (b) used to introduce non-linearity into neural networks.

sided is that once a neuron has reached a saturating value it will change hardly or not at all from it, due to the small or even 0 gradient. This leads to a slow down or stagnation of the learning process.
To alleviate the 0 gradient of the ReLU activation one can introduce a leaky ReLU or Parametric ReLU (PReLU) function. That has a flatter linear part in the negative range:

$$f(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases} \tag{3.26}$$

If α is randomly initialized or static the function is called (Randomized) Leaky ReLU. If α is a parameter of the network and improved during training, the function is called PReLU. A typical value for a static parameter according to [12] is $\alpha = 0.01$.[22]
An even better activation function is the Exponential Linear Unit (ELU), depicted in 3.2b, which offers a one sided saturation with a monotone and smooth gradient. The downside is that while it performs generally better in terms of accuracy, it is also slower in both training and predictions, since a lot of exponential functions have to be evaluated.

$$f(x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases} \tag{3.27}$$

[22]Also some works indicate that for a static parameter $\alpha = \frac{1}{5.5}$ is a better choice.

3.1.4.2 Architecture

The architecture of a network refers to its shape and type of connection. As previously done it is useful to first set down some fundamental terms:

Depth The amount of layers in a network is referred to as the depth of the network. Commonly the input and output layers are omitted for this count, since they are essential for any network.

Width The width of a layer refers to the amount of neurons in that layer. Often networks are build of layers with the same width in each hidden layer. If that is the case one speaks of the width of the network.

Dense layers A fully connected or dense neural network like depicted in figure 3.1 is characterized by connecting every neuron from the previous to all neurons of the following layer. In contrast to other networks this allows for a very high flexibility but also lacks the spatial context of data. This kind of network is especially well suited for data that is given in form of vectors or drawn from an arbitrary parameter space.

insert reference

Feed Forward Networks Any network that propagates information only from input to output is called a feed forward network.

Recurrent networks Recurrent networks allow the use of outputs as inputs. They are typically used in speech and text recognition and processing. Recurrent networks excel in contextualizing information. For example recognizing words as part of a sentence structure.

Autoencoder Networks Autoencoder networks are build symmetrically around a center layer. The output tries to replicate the input instead of a given label. This allows to build a network that simulates a principal component analysis. Looking at the size of the center layer, which is tried to minimize, indicates a set of parameters needed to reconstruct the full information of the input. Hence it allows for dimensionality reduction. This can be used to perform a coordinate transformation of the input data to use for later processing. An example will be shown in the results chapter, section **??**.

Other types There is a wide range of different architectures, some of which can be seen in fig. 3.3. Most of which are not of further importance to this work, but at least convolutional, probabilistic and spiking networks should be mentioned. Since they excel in their respective fields.

3.1.4.3 Cost function

The default cost function is the well known mean squared error formula 3.6, that has already been used as an example before. For the more general case of calculating the cost of multiple training examples the it has to be multiplied by $\frac{1}{n}$, where n is the number of examples.
Alternatively the cross-entropy function is used, if the output is on the scale of 0 to 1, which can be easily achieved by using a sigmoid or softmax activation function in the last layer.

$$C(x) = -\frac{1}{n} \sum_x [y \ln(a) + (1 - y) \ln(1 - a)] \tag{3.28}$$

Here n is the number of inputs x and corresponding labels y, a denotes the activation of the last neuron layer.
The main advantage of the cross entropy function is that its partial derivative to either weights or biases does not depend on the derivative of the activation function, but on the value of the activation function. Thus it prevents slowdown of the learning process.[23]

$$\frac{\partial C_{\text{quadratic}}}{\partial w} = a\sigma'(z) \tag{3.29}$$

$$\frac{\partial C_{\text{crossentropy}}}{\partial w} = \frac{1}{n} \sum_x x_j(\sigma(z) - y) \tag{3.30}$$

Batch size and Epochs During training an update step can occur after each training example[24], after the whole training set has been processed[25] or a after a set amount, called batch size, of training examples has been fed to the network. Research[26] suggests that using a small batch size is optimal, but as with many other hyper parameters the exact size depends on the problem and other hyper parameters.
Smaller batch sizes lead to more noisy updates which has a regularizing effect on the network, though it increases computing time, since more updates are

[23]See chapter 3 of [16] for more detail.
[24]This is called stochastic gradient descend.
[25]This is called batch gradient descend.
[26]add citation see comment

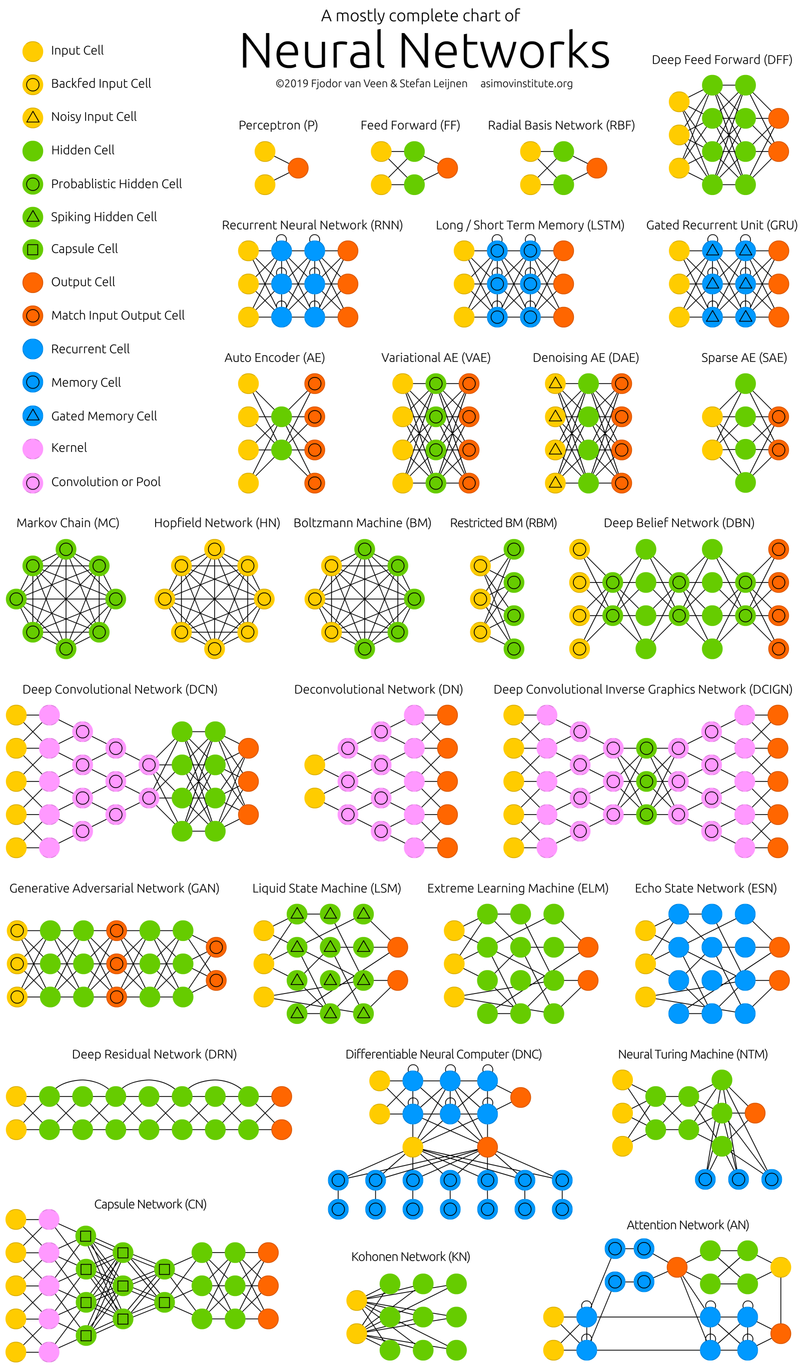

A mostly complete chart of
Neural Networks
©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org
Input Cell
Backfed Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Capsule Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Gated Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Differentiable Neural Computer (DNC)
Neural Turing Machine (NTM)
Capsule Network (CN)
Kohonen Network (KN)
Attention Network (AN)

Figure 3.3: Zoo of Neural network architectures. Image taken from [20]

made.
An epoch marks the time when every training example has been shown to the network once. At that time the next batches for training are created.[27] Epochs are a useful measure of training efficiency. Often training progress is depicted in graphs of epochs against prediction accuracy.

[27]For example with 100 data points and a batch size of 10, 10 random points are assigned to each batch. Meaning that batches over different epochs contain different training examples.

Chapter 4

Results

4.1 Neural Networks

The following subsets of hyper parameters have been investigated:

4.1.1 Architecture

To begin with a general analysis of the architecture is performed. All architectures tested in this section, have been tested with the maximum amount of data to ensure the best possible learning process. Furthermore they have been validated and tested using 2^{15} data points each. Due to assumptions of best suited hyper parameters as explained in chapter 3 section 3.1.4, the initial layer of neurons uses a sigmoid activation function followed by LeakyReLU layers and the output layer has a linear activation function. Dropout has been set to 0.4, the loss function is Root-Mean-Squared and the optimizer is Adam with NAG-momentum and decay. All of these parameters can be found in table **??**.

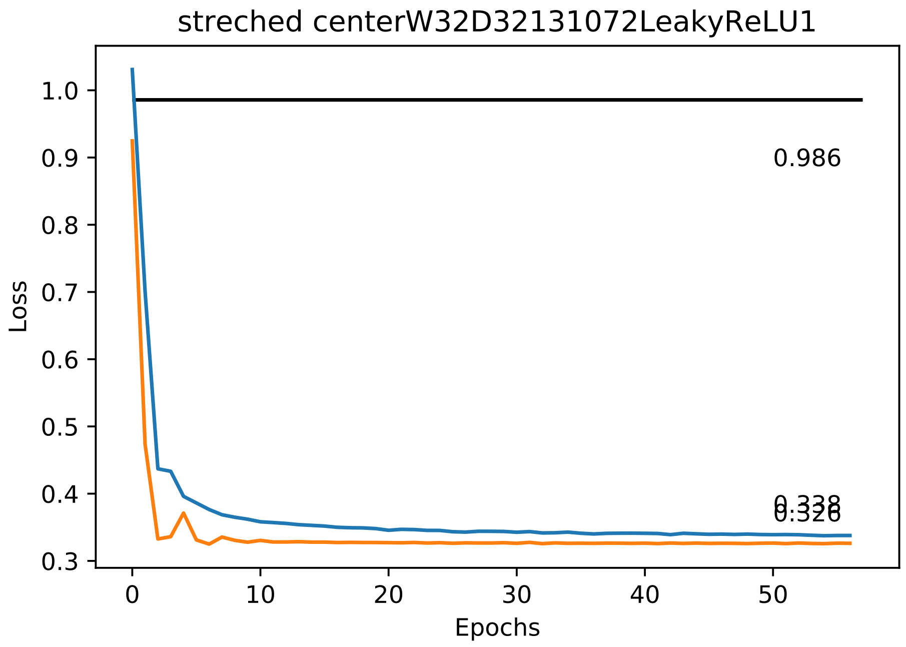


Figure 4.1: Neural network architectrue with best training performance at maximum scale, without fine tuning.

Parameter	Value	Note
Number of Training Points	2^{17}	Sobol Points to ensure coverage of parameterspace
Number of Validation Points	2^{15}	Points Randomly sampled from Parameterspace
Number of Test Points	2^{15}	Randomly sampled from Parameterspace
Optimizer	Adam	$\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$,
Loss Function	RMS	
Activation function	Leaky ReLU	$\alpha = 0.2$
Batch size	2^{14}	8 Batches per Epoch
Maximum Epochs	1000	Early Stopping with patience 50, monitoring validation loss

Shape	Test Loss	Note
Evenly spaced	3.547966	
Triangular	0.971974	
Reversed Triangle	0.895224	
Hourglass	3.320479	
Centered	1.021901	
Stretched Centered	0.986	

Figure 4.1 shows the typical progression of the learning process of a neural network. The blue curve shows the training loss as training progresses. The orange curve shows the validation loss and the black line the test loss. The test loss is just a scalar value since the test evalutaion of the network is done after the training is completed.
It is expected that the training loss has the lowest value followed by the validation loss and the test loss being highest. That being said the expectation is that the test and validation loss are very close.
Initially training and validation loss might be very high due to random initialization of weights. That can also lead to bad learning behaviour, which is why all architectures have been tested multiple times to reduce the influence of bad initialization. Also the initialized weights are he-normalized.
Due to dropout the training loss stays above the validation loss. Showing that the subnetworks formed in training are not able to adequately adapt the underlying behaviour. Giving rise to investigate the influence of the droprate parameter and additionally tuning the batch size to allow the subnetsworks to better adapt during training.
All tested architectures followed the trend as shown in figure 4.1. None showing a systematic plateauing or significantly better or worse training behaviour.
It is surprising to see a much worse value in testing than in validation, since both validation and test data have been sampled from the same random distribution. Hence it would be expected that both have similar values. Especially since no finetuning of the network has been done that would explain a lowered validation loss.

Test with validation and test set switched.

With the general structure locked into place the next test series is used to investigate how the size of the network impacts performance, based on the assumption that a larger network will have better adaptation and hence precision, but takes more time to train, evaluate and make predictions. At first the depth of the network is kept the same while the width is varied, afterwards the width is kept constant. Lastly a test series with about constant amount of weights is done. The Results can be found in tables **??**, **??**, **??**.

Depth	Test Loss

Width	Test Loss

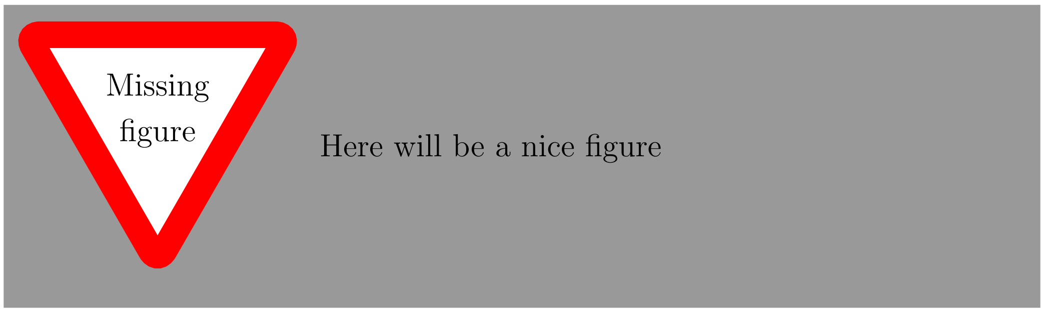


Figure 4.2

Number of Total Parameters	Depth	Width	Test Loss

Summarize final architecture

Conclusion After these tests we can conclude that for the given datastructure and problem the most optimal structure is: .

4.1.2 Training Set Size

Depth and Width together determine the total number parameters and complexity of the network. The question is how complex does the network needs to be to accurately learn the model. Ideally we'd like to trim the network as much as possible without loosing too much accuracy.

Number of Training Points	Number of Validation Points	Number of Test Points

4.1.2.1 Activation

- ReLU

- PReLU

- ELU

- Conclusion

4.1.2.2 Optimizer

- SGD with momentum

- Adam

4.1.3 Derivatives

Bibliography

[1] W.L. Barr, C.G. Bathke, J.N. Brooks, R.H. Bulmer, A. Busigin, P.F. DuBois, M.E. Fenstermacher, J. Fink, P.A. Finn, J.D. Galambos, Y. Gohar, G.E. Gorker, J.R. Haines, A.M. Hassanein, D.R. Hicks, S.K. Ho, S.S. Kalsi, K.M. Kalyanam, J.A. Kerns, J.D. Lee, J.R. Miller, R.L. Miller, J.O. Myall, Y-K.M. Peng, L.J. Perkins, P.T. Spampinato, D.J. Strickler, S.L. Thomson, C.E. Wagner, R.S. Willms, and R.L. Reid. Etr/iter systems code. 4 1988.

[2] Mitja Beckers. Entwicklung eines werkzeugs zur modellierung der nettoerosion im hauptraum der brennkammer eines tokamaks und studium der plasma-wand-wechselwirkung an demo1, September 2017.

[3] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. *CoRR*, abs/1212.0901, 2012. `http://arxiv.org/abs/1212.0901`.

[4] Saptashwa Bhattacharyya. Support vector machine: Kernel trick; mercer's theorem. `https://towardsdatascience.com/understanding-support-vector-machine-part-2-kernel-trick-mercers-theorem-e1e684` 2018. Accessed: 2020-01-31.

[5] Susanne Boll, Qi Tian, Lei Zhang, Zili Zhang, and Yi-Ping Phoebe Chen. *Advances in Multimedia Modeling: 16th International Multimedia Modeling Conference, MMM 2010, Chongqing, China, January 6-8, 2010. Proceedings.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[6] Richard E Brown. The legacy of donald o. hebb: More than the hebb synapse. `https://www.researchgate.net/publication/8954072_The_legacy_of_Donald_O_Hebb_More_than_the_Hebb_Synapse`, 2004. Accessed: 2020-01-31.

[7] Vitaly Bushaev. Stochastic gradient descent with momentum. `https://towardsdatascience.com/`

`stochastic-gradient-descent-with-momentum-a84097641a5d`, 2017. Accessed: 2020-01-31.

[8] Maureen Caudill. Neural networks primer, part i. *AI Expert*, 2(12):46–52, December 1987. `http://dl.acm.org/citation.cfm?id=38292.38295`.

[9] John D. Cook. How to fit an elephant. `https://www.johndcook.com/blog/2011/06/21/how-to-fit-an-elephant/`, 2011. Accessed: 2020-01-31.

[10] D.P Coster, O. Wenisch, A. Kukushkin, M. Stanojevic, and X. Bonnin. Solps 5.0 documentation. `http://solps-mdsplus.aug.ipp.mpg.de:8080/solps/Documentation/solps.pdf`, February 2013. Accessed: 2020-02-09.

[11] Akash Desarda. Understanding adaboost. `https://towardsdatascience.com/understanding-adaboost-2f94f22d5bfe`, 2019. Accessed: 2020-02-09.

[12] Ayoosh Kathuria. Intro to optimization in deep learning: Vanishing gradients and choosing the right activation function. `https://blog.paperspace.com/vanishing-gradients-activation-function/`, 2018. Accessed: 2020-02-09.

[13] Vladislav Kotov, Detlev Reiter, and Andrey S. Kukushkin. Numerical study of the iter divertorplasma with the b2-eirene codepackage. `http://eirene.de/kotov_solps42_report.pdf`, 2007. Accessed: 2020-02-09.

[14] M. Kovari, R. Kemp, H. Lux, P. Knight, J. Morris, and D.J. Ward. “process”: A systems code for fusion power plants—part 1: Physics. *Fusion Engineering and Design*, 89(12):3054 – 3069, 2014. `http://www.sciencedirect.com/science/article/pii/S0920379614005961`.

[15] Alexandre Kowalczyk. *Support Vector Machines Succinctly.* 2017. `https://www.syncfusion.com/ebooks/support_vector_machines_succinctly/introduction`.

[16] Alexandre Kowalczyk. *Neural Networks and Deep Learning.* 2019. Accessed: 2020-02-09.

[17] D. Maisonnier, D. Campbell, I. Cook, L. Di Pace, L. Giancarli, J. Hayward, A. Li Puma, M. Medrano, P. Norajitra, M. Roccella, P. Sardain,

M.Q. Tran, and D. Ward. Power plant conceptual studies in europe. *Nuclear Fusion*, 47(11):1524–1532, oct 2007. `https://doi.org/10.1088\%2F0029-5515\%2F47\%2F11\%2F014`.

[18] Detlev Reiter. The eirene code user manual including: B2-eirene interface. `http://www.eirene.de/html/manual.html`, 2018. Accessed: 2020-02-08.

[19] Antoine Savine. *Modern Computational Finance: AAD and Parallel Simulations*. November 2018. `http://neuralnetworksanddeeplearning.com`.

[20] Fjodor van Veen. Neural network zoo prequel: Cells and layers. `https://www.asimovinstitute.org/author/fjodorvanveen/`, 2017. [Online; accessed January 31, 2020].

[21] Anish Singh Walia. Types of optimization algorithms used in neural networks and ways to optimize gradient descent. `https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-g` 2017. Accessed: 2020-01-31.

[22] Paul Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78:1550 – 1560, 11 1990.

[23] Ryosuke Yano, Kojiro Suzuki, and Hisayasu Kuroda. Formulation and numerical analysis of diatomic molecular dissociation using boltzmann kinetic equation. *Physics of Fluids*, 19, 01 2007.

[24] Tony Yiu. Understanding random forest. `https://towardsdatascience.com/understanding-random-forest-58381e0602d2`. Accessed: 2020-01-31.