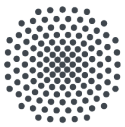
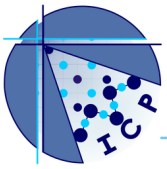


University of Stuttgart
Germany



INSTITUTE FOR
COMPUTATIONAL
PHYSICS

Investigating the Evolution of Fisher Information for Neural Network Dynamics

by

Marc Sauter[†]

A bachelor's thesis submitted to the

Institute for Computational Physics
University of Stuttgart
Allmandring 3
70569 Stuttgart

Advisor:	M. Sc. Samuel Tovey
Supervisor:	Prof. Dr. Christian Holm

Stuttgart, 25. September 2023

[†]github.com/ma-sauter

Try not to have a good time... this is supposed to be educational.

— Charles M. Schulz

Declaration

I, Marc Sauter, hereby declare that I independently authored this work. I affirm that I have used only the specified sources, and that I provided proper attribution whenever necessary. Furthermore, this work, in whole or in part, has not been subject to any other examination procedure. The printed copy matches with the electronic ones.

Signature	*Date*

Zusammenfassung

Deep Learning hat sich als erfolgreiche Methode zur Lösung verschiedenster komplexer Problemstellungen erwiesen. Dies ist auf den bemerkenswert einfachen Aufbau seiner neuronalen Netze zurückzuführen, der in starkem Kontrast zu der Komplexität der reproduzierbaren Funktionen steht. Allerdings ist das Verständnis der genauen Konfigurationen, die den neuronalen Netzen das Lösen der komplexen Probleme ermöglichen, nur sehr gering. Neue Erkenntnisse über die Geometrie hinter der Loss-Funktion und die Architektur der Netzwerke könnten die Entwicklung neuer Methoden auf dem Gebiet des maschinellen Lernens deutlich vorantreiben.

Diese Arbeit beschäftigt sich mit zwei mathematischen Größen, die tiefere Einsichten in die Mechaniken von neuronalen Netzen und ihrem Training versprechen: Der Fisher information und dem Neural Tangent Kernel. Zunächst werden diese mathematisch hergeleitet und ihr Einfluss auf die neuronalen Netze erläutert. Weiterhin wird beschrieben, wie die skalare Ricci Krümmung aus der Fisher information berechnet werden kann. Darauf aufbauend wird eine neue Relation zwischen der Fisher information und dem NTK hergeleitet und diese anhand eines Beispieltrainings für den MNIST Datensatz untersucht. Des Weiteren werden alle zuvor genannten Größen für das Beispiel eines einfachen Netzwerkes berechnet und die erzielten Ergebnisse analysiert.

Abstract

Deep learning has proven to be a powerful approach to tackle computationally intricate tasks. Its success can be attributed to the remarkably simple nature of the models and optimization techniques it employs, especially when contrasted with the complexity of the tasks it can accomplish. Yet, our understanding of how these simple models adapt to meet complex requirements remains limited. Substantial improvements in the development of better models could be gained from understanding how the geometry of the loss function and the architecture of the network affect the training dynamics.

In this work, we investigate the Fisher information and the Neural Tangent Kernel, two observables of neural network training that promise deeper insights into the workings behind neural learning. We derive both of them mathematically and explain their impact on the neural network and its training. Additionally, we mention how to compute the Ricci scalar curvature of a network from the Fisher information. Building on these foundations, we present a novel relation between the trace of the Fisher information and the NTK and investigate its applicability through an example training on the MNIST dataset. Moreover, we provide an example measurement of the complete Fisher information, the trace of the NTK and the scalar curvature for a simple 2-parameter network.

Keywords— Fisher Information - Neural Tangent Kernel - Neural Network Training

This page would be intentionally left blank if we would not wish to inform about that.

Acknowledgments

As I conclude this work, I would like to express gratitude to some of the people that provided me with support and encouragement throughout the process of writing this thesis.
First, I'd like to thank Prof. Dr. Christian Holm for giving me the opportunity to write this thesis and believing in the potential of machine learning applications in a physical context. Further, I'm very grateful to my advisor M. Sc. Samuel Tovey for all the time and effort he put into giving me great advice and guidance. I also owe thanks to M. Sc. Konstantin Nikolaou for teaching me a lot about scientific research and stepping in whenever I needed further guidance. Working at the ICP was a pleasure, since all the great people there created a friendly and inspiring working environment. Here, I also want to specifically thank Julian Hoßbach for great talks, help whenever I needed it and a lot of design inspiration.

Additionally, I want to thank Svenja for keeping me sane, Etienne for being an incredible smorgasbord of knowledge and always being happy to share some of it, and all the other Atzen from the Fachschaft for the right amount of distractions during long working days. I'm also grateful to Tareq, for the fact that he would have always helped if I had needed him and for generally contributing a lot to the person I am today.

Finally, I need to thank my family for all the trust and support throughout all of my life - Zum Schluss möchte ich meiner Familie für die allgegenwärtige Unterstützung und das tiefe Vertrauen in meinen eigenen Lebensweg danken.

Table of Contents

1 | Introduction

In recent years, the widespread adoption of deep learning methods lead to many advancements in the realm of computational problem solving [1–4]. Many tasks that otherwise require vastly complex algorithms can also be solved by comparably simple neural networks. These networks act as function approximators that are optimized to fit certain tasks through simple methods such as Gradient Descent. The backside of this is that, while the networks architecture is remarkably simple, the specific configurations behind trained networks are exceptionally complex. The process of training and the resulting algorithm blackboxes are only poorly understood. That's why many current fields of research aim to find underlying concepts behind neural learning [5] in order to improve optimization algorithms and determine optimal network architectures for different tasks.

This thesis presents two different observables of neural network training — the Fisher information and the NTK — that promise deeper insights into the training and architecture of neural networks. Its main goal is providing a deeper understanding of those observables by first presenting their mathematical background along with interpretable explanations, and then investigating their relation and calculating them for a simple example setup.

To start off, Chapter 2 covers all fundamentals about machine learning needed for readers that are unfamiliar with the basics behind neural networks and supervised learning.

Chapter 3 discusses the mathematical foundation of the Fisher information and some of its use cases. It explains how the Fisher information represents the Riemannian metric of the statistical manifold constructed from the neural network and its loss function. Here, we also explain the concept of scalar curvature and how we can compute it from the Fisher information for neural network training.

Chapter 4 introduces the NTK by going over a derivation and providing explanations on why it arises.

Chapter 5 is divided into two sections. In Section 5.1, we derive a relationship between the trace of the Fisher information and the NTK and examine how accurately the behavior of their traces match up for a training example on the MNIST dataset. In Section 5.2, we compute examples of the Fisher information, the NTK Trace and the scalar curvature for a simple 2-input neural network.

Chapter 6 provides a summary of the mathematical concepts explained and investigated in this work. We also discuss potential further topics of research on the considered observables.

2 | Machine Learning Basics

2.1 Introduction to machine learning

Over 70 years ago in October of 1950 computing hardware was still in its infancy. At a time when computers weighed several tons, could only perform a few thousand operations per second and the pinnacle of machine intelligence were analogous robots that could follow light sources [6], Alan Turing published a paper in the journal of Nature discussing the question "Can machines think?" [7]. In this paper, Turing tries to tackle the question by proposing a game he calls the "imitation game". This game puts a human, whom we will refer to as Alice, in a room where she can communicate via written messages with two different entities, one of which being a human called Bob, the other being a machine. Alice's goal is to determine from this simple communication alone which of the two entities is the human. The goal of both the machine and Bob is to convince Alice that they are the human.

The largest execution of such an experiment to date took place in early 2023 in the form of an online chat portal where players had two minutes to talk to either another human or an Artificial Intelligence (AI) without knowing the type of their interlocutor [8]. After more than two million participants had played the game for a total of more than 15 million conversations, only $68\,\%$ of the attempted classifications were correct.

All of the advanced AI-bots used in this experiment were achieved using machine learning methods. The Oxford Learning Dictionary defines machine learning as "a type of artificial intelligence in which computers use large amounts of data to learn how to do tasks rather than being programmed to do them" [9]. The theoretical foundation of such will be explained in the following sections.

Section 2.2 discusses the workings of neurons and how they form neural networks. They consist of a fixed part, referred to as architecture, and modifiable parameters. These networks will be used as functions which replicate the relation of inputs and outputs in given datasets. Section 2.3 covers how these datasets have to be structured and how one can find the optimal parameters for which the network replicates the given datasets.

2.2 Neural networks

2.2.1 Neurons

The term **neural network** (NN) is a reference to the workings of nervous systems of humans [10]. These systems consist of a net of neurons, biological cells that are intricately connected to other neurons through structures called synapses [11]. These connections carry electric pulses between neurons that can excite them when those pulses exceed certain thresholds. Those thresholds vary from neuron to neuron and change over time. Upon excitation, new pulses in turn propagate from the excited neuron outwards to possibly excite other neurons. This interplay between excitation and transmission through the network of neurons may create what we perceive as thinking.

In attempts to eventually understand and replicate this thinking process, mathematical analyses of such systems have been done as early as the 1940's [12]. The artificial neural networks used in machine learning today are mathematical concepts that replicate the transmission of excitation between neurons. To examine how this is achieved using mathematical functions and values instead of biological cells and electric pulses, let's look at how the artificial neural networks are built.

The **neurons**, which function as building blocks of artificial neural networks, are mathematical entities that take a fixed number of scalar values as inputs and con-

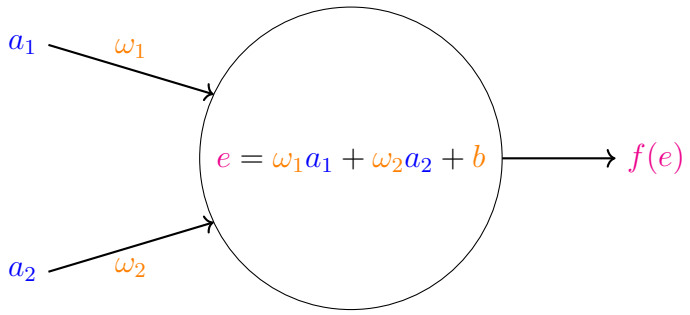


Figure 2.1: This figure aids the explanation of the operating principle of neurons in neural networks. The weights ω_i and the bias b are denoted in orange, the input activations a_i in blue and the activation e with its corresponding activation function f in magenta.

vert them into a single output value. For a more visual explanation let's take a look at Fig. 2.1, which illustrates the example of a neuron with 2 inputs. The big circle in the middle represents the neuron itself. It takes the activation values a_i as input, multiplies them with their corresponding weights ω_i, sums them up, and adds a bias value b to obtain the excitation e. It then applies the activation function onto e to obtain the resulting output value. The input activations a_i correspond to the strength of electronic pulses in the nervous system. The absence of a pulse in the biological system would be represented by an activation of zero in the mathematical model. The weights ω_i are a representation of how important single input values are for the activation of the neuron. In the biological counterpart this might correspond to how thick or conductive the connections between the nerve cells are. Finally, the combination of bias b and activation function determines how large the sum of the input-weight-pairs has to be to activate the neuron and how the resulting value for the activation of the neuron changes for higher input activations. For example, a simple output activation function would be the ReLU

function (Rectified Linear Unit). This function is defined as [13]

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}. \tag{2.2.1}$$

When applying a ReLU activation function, the neuron is activated as soon as the sum of the input-weight-pairs is larger than $-b$. Its value increases linearly with e.

2.2.2 Neural networks

A neural network can be built from these neurons by connecting the outputs of neurons to the inputs of others. An example of such a network is illustrated in Fig. 2.2. This neural network consists of three layers of four neurons each, takes two values as input and outputs one value. For example, it could be used as an approximator of whether a point on a 2D-grid is inside or outside of a given region. The input values would be the x and y coordinates of the point and the output value could represent the predicted probability that the point is inside this region. How to find parameters that make the network correctly classify a desired region will be explained in Section 2.3.

This network serves as an illustrative example, showing what a neural network can look like. For real-world applications, there are various kinds of networks used to learn different tasks [14]. For this thesis we will only talk about "fully connected" or "dense" neural networks. This means that every neuron in the first layer will receive every possible network input value, and every neuron in later layers will receive the output of every neuron in the previous layer as input. All neurons are equipped with the same activation function. The weights and biases vary throughout. How the output of the network gets handled may still differ

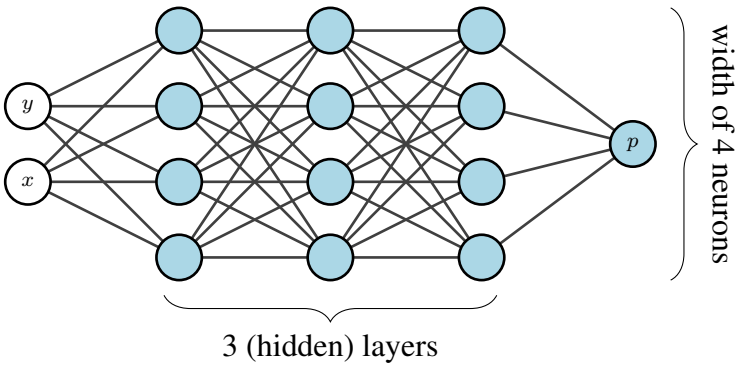


Figure 2.2: This figure shows an example of a neural network. It consists of multiple neurons connected to each other. This network takes in 2 input values and returns one output value. It consists of 3 hidden layers, each having a width of 4 neurons.

through different use cases.

When working on classification tasks, it can be beneficial to feed the output of the neural network into a so-called softmax function before analyzing it. This function converts the scalar outputs of the network, which can be any number from $\mathbb{R}$, into probabilities of the inputs belonging to specific classes. These aren't real probabilities, since every input[1] belongs to a predefined class, but rather represent the network's confidence in the predicted classifications. For a set of outputs z_i, the standard softmax function is defined as

$$\sigma_i(z) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{2.2.2}$$

The resulting values lie between 0 and 1 and add up to a total of 1. A larger value of an output node results in a higher corresponding probability. For further details on the softmax function, see [15].

The structure of a neural network is generally referred to as the **architecture** of the

[1]The term input can refer to single scalar values given to neurons, as well as to whole sets of scalar values given to the network.

network, with the hidden layers[2] referring to the columns of neurons in between the input values and output neurons and the amount of neurons per layer referred to as the "width" of the network. This is also denoted in Fig. 2.2.

2.2.3 Mathematical view

The previous explanations have been very visual and step by step to make the topic more accessible. However, these concepts can be broken down to rather short mathematical expressions.

To start off, we can define the inputs as $a_i^{(0)}$, $i = 1, \ldots, n$. The weights of the first hidden layer can be denoted by $\omega_i^{(1)}$, $i = 1, \ldots, n$. Furthermore, we define $\omega_{i,j}^{(k)}$ as the weight that connects the i-th neuron in the k-th layer to the j-th neuron in the $(k-1)$-th layer. The maximum values of i and j depend on the widths of the respective layers, k can reach values between 1 and the amount of hidden layers plus 1. The bias of the i-th neuron in the k-th layer is denoted as $b_i^{(k)}$. Using this notation we can write out the output of the i-th neuron in the $(k+1)$-th layer as [16]

$$a_i^{(k+1)} = f\left(\sum_j \left(\omega_{i,j}^{(k+1)} a_j^{(k)}\right) + b_i^{(k+1)}\right). \tag{2.2.3}$$

To actually calculate this value, the activations $a_j^{(k)}$ have to be recursively replaced with their full calculation until one arrives at the input values of the network.

For simplicity reasons, we will refer to the weights $\omega_{i,j}^{(k)}$ and biases $b_i^{(k)}$ together as **parameters** of the neural network. We will denote these collected in one ordered set as $\theta = \{\theta_i\}_{i=1}^{N}$, where N is the total number of weights and biases added together. The mapping of the parameters onto θ can be arbitrarily chosen. When talking about all parameters as a single set, the corresponding mapping has to be known, so that it's possible to calculate the output of the network when given

[2] We also sometimes simply refer to hidden layers as layers.

the parameters in the same way as when given the actual weights and biases. Another possible representation that we will use often is to write this set as a vector $\theta \in \mathbb{R}^N$. A short notation for the output of the neural network will be explained in the next section.

2.3 Training of neural networks

In the previous sections we discussed how neural networks are constructed. Now we will give an example of how they can be trained and used to perform specific tasks. Here, the term training refers to the search for a set of parameters θ that make a neural network with a fixed architecture behave in a desired way. Specifically, we will look at how they can be trained on data sets in a process referred to as **supervised learning**. In supervised learning, the desired behavior of a network is defined by a dataset consisting of input and output values [17]. The network should behave in such a way that it produces the desired outputs when fed the corresponding inputs. Training consists of defining a measure of performance of the network, called the loss function, and then using mathematical iterative methods for minimizing the loss.

2.3.1 Datasets

First, we need to assume that we have a given dataset containing input-output pairs that represent the task we want the neural network to perform. The neural network is supposed to learn which output is supposed to be generated from which input by evaluating the given inputs and desired outputs. We will call those desired outputs **targets** to distinct them from the actual outputs of the neural network. A good example of dataset would be the MNIST database, which was first used in 1994 in [18] and has since become a popular entry-level classification task for machine

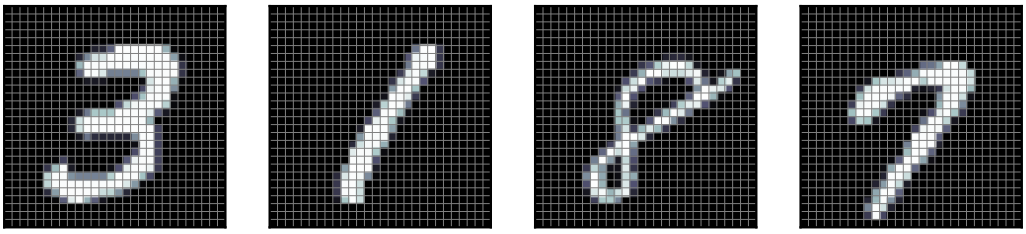


Figure 2.3: Displayed in this figure are 4 examples of input values of the MNIST dataset. This dataset represents handwritten digits from 0 to 9. Each input consists of a 28 by 28 grid of grayscaled pixel values.

learning. Examples of input data for this database are shown in Fig. 2.3. This dataset consists of various handwritten digits from 0 to 9 represented by a 28 by 28 grid of grayscale pixel values. The values of the pixels in these grids act as the 784 input values for the neural network. The targets and the output of the neural network should be of the same shape. For example one might use a scalar output of the neural network that should be equal to the value of the digit drawn in the input pixel grid. In this case the targets are scalar values from 0 to 9 corresponding to their input pictures. Another way would be to use 10 output values together with the previously mentioned softmax function (see Section 2.2.2) to make the outputs interpretable as probabilities of each prediction [15]. We would then change our targets to vectors with 10 elements, where the entry corresponding to the handwritten digit in the input picture would be 1, every other value would be 0. For example, in this case, an image depicting the number 3 would correspond to a target of $(0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$.

We will denote the data sets as mathematical sets of input-target pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$, where $\mathbf{x}_i$ are the input values and $\mathbf{y}_i$ are the targets. Here, we referred to the total number of training samples as N. The mathematical dimension of $\mathbf{x}_i$ and $\mathbf{y}_i$ can vary and are dependent on the network architecture, but we will assume that they are vectors of the real numbers $\mathbf{x}_i \in \mathbb{R}^a$ $\mathbf{y}_i \in \mathbb{R}^b$. Due to this property we also

sometimes refer to them as points. If our input values are organized in a different manner, for example the MNIST data being matrices of $\mathbb{R}^{28\times 28}$, we can rearrange these numbers into a vector of $\mathbb{R}^{784}$ in any way we want, as long as every input is rearranged in the same manner.

Going further, we will denote an output of the neural network for an input $\mathbf{x}_i$ by $f_\theta(\mathbf{x}_i)$. This means that we mathematically describe the entire neural network as a function

$$f : \mathbb{R}^a \times \mathbb{R}^p \to \mathbb{R}^b \tag{2.3.1}$$

with p being the amount of parameters in the network.

2.3.2 Loss function

Thus far, we have defined what a neural network is and how the input data we want to train on is structured. In order to train our network to learn the underlying function of our dataset, we need to introduce a way to measure how well our network is performing. Once we can evaluate the performance of our network, we can introduce ways to optimize that performance.

This measure of the performance of a neural network is called the **loss function** $\mathscr{L}$. It's also sometimes referred to as the cost function in literature [19]. Within this work, we will only consider loss functions of the form

$$\mathscr{L}\left(\{(f_\theta(\mathbf{x}_i), \mathbf{y}_i)\}_{i=1}^N\right) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(f_\theta(\mathbf{x}_i), \mathbf{y}_i\right). \tag{2.3.2}$$

Let's call ℓ the subloss of the system. As a reminder, θ is a set containing the parameters of the neural network, f_θ is its output function. Sometimes it's more convenient to explicitly denote the dependency on θ, which would make the loss

function

$$\mathscr{L}\left(\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N, \theta\right) = \frac{1}{N} \sum_{i=1}^{N} \ell_\theta\left(\mathbf{x}_i, \mathbf{y}_i\right). \tag{2.3.3}$$

How exactly the loss function should be defined cannot be generally stated. In any case, the value of the loss function should generally be smaller the closer the output of the neural network is to the corresponding targets.
In most of our cases we use loss functions with

$$\ell_\theta\left(\mathbf{x}_i, \mathbf{y}_i\right) = d\left(f_\theta(\mathbf{x}_i), \mathbf{y}_i\right), \tag{2.3.4}$$

where d represents the distance between the output vector of the neural network and the target vector according to different metrics. One example of a loss function would be

$$\mathscr{L}\left(\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N, \theta\right) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \left(f_\theta(\mathbf{x}_i)_j - y_{i,j}\right)^2, \tag{2.3.5}$$

with $f_\theta(\mathbf{x}_i)_j$ and $y_{i,j}$ being the j-th component of the network output and target vectors. Further examples and loss functions of different forms can be found in [20].

2.3.3 Optimization

To quickly recap, we have now defined what a neural network is and that we want a fixed network architecture during training, for which the parameters can vary to optimize the performance. We also assume that we have a data set that we want our network to learn the structure of, and a loss function that measures how well the current setup of our network is performing. We will now talk about how we can change the parameters to optimize performance.
This is achieved by minimizing the value of the loss function. The simplest and

most common algorithm to do this with is **Gradient Descent** (GD). Here, the rule for updating the parameters looks like [21]

$$\theta' = \theta - \eta \nabla_\theta \mathscr{L}\left[\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N, \theta\right], \tag{2.3.6}$$

where η is a parameter of training called the **learning rate**, which is a scalar value for GD but could also vary through training and even be a matrix for more advanced optimization methods [22].

The idea behind GD is that the gradient with respect to θ points in the estimated direction of the steepest ascent of the loss when the parameters are varied. If we subtract this gradient from the parameters we update the loss in the opposite direction, which is the estimated direction of steepest descent. To visualize this, let's examine a single parameter θ_1. The update rule for this single parameter can be obtained from writing out the previous Eq. (2.3.6) in its full vectorial component form and then picking out the line of θ_1. It reads

$$\theta_1' = \theta_1 - \eta \frac{\partial}{\partial \theta_1} \mathscr{L}\left[\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N, \theta\right]. \tag{2.3.7}$$

To explain the concept of this algorithm more visually, let's take a look at Fig. 2.4. The cross markings symbolize the position of parameter θ_1 before the update step. The algorithm then calculates the derivative of the loss $\frac{\partial}{\partial \theta_1}\mathscr{L}$, which geometrically is the slope of the tangent. To update the parameter, the slope is multiplied with -1 times the learning rate η and added to θ_1, which is represented by the arrows. The length of these arrows depicts the resulting change in θ_1. One can see that the red arrow on the right covers more distance in the direction of θ_1 than the green one on the left, which is a result of the corresponding slope being steeper. The circle markers then mark the updated parameter value θ_1'.

This process of updating the parameters is repeated several times. One update is

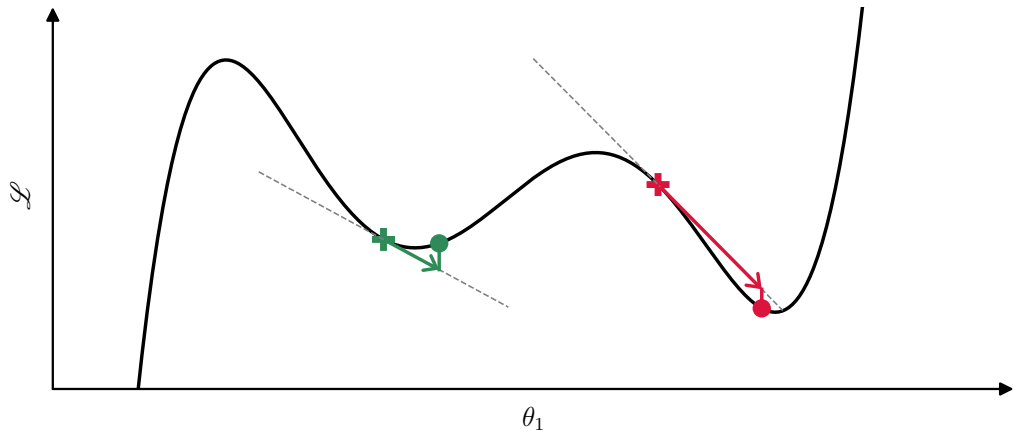


Figure 2.4: Visual explanation of the Gradient Descent algorithm. The dot markers show where a parameter that starts at the cross marker might end up if the algorithm is applied.

called an update step. When using batches of the dataset for parameter updates (see Section 2.3.4), an epoch refers to a period of training after which the algorithm calculated gradients on each input-output data pair. The number of update steps in an epoch depends on the batch size.

This visual example already illustrates some of the drawbacks of the GD algorithm: Starting training with the green parameter will yield a higher final loss than starting with the red parameter. Also going further to the left would make the loss function smaller than both of these local minima. GD just tends to find the closest local minimum instead of a global one [23]. Another drawback results from the step size being finitely big, which makes the green parameter hop over the local minimum. Both of these problems can't generally be solved by simply optimizing the learning rate η.

2.3.4 Other optimizers

The standard GD calculates the gradient for parameter updates from the whole data set according to Eq. (2.3.6). It can be sufficient or even beneficial to compute

the loss gradient and update the parameters for a subset of the whole data set [23], which is usually called batching. The extreme end of this would be to calculate the gradient and update the parameters for one data point each, which is then called Stochastic Gradient Descent. GD and its variations are very simple and fast to compute algorithms, but they generally only converge to local minima and can quite easily hop over them. That's why other algorithms have been developed, aiming to solve or mitigate some of the problems of GD. In all applications of GD for training models in this thesis, the so-called ADAM optimizer was used. The intricacies of this optimization method are beyond the scope of this work. We refer the interested reader to [24], where the details of this algorithm are discussed.

2.4 Which assumptions are actually necessary?

In the previous sections we took a brief look at machine learning and neural networks. Although this barely scratches the surface of the methods used nowadays, it's still more specific than what's needed for Chapter 3 and Chapter 4. The specific methods were given to explain how NNs in this thesis are trained, but include restrictions that don't need to be made for the mathematical considerations coming up.

For those chapters it is sufficient to view mathematical functions $f_\theta(\mathbf{x}_i)$ that depend on parameters θ, accept input vectors $\mathbf{x}_i$ of fixed dimension and can be evaluated through a loss function of the same shape as Eq. (2.3.3). For the discussions of the NTK, we can even disregard the assumption of the loss function splitting up into a sum of subloss functions. This means that the exact properties of the neural networks we looked at earlier can be ignored, allowing us to generalize the observations to many more network architectures or completely different systems than previously mentioned.

3 | Fisher information

Before starting this chapter, let's go over the ideas behind its structure.
First, Section 3.1 will describe the Fisher information as it is used in statistics. Some readers may wonder why we're discussing a statistical method describing probabilities when we've only talked about machine learning and neural networks before. The answer is, that the Fisher information matrix also acts as the Riemannian metric describing the statistical manifold of the network regarding its loss, which is explained in Section 3.2. To make the mathematically abstract concepts in this section more accessible, Section 3.3 provides a brief recap with some added intuitive explanations. To conclude this chapter, Section 3.4 presents another application of the Fisher information in a physical context, where it can be used to find phase transitions in thermodynamic systems.

3.1 Use in Statistics

To introduce the Fisher information (FI), we will start off with how it's defined and used in statistics.
We will consider a **statistical model** $f(x_i|\theta)$ that represents how a parameter θ is related to the outcomes x_i of random variables X_i [25]. Let's look at an example of a statistical model. You can see a picture of a Galton board in Fig. 3.1. It's a famous mechanical model that visualizes binomial distributions, which are discrete approximations of the normal distribution. If we place many balls at the top of the board and let them fall to the bottom, the amount of balls that end up in each cell are distributed according to the binomial distribution [27]. In this case, x_i could assume the slot number which a ball can fall into. The i could label multiple throws into the board, but for now we'll assume that there is only one experiment

Figure 3.1: Photograph of a Galton board, taken from [26].

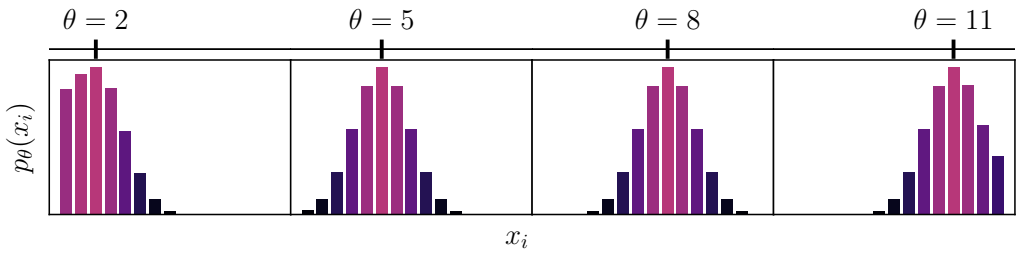


Figure 3.2: This figure shows the probability distributions for a Galton board with different drop-in positions. The slots where the ball can end up are labeled by the value of x_i.

i. To introduce a parameter that influences the distribution of the balls, let's say one can throw from different spots above the Galton board which we now control with the the value of θ. For a known θ, the resulting function of the statistical model represents the probability distribution $f(x_i|\theta) = p_\theta(x_i)$ for the probability of the different outcomes x_i. As a sidenote, if we instead fixed the value of x_i and viewed the statistical model as a function of θ it would be called a likelihood function. A visual representation of the probability distributions for different θ can be seen in Fig. 3.2.

In general, the statistical models might be more complex, where θ contains several parameters, x is an element of a mathematical space other than $\mathbb{R}$ and the index

i denotes various different experiments, all depending on the same parameter but having different possible outcomes and probability distributions.
What's of interest for the FI are cases, where the parameters are not known before conducting the experiment and have to be approximated by the different outcomes x_i.
Before we introduce the FI, let's look at an example from Ly et al. [25]. Let's consider a biased coin where we denote the probability of heads ($x_i = 1$) with θ and the probability of tails ($x_i = 0$) with $1-\theta$. We will now take a look at the outcome of n tosses, represented by the variable X^n. For example, an observed result for X^5 could be $x^5 = (1, 1, 0, 1, 0)$. Let's consider another variable Y, observing the sum of the total head throws $y = \sum x^n$. In our example case of $x^5 = (1, 1, 0, 1, 0)$, this would result in a value of $y = 3$. The probability for this variable y is distributed according to the binomial distribution $f(y|\theta) = \binom{n}{y}\theta^y(1-\theta)^{n-y}$ [28]. Here, the binomial coefficient $\binom{n}{y}$ represents the different combinations that result in the same value of y. This is needed because there are 2^n different possibilities for x, while there are only n different possibilities for y.
If we now fix the outcome of y and look at the conditional probability for the different x^n that could have resulted in that y value, we get $p(x|y, \theta) = 1/\binom{n}{y}$. With $p(x|y, \theta)$ we denote the probability depending on x for fixed y and θ. Although the probability of y and x both depend on θ, the probability for x when y is fixed doesn't. After measuring y, there is no information about θ left in the measurement of x. This means that y is fully descriptive of, or sufficient for the parameter θ. Measuring y results in the same amount of information about the parameter θ as measuring the whole observation x. To quantify how much information a certain function contains about the parameters θ, Fisher introduced the **Fisher information**.

The Fisher information is defined as

$$I_{X,ij}(\theta) = \underset{x\in X}{E}\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i}\log f(x|\theta)\cdot\frac{\mathrm{d}}{\mathrm{d}\theta_j}\log f(x|\theta)\right], \tag{3.1.1}$$

where we used the expectation E

$$\underset{x\in X}{E}[A(x)] = \begin{cases}\sum_{x\in X}(A(x)p(x)) & \text{if } X \text{ is discrete,}\\ \int_{x\in X} A(x)p(x)\mathrm{d}x & \text{if } X \text{ is continuous.}\end{cases} \tag{3.1.2}$$

We will later use an alternative notation where we denote $\log f$ as ℓ. As will be evident later, this notation does not interfere with the definition of ℓ in Eq. (2.3.3). Since the Fisher information is dependent on θ, we can fix the value of θ during calculation, which makes $f(x|\theta)$ equal to the probability density $p_\theta(x)$.

For n independent experiments X^n, where $f(x^n|\theta) = \prod_{i=1}^n f(x_i|\theta)$, one can split the FI into

$$I_{X^n,ij}(\theta) = \prod_{i=1}^{n} I_{X_i,ij}(\theta). \tag{3.1.3}$$

A proof of this can be found in Section A.1.

For our example, the FI yields $I_{X^n}(\theta) = I_Y(\theta) = n/(\theta(1-\theta))$ [25]. This means that there's as much information about the θ contained in the measurement of Y as in the measurement of X^n, which coincides with Y being a sufficient measurement for θ.

To give another example of how the FI represents the information obtainable about a parameter from a measurement, let's consider the family of normal distributions

$$\mathcal{N}(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-(x-\mu)^2/(2\sigma^2)}. \tag{3.1.4}$$

These will now act as our statistical model $p_\theta(x) = \mathcal{N}(x|\theta)$, where $\theta = \{\theta_1,\theta_2\} = \{\mu,\sigma\}$. An observation would consist of a resulting value $x \in \mathbb{R}$, with its probabil-

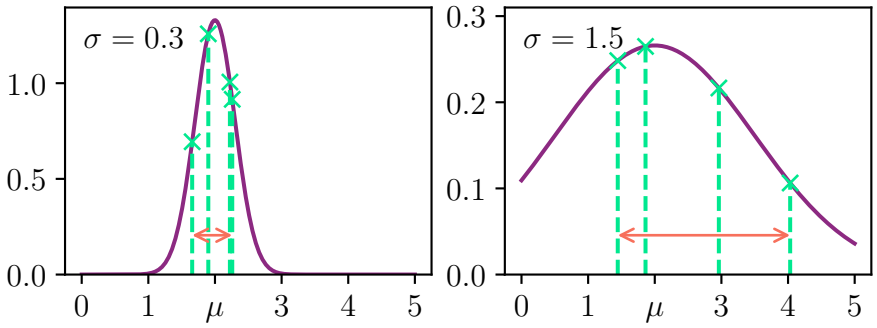


Figure 3.3: This figure shows two normal distributions centered around $\mu = 2$ with varying σ parameters. It also shows four samples chosen randomly according to the distribution. It's visible that for the case of a smaller variance σ, the points tend to be closer to the center and also less spread apart, which makes the information about μ contained in a measurement larger for a smaller variance.

ity distributed according to the statistical model. The FI from equation Eq. (3.1.1) can be derived as

$$I(\mu, \sigma) = \frac{1}{\sigma^2} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} . \tag{3.1.5}$$

We can now interpret the diagonal elements as measures of how much information a measurement contains about the corresponding parameter, and the off-diagonal elements as measurements of how similarly the model changes when varying the corresponding parameters. To give a specific example, let's look at the diagonal component corresponding to μ, $I_{11}(\mu, \sigma) = 1/\sigma^2$. This value indicates that for smaller σ, random values drawn from the distribution contain more information about μ than samples drawn from distributions with larger σ. For a visual guide, consider Fig. 3.3. It can be seen that the smaller value of σ results in a narrower spread of randomly drawn values, as indicated by the orange arrow. The values also tend to be closer to the mean value μ the smaller the variance σ is. Therefore, if we had to predict the value of μ from knowing only a few drawn samples, it would be easier to use the values drawn from the distribution with the smaller

variance. This is because the information contained in the samples measured by the FI is greater there. Keep in mind that all of these drawn values are randomly distributed. Therefore it's also possible to have two sets of random samples where the samples from the larger variance are better at predicting μ, but statistically speaking the smaller variance tends to perform better.

To conclude this chapter, the Fisher information is used in statistics to measure the amount of information one can gather about a parameter θ by measuring the outcome of a probability distribution $p_\theta(x_i)$. It is defined in Eq. (3.1.1).

3.2 Fisher information as the Riemannian metric

This section covers the basics of Riemannian geometry for neural networks. For more details, please refer to [29], from which most of the following information is taken.

3.2.1 Differentiable manifolds

To state the definition, a n-dimensional **manifold** S is a topological space so that for every point you can define a neighborhood around that point which is homeomorphic to an open subset of $\mathbb{R}^n$ [29]. A good example of this would be the surface of the earth, where locally viewed in the scale that we usually see things, the earth appears flat, but on a global scale the earth is obviously a sphere. This results, for example, in the shortest path between to points not being a straight line in maps of the world as a whole. Also the angles of a triangle don't sum up to $180°$ as they would in a subspace of $\mathbb{R}^n$. This results from conventional maps being subspaces of $\mathbb{R}^2$, although the earth is only homeomorphic to $\mathbb{R}^2$ in smaller local scales. If one tries to map the whole sphere into a map without gaps, one has to map the coordinates in a way that makes the shortest lines curved for example.

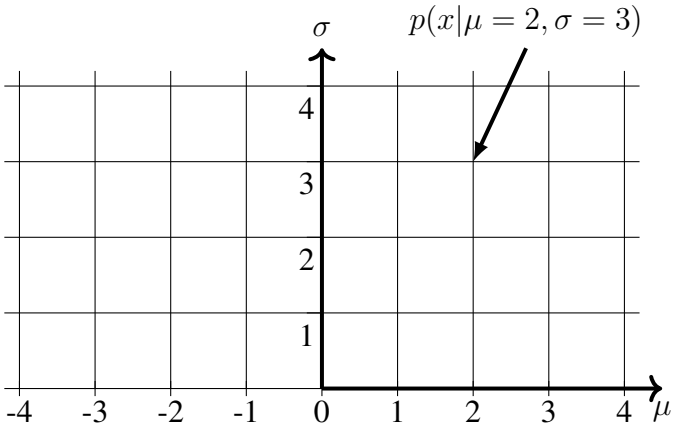


Figure 3.4: This figure illustrates the manifold of normal distributions. As the coordinates, μ and σ are used. Every point in this manifold represents a probability distribution, as indicated by the arrow.

Let's come back to the statistical models $f(x|\theta)$ we talked about in the last section. If a model is sufficiently smooth in θ, we can consider it to be a n-dimensional manifold, called a **statistical manifold**, where the n different θ components play the role of the coordinate system of the manifold. Since at one specific point in that manifold, the coordinate system and therefore the parameters are fixed, we can also refer to points in the statistical manifold as probability distributions p_θ as mentioned in Section 3.1.

For example let's consider normal distributions [29]

$$p_\theta(x) = \frac{1}{\sqrt{(2\pi\sigma^2)}} \mathrm{e}^{-(x-\mu)^2/(2\sigma^2)}, \tag{3.2.1}$$

where $\theta = \{\theta_1, \theta_2\} = \{\mu, \sigma\}$. We can now consider this family of distributions as a manifold, displayed in Fig. 3.4. This is a space in which every point represents a distribution $p_\theta(x)$.

It might also be clear that the coordinate system of a manifold is definable in multiple different ways. Although it's always given for our use case later, let's

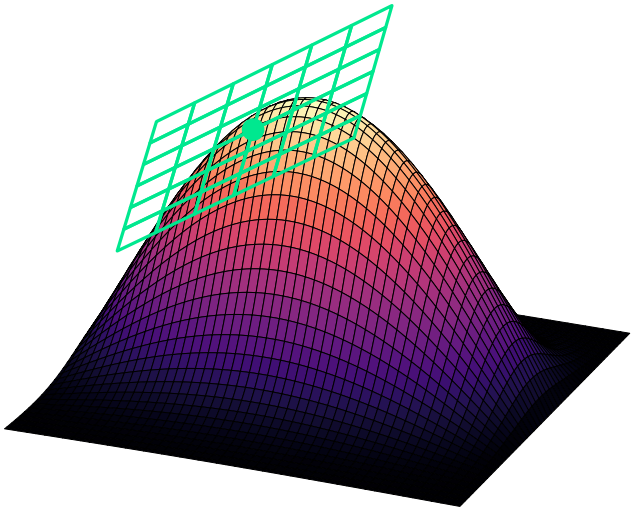


Figure 3.5: This figure contains a visualization of a tangent space of a manifold, which in this case is a 2D-surface.

therefore denote that in general, if we have coordinates θ we also need a mapping ϕ, which maps coordinates to points on a manifold. This means that by applying $\phi(p)$ to a point $p \in S$ the resulting vector in $\mathbb{R}^n$ resembles the coordinates of that point. We can also apply the inverse of that mapping to a set of coordinates to the point in the manifold that's represented by those coordinates.

3.2.2 Tangent space

The tangent space T_p of a manifold at point p is a vector space obtained by linearization of the manifold around p. For intuition purposes, let's consider the tangent plane of a 2D-surface in Section 3.2.2. There, the tangent space is simply a plane that touches the surface in one point, with derivatives adjusted to match the surface at that point. For the general case of n-dimensional manifolds, tangent spaces aren't simply tangent planes of surfaces in every case, therefore let's introduce a way to calculate a tangent space.

First, we will define curves $c(t)$ that are continuous mappings from an interval $[a, b] \in \mathbb{R}$ into the manifold S. In the parametric representation, the curve is given by $\theta(t)$. Now we can define what a tangent vector is.

Imagine a smooth, real function $f(\theta) : S \to \mathbb{R}$. We can now restrict this function to our predefined curve c by $f \circ c : [a, b] \to \mathbb{R}$. We'll denote this via $f(\{\theta(t)\})$ in the coordinate expression. The derivative Cf of this function is then given by [29]

$$Cf = \frac{\mathrm{d}f \circ c}{\mathrm{d}t} = \sum_{i=1}^{n} \frac{\partial f}{\partial \theta_i} \frac{\mathrm{d}\theta_i(t)}{\mathrm{d}t} = \underbrace{\left(\sum_{i=1}^{n} \frac{\mathrm{d}\theta_i}{\mathrm{d}t} \frac{\partial}{\partial \theta_i} \right)}_{\text{Operator } C} f. \tag{3.2.2}$$

Therefore we can associate a directional derivative operator C with each curve. The only dependence of this operator regarding the curve is $\frac{\mathrm{d}\theta_i}{\mathrm{d}t}$, where we might also clarify that this derivative itself depends on the point where it is calculated at. If the manifold is infinitely differentiable, the set of these mappings C at a fixed point on the manifold forms a n-dimensional vector space, called the **tangent space** T_p of that point p.

To make this more clear, let's look at the most simple basis for this vector space, which we call the natural basis of the tangent space. For this, we consider curves $c_1, c_2, \ldots, c_n$ through a point p_0, with

$$c_i(t) = \{\theta_1^0, \theta_2^0, \ldots, \theta_i^0 + (t - t_0), \ldots, \theta_n^0\}, \tag{3.2.3}$$

so that $c_i(t_0) = \{\theta_1^0, \theta_2^0, \ldots, \theta_n^0\} = p_0$ for every curve. The tangent vectors C_i then are simply the derivatives regarding their corresponding coordinate $C_i f = \partial/\partial\theta_i f$. We denote this in short by $C_i = \partial_i$. The n vectors ∂_i are linearly independent and form the natural basis for the tangent space [29] . This means that

any tangent vector A can be represented by [29]

$$A = \sum_{i=1}^{n} A_i \partial_i, \tag{3.2.4}$$

with components with respect to the natural basis A_i. If we are given a curve $c(t)$ going through $c(t_0)$ where the derivative operator C in t_0 is equivalent to the vector A, we can find the natural basis representation of A as [29]

$$A_i = \left. \frac{\mathrm{d}\theta_i}{\mathrm{d}t} \right|_{t_0}. \tag{3.2.5}$$

Now let's take a look at the case of manifolds of statistical models. First, we revisit the definition

$$\ell(x|\theta) = \log f(x|\theta), \tag{3.2.6}$$

and assume that for fixed θ, the n functions $\partial_i \ell(x|\theta)$ are linearly independent. This means that we can construct a vector space by defining [29]

$$T_\theta^{(1)} = \{A(x) | A(x) = A_i \partial_i \ell(x|\theta)\}. \tag{3.2.7}$$

We do this because there is a natural isomorphism between the tangent space T_θ and this space $T_\theta^{(1)}$ through [29]

$$\partial_i \in T_\theta \leftrightarrow \partial_i \ell(x|\theta) \in T_\theta^{(1)}. \tag{3.2.8}$$

We will call $T_\theta^{(1)}$ the **1-representation** of our tangent space for the statistical models [29]. We will now use this vector to define an inner product on the tangent space and its 1-representation.

3.2.3 Riemannian metric and Fisher information

When the inner product of the tangent spaces T_p is defined, the manifold is called a **Riemannian manifold** [29].

Let's first consider the inner product of the 1-representation space. Let $A(x)$ and $B(x)$ be 1-representations of A and $B \in T_\theta$. It is intuitive to define the inner product as

$$\langle A(x), B(x) \rangle = \underset{x \in X}{E}\Big[A(x)B(x)\Big], \tag{3.2.9}$$

with the expectation value $E[\cdot]$ [29]. Since the tangent space is isomorphic to its 1-representation, the inner product also translates via

$$\langle A, B \rangle = \langle A(x), B(x) \rangle. \tag{3.2.10}$$

This also means that we can calculate the inner product of the basis vectors as [29]

$$\begin{aligned} g_{ij}(\theta) = \langle \partial_i, \partial_j \rangle &= \langle \partial_i \ell(x|\theta), \partial_j \ell(x|\theta) \rangle \\ &= \underset{x \in X}{E}\Big[\partial_i \ell(x|\theta), \partial_j \ell(x|\theta)\Big]. \end{aligned} \tag{3.2.11}$$

The resulting object $g_{ij}(\theta)$ is called the **Riemannian metric tensor** of the manifold [29]. We can see that the Riemannian metric tensor for the statistical model is equivalent to the Fisher information. It might be of interest to note here that we assume that ℓ only depends explicitly on θ. If we denote these as implicit dependencies, we have to replace the partial derivatives with absolute ones.

The inner product of two vectors can now be expressed with the metric tensor as [29]

$$\langle A, B \rangle = \sum_{i,j} A_i B_j g_{ij}(\theta) \tag{3.2.12}$$

in the component form.

Using this representation of the inner product, we can define various things. For example, the length of a vector A is defined as $|A|^2 = \sum_{i,j} A_i A_j g_{ij}$, the orthogonality of two vectors as their inner product being zero, and the distance between two points $\theta^{(0)}$ and $\theta^{(1)}$ along the curve c as [29]

$$s = \int_{t_0}^{t_1} \sum_{i,j} \sqrt{g_{ij} \frac{\mathrm{d}\theta_i}{\mathrm{d}t} \frac{\mathrm{d}\theta_j}{\mathrm{d}t}} \mathrm{d}t. \tag{3.2.13}$$

This also introduces the concept of **Riemannian geodesics**. These are the curves that connect two points via the minimal distance between the two.

Another representation of the metric tensor or the FI is [29]

$$g_{ij}(\theta) = -\underset{x \in X}{E} \Big[\partial_i \partial_j \ell(x|\theta) \Big]. \tag{3.2.14}$$

A proof of this is denoted in Section A.2.

3.2.4 Scalar curvature and Christoffel symbols

Here, we will only give the definitions needed to compute the scalar curvature of a statistical manifold along with brief explanations. A full understanding requires much more mathematics than we will go over here. We will refer the interested reader to [29] and [30] for further details.

Note that we don't use covariant and contravariant indices, since we don't need them in the scope of this thesis. The notation of uppercase or lowercase indices is just to be consistent with the notation from general relativity and should be of no further concern for our experiments. The parameters with lowercase indices defined previously translate directly to the ones with uppercase indices here. We will also use Einstein notation for these equations.

First we define the Christoffel symbols which can be calculated from the Riemannian metric via [30]

$$\Gamma^i_{jk} = \frac{1}{2} g^{im} \left(\frac{\partial g_{mk}}{\partial \theta^l} + \frac{\partial g_{ml}}{\partial \theta^k} - \frac{\partial g_{kl}}{\partial \theta^m} \right), \tag{3.2.15}$$

where $g^{ij} = (g^{-1})_{ij}$ are the components of the inverse of the metric. They originally arise from the formula $\partial_k \vec{e}_j = \Gamma^i_{jk} \vec{e}_k$, where they describe how basis vectors change when moving in the parameter space [30]. From these, we can define the Riemannian curvature tensor as [30]

$$R^i_{jkl} = \partial_k \Gamma^i_{jl} - \partial_l \Gamma^i_{jk} + \Gamma^i_{mk} \Gamma^m_{jl} - \Gamma^i_{ml} \Gamma^m_{jk}. \tag{3.2.16}$$

The Riemannian curvature tensor is the full description of curvature on a manifold. It generalizes the well known concept of curvature on a curve to multidimensional, abstract manifolds. We can compress its information down into a single scalar number by defining the Ricci scalar curvature as [30]

$$R = g^{ij} R^m_{imj}. \tag{3.2.17}$$

The scalar curvature can be viewed as a measure of how much the manifold at the point of computation differs from flat space. For euclidean spaces, the scalar curvature is 0 [30]. To give more intuition about the curvature we can refer to its interpretation in general relativity. It considers the physical space to be a curved manifold with the coordinate of time added to the 3 spatial dimensions. Curvature results from objects with mass in space [30]. The concept of curvature here can be imagined similar to how a ball placed on an elastic surface will curve it around the ball. When viewed from the top, the space seems flat, but when moving towards the ball on the surface, the distance traveled on the elastic band gets larger

compared to the distance traveled in the top down perspective, the closer we come to the ball. This results from the ball stretching the manifold represented by the elastic band. Similarly for neural networks, the distance from the top down view is represented by the distance in the coordinate frame. The actual distance moved on the elastic band is described by the distance between points in the manifold, which in our case means how much the loss functions at those coordinates differ from each other. How exactly this translates to neural networks is explained in Section 3.2.5.

3.2.5 Application to neural networks

Now that we've laid down the mathematical fundamentals of how the FI can be considered the metric of a statistical manifold, we can apply this to our neural network optimization by introducing a way of viewing them as statistical models. Let's first define the statistical model by the probability $p_\theta(\mathbf{x}_i, \mathbf{y}_i) = e^{-\ell_\theta(\mathbf{x}_i, \mathbf{y}_i)}$, where ℓ is the subloss function defined in Eq. (2.3.3). You can think of this as a kind of probability that the network characterized by the parameters θ and the subloss function ℓ can generate the outputs $\mathbf{y}_i$ from the inputs $\mathbf{x}_i$. In reality, of course, it's not a probability that the network reproduces the output, since that would be either true or false. It's rather a measure of how close the network's prediction is to the actual target. We can think of it and use it as a probability because of the exponential function, which results in a probability of 1 if the network output matches the target (which means $\ell = 0$) and shrinks towards 0 for larger discrepancies.

Now we can also see that the definitions of two different ℓ don't interfere with each other. In the context of a statistical model, ℓ is defined as the logarithm of $f(x|\theta)$, in the context of neural network training, ℓ is defined as the subloss (see Eq. (2.3.3)) that we used for the probability above. This means that in the case of

a statistical manifold of neural network training, the negative subloss $-\ell_\theta(\mathbf{x}_i\mathbf{y}_i)$ is equal to the logarithm of the statistical model $\ell(\mathbf{x}_i, \mathbf{y}_i|\theta)$. Since we will only use ℓ to calculate the FI, where the two minus signs cancel, we can think of the ℓ in the definition of the FI in Eq. (3.1.1) as being the subloss of a network defined in Eq. (2.3.3). How the components of the resulting matrix can be interpreted will be discussed at the end of Section 3.3.

3.3 Intuitive explanations

In the previous sections we've introduced Fisher information, first in a statistical context and later as the metric of a statistical manifold, in order to apply some of its insights to neural network training.

Since the information provided in the last sections has been mathematically abstract and lacking intuition, let's quickly recap the basics of what we need to know and understand about the Fisher information for the rest of this work.

First, let's state the definition again. The FI is defined as

$$\begin{aligned} I_{ij} &= \underset{x\in X}{E}\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i}\log f(x|\theta)\cdot\frac{\mathrm{d}}{\mathrm{d}\theta_j}\log f(x|\theta)\right] \\ &= \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E}\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i}\ell_\theta(\mathbf{x}_i,\mathbf{y}_i)\cdot\frac{\mathrm{d}}{\mathrm{d}\theta_j}\ell_\theta(\mathbf{x}_i,\mathbf{y}_i)\right], \end{aligned} \tag{3.3.1}$$

where the notation in the first line is used in the context of statistics and the one from the second line in the context of the manifold of neural network training.

We first introduced the FI as a statistical measure of how much information the measurement of a random variable contains about its underlying parameters.

For example, let's say our experiment consists of throwing a biased coin, where the probability of heads is the underlying parameter. We can conduct an experiment to estimate the probability of the biased coin. Let's say we make 10 throws

twice. The first time, we will use the full observation for the parameter estimation. We will exactly know the results of the coin toss *for every single one* of the 10 throws. The second time, we will only know how often heads came up *in total* for the 10 throws. We will disregard information about when exactly during these 10 trials we measured heads or tails. The Fisher information with respect to the parameter yields the same value for both of these observables. This therefore means that the information about the probability of heads contained in the measurement is the same for both observations [25]. To estimate the bias of the coin, it is perfectly sufficient to only write down the total number of heads instead of the entire observation.

Going further, we introduced another application of the FI as the Riemannian metric of the statistical manifold corresponding to a neural network. For this, we considered a neural network along with a dataset and loss function as a statistical model. We did this by introducing a kind of probability distribution $p_\theta(\mathbf{x}_i, \mathbf{y}_i) = \mathrm{e}^{-\ell_\theta(\mathbf{x}_i, \mathbf{y}_i)}$. This probability measures whether our network with parameters θ produces the correct output $\mathbf{y}_i$ when given the input $\mathbf{x}_i$. The family of probability distributions $\{p_\theta\}$ forms a statistical manifold [29]. It is a topological space with coordinates $\theta = \{\theta_1, \ldots, \theta_n\}$, where every point in the space corresponds to a probability function that depends on the θ coordinates. Locally, it is isometric to a subset of $\mathbb{R}^n$. Globally, euclidean geometry doesn't apply, which for example makes the shortest paths between points in the manifold curves in the coordinate system. Because every point in the space is a probability function, it's very hard to properly imagine lines between points and even the seemingly simple concept of distance becomes very abstract. Therefore, our next goal was defining how one can measure distance and characterize curvature in this space.

To be able to define what distance means, we first introduced the concept of a tangent spaces. A tangent space is an abstract vector space defined at a point in the

manifold, where the vectors are differential operators corresponding to the derivatives along curves through the point [29]. The reason we defined it is because by defining an inner product $\langle \cdot, \cdot \rangle$ on the tangent spaces, we can obtain a metric on the manifold through $g_{ij} = \langle \mathbf{e}_i, \mathbf{e}_j \rangle$ [29]. The basis vectors $\mathbf{e}_i$ on the tangent spaces are abstract concepts, which makes finding an intuitive definition of an inner product non trivial. That's why we introduced a new space, isomorph to the tangent space, where defining the inner product feels natural. The ismorphism was introduced as [29]

$$\partial_i \leftrightarrow \partial_i \ell(x|\theta) \tag{3.3.2}$$

and maps the derivative operators onto actual derivatives of the logarithm of the statistical model f. To obtain the inner product of two derivative operators we then defined [29]

$$\langle \partial_i, \partial_j \rangle = \langle \partial_i \ell(x|\theta), \partial_j \ell(x|\theta) \rangle, \tag{3.3.3}$$

where we naturally assume the inner product between the two distributions as

$$\langle \partial_i \ell_\theta(\mathbf{x}_i, \mathbf{y}_i), \partial_j \ell_\theta(\mathbf{x}_i, \mathbf{y}_i) \rangle = \underset{(\mathbf{x}_i, \mathbf{y}_i) \in D}{E} \big[\partial_i \ell_\theta(\mathbf{x}_i, \mathbf{y}_i) \cdot \partial_j \ell_\theta(\mathbf{x}_i, \mathbf{y}_i) \big]. \tag{3.3.4}$$

Now we arrived at the definition of the metric of our manifold, which is equivalent to the Fisher information matrix

$$I_{ij} = g_{ij} = \underset{(\mathbf{x}_i, \mathbf{y}_i) \in D}{E} \big[\partial_i \ell_\theta(\mathbf{x}_i, \mathbf{y}_i) \cdot \partial_j \ell_\theta(\mathbf{x}_i, \mathbf{y}_i) \big]. \tag{3.3.5}$$

Since this equation is a bit easier to grasp, we can finally try to get some intuition about the connections of the concepts mentioned before.

Distance between two points in a curved manifold along a curve c is defined as

[29]

$$s = \int_{t_0}^{t_1} \sum_{i,j} \sqrt{g_{ij} \frac{\mathrm{d}\theta_i}{\mathrm{d}t} \frac{\mathrm{d}\theta_j}{\mathrm{d}t}} \mathrm{d}t. \tag{3.3.6}$$

If we only move along one coordinate θ_i on the curve, the distance reduces to

$$s = \int_{t_0}^{t_1} \sqrt{g_{ii}} \left| \frac{\mathrm{d}\theta_i}{\mathrm{d}t} \right| \mathrm{d}t. \tag{3.3.7}$$

Therefore the diagonal components of the FI tell us how far we move across the manifold when we change parameter θ_i. Through inspection of the equation for the metric [1], it's clear that the distance between points created by changing a parameter relates to the expected change in the logarithm of the probability distribution. We can directly map the parameter space onto the manifold of probabilities, but we need to reconsider the notion of distance for the manifold because distance in the euclidean parameter space doesn't generally translate to distance in the manifold, which is a measure of difference between the probabilities.

Finally let's examine the components of the metric for the neural network. The diagonal components are the expectation values of the squared derivative of the loss. This means that the diagonal values represent how much our subloss changes when we vary the corresponding parameter. The off-diagonal values represent how similar the change in the subloss function is under changes in the two corresponding parameters. This information about the change of the loss regarding the parameters is now of high interest when considering neural network training. In general, calculating the FI computationally is very costly though, because the number of parameters can be very high for complex tasks. That's why the results of this work consider a few observations resulting from the Fisher matrix and investigate some possibilities to obtain them from other, computationally easier, observables of training instead.

[1] See Eq. (3.3.5).

3.4 Investigation of physical phase transitions using Fisher information

The Fisher information can also be useful in physical context. It's possible to use it to find phase transitions in thermodynamic systems. We will later look at the possibility of applying this to neural network training in search of processes equivalent to phase transitions, which may lead to more insight into what exactly influences training.

To explain how the FI can be used to find phase transitions let's briefly review [31].

Given an equilibrated physical system in a large thermal heat bath, statistical models of those systems usually deal with Gibbs measures of the form

$$p(x|\theta) = \frac{1}{Z(\theta)} \exp\left(\sum_i \theta_i X_i(x)\right). \tag{3.4.1}$$

Here, x represent the microstates of the system, X_i are time-independent functions called collective variables and θ_i represent the time-dependent thermodynamic variables. These θ_i could be, for example, temperature, pressure, magnetic fields etc. They will be used as parameters θ of the FI.

Using the thermodynamic variables one can construct thermodynamic potentials. One example used in the paper is the Helmholtz free energy

$$A = -k_B T \ln Z(\theta), \tag{3.4.2}$$

where we consider $k_B T = 1/\beta$ to be one of the thermodynamic variables.

Furthermore, it's stated that a classification of phase transitions typically requires an examination of the derivatives of the thermodynamic potential. Specifically,

there are cases where an order parameter ϕ^i describing the phase transition is representable as a negative derivative of the potential over some thermodynamic variable θ^i. In this case, the diagonal components Fisher information defined by the probability distributions in Eq. (3.4.1) and Eq. (3.1.1) can be written as

$$I_{ii} = \beta \frac{\partial \phi^i}{\partial \theta^i}. \tag{3.4.3}$$

There are second order phase transitions, where the order parameter (which is a derivative of the thermodynamic potential) changes continuously while its derivative diverges. Using this relationship, one can identify those phase transitions by searching for divergences in the diagonal components of the Fisher information.
In addition to that, [32] visits the same thermodynamic metric described by the Fisher information. Here, the scalar curvature corresponding to the metric R is introduced as a measure of complexity for the physical systems. It is stated that for all models that they've considered so far, R diverges at, and only at, the phase transition.
This gives us two ways of finding phase transitions, which could also be applied to the Fisher information of neural network training to possibly search for analogues of physical phase transitions in the training.

4 | Neural Tangent Kernel

This chapter introduces the **Neural Tangent Kernel** (NTK). While the Fisher information describes the parameter derivative correlations of the loss functions, the NTK describes the similarities in the neural network output regarding the parameters.

Section 4.1 presents a derivation of the NTK using the Gradient Flow assumption. Section 4.2 provides further explanations on why the NTK arises in this context and how it can be interpreted.

4.1 Derivation from Gradient Flow

4.1.1 What we call time

A simple and intuitive way to introduce the NTK is via **Gradient Flow**, which is an assumption related to the GD algorithm from Section 2.3.3. To quickly recap the update step from Gradient Descent, it's defined as

$$\theta' = \theta - \eta \nabla_\theta \mathscr{L} \left(\{ (f_\theta(\mathbf{x}_i), \mathbf{y}_i) \}_{i=1}^N \right). \tag{4.1.1}$$

The "flow" aspect arises when we start to ignore the discrete nature of these update steps and assume that the θ parameters change continuously by assuming $\eta \to 0$. A visual demonstration of how this changes the evolution of the parameters can be seen in Fig. 4.1. The dashed markers represent the evolution under regular GD, while the solid line represents the evolution for gradient flow. To do this, we first introduce a notion of "time" into our system. We try to visualize the optimization process as an evolution of our parameters θ through this variable called time, which converts updating the parameters into moving further along the

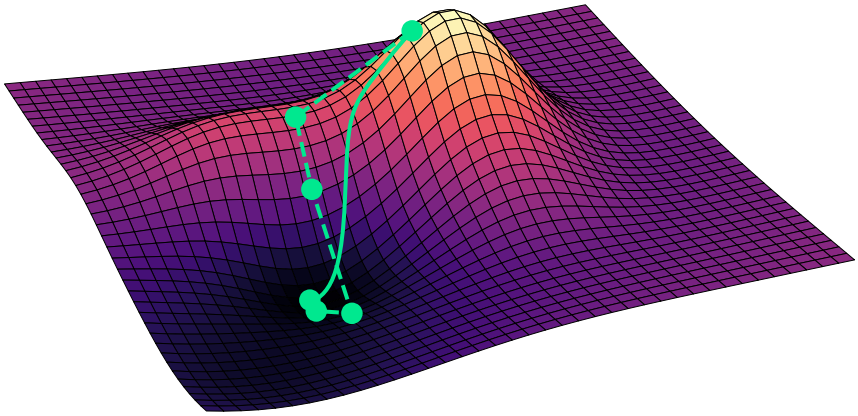


Figure 4.1: This graph shows the difference between Gradient Descent and Gradient Flow. The dashed line along with the circle markers show the positions during regular GD with finite step sizes, the solid line shows the path of the parameters under gradient flow.

timeline of our parameters. We translate $\theta \to \theta(t)$ and $\theta' \to \theta(t + \Delta t)$. This time in our system doesn't work exactly the same way as physical time, but since the process of optimizing parameters and changing them is always associated with the expenditure of physical time, it is intuitive to refer to our system's propagation variable as "time".

Returning to the GD algorithm, since the learning rate η affects the size of our update step, we will refer to η as the amount of time it takes to update a parameter $\theta' \to \theta(t + \eta)$. The whole GD algorithm then becomes

$$\theta(t + \eta) = \theta(t) - \eta \nabla_{\theta(t)} \mathscr{L} \left(\{ (f_{\theta(t)}(\mathbf{x}_i), \mathbf{y}_i) \}_{i=1}^{N} \right) \tag{4.1.2}$$

which we can rewrite to

$$\frac{\theta(t + \eta) - \theta(t)}{\eta} = -\nabla_{\theta(t)} \mathscr{L} \left(\{ (f_{\theta(t)}(\mathbf{x}_i), \mathbf{y}_i) \}_{i=1}^{N} \right). \tag{4.1.3}$$

When observing the term on the left side, readers who are familiar with calculus might recognize that it looks similar to the definition of a derivative with respect to time when $\eta \to 0$

$$\frac{\mathrm{d}}{\mathrm{d}t}\theta(t) = \lim_{\eta \to 0} \frac{\theta(t+\eta) - \theta(t)}{\eta}. \tag{4.1.4}$$

This means that for very infinitesimal learning rates we can approximate the GD as

$$\frac{\mathrm{d}}{\mathrm{d}t}\theta(t) = -\nabla_{\theta(t)} \mathscr{L}\left(\{(f_{\theta(t)}(\mathbf{x}_i), \mathbf{y}_i)\}_{i=1}^{N}\right) \tag{4.1.5}$$

with the derivative of θ along the assumed time variable.

For visual simplicity reasons, let's define the j-th component of the network output for the i-th input point $f_{\theta(t)}(\mathbf{x}_i)_j$ as F_{ij} and assume Einstein summation for the rest of the chapter. This means that when an index is occuring twice, we assume a hidden summation over all possible values for this index (for example $a_k b_k = \sum_k a_k b_k$).

Because it will be convenient later, we also spell out one component of the ∇_θ derivation of the loss function further by using the chain rule as

$$\begin{aligned} \frac{\mathrm{d}\theta_k}{\mathrm{d}t} &= -\frac{\mathrm{d}}{\mathrm{d}\theta_k} \mathscr{L}\left(\{(f_{\theta(t)}(\mathbf{x}_i), \mathbf{y}_i)\}_{i=1}^{N}\right) \\ &= -\frac{\partial \mathscr{L}}{\partial F_{ij}} \cdot \frac{\mathrm{d}F_{ij}}{\mathrm{d}\theta_k}. \end{aligned} \tag{4.1.6}$$

4.1.2 Derivation of the NTK

This notion of time affects not only the parameters, but also everything that depends on them. For example, since the network output of a fixed architecture for a given input data point only depends on the parameters of the network, it can also be mathematically viewed as dependent on the time $f_\theta(\mathbf{x}_i) \to f_{\theta(t)}(\mathbf{x}_i)$. This means we can also calculate the derivative of one of the network outputs

$f_{\theta(t)}(\mathbf{x}_i)_j = F_{ij}$ to

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}F_{ij} &= \frac{\partial F_{ij}}{\partial \theta_k}\frac{\mathrm{d}\theta_k}{\mathrm{d}t} \\ &= \frac{\partial F_{ij}}{\partial \theta_k}\left(-\frac{\partial \mathscr{L}}{\partial F_{ij}}\frac{\mathrm{d}F_{ij}}{\mathrm{d}\theta_k}\right) \\ &= -\underbrace{\frac{\partial F_{ij}}{\partial \theta_k}\frac{\partial F_{lm}}{\partial \theta_k}}_{=:\,\Lambda_{iljm}}\frac{\partial \mathscr{L}}{\partial F_{lm}}.\end{aligned} \tag{4.1.7}$$

In the last line, we used the fact that F_{ij} explicitly depends on θ, which means that the total and partial derivative ar interchangeable. The rank 4 tensor Λ is what we call the Neural Tangent Kernel for GD. We sorted the indices of this matrix so that the first two refer to the input points of f and the last two refer to the components of the output dimensions of the neural network. Note that this NTK is derived directly from the update algorithm of the GD for infinitely small η and only holds for linear gradient based algorithms.

Another way to derive the NTK using an approximation of $\Delta\mathscr{L}$ for small η can be seen around page 196 of [22]. The NTK is derived there in a more generic form using a learning rate tensor η_{ij}. Here, it's important to note that the assumption of continuous parameter updates isn't necessary for the NTK. It also arises from the simple assumption of $\eta \to 0$, which makes it a viable observable for GD algorithms with small learning rates at all times t.

4.2 Interpretation

Through Eq. (4.1.7), the NTK is defined as

$$\Lambda_{ij\alpha\beta} = \nabla_\theta f_\theta(\mathbf{x}_i)_\alpha \cdot \nabla_\theta f_\theta(\mathbf{x}_j)_\beta. \tag{4.2.1}$$

For this explanation, let's assume that the neural network has only one output $f_\theta(\mathbf{x}_i)_\alpha \rightarrow f_\theta(\mathbf{x}_i)$, which gets rid of the α and β indices for us and makes the NTK a regular matrix. We can use this matrix to calculate the time derivative of the neural network output similar to Eq. (4.1.7) by

$$\frac{\mathrm{d}}{\mathrm{d}t} f_{\theta(t)}(\mathbf{x}_i) = \sum_j \Lambda_{ij} \left(-\frac{\partial \mathscr{L}}{\partial f_{\theta(t)}(\mathbf{x}_j)} \right). \tag{4.2.2}$$

This means that the evolution of the network output for input $\mathbf{x}_i$ is influenced by the outputs for other input values $\mathbf{x}_j$ through the NTK. We can investigate this further by taking a look at the definition of the NTK above in Eq. (4.2.1).
A mathematical kernel K is defined as a function [33]

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_i), \tag{4.2.3}$$

if it can be expressed using a so-called "feature map" function ϕ, that maps the points $\mathbf{x}_i$ into a higher dimensional inner product space called the "feature space". The kernel K assigns a scalar value to the two points by comparing their "features" via a scalar product in the feature space. For more specific information on Kernels and how they are used, please refer to [33].
In our case the feature map is ∇_θ which maps our scalar network output onto a vector that contains the derivatives of the network output with respect to all of the various parameters. This is our feature space, because what we're interested in is how moving in θ space changes the output values of our network. If we compare those two mappings we get a value that measures how closely the direction that increases $f_\theta(\mathbf{x}_i)$ most effectively matches with the direction that increases $f_\theta(\mathbf{x}_j)$ most effectively (which are the directions the gradients point to). For example, $\Lambda_{ij} = 0$ means that varying θ in the direction that changes $f_\theta(\mathbf{x}_i)$ most effectively results in 0 change for $f_\theta(\mathbf{x}_j)$.

Coming back to Eq. (4.2.2) the right side is the negative loss derivative with respect to $f_\theta(\mathbf{x}_j)$. Since we update the parameters in a way that minimizes the loss most effectively in gradient flow, the negative loss derivative with respect to $f_\theta(\mathbf{x}_j)$ describes how beneficial increasing $f_\theta(\mathbf{x}_j)$ is for our system. If we now multiply this with the NTK, which is a kernel that describes how similar $f_\theta(\mathbf{x}_i)$ and $f_\theta(\mathbf{x}_j)$ behave when changing theta, and sum everything up, we get the change of the function value $f_\theta(\mathbf{x}_i)$ as result.

In θ space, we can directly relate the evolution of θ to its negative loss derivative, because the parameters themselves are what we update with our algorithm. For the derivative of the output values we need the NTK as well, because we don't change the output values directly. If the loss derivative with respect to an output value tells the system that it *should increase this output value*, the system *changes the parameters* in a way that increases this output value. The difference to the derivative of θ is that the change of parameters now doesn't reflect in just one output, but in every other output value as well. That's where the NTK arises in the equation. The NTK is, in a way, a translation of the GD into the space of outputs of the neural network.

All of the explanations above apply to the 4 dimensional hypermatrix-form of the NTK for multidimensional neural network output as well, which can be seen when comparing Eq. (4.2.2) to Eq. (4.1.7).

5 | Investigations

Building on the foundations from Chapter 2, Chapter 3, and Chapter 4, this chapter presents and investigates a novel relation between the trace of the Fisher information and the NTK in Section 5.1. Furthermore, it provides example calculations of all previously mentioned observables for a simple 2-parameter network in Section 5.2.

5.1 Fisher Trace - NTK relation

To motivate the novel relation presented in this section, let's first investigate the size of the Fisher information. For a neural network with N parameters, the Fisher information is a matrix with N^2 entries[1]. To illustrate how large those resulting matrices are, let's consider a network containing 3 hidden layers with a respective width of 128 neurons that's trained on the 28×28 matrices of the MNIST dataset. Such a network consists of over 130000 parameters, which results in over 10^{10} entries in the FI. When saving the entries as 32-bit floating point numbers, the matrix would take over 67 GB of space. Calculating this matrix in under 1 hour would require over 2 million entries to be calculated per second. Calculating the full matrix to optimize training is therefore computationally intractable. That's why we're going to look at an equation for calculating the trace of the Fisher information, or Fisher Trace for short, through other mathematical objects.

[1] see Eq. (3.1.1).

Let's first inspect the Fisher Trace by using the chain rule of derivation as

$$\begin{aligned}
\operatorname{tr}(I) &= \sum_{\alpha=1}^{N_P} I_{\alpha\alpha} \\
&= \sum_{\alpha=1}^{N_P} \left\{ \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \left[\frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \cdot \frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \right] \right\} \\
&= \sum_{\alpha=1}^{N_P} \left\{ \frac{1}{N} \sum_{i=1}^{N_D} \left[\sum_{a=1}^{N_O} \left(\frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_a} \frac{\mathrm{d} f_\theta(\mathbf{x}_i)_a}{\mathrm{d}\theta_\alpha} \right) \cdot \sum_{b=1}^{N_O} \left(\frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_b} \frac{\mathrm{d} f_\theta(\mathbf{x}_i)_b}{\mathrm{d}\theta_\alpha} \right) \right] \right\} \\
&= \frac{1}{N} \sum_{i=1}^{N_D} \sum_{a=1}^{N_O} \sum_{b=1}^{N_O} \left[\frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_a} \frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_b} \cdot \underbrace{\sum_{\alpha=1}^{N_P} \left(\frac{\mathrm{d} f_\theta(\mathbf{x}_i)_a}{\mathrm{d}\theta_\alpha} \frac{\mathrm{d} f_\theta(\mathbf{x}_i)_b}{\mathrm{d}\theta_\alpha} \right)}_{\Lambda_{iiab}} \right],
\end{aligned} \tag{5.1.1}$$

with the amount of parameters N_P, the amount of dataset pairs N_D and the output dimension of the network N_O. By identifying the entries of the NTK in the last line and simplifying the notation, we can rewrite the relation as

$$\operatorname{tr}(I) = \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \left[\sum_{a,b} \left(\frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_a} \frac{\partial \ell}{\partial f_\theta(\mathbf{x}_i)_b} \cdot \Lambda_{iiab} \right) \right]. \tag{5.1.2}$$

As far as we know, this relation has never been presented before in literature. Let's look at how we can interpret its components as a splitting of the driving force behind neural network training into a part dependent on the architecture and a part dependent on the loss function. It is important to note that there are two main parts on the right hand side: One is the loss derivative with respect to the neural network output, the other consists of diagonal components of the NTK. By diagonal, we refer to components of the NTK where the first two indices refer to the same input point. The loss derivatives are independent of the neural network architecture, whereas the NTK is independent of the loss function.

We can also rewrite the Fisher Trace as

$$\begin{aligned}\mathrm{tr}(I) &= \sum_{\alpha=1}^{N_P} I_{\alpha\alpha} \\ &= \sum_{\alpha=1}^{N_P} \left\{ \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \left[\frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \cdot \frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \right] \right\} \\ &= \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \left\{ \sum_{\alpha=1}^{N_P} \left[\frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \cdot \frac{\mathrm{d}}{\mathrm{d}\theta_\alpha} \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \right] \right\} \\ &= \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \left| \nabla_\theta \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i) \right|^2 ,\end{aligned} \tag{5.1.3}$$

which is closely related to the size of the parameter update for Gradient Descent

$$\theta' - \theta = \eta \underset{(\mathbf{x}_i,\mathbf{y}_i)\in D}{E} \nabla_\theta \ell(f_\theta(\mathbf{x}_i),\mathbf{y}_i). \tag{5.1.4}$$

This means that we can split the Fisher Trace, which describes the size of the update of the GD algorithm, into a part depending on the loss function and a part depending on the architecture of the network.
Going further, we will conduct some experiments to investigate which of those two parts is the dominant driving force in the evolution of the Fisher Trace, and therefore, also the driving force behind the training of the neural network. Specifically, we will look at the time evolution of the Fisher Trace in comparison to the trace of the NTK, to see how similar they evolve during training.

5.1.1 Comparison of traces of NTK and Fisher information

To test how similar the the traces of NTK and Fisher information evolve, we trained multiple networks on the MNIST dataset[2] and recorded the traces of the FI and NTK. We used networks consisting of 2 hidden layers that were equipped

[2] See Section 2.3.1.

with the ReLU activation function. The network widths varied between 128 and 784. The losses used were the L_p-Norm loss [34]

$$d_p(f_\theta(\mathbf{x}), \mathbf{y}) = \sqrt[p]{\sum_{i=1}^{N_O} |f_\theta(\mathbf{x}_i) - \mathbf{y}_i|^p}, \tag{5.1.5}$$

the Mean power loss of order n

$$d_n(f_\theta(\mathbf{x}), \mathbf{y}) = \frac{1}{N_O} \sum_{i=1}^{N_O} (f_\theta(\mathbf{x}_i) - \mathbf{y}_i)^n, \tag{5.1.6}$$

and the Cross-entropy loss [20]

$$d(f_\theta(\mathbf{x}), \mathbf{y}) = \sum_{i=1}^{N_O} -\mathbf{y}_i \log(\sigma_i(f_\theta(\mathbf{x}))). \tag{5.1.7}$$

Here, we used the definition of the distance d from Eq. (2.3.4), and of the softmax function σ_i from Eq. (2.2.2). The experiment was conducted for orders of n or p equal to 2, 4 and 6. We trained each combination of architecture and loss function for a total of 5 times and calculated mean values and standard deviations of the traces using the results. The networks were trained using the Adam optimizer [24].

As an illustrative example, the results for L_2-norm loss, Mean power loss of order $n = 2$, as well as the result for Cross-entropy loss for a network with a width of 128 neurons are depicted in Fig. 5.1. The solid line represents the mean of the 5 experiments, the translucent area marks one standard deviation above and under the mean value.

It's visible that for the L_2-norm, the traces evolve almost proportionally to each other, but for the other loss functions they evolve significantly different. This indicates that the loss derivatives in Eq. (5.1.2) play an important role and can't be

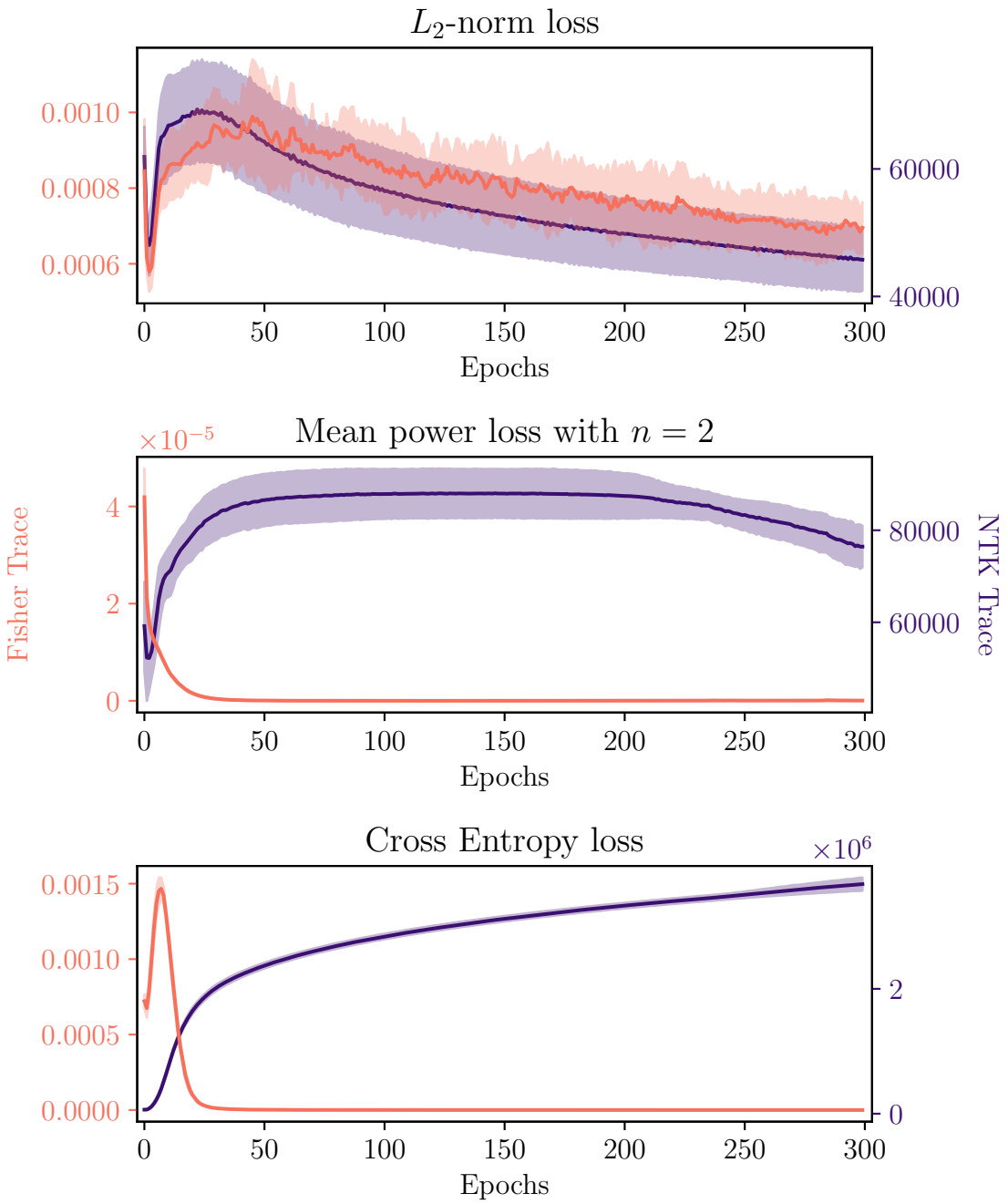


Figure 5.1: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 128 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

neglected. The NTK Trace doesn't seem to be a good approximation of the Fisher Trace in general. For which cases the NTK is the driving force of training and can be used as a good approximation of the Fisher Trace needs to be investigated further and can't be answered from just the data generated here.
More results from the experiment for different losses and also a larger network width can be found in Appendix B. From there it seems like the deviations between the two traces increase for growing network width.

5.2 2-parameter network analysis

As previously mentioned, it's computationally expensive to compute the full Fisher information. A single matrix for the smallest network from the previous section would already be over 50 GB in size. Therefore, to be able to compute and visualize the trace of the NTK, the Fisher information and the curvature R, this section covers the training of one of the simplest networks possible. Its simplicity allows us to compute the observables for a whole grid of parameters instead of just the path of parameters during training. The measurements of the full surfaces provided here might lead us to a better understanding of the meaning and impact of the Fisher information, NTK and scalar curvature.

5.2.1 The Network

The network used for the experiment in Section 5.2.3 consists of a single neuron with two inputs, two weights, and no bias. It uses a sigmoid function [13] as activation function. This means that we can express the output of the network by

$$f_\theta(x, y) = \frac{1}{1 + \exp(-5(\theta_1 x + \theta_2 y))}, \tag{5.2.1}$$

with inputs x and y. The factor 5 was added to make the sigmoid function steeper. The network output, which is a number between 0 and 1, represents a prediction whether the input data belongs to the class with target 0 or the class with target 1. The loss functions considered are the same as in Section 5.1.

5.2.2 The Dataset

The dataset used in Section 5.2.3 is depicted in Fig. 5.2. We created the input data by generating a set of randomly distributed (x, y) pairs within the square from 0 to 1. The input pair is sorted into the class of target 1 if the y coordinate is bigger than the x coordinate, and into the class of target 0 otherwise.

5.2.3 Results

Let's inspect the loss surfaces of the network, depicted in Fig. 5.3, to start off this analysis[3]. For every investigated loss function, the loss surface has a valley along the line of $-\theta_1 = \theta_2$, leading down towards increasing values of θ_2. This can be mathematically understood by considering that the network output from Eq. (5.2.1) is larger than 0.5 for $\theta_1 x + \theta_2 y < 0$ and smaller than 0.5 otherwise. Since we want the network outputs for inputs with $y > x$ to be 1, we want the parameters to be $-\theta_1 = \theta_2$ with large absolute values of θ_2. We can see the effect of applying the different loss functions in the actual shapes of the valleys.
The path of the parameters during training with Gradient Descent is depicted as a dotted line in the figure. To make the training look similar throughout different loss surfaces and different trials, we initialized the training at $\theta = \{0.5, 0.5\}$ instead of choosing random starting points. Since the actual values of the loss

[3]For some of the surfaces depicted here and in later figures, parts of the heatmap are missing. This is due to computational failure. We didn't investigate this further, since the missing parts seem to be predictable from the visible data in most cases.

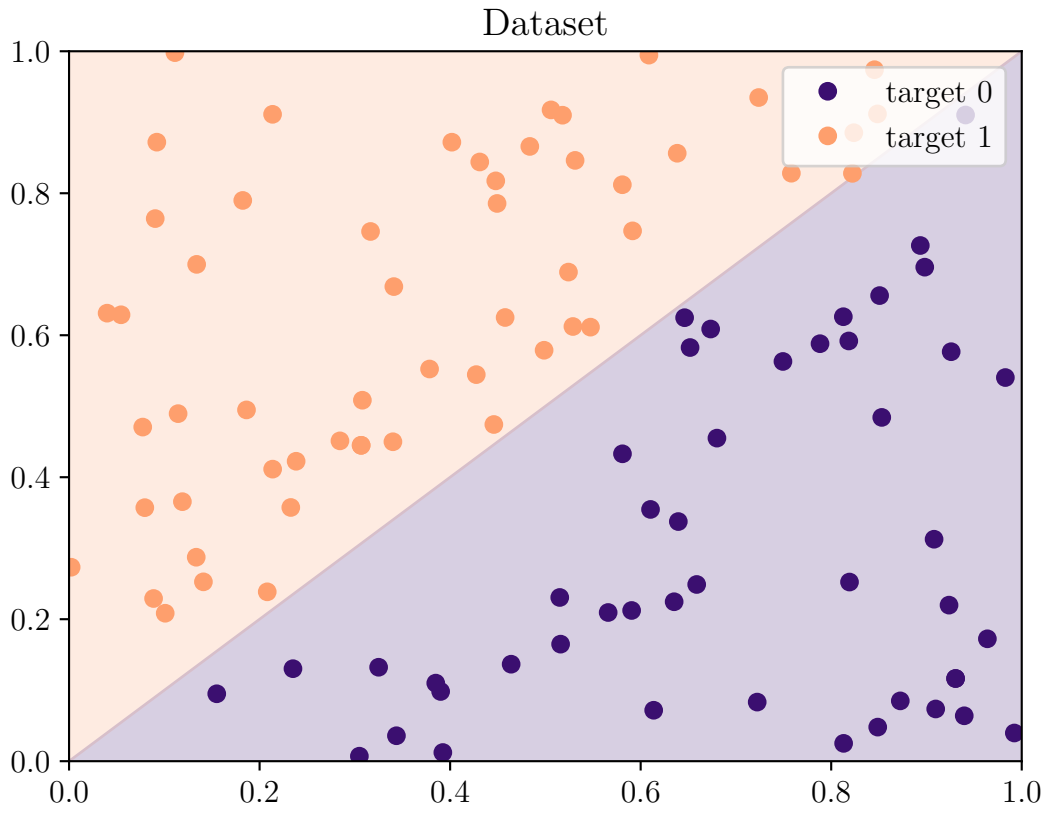


Figure 5.2: The decision boundary dataset used in the creation of the data from Section 5.2.3.

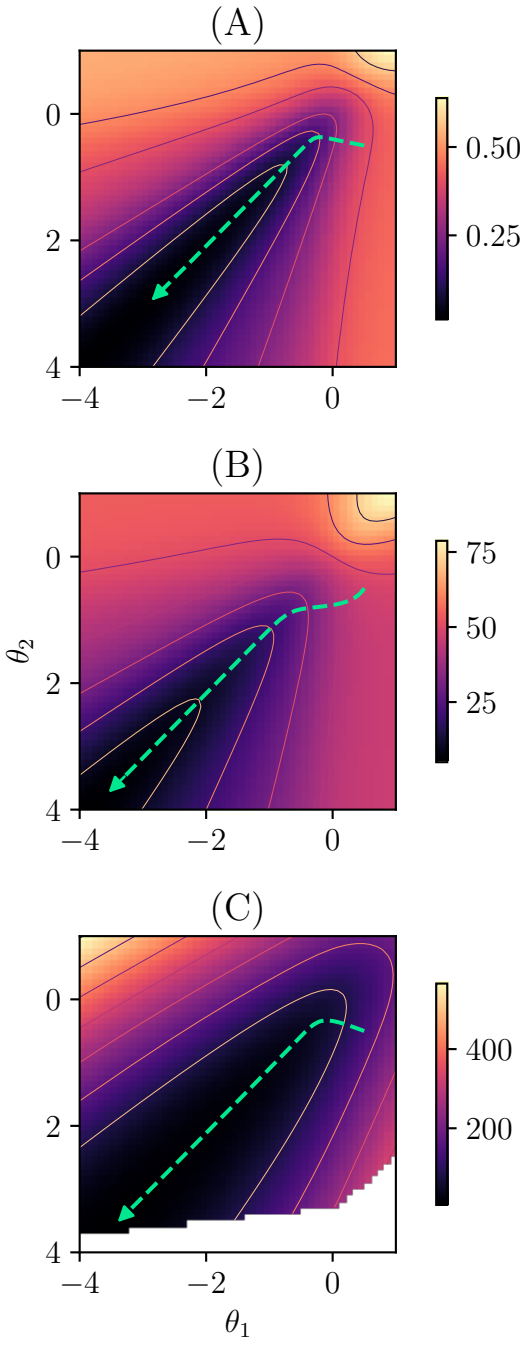


Figure 5.3: Loss surfaces for the Mean power loss of order $n = 2$ (A), the L_2-norm loss (B) and the Cross-entropy loss (C). The path of the parameters during Gradient Descent with initialization at $\theta = \{0.5, 0.5\}$ is depicted as a dotted line.

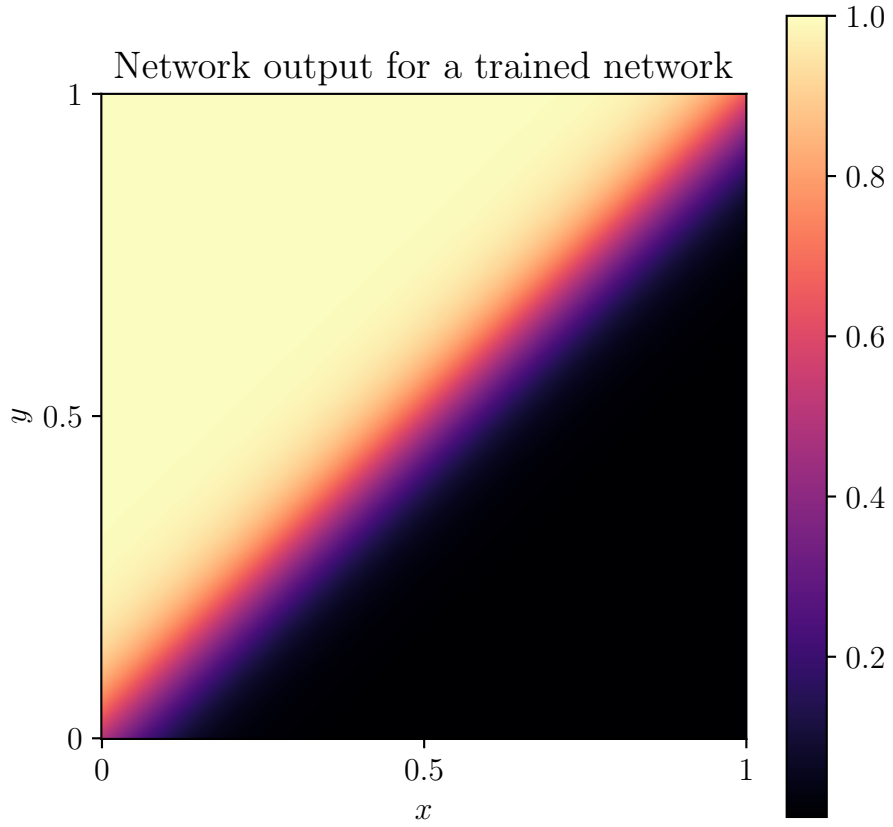


Figure 5.4: Network output at the final parameter configuration of training on the Cross-entropy loss.

also vary throughout the loss surfaces, we adjusted the learning rate and number of training epochs to account for the resulting variations in the size of gradients. In all cases, the network predicted all of the datapoints correctly at the end of training. To visualize the network behavior at the end of training, the output as a function of input values for the final parameters of the Cross-entropy training is depicted in Fig. 5.4.

Now let's look at the surfaces for the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R. The results for Mean power loss of order $n = 2$ are depicted in Fig. 5.5, the results for L_2-norm loss are depicted in Fig. 5.6 and the results for Cross-entropy loss are depicted in Fig. 5.7.

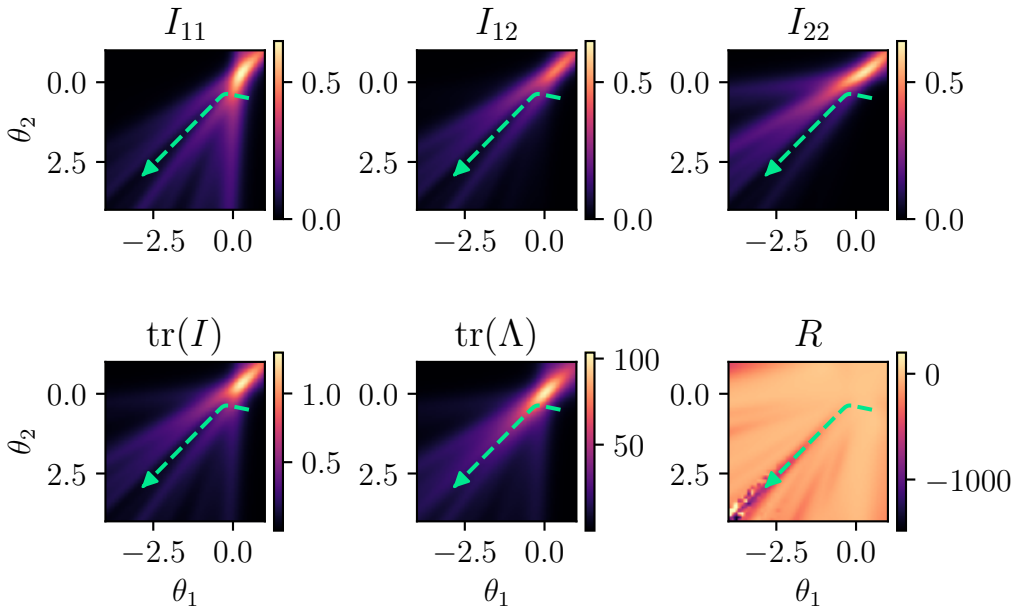


Figure 5.5: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *Mean power loss of order* $n = 2$.

The path of the parameters is depicted in every heatmap, similar to Fig. 5.3. Appendix C provides larger versions of these plots with added contours, to provide a more detailed view.

To add a more specific perspective on these surfaces, the evolution of the traces and the curvature along the path of parameters during training are depicted in Fig. 5.8 for the Mean power loss of order $n = 2$, Fig. 5.9 for L_2-norm loss and Fig. 5.10 for Cross-entropy loss.

Let's note the key details visible in these figures. First, similar to the results of Section 5.1, the Fisher Trace and the NTK trace behave similarly for the L_2-norm loss. In fact, for this experiment they are indistinguishable to each other apart from a constant factor. This means that the loss derivatives from Eq. (5.1.2) don't depend on the input index i. For the other loss functions, the traces also behave more similar than the traces in the MNIST experiment. We'd also expect

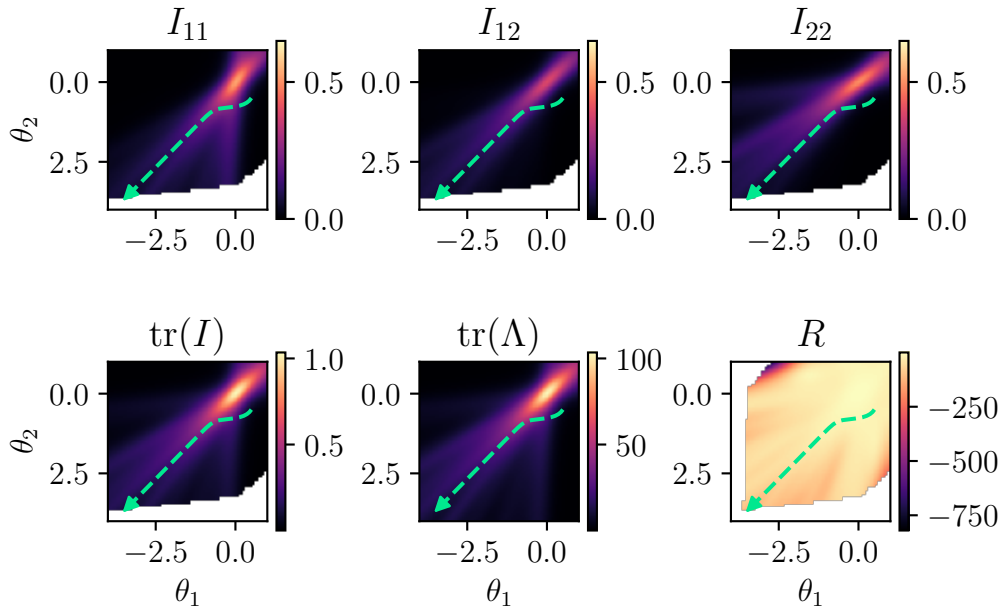


Figure 5.6: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *L_2-norm loss*.

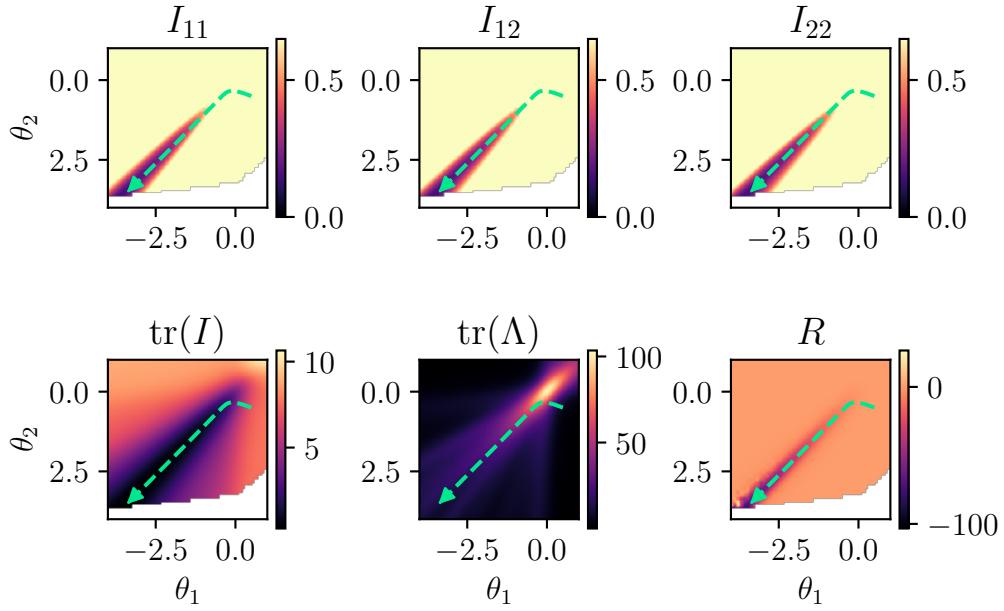


Figure 5.7: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *Cross-entropy loss*.

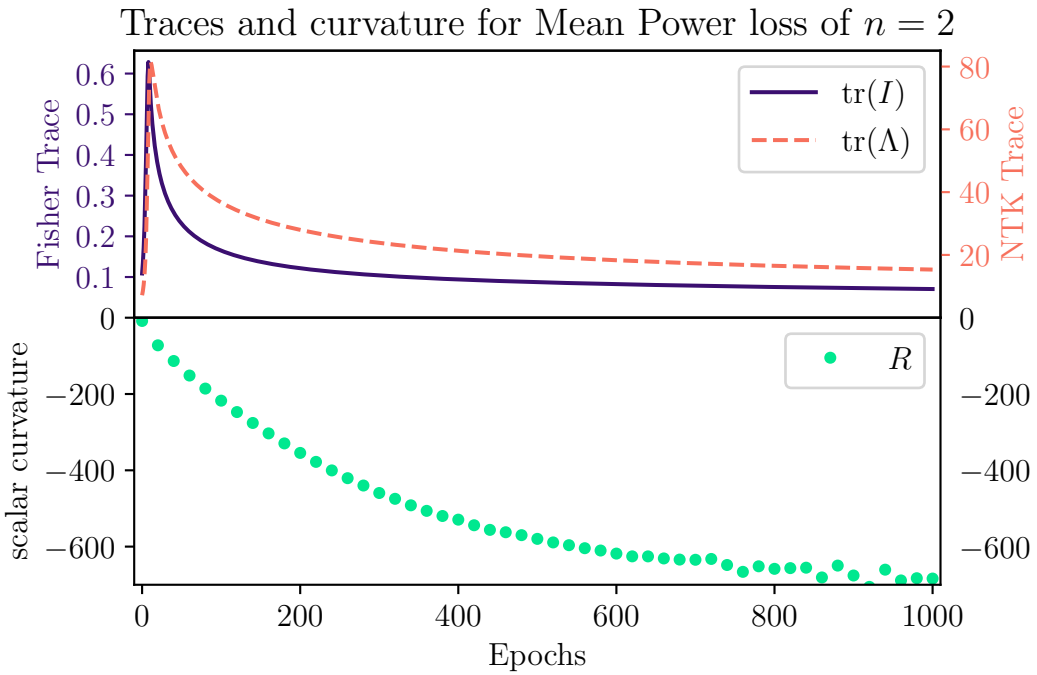


Figure 5.8: Evolution of the Fisher Trace, the NTK Trace and the scalar curvature during training with GD and *Mean power loss of order* $n = 2$.

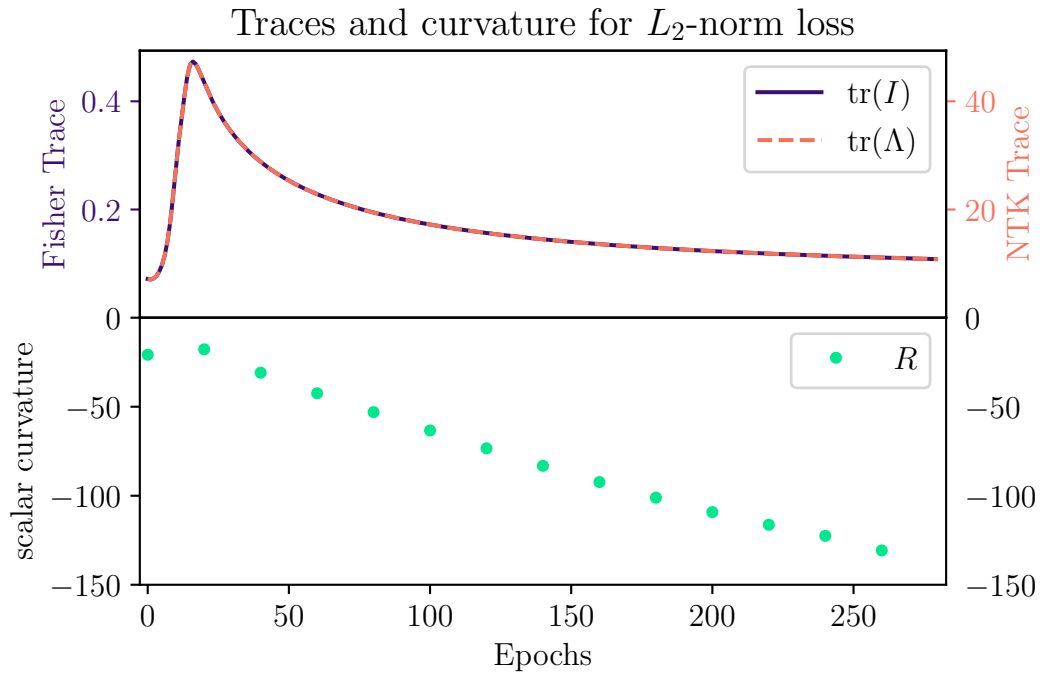


Figure 5.9: Evolution of the Fisher Trace, the NTK Trace and the scalar curvature during training with GD and L_2-*norm loss*.

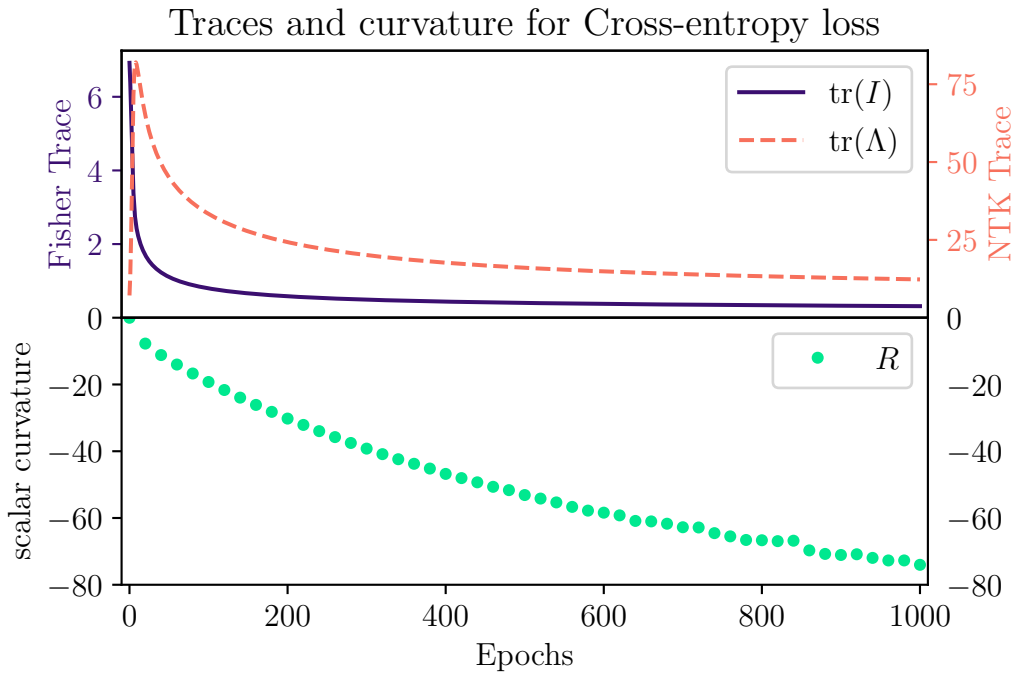


Figure 5.10: Evolution of the Fisher Trace, the NTK Trace and the scalar curvature during training with GD and *Cross-entropy loss*.

this behavior, since the input values representing points on a 1 by 1 square in $\mathbb{R}^2$ intuitively include less variety than the $\mathbb{R}^{784}$ pictures of handwritten digits. It's a reasonable assumption that this also makes the loss derivatives with respect to the output of those input points less diverse, which would make the proportionality to the NTK Trace a better approximation of Eq. (5.1.2). Since the network used here only outputs one value, we also lose an extra summation in Eq. (5.1.2). Those factors in combination lead to the Fisher Trace being more closely related to the NTK Trace.

For this experiment we also measured the scalar curvature R, depicted in Fig. 5.8, Fig. 5.9 and Fig. 5.10. We calculated it from the Fisher information using the equations presented in Section 3.2.4. The resulting algorithm was tested against multiple reference matrices with the help of analytical results from the EinsteinPy package [35]. Here, we can note that the curvature seems to descend roughly linearly in the first parts of training. The graph for the Mean power loss hints at it possibly converging for further training. It's also important to note that it's always negative and the path of the parameters descends into a valley of high negative curvature, that is more narrow than the valley of the loss function. For the L_2-norm loss the valley isn't visible in the surface heatmap, but it becomes clear that the curvature still descends in Fig. 5.9. What exactly large negative curvature means for the manifold of neural network training, and if the networks always tend to negative values has to be investigated in further research.

6 | Conclusion and Outlook

This work mostly explained the mathematical foundation of Fisher information, Neural Tangent Kernel and Ricci scalar curvature in their application for neural network training. We described the mathematical basics behind machine learning in Chapter 2. Chapter 3 dealt with the statistical background of the Fisher information, how it represents the Riemannian metric of statistical manifolds, its application for neural networks and how it can be utilized to find phase transitions in physical systems. Furthermore, we stated how the scalar curvature of manifolds is defined and why we can derive it from the Fisher information in the context of neural networks. In Chapter 4 we derived the NTK and noted which aspect of neural network training it describes.

Chapter 5 started with the derivation of a new relation between the trace of the Fisher information and the NTK in Eq. (5.1.2). This relation is of great interest, since we can obtain information about the Fisher Trace from the NTK, which is a smaller and also easier to compute observable of neural network training. It also factorizes the Fisher Trace, which is a description of the size of the parameter updates in GD, into a component that only depends on the loss function and a component that depends only on the network architecture. To investigate whether the NTK is sufficient to describe the evolution of the Fisher Trace through this equation, we compared the two traces for an example training on the MNIST dataset in Section 5.1.1. The result indicated that for some combinations of loss function and network, the behavior of the Fisher Trace can be approximated by the NTK Trace, but not in general. Further investigations are needed to determine the characteristics of network and loss function, for which the traces behave similarly.

In Section 5.2.3 we calculated the full Fisher information, the NTK Trace and the

scalar curvature for a simple two-input neural network. The results obtained are a first step towards understanding how these observables describe neural network training.

Overall, this work investigated the Fisher information, the NTK Trace and the scalar curvature of neural networks. It provided mathematical backgrounds and first calculations of these observables for simple problems. Further research possibilities are vast. For example, there is already ongoing research attempting to make its computation feasible [36]. The Fisher information could be used for training optimization, as in the Natural Gradient Descent algorithm proposed by Amari in [37], but one could also investigate its use for finding efficient initial states or even look for analogies of physical phase transitions in neural networks as described in Section 3.4. Regarding the NTK, one could also investigate the possibility of using it for training optimization in similar ways. For the scalar curvature, further research should delve deeper into its theoretical implications, to reveal insight into the specific information it contains.

Literature

[1] Michael KK Leung et al. “Deep learning of the tissue-regulated splicing code”. In: *Bioinformatics* 30.12 (2014), pp. i121–i129.

[2] Junshui Ma et al. “Deep neural nets as a method for quantitative structure–activity relationships”. In: *Journal of chemical information and modeling* 55.2 (2015), pp. 263–274.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[4] Sébastien Jean et al. “On using very large target vocabulary for neural machine translation”. In: *arXiv preprint arXiv:1412.2007* (2014).

[5] Tilman Räuker et al. *Toward Transparent AI: A Survey on Interpreting the Inner Structures of Deep Neural Networks*. 2023. arXiv: 2207.13243 [cs.LG].

[6] Esther Inglis-Arkell. Mar. 7, 2012. URL: https://gizmodo.com/the-very-first-robot-brains-were-made-of-old-alarm-cl-5890771 (visited on 08/23/2023).

[7] A. M. TURING. “I.COMPUTING MACHINERY AND INTELLIGENCE”. In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[8] Daniel Jannai et al. “Human or Not? A Gamified Approach to the Turing Test”. In: (2023). arXiv: 2305.20010 [cs.AI].

[9] URL: https://www.oxfordlearnersdictionaries.com/us/definition/english/machine-learning (visited on 08/23/2023).

[10] SS Khare and AR Gajbhiye. "Literature Review on Application of Artificial Neural Network (Ann) In Operation of Reservoirs". In: *International Journal of computational Engineering research (IJCER) IJCER| June 2013| VOL 3 ISSUE 6* (1943), p. 63.

[11] Hermann Haken. *Principles of brain functioning: a synergetic approach to brain activity, behavior and cognition*. Vol. 67. Springer Science & Business Media, 2013.

[12] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133. DOI: https://doi.org/10.1007/BF02478259.

[13] Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sinák. "A review of activation function for artificial neural network". In: *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE. 2020, pp. 281–286.

[14] Iqbal H Sarker. "Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions". In: *SN Computer Science* 2.6 (2021), p. 420.

[15] Bolin Gao and Lacra Pavel. "On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning". In: (2018). arXiv: 1704.00805 [math.OC].

[16] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[17] Vladimir Nasteski. "An overview of the supervised machine learning methods". In: *Horizons. b* 4 (2017), pp. 51–62.

[18] L. Bottou et al. “Comparison of classifier methods: a case study in handwritten digit recognition”. In: *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5)*. Vol. 2. 1994, 77–82 vol.2. DOI: 10.1109/ICPR.1994.576879.

[19] S. Amari and S.C. Douglas. “Why natural gradient?” In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP ’98 (Cat. No.98CH36181)*. Vol. 2. 1998, 1213–1216 vol.2. DOI: 10.1109/ICASSP.1998.675489.

[20] Qi Wang et al. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Annals of Data Science* 9 (2 2022), pp. 2198–5812. DOI: 10.1007/s40745-020-00253-5.

[21] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[22] Daniel A. Roberts, Sho Yaida, and Boris Hanin. *The Principles of Deep Learning Theory*. Cambridge University Press, May 2022. DOI: 10.1017/9781009023405. URL: https://doi.org/10.1017%2F9781009023405.

[23] Bobby Kleinberg, Yuanzhi Li, and Yang Yuan. “An alternative view: When does SGD escape local minima?” In: *International conference on machine learning*. PMLR. 2018, pp. 2698–2707.

[24] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: (2017). arXiv: 1412.6980 [cs.LG].

[25] Alexander Ly et al. “A Tutorial on Fisher information”. In: *Journal of Mathematical Psychology* 80 (2017), pp. 40–55. ISSN: 0022-2496. DOI: https://doi.org/10.1016/j.jmp.2017.05.006. URL: https://www.sciencedirect.com/science/article/pii/S0022249617301396.

[26] *atemateca (IME/USP)/Rodrigo Tetsuo Argenton*. URL: https://commons.wikimedia.org/wiki/File:Galton_box.jpg#/media/Datei:Galton_box.jpg (visited on 08/24/2023).

[27] MathWorld. *Galton Board – from Wolfram MathWorld*. [Online; accessed 17-September-2023]. 2023. URL: https://mathworld.wolfram.com/GaltonBoard.html.

[28] James Sneyd, Rachel M Fewster, and Duncan McGillivray. "Binomial distribution". In: *Mathematics and Statistics for Science*. Springer, 2022, pp. 561–582.

[29] Shun-ichi Amari. *Differential-Geometrical Methods in Statistics*. 1st ed. ISBN: 978-0-387-96056-2. DOI: https://doi.org/10.1007/978-1-4612-5056-2.

[30] Bernard Schutz. *A First Course in General Relativity*. 2nd ed. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511984181.

[31] Mikhail Prokopenko et al. "Relating Fisher information to order parameters". In: *Phys. Rev. E* 84 (4 Oct. 2011), p. 041116. DOI: 10.1103/PhysRevE.84.041116. URL: https://link.aps.org/doi/10.1103/PhysRevE.84.041116.

[32] W Janke, DA Johnston, and Ralph Kenna. "Information geometry and phase transitions". In: *Physica A: Statistical Mechanics and its Applications* 336.1-2 (2004), pp. 181–186.

[33] Colin Campbell. "An introduction to kernel methods". In: *Studies in Fuzziness and Soft Computing* 66 (2001), pp. 155–192.

[34] Tien D Bui et al. "LP norm relaxation approach for large scale data analysis: a review". In: *Image Analysis and Recognition: 15th International Conference, ICIAR 2018, Póvoa de Varzim, Portugal, June 27–29, 2018, Proceedings 15*. Springer. 2018, pp. 285–292.

[35] EinsteinPy Development Team. *EinsteinPy: Python library for General Relativity*. 2021. URL: https:/einsteinpy.org/.

[36] Abdoulaye Koroko et al. "Efficient approximations of the fisher matrix in neural networks using kronecker product singular value decomposition". In: *arXiv preprint arXiv:2201.10285* (2022).

[37] Shun-Ichi Amari. "Natural gradient works efficiently in learning". In: *Neural computation* 10.2 (1998), pp. 251–276.

[38] Anders Krogh. "What are artificial neural networks?" In: *Nature biotechnology* 26.2 (2008), pp. 195–197.

[39] Nan Du et al. "Glam: Efficient scaling of language models with mixture-of-experts". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5547–5569.

[40] Piyush Kaul and Brejesh Lall. "Riemannian Curvature of Deep Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 31.4 (2020), pp. 1410–1416. DOI: 10.1109/TNNLS.2019.2919705.

[41] S. Amari and S.C. Douglas. "Why natural gradient?" In: 2 (1998), 1213–1216 vol.2. DOI: 10.1109/ICASSP.1998.675489.

[42] Enzo Grossi and Massimo Buscema. "Introduction to artificial neural networks". In: *European Journal of Gastroenterology and Hepatology* 19.12 (Dec. 2007), pp. 1046–1054. DOI: 10.1097/MEG.0b013e3282f198a0.

A | Auxiliary mathematical proofs

A.1 Proof of Eq. (3.1.3)

For all variables X^n that satisfy

$$f(x^n|\theta) = \prod_{\alpha=1}^{n} f(x_\alpha|\theta) \tag{A.1.1}$$

we can prove

$$I_{X^n,ij} = \sum_{\alpha} I_{X_\alpha,ij} \tag{A.1.2}$$

by

$$
\begin{aligned}
I_{X^n,ij} &= \underset{x^n \in X^n}{E} \left\{ \frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x^n|\theta) \frac{\mathrm{d}}{\mathrm{d}\theta_j} \log f(x^n|\theta) \right\} \\
&= \sum_{x^n \in X^n} \left\{ \left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x^n|\theta) \right] \left[\frac{\mathrm{d}}{\mathrm{d}\theta_j} f(x^n|\theta) \right] [f(x^n|\theta)]^{-1} \right\} \\
&\overset{A.1.1}{=} \sum_{x^n \in X^n} \left\{ \frac{\mathrm{d}}{\mathrm{d}\theta_i} \left[\prod_\alpha f(x_\alpha|\theta) \right] \frac{\mathrm{d}}{\mathrm{d}\theta_j} \left[\prod_\beta f(x_\beta|\theta) \right] \left[\prod_\gamma f(x_\gamma|\theta) \right]^{-1} \right\} \\
&= \sum_{x^n \in X^n} \left\{ \left(\prod_\alpha f(x_\alpha|\theta) \right) \left(\sum_\alpha \frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right) \right. \\
&\qquad \left. \left(\prod_\beta f(x_\beta|\theta) \right) \left(\sum_\beta \frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right) \left[\prod_\gamma f(x_\gamma|\theta) \right]^{-1} \right\} \\
&= \underset{x^n \in X^n}{E} \left\{ \left[\sum_\alpha \frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right] \left[\sum_\beta \frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\beta|\theta) \right] \right\} \\
&\overset{(*)}{=} \underset{x^n \in X^n}{E} \left\{ \sum_\alpha \left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right] \left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right] \right\} \\
&= \sum_\alpha \underset{x^n \in X^n}{E} \left\{ \left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right] \left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} \log f(x_\alpha|\theta) \right] \right\} \\
&= \sum_\alpha I_{X_\alpha,ij}.
\end{aligned}
\tag{A.1.3}
$$

The equality at $(*)$ holds because for all $\alpha \neq \beta$

$$\begin{aligned}
&\underset{x^n \in X^n}{E}\left\{\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i}\log f(x_\alpha|\theta)\right]\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i}\log f(x_\beta|\theta)\right]\right\} \\
&\propto \sum_{x_\alpha}\sum_{x_\beta}\left\{\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\alpha|\theta)\right]\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\beta|\theta)\right]\right\} \\
&= \sum_{x_\alpha}\left\{\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\alpha|\theta)\right]\sum_{x_\beta}\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\beta|\theta)\right]\right\} \\
&= \sum_{x_\alpha}\left\{\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\alpha|\theta)\right]\sum_{x_\beta}\frac{\mathrm{d}}{\mathrm{d}\theta_i}\left[f(x_\beta|\theta)\right]\right\} \\
&= \sum_{x_\alpha}\left\{\left[\frac{\mathrm{d}}{\mathrm{d}\theta_i} f(x_\alpha|\theta)\right]\sum_{x_\beta}\frac{\mathrm{d}}{\mathrm{d}\theta_i}\left[1\right]\right\} \\
&= 0.
\end{aligned} \tag{A.1.4}$$

This prove still holds when using continuous variables instead of discrete ones.

A.2 Proof of Eq. (3.2.14)

Here we will prove Eq. (3.2.14) which states

$$\underset{x\in X}{E}\Big[\partial_i \log p(x|\theta)\cdot\partial_j \log p(x|\theta)\Big] = -\underset{x\in X}{E}\Big[\partial_i\partial_j \log p(x|\theta)\Big]. \tag{A.2.1}$$

To prove this, we will evaluate the right side by

$$\begin{aligned}
-\underset{x\in X}{E}\Big[\partial_i\partial_j \log p(x|\theta)\Big] &= -\underset{x\in X}{E}\left\{\partial_i\left[\partial_j \log p(x|\theta)\right]\right\} \\
&= -\underset{x\in X}{E}\left\{\partial_i\left[\frac{\partial_j p(x|\theta)}{p(x|\theta)}\right]\right\} \\
&= -\underset{x\in X}{E}\left[\frac{\partial_i\partial_j p(x|\theta)}{p(x|\theta)} - \frac{\partial_i p(x|\theta)\partial_j p(x|\theta)}{p(x|\theta)^2}\right] \\
&= -\underset{x\in X}{E}\left[\frac{\partial_i\partial_j p(x|\theta)}{p(x|\theta)} - \partial_i \log p(x|\theta)\partial_j \log p(x|\theta)\right] \\
&= \underset{x\in X}{E}\Big[\partial_i \log p(x|\theta)\cdot\partial_j \log p(x|\theta)\Big],
\end{aligned} \tag{A.2.2}$$

Where the last equation holds because

$$\begin{aligned}
-\underset{x\in X}{E}\left[\frac{\partial_i\partial_j p(x|\theta)}{p(x|\theta)}\right] &= \sum_{x\in X}\left[\partial_i\partial_j p(x|\theta)\right] \\
&= \partial_i\partial_j \sum_{x\in X}\left[p(x|\theta)\right] \\
&= -\partial_i\partial_j \underset{x\in X}{E}\left[1\right] \\
&= 0.
\end{aligned} \tag{A.2.3}$$

Here, we assumed that we can swap the sum and the derivatives. For continuous variables, the swapping of the integral and the derivatives has to be assumed for the equation to hold.

B | MNIST experiment for trace comparison

This section contains more results of the experiment from Section 5.1.1. The results for loss functions of order 4 and 6 for the network of width 128 are depicted in Fig. B.1 and Fig. B.2. The results for loss functions of order 2, 4 and 6 for the network of width 784 are depicted in Fig. B.3, Fig. B.4 and Fig. B.5. The Cross-entropy loss doesn't vary with order but is still depicted for reference in every plot.

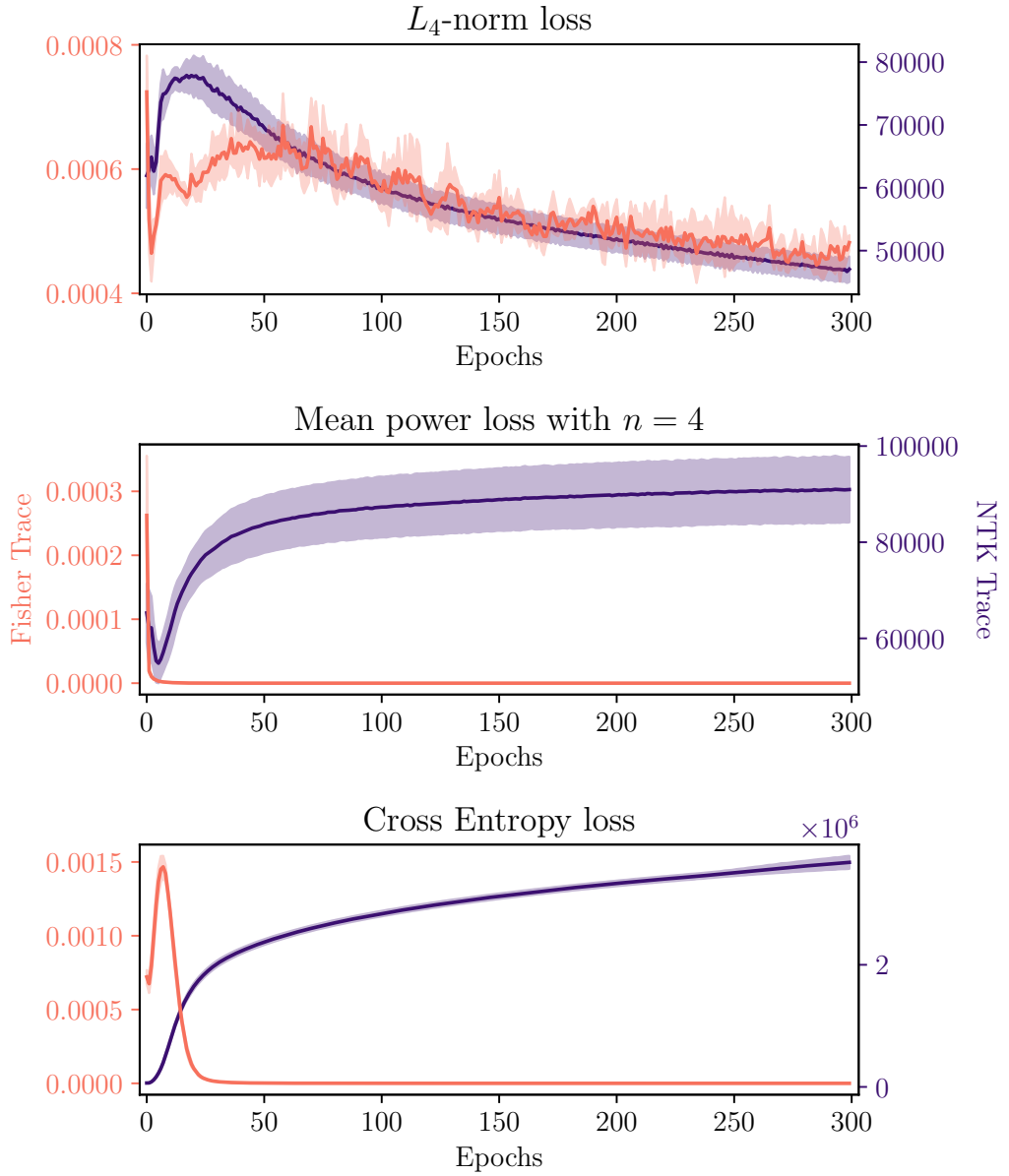


Figure B.1: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 128 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

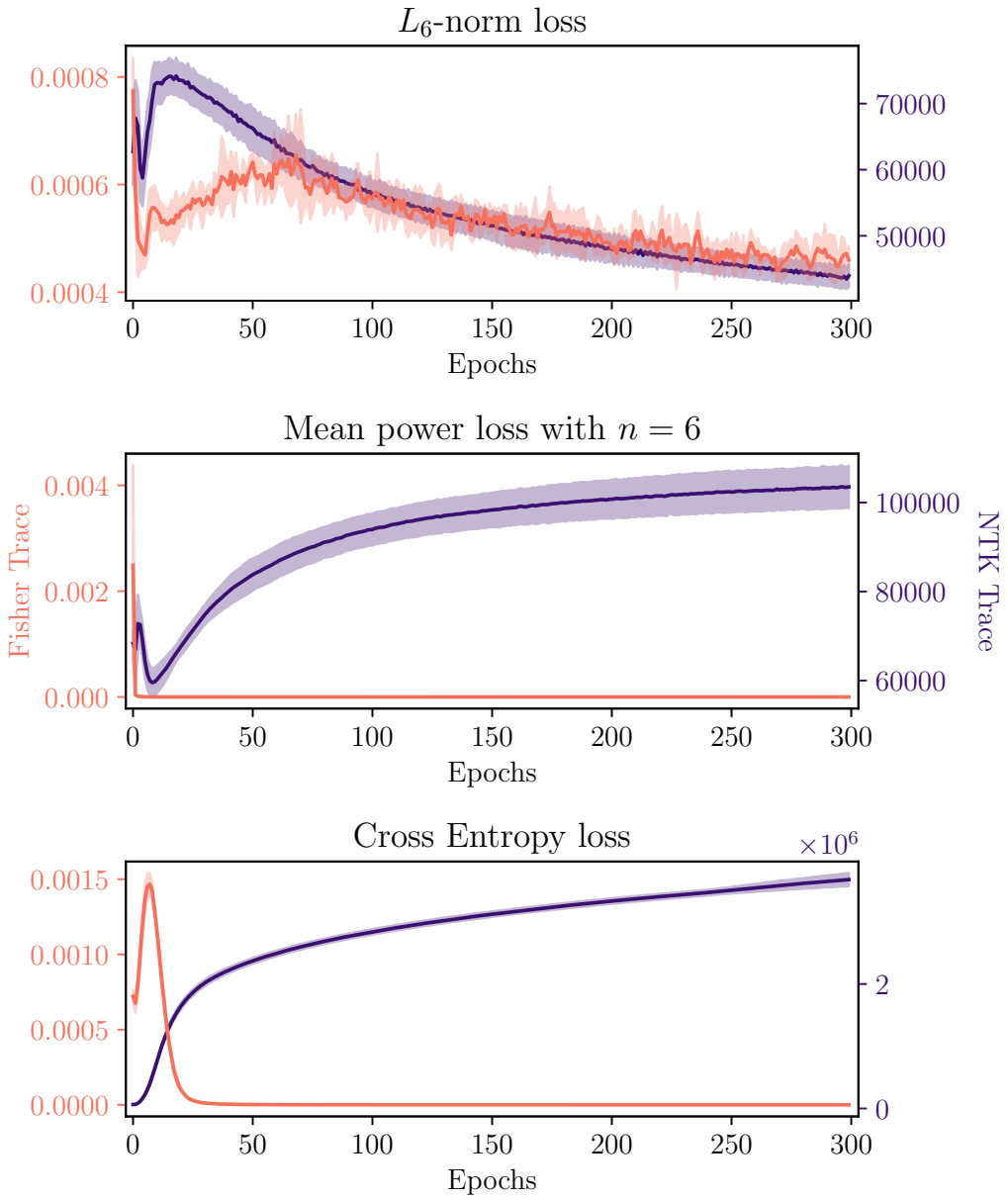


Figure B.2: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 128 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

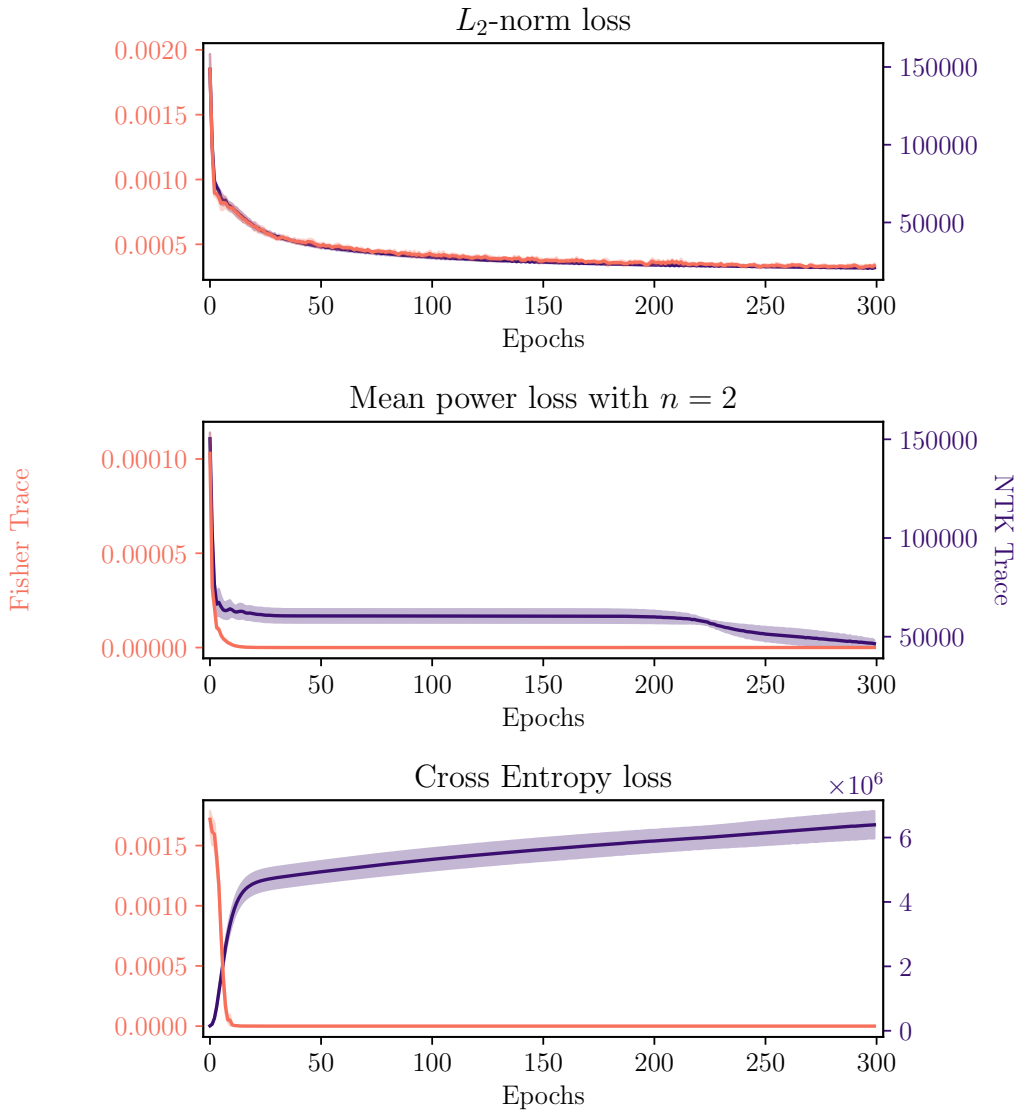


Figure B.3: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 784 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

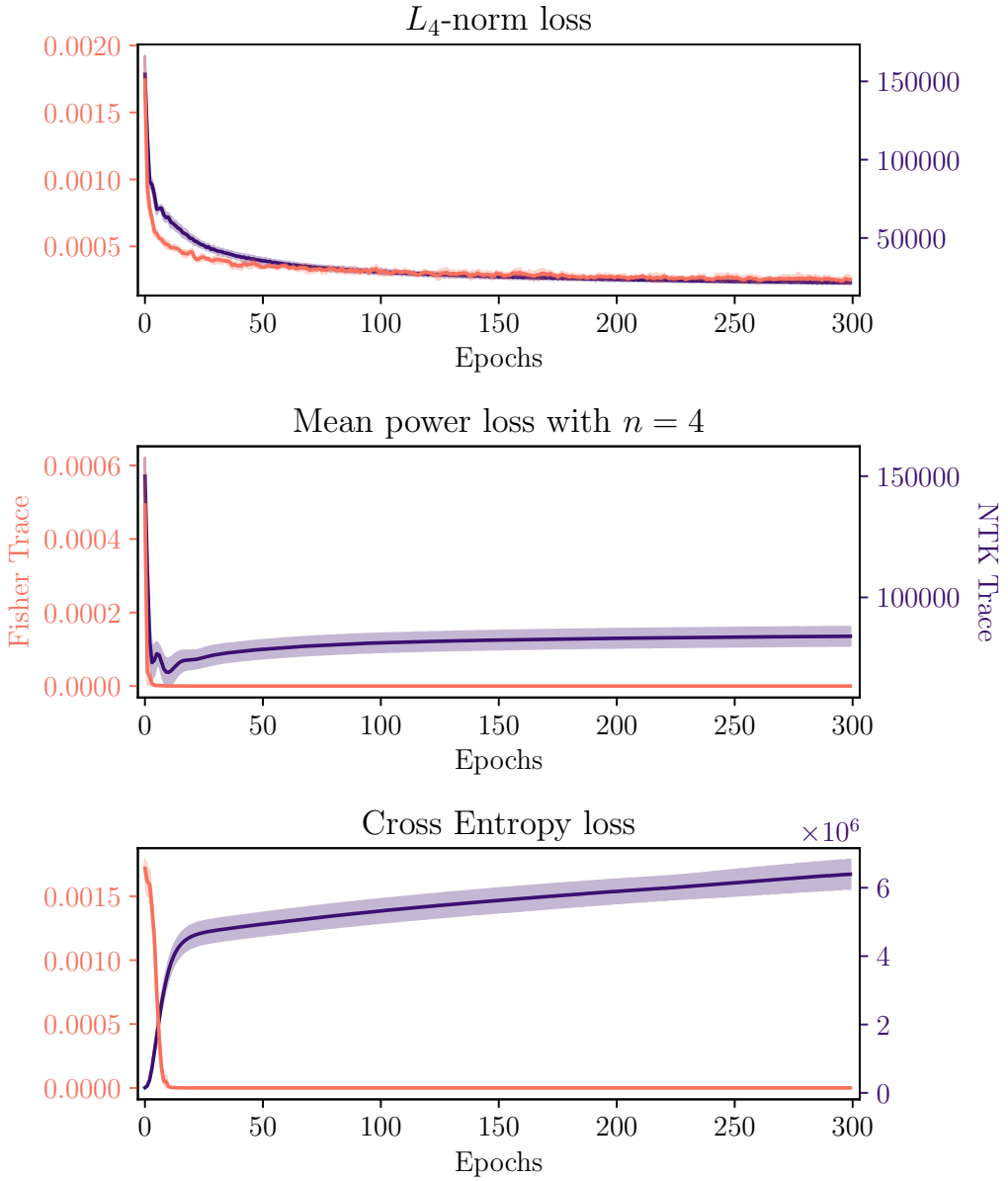


Figure B.4: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 784 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

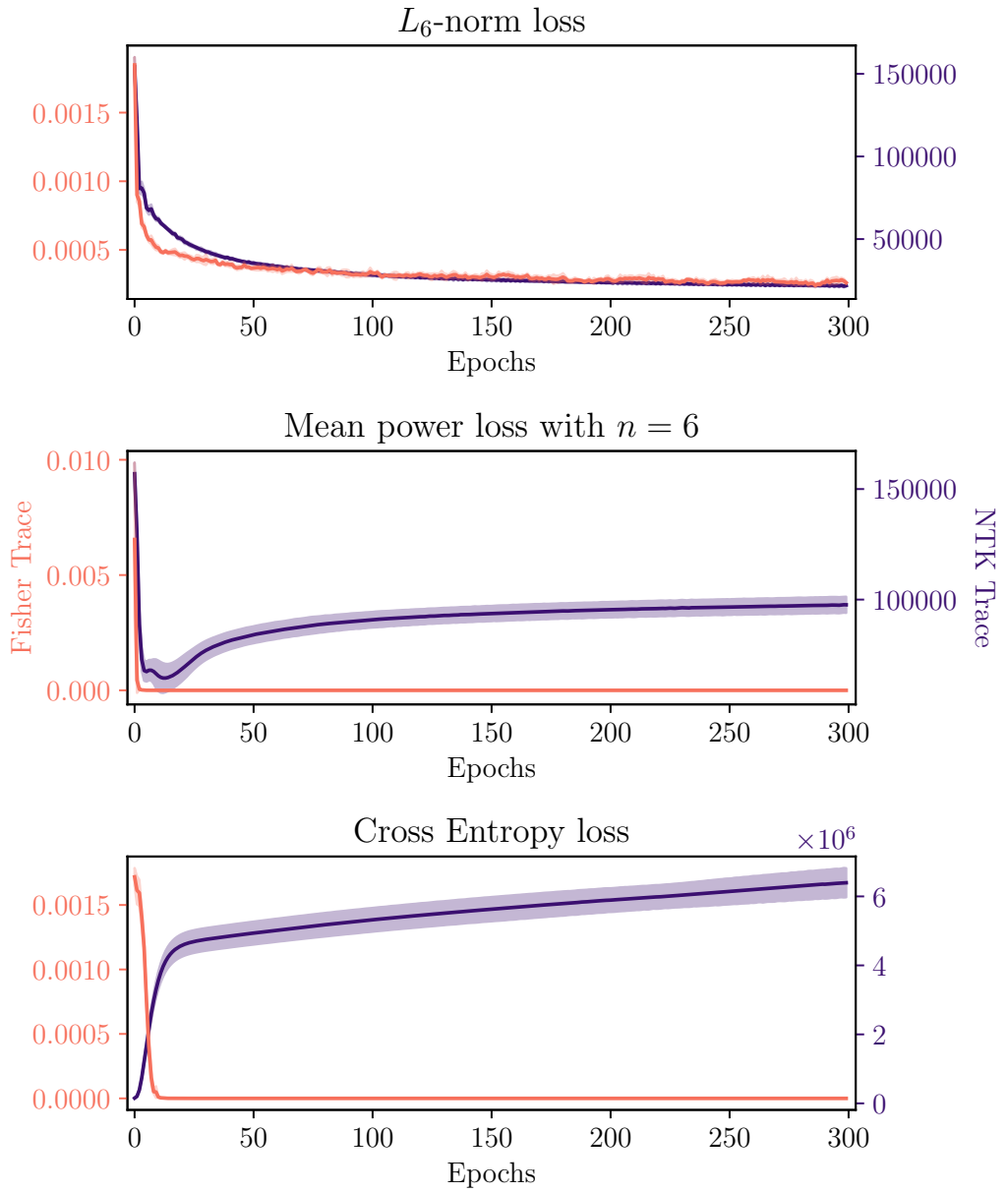


Figure B.5: Trace of NTK and Fisher information during 300 epochs of training on the MNIST dataset using the Adam optimizer. The network consisted of 2 hidden layers with a *width of 784 neurons* that were equipped with the ReLU activation function. The loss functions used for optimization are denoted in each subplot. The solid line represents the mean value of 5 experiments, the translucent area represents one standard deviation from the mean value.

C | Additional plots for the 2-parameter network experiment

This section contains larger versions of Fig. 5.5, Fig. 5.6 and Fig. 5.7. These are depicted in Fig. C.1, Fig. C.2 and Fig. C.3.

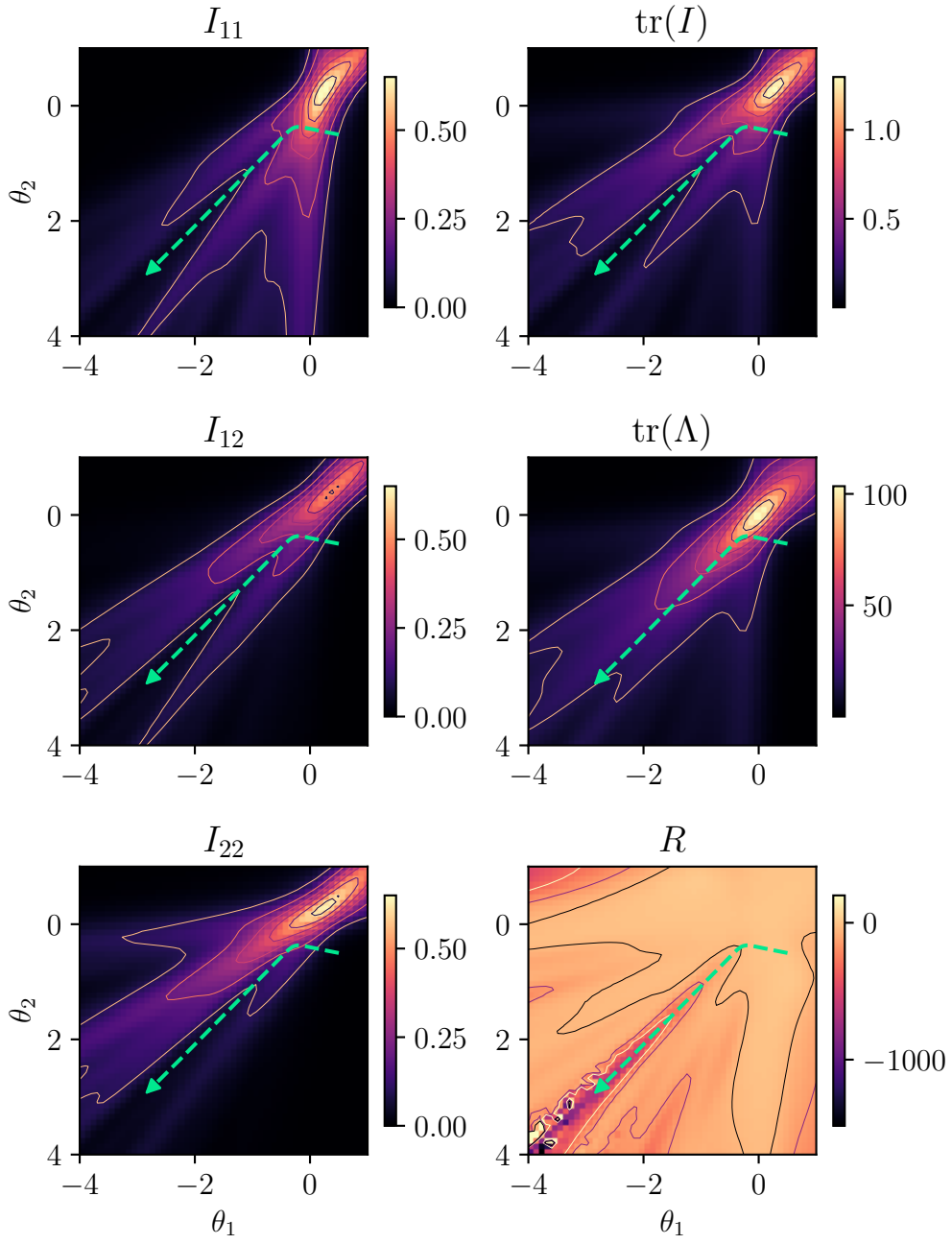


Figure C.1: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *Mean power loss of order* $n = 2$. This is a larger version of Fig. 5.5.

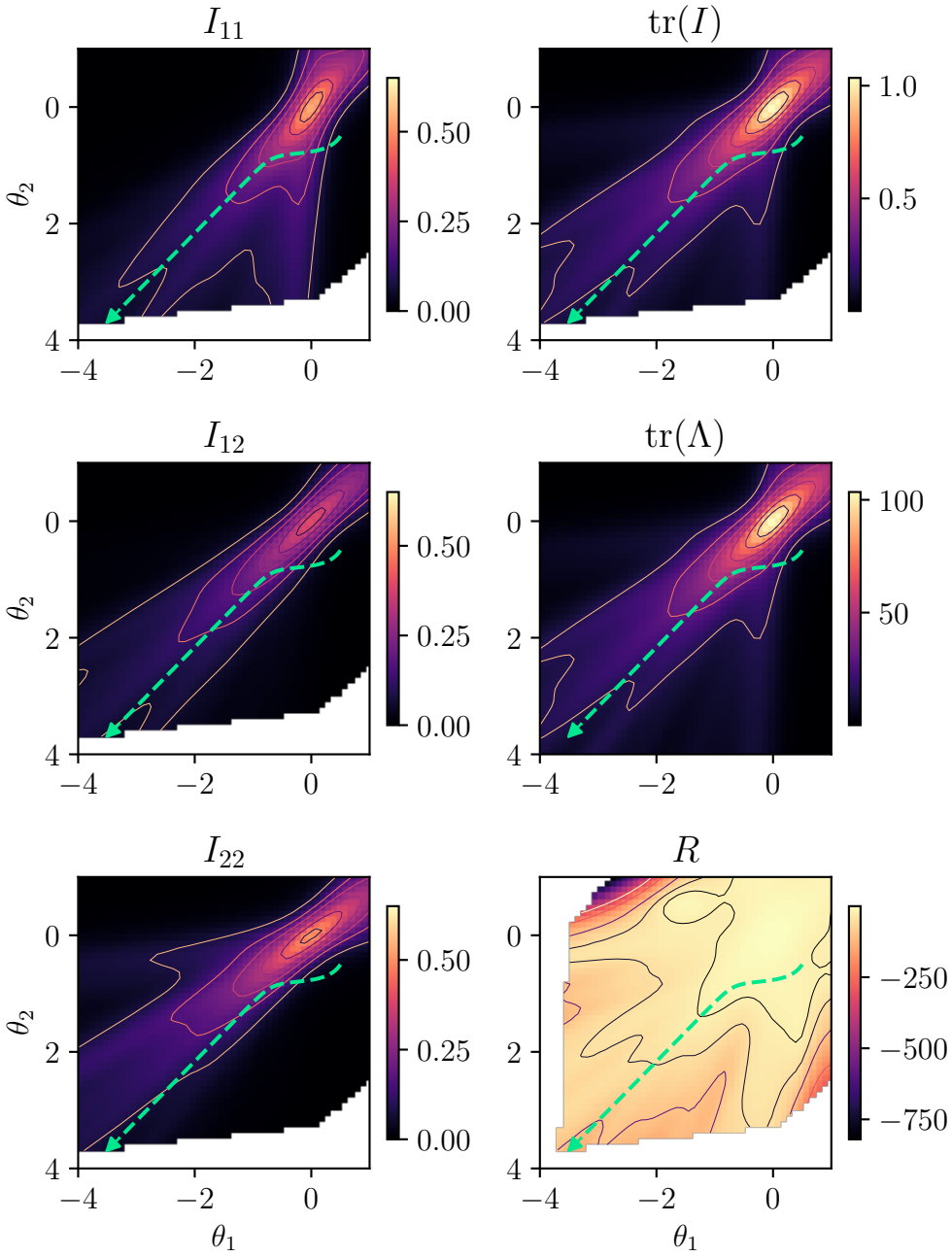


Figure C.2: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *L_2-norm loss*. This is a larger version of Fig. 5.6.

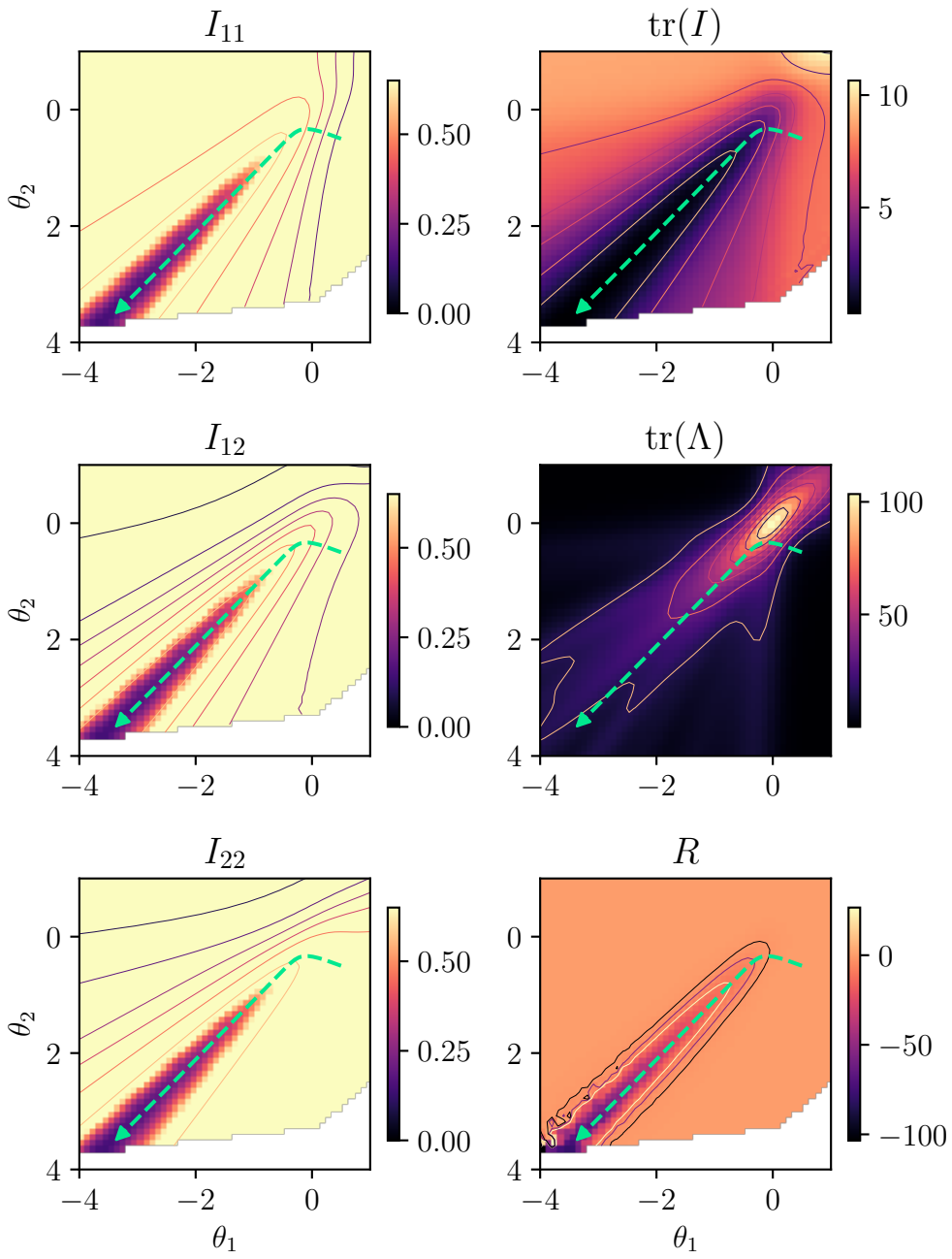


Figure C.3: Surfaces of all components of the Fisher information I, its trace $\mathrm{tr}(I)$, the trace of the NTK $\mathrm{tr}(\Lambda)$ and the scalar curvature R for *Cross-entropy loss*. This is a larger version of Fig. 5.7.