Julius–Maximilians–University of Würzburg
Chair of Theoretical Physics III
Prof. Dr. Johanna Erdmenger

Quantifying the propagation of information in deep artificial neural networks through the relative entropy

by

Marwin Schmidt

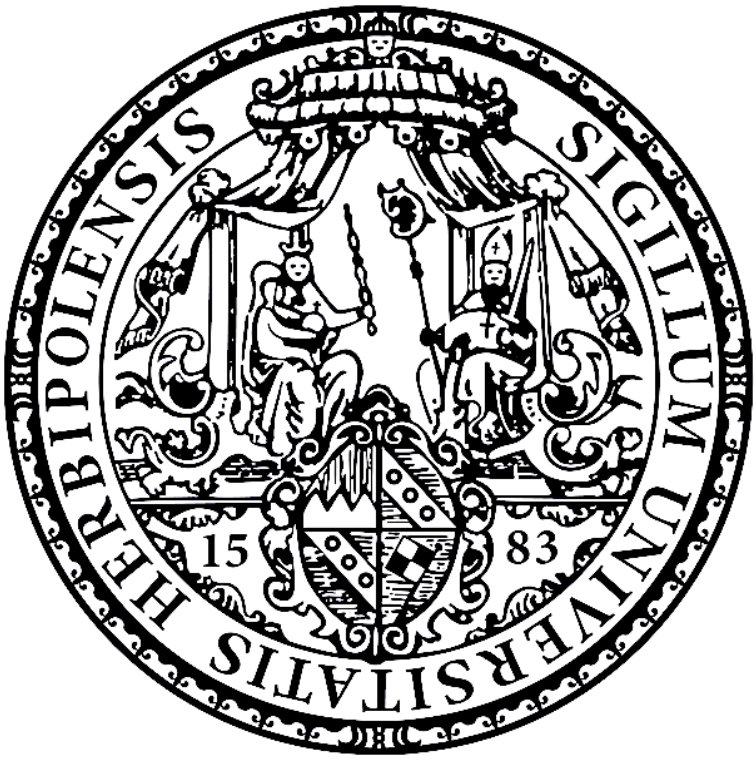

Table of Contents

List of Figures

1 Introduction

The advent of Deep Learning — the use of deep neural networks trained on large data sets has revolutionized and connected industries and scientific disciplines. From quantitative finance, where Deep Learning models are used to make investment decisions to the life sciences, where these models have been able to predict the structure of proteins, i.e. the fundamental building blocks of life. What allowed these models to achieve this is their ability to uncover complex nonlinear relationships between high-dimensional data. This ability also gives Deep Learning models the potential to discover new vaccines, nanomaterials, and disease treatments, and to open a path to a deeper understanding of nature and human behavior. However, these models have shortcomings too – mainly that we are not able to fundamentally and comprehensively understand their behavior. As they are used for increasingly important tasks, this is becoming a big problem. This is because not understanding their behavior also means that we are unable to prevent them from making mistakes with potentially severe consequences. To address the issue, we take a step towards fundamentally understanding Deep Learning models by looking at them from the perspective of theoretical physics, where the pursuit has always been to find out why things are the way they are.
First, we want to fundamentally understand the architecture of two deep neural networks, the multilayer perceptron, and the sparse autoencoder. For this, we use a biological neural network as inspiration. Next, we explain how the multilayer perceptron is able to recognize images and how the autoencoder is able to reconstruct images. To ensure that the networks are able to do this for images they have not already seen, we address the issue of generalization. After this intuitive and fundamental view of Deep Learning, we next attempt to understand how information from an input image is processed by a multilayer perceptron for image recognition and by an autoencoder for image reconstruction. To do this, we shed light on what happens inside these networks by measuring how information propagates through them. To do this, we take inspiration from [6] and use the relative entropy, also known as Kullback-Leibler divergence. Results indicate that networks must be well-trained before we can use the relative entropy. For a multilayer perceptron, the measurements show that the relative entropy as a function depth increases up to an asymptote. This behavior is independent of the network's number of layers (depth). For the autoencoder, the relative entropy as a function depth increases to a peak and then decreases again. This behavior is independent of whether we use MNIST images or spin configurations of a 2d Ising model as the input data set. For both networks, we interpret the behavior of the relative entropy with the help of the renormalization group and look at Deep Learning as being a coarse-graining procedure that removes irrelevant information with respect to image recognition and reconstruction. However, we observe that in both cases the relative entropy pre-training can be above or below the relative entropy post-training. This raises questions about the validity of the interpretation using the renormalization group and whether the relative entropy is able to accurately measure how information propagates through deep neural networks.

2 Deep neural networks

Neural networks are information-processing systems that can learn to recognize objects. From an intuitive point of view, they do this by extracting high-level features such as patterns and shapes from input data [6]. In reality, the process is not that clear. Thus, we want to understand how these networks can learn to recognize objects on a fundamental level.
For inspiration, we first consider how the human brain recognizes digital grayscale images of numbers from the MNIST database[1]. Every digital image inside this data set is composed of a large ensemble of pixels with different grayscale values. When recognizing these numbers, our brain does not seem to consider the grayscale values of each pixel, which is why we can say that the recognition process qualitatively resembles a coarse-graining procedure (extraction of coarse-grained features from fine-grained input information) from the perspective of the renormalization group [6].

2.1 Multilayer perceptron

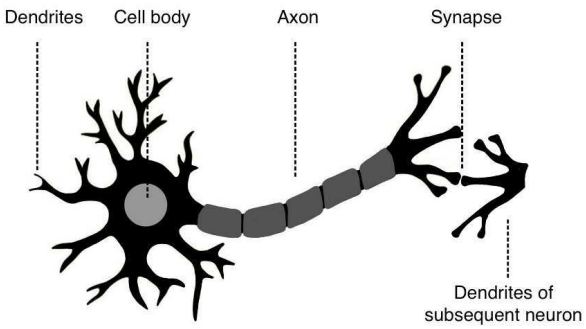

FIGURE 1: Simplified representation of the biological neuron. It receives information via its dendrites, processes it in the cell body, and then passes it on to the following neuron via axons. Axons and dendrites are connected by synapses, which control the strength of the transmitted signals. This figure is inspired by figure 2 in [1].

Inspired by the human brain, we want to create a mathematical model that is also capable of recognizing images of handwritten numbers. Therefore, we resort to an artificial neural network; the multilayer perceptron (Figure 2). To develop its architecture, we first consider the human brain's foundational computational units, the neurons. These are interconnected in such a way that they can receive and transmit signals (Figure 1). The neuron receives information signals from other neurons via its dendrites. These signals are summed up inside the cell body and when reaching a certain threshold a signal is fired throughout the axons in the direction of the subsequent neuron [9]. The axon of the neuron is connected to

[1]The MNIST database is a database of many handwritten digits which are commonly used to train and test neural networks.

the dendrites of another neuron via synapses. Since these determine the connection strength between the neurons, synapses act like a weight for the information signal [4].
We translate this biological architecture into a mathematical model by interpreting a neuron as a computational unit that receives input data, processes it, and then produces an output. We call this an artificial neuron or perceptron. Similar to a biological neuron, we connect these artificial neurons to each other and control the connection strength between them through weights (Figure 2) [1]. When the perceptron receives an input, this input is multiplied by the associated weights. In our case, a perceptron's input is a pixel's grayscale value. All these grayscale values are multiplied by weights and then summed. The weighted sum is then passed through an activation function that represents a threshold that it must overcome in order to be transmitted to subsequent neurons [1]. The output of the activation function is referred to as activation.

2.1.1 Architecture

This chapter uses ideas from [6], [13], [3],[14], [2], and [17].

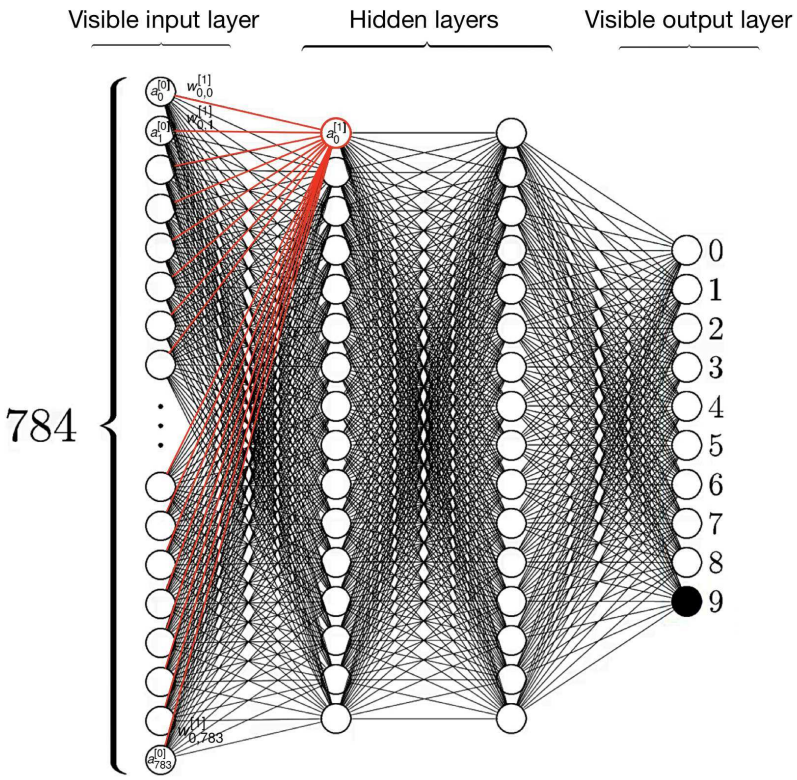

FIGURE 2: Architecture of a multilayer perceptron consisting of several neurons arranged in layers. These layers are stacked in such a way that the neurons of one layer are connected to all neurons of the neighboring layers, while neurons within a layer are not connected to each other.

We start building the network's architecture by assigning a perceptron to each pixel of the image. The image displays a handwritten one-digit number from the MNIST data set[2]. Together, these artificial neurons form the input layer of the network (Figure 2). Next, we create subsequent layers of neurons, where each neuron receives input information from every neuron in the previous layer and computes an output respectively. The final layer of the network is called the output layer and consists of ten neurons, each representing one of the ten possible numbers (Figure 2). Only the input layer and the output layer are visible to us, which is why we call the layers in between hidden layers. For simplicity, we do not allow connections between neurons of the same layer only between subsequent layers.[3] Thus, the architecture we obtain resembles a network with multiple layers, each consisting of several neurons (Figure 2), which is why we refer to this neural network as the multilayer perceptron.

We will now develop the mathematics for the multilayer perceptron architecture by starting with the first neuron inside the input layer $a_0^{[0]}$. Here, the superscript index indicates the layer and the subscript index indicates the neuron within that layer. The specific value a neuron in the input layer holds depends on the greyscale value of one image's pixel, i.e. a neuron that represents a white pixel has the value 255 and a neuron that represents a black pixel has the value 0. Since one MNIST image has 28×28 pixels, the computer represents it as a $\mathbb{N}^{28x28}$ matrix. We enroll this matrix into a feature vector $\vec{x}\,\epsilon\,\mathbb{N}^{784}$ with 784 entries. This feature vector $\vec{x}$ is essentially a numerical representation of our input image's characteristics where one entry of the vector is called a feature. This feature is a numerical property of a certain aspect of the image. We pass the feature vector $\vec{x}$ with 784 entries to our neural network and accordingly require 784 neurons in the input layer to hold the grey scale value of each pixel (Figure 3 and figure 4). The mathematical representation of the input layer is a 784-dimensional vector

$$\vec{a}^{[0]} = \begin{bmatrix} a_0^{[0]} \\ a_1^{[0]} \\ \vdots \\ a_{783}^{[0]} \end{bmatrix}. \tag{2.1}$$

[2]This data set is labeled and has ten mutually exclusive classes, namely numbers from 0 to 9.

[3]Since neurons within a layer are not connected to each other, they are independent of each other. For a sufficiently large number of neurons in a layer, this allows us to see the layer in the mean field limit and therefore to describe a layer by a Gaussian. This is essential in our derivation of the relative entropy for a hierarchical network (see chapter 2.1.3.2).

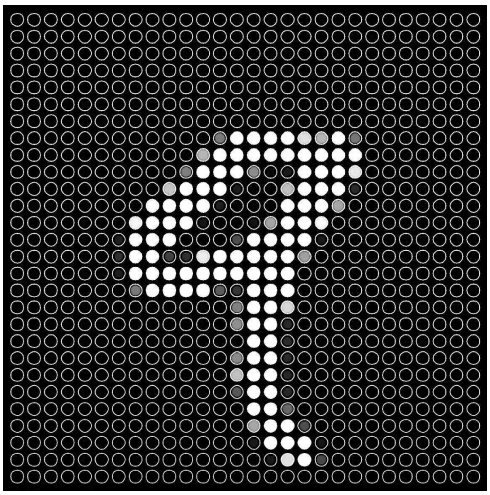

FIGURE 3: MNIST input image with 28×28 pixels which are represented as a 784-dimensional feature vector. An element of the feature vector that represents a white pixel has the value 255 and an element representing a black pixel has the value 0.

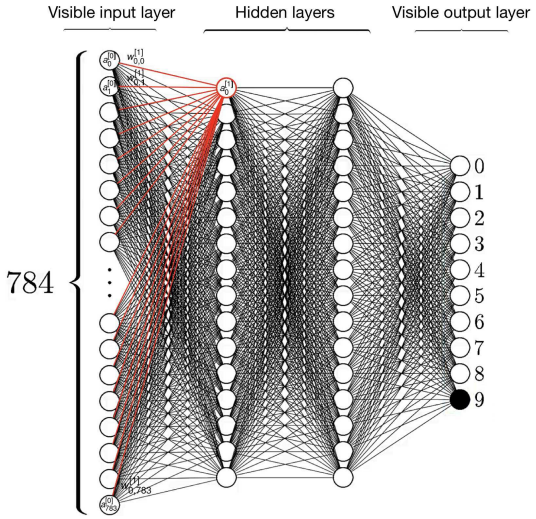

FIGURE 4: Multilayer perceptron with an input layer of 784 neurons, each representing one entry of the feature vector. Two hidden layers with 16 neurons each and an output layer with 10 neurons each, representing a number from 0 to 9.

To pass the information stored in these activations to the first neuron of the first hidden layer, connections are built to which we assign weights (Figure 2). These weights specify the connection strength between the neurons of the input layer and the first neuron inside the first hidden layer. They will be represented by the following vector

$$\vec{w}_0^{[1]} = \begin{bmatrix} w_{0,0}^{[1]} \\ w_{0,1}^{[1]} \\ \vdots \\ w_{0,783}^{[1]} \end{bmatrix}. \tag{2.2}$$

The first subscript indicates the neuron of the first hidden layer and the second subscript indicates the neuron of the input layer e.g. $w_{0,0}$ is a connection between the first neuron of the input layer and the first neuron of the hidden layer. We then compute the weighted sum for the first neuron of the first hidden layer via the dot product

$$\vec{w}_0^{[1]}\vec{a}^{[0]} = w_{0,0}^{[1]}a_0^{[0]} + w_{0,1}^{[1]}a_1^{[0]} + \ldots + w_{0,783}^{[1]}a_{783}^{[0]}. \tag{2.3}$$

Now we apply an activation function to the weighted sum. The one we choose is the hyperbolic tangent σ_T, which is differentiable, monotonic, and nonlinear. It scales the weighted sum between -1 and 1 (Figure 5).

Thus, we get the activation of the first neuron in the first hidden layer

$$a_0^{[1]} = \sigma_T(w_{0,0}^{[1]}a_0^{[0]} + w_{0,1}^{[1]}a_1^{[0]} + ... + w_{0,783}^{[1]}a_{783}^{[0]}). \tag{2.4}$$

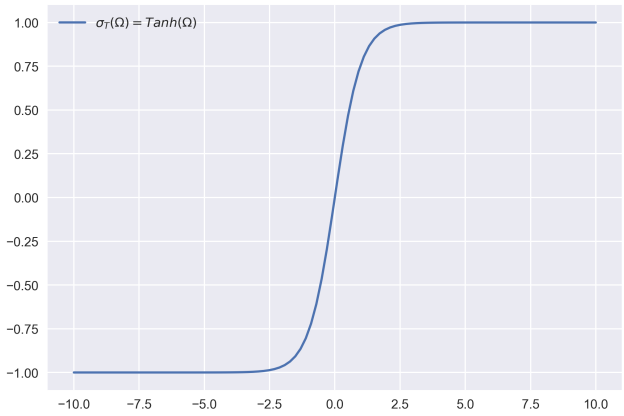

FIGURE 5: The hyperbolic tangent is used as an activation function σ_T and takes a weighted sum Ω as an input to yield an activation $a \epsilon \{-1, 1\}$.

The hyperbolic tangent does not represent an actual threshold, it rather normalizes and maps a positive weighted sum close to 1, a negative weighted sum close to -1, and a weighted sum near zero even closer to zero. In the following chapters, we will see that the activation function's non-linearity and differentiability are indispensable for the learning process.[4] For now, we find their use in scaling the weighted sum.
We can include an additional parameter b called the bias. It shifts the weighted sum to the left or right. Since the bias is independent of the inputs, it is simply added to the weighted sum

$$a_0^{[1]} = \sigma_T(w_{0,0}^{[1]}a_0^{[0]} + w_{0,1}^{[1]}a_1^{[0]} + ... + w_{0,783}^{[1]}a_{783}^{[0]} + b_0^{[1]}). \tag{2.5}$$

To be more concise we use the summation notation and call the term inside the activation function the pre-activation $z_0^{[0]}$. Hence,

$$a_0^{[1]} = \sigma_T(z_0^{[1]}) \text{ where } z_0^{[1]} := \sum_{k^{[0]}} w_{0,k^{[0]}}^{[1]}a_{k^{[0]}}^{[0]} + b_0 \text{ with } k^{[0]} \epsilon \{0, ..., 783\}. \tag{2.6}$$

Consequently, $a_0^{[1]}$ is essentially a function that takes the values of all the neurons in the previous layer, calculates their weighted sum, and runs it through an activation function that yields an activation between -1 and 1.

[4]The nonlinearity of the activation function is key to the neural network's ability to learn nonlinear relationships between high-dimensional data (see chapter 2.1.2.1). The differentiability of the activation function allows the network to backpropagate errors, which is essential for optimizing the weights and biases of the network and thus for learning the network (see chapter 2.1.2.4).

If we now connect each of the 16 neurons of the first hidden layer to the 784 neurons of the input layer, each connection having its own weights and biases, we get a total of 784×16 weights and 16 biases (Figure 4). This allows us to compute the activation of an arbitrary neuron $a_i^{[1]}$ in the first hidden layer

$$a_{i^{[1]}}^{[1]} = \sigma_T\left(\begin{bmatrix} w_{0,0}^{[1]} & w_{0,1}^{[1]} & \cdots & w_{0,783}^{[1]} \\ w_{1,0}^{[1]} & w_{1,1}^{[1]} & \cdots & w_{1,783}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i^{[1]},0}^{[1]} & w_{i^{[1]},1}^{[1]} & \cdots & w_{i^{[1]},783}^{[1]} \end{bmatrix} \begin{bmatrix} a_0^{[0]} \\ a_1^{[0]} \\ \vdots \\ a_{783}^{[0]} \end{bmatrix} + \begin{bmatrix} b_0^{[1]} \\ b_1^{[1]} \\ \vdots \\ b_{i^{[1]}}^{[1]} \end{bmatrix}\right), \text{ with } i^{[1]} \epsilon \{0,...,15\}. \tag{2.7}$$

Here, each row of the matrix corresponds to the connections between the first layer and one particular neuron in the second layer. In a more concise notation, this can be written as

$$a_{i^{[1]}}^{[1]} = \sigma_T(z_{i^{[l]}}^{[1]}) \text{ with } z_{i^{[1]}}^{[1]} = \sum_{k^{[0]}} w_{i^{[1]},k^{[0]}} a_{k^{[0]}}^{[0]} + b_{i^{[1]}}^{[1]}. \tag{2.8}$$

This equation allows us to pass the information of the input layer to the next layer. By repeating this for the subsequent layers we get a total of $784 \times 16 + 16 \times 16 + 16 \times 10$ weights and $16 + 16 + 10$ biases. Thus, in our example, we have a 13002-dimensional parameter space of possible weights and biases that can be adjusted to alter the behavior of the network (see chapter 2.1.2.4). Therefore, we can think of the entire network as being a function with 13002 parameters that takes in 784 numbers as an input and computes ten numbers as an output.

So, we started with one perceptron as the fundamental building block of the network and built the rest of the network on top of it. We then derived equation (2.8) which computes the activation of an arbitrary neuron in the first hidden layer. We now want to generalize the equation for a multilayer perceptron with arbitrary many layers L where one layer $l \epsilon \{0, 1, ..., L-1\}$ consists of $N^{[l]}$ neurons $z_{i^{[l]}}^{[l]}$ where $i^{[l]} \epsilon \{0, 1, ..., N^{[l]} - 1\}$. With respect to equation (2.8) the pre-activation of a neuron is given by

$$z_{i^{[l]}}^{[l]} = \sum_{k^{[l-1]}} w_{i^{[l]},k^{[l-1]}}^{[l]} a_{k^{[l-1]}}^{[l-1]} + b_{i^{[l]}}^{[l]} \text{ with } k^{[l-1]} \epsilon \{0, ..., N^{[l-1]} - 1\}, \tag{2.9}$$

where $w_{i,k^{[l-1]}}^{[l]}$ is an $N^{[l]} \times N^{[l-1]}$ matrix of weights. This matrix assigns a weight to each connection between a neuron $k^{[l-1]}$ in layer $[l-1]$ and a neuron $i^{[l]}$ in layer $[l]$. Equation (2.9) essentially governs the signal propagation through the network. The resulting activation function of an arbitrary neuron is therefore

$$a_{i^{[l]}}^{[l]} = \sigma_T(z_{i^{[l]}}^{[l]}). \tag{2.10}$$

With it, the information in the activations of the input layer can lead to a specific pattern of activation values in the first hidden layer, and thus to a specific pattern in the subsequent hidden layer, etc. Consequently, these activations determine the specific pattern of activations

in the output layer.[5] In chapter 2.1.2.3, we will see how we translate the activations of the output layer into probabilities. Since each neuron in the output layer represents one of the handwritten digits, the values of these activations can then be interpreted as the network's confidence that an input image belongs to that handwritten digit. Thus, the neuron in the output layer with the largest probability is the network's answer to the question of which digit is seen in the input image.

2.1.2 Learning

This chapter uses ideas from [13].

By simply implementing the architecture and feeding it with an image of a number, the neural network will not recognize the numbers or distinguish between them. This is because it first has to learn valid settings for all parameters with respect to the input image. Finding these parameter settings will be the main challenge of this chapter.

Recognizing handwritten numbers or, more specifically, predicting the class label for a given sample of input data is referred to as a classification problem. To solve this problem, our deep neural network will need to predict the class label y for a given input $\vec{x}$. Here, y can be understood as what number the feature vector $\vec{x}$ truly represents.
To get an intuitive sense of how these problems are solved by machine learning algorithms, we consider a binary classification problem, i.e., a problem with two mutually exclusive classes. A good example is to determine if a patient has a cold or COVID. To solve this binary classification problem, the selection of appropriate features is crucial. They should be fine-grained and represent the most important properties of the data with respect to the classification problem. Otherwise, the neural network can not learn to characterize concepts and distinguish between them. Irrelevant information does not help in solving the classification problem, it only increases its complexity.
We now assume having a two-dimensional feature vector $\vec{x}$ with features x_1 and x_2. To check if the selected features are appropriate to solve the classification problem we represent the data points in feature space. Note that each data point in the feature space represents a feature vector. The better our feature selection was, the more clearly data points of the same class cluster together. So, the clearer the clustering of data points the better our neural network can learn to discriminate between different classes (Figure 6) and therefore classify if a patient has COVID or not. With poorly selected features, we do not see clustering and therefore the network cannot learn to distinguish between classes (Figure 7). For our MNIST image classification problem, the features are the greyscale values of pixels. Hence,

[5] Since the information only moves in the forward direction, namely from the input layer through the hidden layers to the output layer, the multilayer perceptron falls under the category of feedforward networks.

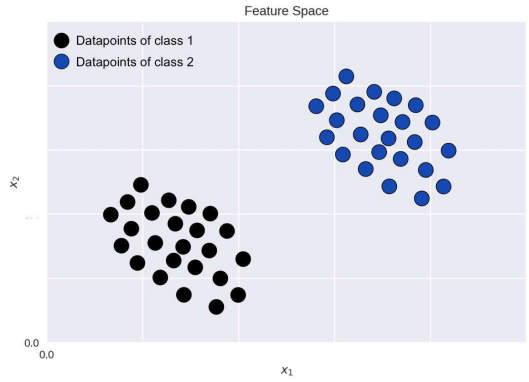

FIGURE 6: Idealized visualization of a clear clustering of data points after appropriate feature selection with respect to the classification problem.

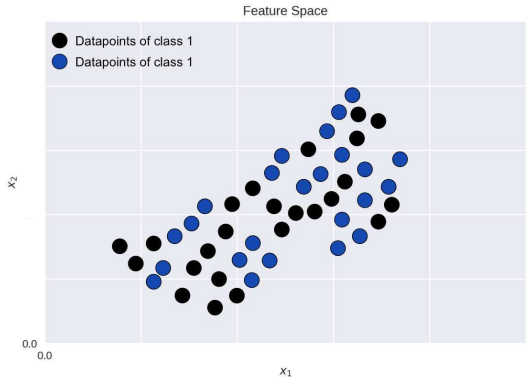

FIGURE 7: Idealized visualization of an unclear clustering of data points after bad feature selection with respect to the classification task.

we have a 784−dimensional feature space. [22][6] has demonstrated that the MNIST numbers represented by the grayscale values form clusters that allow us to discriminate between them. This makes the grayscale values appropriate features for the classification of MNIST numbers.

2.1.2.1 Discriminative Learning Algorithm

This chapter uses ideas from [13] and [17].

The approach we will use to discriminate between different classes is called the discriminative learning algorithm. It classifies by fitting a decision boundary to a given training data set that allows for discrimination between different classes. This decision boundary depends on the parameters of the network. Given the two classes we are considering in our COVID example the discriminative learning algorithm searches for a proper decision boundary in the following way:
Initially, the parameters of the multilayer perceptron are randomly initialized, resulting in the random decision boundary (red line in figure 8). As we can see, this decision boundary is not precise enough to distinguish between the two classes. By using an optimization algorithm called gradient descent[7], we optimize the parameters and generate the orange decision boundary in figure 8. This decision boundary is still not good enough to distinguish between the two classes. After another application of gradient descent, we get the green decision boundary in figure 8.

[6]For the 2d visualization in [22]'s figure 6 the t-SNE technique was used. It is a nonlinear dimensionality reduction technique for visualizing high-dimensional data in a low-dimensional space.

[7]The gradient descent optimization algorithm essentially finds an optimal set of parameters with respect to the classification problem by iteratively adjusting the network's weights and biases such that the classification error becomes smaller at each iteration. We will explain in detail how this optimization algorithm works in chapter 2.1.2.4.

As we can see, this one allows us to make a clear distinction between the classes.

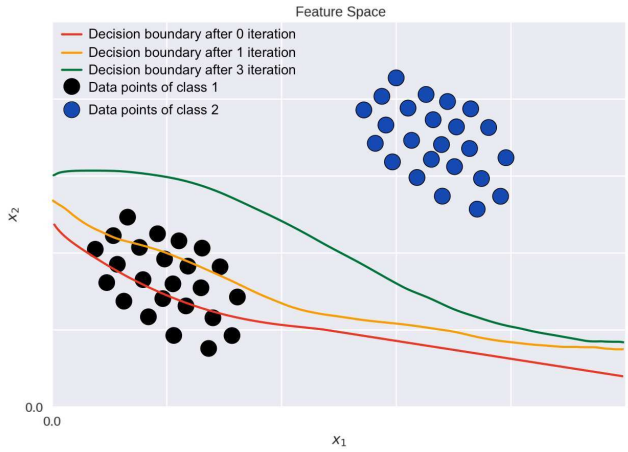

FIGURE 8: Optimization of the decision boundary with a discriminative learning algorithm. The decision boundary is optimized twice. It is defined by the parameters of the network (W, B), which are updated by the optimization algorithm so that the decision boundary better and better describes the data. The red line illustrates a decision boundary initialized randomly. The orange line illustrates the decision boundary after the first iteration of the optimization algorithm. The green line illustrates the decision boundary after the last iteration and appropriately describes the data.

Given a new input, the algorithm checks on which side of the decision boundary the input falls and makes a classification prediction accordingly. For instance, if a new input is found on the left side of the decision boundary, then the class label prediction of the deep neural network is "class 1", e.g. COVID.
Essentially, through the discriminative learning algorithm, the neural network learns a function f that accurately describes the feature vectors that are included in the training set and hopefully those that are not [8]. In other words, the network learns a function f, parameterized by all the network's weights W and all the network's biases B. This function maps a feature vector of a given image $\vec{x}$ to its corresponding label vector $\vec{y}$

$$f : \vec{x} \rightarrow \vec{y}, \tag{2.11}$$

such that the network's prediction $\hat{\vec{y}}$ is as close as possible to the label vector $\vec{y}$ [9]. Note that the prediction $\hat{\vec{y}}$ also called hypothesis depends on all the network parameters (W, B) and the feature vector $\vec{x}$.
The activation functions we mentioned in the previous chapter play a crucial role in accurately predicting the label. This is because they are essential to introduce nonlinearities into

[8] Learning a function f that accurately describes feature vectors that are not included in the training data set is the holy grail of learning and is referred to as the generalization problem. We will discuss this in chapter 2.1.2.6.

[9] The vectorization of the labels is done through one-hot encoding. We will explain this in chapter 2.1.2.3.

the decision boundaries. These nonlinearities allow us to approximate arbitrarily complex functions that can describe the nonlinear MNIST data. This is not possible with linear activation functions, since a linear combination of linear functions again yields a linear function. Thus, we essentially get a network that behaves like a single-layer neural network that cannot describe nonlinear data (demonstrated in A.1).

2.1.2.2 Supervised Learning

This chapter uses ideas from [3], [13],[14] , and [17].

For the discriminative learning algorithm we use supervised learning. In supervised learning, the neural network is fed with various different training samples, along with labels indicating what each example is supposed to represent. In our case, these training samples are handwritten images of the MNIST database. Therefore, we mathematically define a training set $\{(\vec{x}^{(1)}, y^{(1)}), ..., (\vec{x}^{(m)}, y^{(m)})\}$ consisting of multiple feature vectors $\vec{x}^{(j)} \epsilon \mathbb{N}^{784}$ with corresponding scalar labels $y^{(j)} \epsilon \{0, 1, ..., 9\}$. We will index both of them by $j \epsilon \{1, 2, ..., m\}$, where m tells us how many training samples the training set has and j which of them we are considering. We can represent the feature vectors inside the training set in a more compact notation $X \epsilon \mathbb{N}^{784 \times j}$, where each column is a feature vector $\vec{x}^{(j)}$. Then we convert the scalar labels $y^{(j)}$ into vector labels $\vec{y}^{(j)} \epsilon \mathbb{N}^{10}$ by making use of one-hot encoding (see chapter 2.1.2.3). This allows us to define a matrix $Y \epsilon \mathbb{N}^{10 \times j}$ for the label vectors, where each column is a label vector $\vec{y}^{(j)}$.
Now we can show the deep neural network the training data set (X, Y) and give it feedback on whether the prediction $\hat{\vec{y}} \epsilon \mathbb{R}^{10}$ and hence its estimated function $\hat{f}$ is accurate or not for (X, Y). We do this by using labels. Giving the network feedback using labels is the fulcrum of supervised learning and it is done by measuring the deviation of the network's prediction $\hat{\vec{y}}^{(j)}$ from the labels $\vec{y}^{(j)}$ of the training sample $\vec{x}^{(j)}$. We will explain in the upcoming chapter how this deviation is quantified.
Intuitively, supervised learning is similar to the learning process of a human child when his parents give him active feedback on his actions.

2.1.2.3 Cost function

The following chapter uses ideas from [17], [14] and [2].

The question now is how does the network measure the deviation of its predictions from the labels. The answer lies in the use of a loss function. A basic example is the mean squared error loss function

$$\mathcal{L}(\hat{\vec{y}}^{(j)}, \vec{y}^{(j)}) = \frac{1}{N^{[L-1]}} \left\| \vec{y}^{(j)} - \hat{\vec{y}}^{(j)} \right\|^2, \tag{2.12}$$

which gives us the prediction error of one training example by comparing the activation value of each neuron in the output layer with the corresponding element in the one-hot

encoded label vector. To let the network learn from all examples, we compute the average of the loss function with respect to all examples in the training set. This average is called the mean squared error cost function

$$J(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \frac{1}{m} \sum_{j=1}^{m} \frac{1}{N^{[L-1]}} \left\| \vec{y}^{(j)} - \hat{\vec{y}}^{(j)} \right\|^2 . \tag{2.13}$$

Note that a loss function $\mathcal{L}$ yields the classification error for one sample and the cost function J yields the average value of the loss function $\mathcal{L}$ over all samples inside the data set. Since J is a function of $\vec{y}^{(j)}$ and $\hat{\vec{y}}^{(j)}$, where the latter is a function of the network parameters W and B, the cost function is also a function of these parameters which is why we can denote the loss function as $J(W, B)$.[10]

In our classification problem, we want to classify single-digit numbers from 0-9 and are therefore dealing with a multi-class problem with ten mutually exclusive classes. So, for an input $\vec{x}^{(j)}$ to our multilayer perceptron, we want to know if the network can determine to which of the ten mutually exclusive classes this input belongs. For this purpose, we use an output layer with ten neurons each representing one class, i.e. $y^{(j)}$. To compare the prediction of the network with the label $y^{(j)}$, we have to introduce a one-hot encoded label vector $\vec{y}^{(j)}$. This vector has entries that are all 0 except for one, which is 1, and the sum of all entries is 1. Thus, a one-hot encoded label vector representing the MNIST digit 6 is

$$\vec{y}^{(j)} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T . \tag{2.14}$$

This vector shows that the elements of the vector are not independent of each other, because if one of them becomes one, the others are forced to become zero. Consequently, we also need to change each of the outputs $\hat{y}^{(j)}_{i^{[L-1]}}$ of our multilayer perceptron such that they are no longer independent from each other, sum up to 1, and thus can be compared to the respective labels. Moreover, we want each of the ten neurons output $\hat{y}^{(j)}_{i^{[L-1]}}$ to have a probabilistic interpretation, such that each output indicates how convinced the network is that the given input $\vec{x}^{(j)}$ belongs to one of the ten classes. In other words, we want a categorical probability distribution for the output vector. We obtain this by first scaling the output of each neuron such that $0 \leq \hat{y}^{(j)}_{i^{[L-1]}} \leq 1$ and then using a softmax activation function that essentially normalizes the outputs to a probability distribution. To do this, the softmax is simultaneously applied to the pre-activations $z^{[L-1]}_{i^{[L-1]}}$ with $i \, \epsilon \, \{0, 1, ..., 9\}$. This means that the inputs to the softmax activation function are the pre-activations which are combined to calculate the activation of the respective neuron $a^{[L-1]}_{i^{[L-1]}}$. In detail, this combination is done by dividing the exponential of the corresponding pre-activation by the sum of the exponentials of all pre-activations. Consequently, we get equation (94) in [2]

$$\hat{y}^{(j)}_{i^{[L-1]}} := a^{[L-1]}_{i^{[L-1]}} = \frac{e^{z^{[L-1]}_{i^{[L-1]}}}}{\sum_{l^{[L-1]}=0}^{N^{[L-1]}=9} e^{z^{[L-1]}_{l^{[L-1]}}}} \text{ with } i^{[L-1]} \, \epsilon \, \{0, 1, ..., 9\}. \tag{2.15}$$

[10] We use the mean square error for the autoencoder in chapter 2.2.

The exponential function ensures that $\hat{y}^{(j)}_{i^{[L-1]}} \geq 0$ and the denominator $\sum_{l^{[L-1]}=0}^{N^{[L-1]}=9} \mathrm{e}^{z^{L-1}_{l^{[L-1]}}}$ ensures, that $\hat{y}^{(j)}_{i^{[L-1]}} \leq 1$ as well as that the neurons in the output layer are interdependent. Furthermore, when summing over all activations generated by the softmax function, we obtain

$$\sum_{i^{[L-1]}=0}^{N^{[L-1]}=9} \hat{y}_{i^{[L-1]}} = \sum_{i^{[L-1]}=0}^{N^{[L-1]}=9} \left(\frac{\mathrm{e}^{z^{[L-1]}_{i^{[L-1]}}}}{\sum_{l^{[L-1]}=0}^{N^{[L-1]}=9} \mathrm{e}^{z^{[L-1]}_{l^{[L-1]}}}} \right) = 1. \tag{2.16}$$

This is because the softmax function takes into account not only the corresponding pre-activation but all pre-activations simultaneously when calculating $\hat{y}^{(j)}_{i^{[L-1]}}$. Now, the output layer can be regarded as a probability distribution, where the neuron with the highest value indicates to which class $\vec{x}^{(j)}$ belongs.

The softmax allows us to compare $\hat{\vec{y}}^{(j)}$ to $\vec{y}^{(j)}$. To measure the deviation between the prediction and the label we use an error function for multiclass classification. To derive it, we introduce the general concept of error functions by making use of Bayes' theorem

$$P(M|O) = \frac{P(O|M)P(M)}{P(O)}. \tag{2.17}$$

The crucial term in this equation is $P(O|M)$ which is referred to as the likelihood $L(M|O)$ since $P(O|M) = L(M|O)$. In the context of Deep Learning, the observations O are the training set $O := (\vec{x}^{(j)}, \vec{y}^{(j)})$. Moreover, the model M which describes a stochastic process is replaced by the parameters of the multilayer perceptron. Consequently, we rewrite (2.18) into

$$P(\vec{x}^{(j)}, \vec{y}^{(j)})|W, B) = \frac{P((\vec{x}^{(j)}, \vec{y}^{(j)})|W, B)P(W, B)}{P((\vec{x}^{(j)}, \vec{y}^{(j)}))}. \tag{2.18}$$

The likelihood can be simplified into

$$P((\vec{x}^{(j)}, \vec{y}^{(j)})|W, B) = P(\vec{y}^{(j)}|\vec{x}^{(j)}, W, B)P(\vec{x}^{(j)}|W, B), \tag{2.19}$$

where the lefthand side of the equation is the conditional probability of observing $(\vec{x}^{(j)}, \vec{y}^{(j)})$ when the network parameters are (W, B). The first term on the right side is the conditional probability of observing label $\vec{y}^{(j)}$ when the input is $\vec{x}^{(j)}$ and the parameters are (W, B). In other words, the probability that the network predicts $\hat{\vec{y}}^{(j)} = \vec{y}^{(j)}$ given input $\vec{x}^{(j)}$ and parameters (W, B). The second term is the probability of observing $\vec{x}^{(j)}$ when the network parameters are (W, B). However, since the input $\vec{x}^{(j)}$ is independent of (W, B), we can write

$$P((\vec{x}^{(j)}, \vec{y}^{(j)})|W, B) = P(\vec{y}^{(j)}|\vec{x}^{(j)}, W, B)P(\vec{x}^{(j)}). \tag{2.20}$$

By substituting this back into (2.18), applying the logarithm, and multiplying by -1, we get an intermediate function that we will define as the error function

$$\begin{aligned} \text{Error} &:= -\ln(P((\vec{x}^{(j)}, \vec{y}^{(j)})|W, B)) \\ &= -\ln(P(\vec{y}^{(j)}|\vec{x}^{(j)}, W, B)P(\vec{x}^{(j)}) - \ln(P(W, B)) + \ln(P((\vec{x}^{(j)}, \vec{y}^{(j)}))). \end{aligned} \tag{2.21}$$

We will see in a bit why we chose this name. Considering multiple training samples, which we assume to be independent of each other, we get the

$$\begin{aligned}\text{Error}_\Sigma &= -\ln\left(\prod_{j=1}^{m} P((\vec{x}^{(j)}, \vec{y}^{(j)}))|W, B)\right) \\ &= -\sum_{j=1}^{m} \ln(P(\vec{y}^{(j)}|\vec{x}^{(j)}, W, B)) - \sum_{j=1}^{m} \ln(P(\vec{x}^{(j)})) - \ln(P((W, B))) \\ &\quad + \sum_{j=1}^{m} \ln(P((\vec{x}^{(j)}, \vec{y}^{(j)}))).\end{aligned} \tag{2.22}$$

Since our network knows the label vector $\vec{y}^{(j)}$ but does not know which (W, B) to use in order to predict it, we want to find the values of (W, B) that maximize observing $\vec{y}^{(j)}$ as the network's prediction given the input $\vec{x}^{(j)}$. Therefore, we have to find the values of (W, B) that maximize the likelihood of making this observation. This is called maximum likelihood estimation. We do this by finding the parameters (W, B) that minimize the error (2.22). When taking a closer look at (2.22), we see that the terms $-\sum_{j=1}^{m} \ln(P(\vec{x}^{(j)}))$ and $\sum_{j=1}^{m} \ln(p((\vec{x}^{(j)}, \vec{y}^{(j)})))$ can be neglected since they do not depend on these parameters. Additionally, the term $-\ln(P(W, B))$ can be considered as an additional regularization that we will also neglect since we do not want to constrain the complexity of our neural network at this point (see chapter 2.2.2). A more straightforward explanation for omitting this element is that it results in a significant simplification of (2.22). Consequently,

$$\text{Error}_\Sigma = -\sum_{j=1}^{m} \ln(P(\vec{y}^{(j)}|\vec{x}^{(j)}, W, B)). \tag{2.23}$$

To find the parameters that minimize (2.23) we have to define the likelihood function for our multiclass classification problem first.
Therefore, we consider the one-hot encoded label vector $\vec{y}^{(j)}$ in more detail. If $\vec{y}^{(j)}$ represents one of the numbers from 0 to 9, then the corresponding element of the vector $y_i^{(j)} = 1$, while all other elements are zero [2].

This allows us to consider only $y_i^{(j)}$ and introduce the following likelihood function

$$L(\vec{x}^{(j)}, W, B | y_{i^{[L-1]}}^{(j)}) = P(y_{i^{[L-1]}}^{(j)} | \vec{x}^{(j)}, W, B) = \begin{cases} \hat{y}_0^{(j)} & \text{if } y_0^{(j)} = 1 \\ \hat{y}_1^{(j)} & \text{if } y_1^{(j)} = 1 \\ \hat{y}_2^{(j)} & \text{if } y_2^{(j)} = 1 \\ \hat{y}_3^{(j)} & \text{if } y_3^{(j)} = 1 \\ \hat{y}_4^{(j)} & \text{if } y_4^{(j)} = 1 \\ \hat{y}_5^{(j)} & \text{if } y_5^{(j)} = 1 \\ \hat{y}_6^{(j)} & \text{if } y_6^{(j)} = 1 \\ \hat{y}_7^{(j)} & \text{if } y_7^{(j)} = 1 \\ \hat{y}_8^{(j)} & \text{if } y_8^{(j)} = 1 \\ \hat{y}_9^{(j)} & \text{if } y_9^{(j)} = 1 \end{cases}, \tag{2.24}$$

which is a function of $\vec{x}^{(j)}$ and parameters (W, B) given $y_{i^{[L-1]}}^{(j)}$. In other words, it is the conditional probability to observe $y_{i^{[L-1]}}^{(j)}$ given the input $\vec{x}^{(j)}$ and parameters (W, B). We summarize (2.24) by a multinomial distribution

$$P(y_{i^{[L-1]}}^{(j)} | \vec{x}^{(j)}, W, B) = \prod_{i=0}^{n=9} \hat{y}_{i^{[L-1]}}^{(j)}{}^{y_{i^{[L-1]}}^{(j)}}. \tag{2.25}$$

and substitute (2.25) into (2.23), which leads to equation (137) in [2]

$$\begin{aligned} \text{Error}_\Sigma &= -\sum_{j=1}^{m} \ln \left(\prod_{i=0}^{n=9} \hat{y}_{i^{[L-1]}}^{(j)}{}^{y_{i^{[L-1]}}^{(j)}} \right) \\ &= -\sum_{j=1}^{m} \sum_{i=0}^{n=9} y_{i^{[L-1]}}^{(j)} \ln(\hat{y}_{i^{[L-1]}}^{(j)}). \end{aligned} \tag{2.26}$$

To finally get the network parameters (W, B) that minimize this error so that the network's prediction $\hat{y}_{i^{[L-1]}}^{(j)}$ is as close as possible to $y_{i^{[L-1]}}^{(j)} = 1$, we apply the argument of the minima with respect to the network parameters. Also, we add a positive constant $1/m$ that does not change at which (W, B) the minimum occurs. Consequently,

$$\text{argmin}_{\text{W,B}}(\text{Error}_\Sigma) = \text{argmin}_{\text{W,B}} \left(-\frac{1}{m} \sum_{j=1}^{m} \sum_{i=0}^{n=9} y_{i^{[L-1]}}^{(j)} \ln(\hat{y}_{i^{[L-1]}}^{(j)}) \right), \tag{2.27}$$

yields the parameters (W^*, B^*) for which the error is minimized (see equation (139) in [2]). The exact procedure to perform this minimization is discussed in detail in the upcoming chapter. The term inside (2.27) is known as the cross-entropy cost function for the multiclass problem

$$J(W, B) = -\frac{1}{m} \sum_{j=1}^{m} \sum_{i=0}^{n=9} y_{i^{[L-1]}}^{(j)} \ln(\hat{y}_{i^{[L-1]}}^{(j)}). \tag{2.28}$$

It quantifies the average prediction error known as cost by comparing the network's predictions to the labels of the corresponding input images. This comparison is done by measuring how much the distributions describing the predictions differ from the distributions describing the labels.

2.1.2.4 Gradient Descent and Backpropagation

This chapter uses ideas from [13], [19], [14], and [17].

Using the loss function (2.28) we just introduced, we can now solve our MNIST classification problem in a supervised learning environment.
We do this by giving the neural network the entire training data set (X, Y), i.e., images of handwritten digits X together with their associated labels Y for what they are supposed to represent. At the beginning, the weights and biases are randomly initialized according to a Gaussian distribution, which results in a random decision boundary. Consequently, the network will do something random and there will be a discrepancy between the prediction and the true answer. This discrepancy is signaled to the network by the cost function (2.28) in the form of a high cost. The cost essentially tells the network how wrong its predictions are on average for the given training data set (X, Y). Since (2.28) is a function of all the network's weights and biases (W, B), it also tells the network how good these parameters are for classifying the given images. In response to this feedback, the network reduces the cost by adjusting the weights and biases of the network. With these adjusted weights, the network produces a better decision boundary with respect to the data and thus provides better classification prediction. By repeating this process, the network eventually minimizes the cost and ideally converges to the correct answer. Thus, minimizing the cost function (2.27) is key to finding the optimal parameters (W^*, B^*) for making an accurate classification prediction. However, with 13002 parameters, this is a non-trivial undertaking.
To understand how minimization is performed, we simplify the cost function to $J_{X,Y}(w_0, w_1)$, resulting in a three-dimensional loss landscape (Figure 9). To find the parameter set that minimizes this cost function (w_0^*, w_1^*), we first choose a random location on the loss landscape $(w_{0,start}, w_{1,start})$. From this random location, we want to find out in which direction in the input space we need to step to reduce the cost function the fastest. In other words, we are looking for the direction of steepest descent. To find this direction, we take the negative gradient of the cost function with respect to these weights. The resulting vector then indicates the direction of steepest descent. Subsequently, we move one step towards this direction. How large this step is is determined by the learning rate α. Repeating this over and over and choosing the appropriate step size, ideally allows us to converge towards the global minimum of the loss landscape (w_0^*, w_1^*). We call this approach the gradient descent optimization algorithm. It essentially minimizes the cost function by iteratively moving towards the lowest point of the loss landscape.

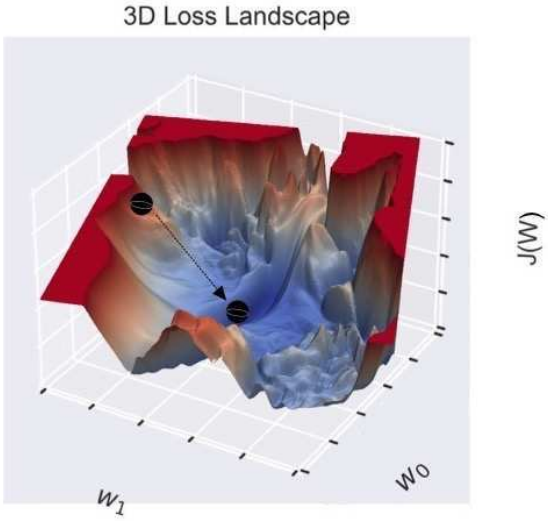

FIGURE 9: Visualized cost function $J_{X,Y}(w_0, w_1)$ in 3d. This image is taken from [11] and illustrates the loss landscape of a neural network with millions of parameters, collapsed down to two dimensions. However, we simply use it to represent a three-dimensional loss landscape, where we start at $(w_{0,start}, w_{1,start})$ and move towards the direction that reduces the cost function the fastest until we eventually land in (w_0^*, w_1^*).

We now formalize this process for our example network with 13002 adjustable parameters (Figure 4).[11] For this, we consider the cost function (2.28), which is multivariable, differentiable, and a function of all parameters (W, B) of our network. We organize all weights and biases of our neural network into a vector $\vec{P}$ that refers to a point in the n-dimensional loss landscape so that the cost function becomes $J(\vec{P})$. The minimization of the cost function with respect to all its variables using the gradient descent algorithm can now be summarized in a two-step process:

1. First, the gradient descent algorithm randomly initializes the parameters according to a Gaussian $\sim N(0, \sigma^2)$ such that it starts at $\vec{P}_{initial}$.
2. Second, the algorithm iteratively steps towards a new point using the previous point $\vec{P} \leftarrow \vec{P} + \Delta\vec{P}$. The current point is replaced by a new point until it is close enough to the optimal point $\vec{P}^*$.

The fulcrum of this method is $\Delta\vec{P}$. To determine it, we make use of the total differential of the cost function (2.28)

$$dJ_{X,Y}(\vec{P}) = \sum_{i=0}^{n-1} \frac{\partial J_{X,Y}(\vec{P})}{\partial p_i} dp_i = \frac{\partial J_{X,Y}(\vec{P})}{\partial p_0} dp_0 + \ldots + \frac{\partial J_{X,Y}(\vec{P})}{\partial p_{n-1}} dp_{n-1}. \tag{2.29}$$

[11] Since the network has adjustable 13002 parameters, we consider a 13002-dimensional parameter space. We can no longer imagine this spatially, which is why we only consider the gradient descent optimization algorithm mathematically from here on.

By assuming a small change in p_i we can write (2.29) into

$$\Delta J_{X,Y}(\vec{P}) \approx \frac{\partial J_{X,Y}(\vec{P})}{\partial p_0}\Delta p_0 + \ldots + \frac{\partial J_{X,Y}(\vec{P})}{\partial p_{n-1}}\Delta p_{n-1}, \tag{2.30}$$

which by using the gradient leads to

$$\Delta J_{X,Y}(\vec{P}) \approx \vec{\nabla}_{\vec{P}} J_{X,Y}(\vec{P})\Delta\vec{P}. \tag{2.31}$$

Since we want to step towards the direction that decreases $\Delta J_{X,Y}(\vec{P})$ the most, $\Delta\vec{P}$ should point in the direction $-\Delta J_{X,Y}(\vec{P})$. Consequently, $\Delta\vec{P} = -\alpha\Delta J_{X,Y}(\vec{P})$ where $\alpha \epsilon \mathcal{R}^+$ is the step size known as learning rate. From this follows

$$\vec{P} \leftarrow \vec{P} - \alpha\vec{\nabla}_{\vec{P}} J_{X,Y}(\vec{P}), \tag{2.32}$$

which we will use to update the weights and biases of an arbitrary layer $[l]$

$$W^{[l]} \leftarrow W^{[l]} - \alpha\vec{\nabla}_{W^{[l]}} J_{X,Y}(\vec{P}) \tag{2.33}$$

$$\vec{b}^{[l]} \leftarrow \vec{b}^{[l]} - \alpha\vec{\nabla}_{\vec{b}^{[l]}} J_{X,Y}(\vec{P}). \tag{2.34}$$

Here the negative gradient of the cost function $-\vec{\nabla}_{W.B} J_{(X,Y)}(\vec{P})$ yields a direction that indicates which weights or biases will cause the fastest decrease in the cost function. These equations are the equations (153) and (154) in [2].

The algorithm for efficiently computing this gradient is at the heart of our neural network's learning process and is called backpropagation.
To understand backpropagation intuitively, we assume a state in which the network is not well-trained and give it a training sample of a handwritten number, e.g., the number nine. Consequently, the network's output will be random. In other words, the neurons in the output layer, each representing a particular handwritten number from 0 to 9, will have a random activation value. The goal is for the neuron in the output layer representing the number nine to have an activation value of one and the other neurons to have an activation value of zero (Figure 4). To accomplish this, we recall that a neuron's activation is essentially a weighted sum of all the activations of the neurons in the previous layer plus a bias, all put into an activation function (2.4). Consequently, we can adjust either weights, biases, or activations of the previous layer to increase the activation of the neuron representing the number nine and decrease the activations of the other neurons in the output layer.
We start by considering the activation of the $i-$th neuron in the output layer $[L-1]$. We want to measure the effect that a change in the neuron's activation has on the loss function by introducing an intermediate quantity called the responsibility term θ. The detailed derivation of this term can be found in A.2. θ is the partial derivative of the loss function with respect to the activation of the $i-$th neuron in the output layer

$$\theta^{[L-1](j)}_{i^{[L-1]}} := \frac{\partial\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a^{[L-1](j)}_{i^{[L-1]}}}. \tag{2.35}$$

Note that we use a loss function $\mathcal{L}$ since we consider a single training example (j). The one, we resort to is the cross-entropy loss function from the previous chapter and we, therefore, need to consider the softmax (2.15). With a softmax, the activations of individual neurons are interdependent. This, together with the derivative of the logarithm, leads us to

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)} \frac{\frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial a_{i^{[L-1]}}^{[L-1](j)}}}{\hat{y}_{k^{[L-1]}}^{(j)}} \text{ with } k \in \{0, ..., N^{[L-1]} - 1\}. \tag{2.36}$$

Here $N^{[L-1]}$ indicates how many neurons the output layer $[L-1]$ has. Since we are considering the layer $[L-1]$, we can write $a_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{[L-1](j)}$, which allows us to rewrite (2.36) as

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} - \sum_{k^{[L-1]} \neq i^{[L-1]}} \frac{y_{k^{[L-1]}}^{(j)}}{\hat{y}_{k^{[L-1]}}^{(j)}} \frac{\frac{d\hat{y}_{i^{[L-1]}}^{(j)}}{dz_{i^{[L-1]}}^{[L-1]}}}{\frac{d\hat{y}_{k^{[L-1]}}^{(j)}}{dz_{i^{[L-1]}}^{[L-1]}}}. \tag{2.37}$$

Given that $\hat{y}_{k^{[L-1]}}^{(j)}$ is a softmax, we need to determine the derivative of the softmax with respect to $z_{i^{[L-1]}}^{[L-1]}$ in (2.37). We distinguish between two cases

$$\frac{d\hat{y}_{k^{[L-1]}}}{dz_{i^{[L-1]}}^{[L-1]}} = \begin{cases} \hat{y}_{i^{[L-1]}}(1 - \hat{y}_{i^{[L-1]}}) & \text{if } k = i \\ -\hat{y}_{k^{[L-1]}} \hat{y}_{i^{[L-1]}} & \text{if } k \neq i\,, \end{cases} \tag{2.38}$$

which simplifies (2.37) into

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} + \frac{1}{(1 - \hat{y}_{i^{[L-1]}}^{(j)})} \sum_{k^{[L-1]} \neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)}. \tag{2.39}$$

Recalling that $y_{k^{[L-1]}}^{(j)}$ is a one-hot encoded label vector's element allows us to rewrite (2.39) into equation (222) from [2]

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} + \frac{1 - y_{i^{[L-1]}}^{(j)}}{(1 - \hat{y}_{i^{[L-1]}}^{(j)})}. \tag{2.40}$$

This equation essentially measures the effect a change in a neuron's prediction has on the loss function.

We now go back one step and compute the effect a change in the pre-activation of the $i-$th output layer neuron has on the loss function. To do this, we introduce another intermediate quantity which we call the responsibility term δ. The detailed derivation of this term

can be found in A.2. It is essentially the partial derivative of the loss function with respect to the pre-activation of the $i-$th output layer neuron

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[L-1]}}^{[L-1](j)}}. \tag{2.41}$$

Knowing how a change in the activation of a neuron in the output layer affects the loss function (2.40) and $a_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{[L-1](j)}$, we use the chain rule to write δ in terms of (2.35). Thus,

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[L-1]}}^{[L-1](j)}} = \sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \hat{y}_{k^{[L-1]}}^{(j)}} \frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial z_{i^{[L-1]}}^{[L-1](j)}} \text{ with } k \,\epsilon\, \{0, ..., N^{[L-1]} - 1\}. \tag{2.42}$$

The cross-entropy loss function in combination with the softmax allows us to rewrite (2.42) into

$$\delta_{i^{[L-1]}}^{[L-1](j)} = -\sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)} \frac{\frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial z_{i^{[L-1]}}^{[L-1](j)}}}{\hat{y}_{k^{[L-1]}}^{(j)}}. \tag{2.43}$$

Then, using (2.38) together with the fact that $y_{k^{[L-1]}}^{(j)}$ is one of the one-hot encoded label vector's elements results in equation (210) from [2]

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{(j)} - y_{i^{[L-1]}}^{(j)}. \tag{2.44}$$

This equation measures the effect a change in the pre-activation of the $i-$th output layer neuron has on the loss function.

Now we determine θ and δ for an arbitrary layer $[l]$ prior to the output layer. Again, the detailed derivation of these terms can be found in A.2.
The responsibility term θ for the $i-$th neuron in the $l-$th layer for a single training sample (j) is

$$\theta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[l]}}^{[l](j)}}. \tag{2.45}$$

A change in activation of this neuron propagates from layer $[l]$ through the other layers until it reaches the output layer and changes the loss function's output. Therefore, (2.45) measures the effect that a change in activation of an arbitrary neuron has on the loss function. To derive this term, we consider θ for a neuron in layer $[l-1]$ and look at how it affects a neuron in the subsequent layer. This is the reason why we use the chain rule to write (2.45) in terms of pre-activation of layer $[l]$

$$\theta_{i^{[l-1]}}^{[l-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}} \frac{\partial z_{k^{[l]}}^{[l](j)}}{\partial a_{i^{[l-1]}}^{[l-1](j)}}. \tag{2.46}$$

According to (2.9) and (2.10), we can write $z_{k^{[l]}}^{[l](j)}$ in terms of the activation of layer $[l-1]$. Thus the derivative with respect to $a_{i^{[l-1]}}^{[l-1](j)}$ becomes

$$\frac{\partial z_{k^{[l]}}^{[l](j)}}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = \frac{\partial(\sum_{i^{[l-1]}=0}^{N^{[l-1]}-1} w_{k^{[l]}i^{[l-1]}}^{[l]} a_{i^{[l-1]}}^{[l-1](j)} + b_{k^{[l]}}^{[l]})}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = w_{k^{[l]}i^{[l-1]}}^{[l]}, \tag{2.47}$$

which we substitute into (2.46). Hence,

$$\theta_{i^{[l-1]}}^{[l-1](j)} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}} w_{k^{[l]}i^{[l-1]}}^{[l]} = \sum_{k^{[l]}=0}^{N^{[l]}-1} [(W^{[l]})^T]_{ik} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}}, \tag{2.48}$$

where $[(W^{[l]})^T]_{ik}$ is the element ik of the transposed weight matrix of layer $[l]$. We see that we can substitute the partial derivative in (2.48) by the definition of δ (2.41). Here, δ measures the effect of a change in the pre-activation of the neuron in layer $[l]$ on the loss function[12]. Hence, we write (2.48) as

$$\theta_{i^{[l-1]}}^{[l-1](j)} = [(W^{[l]})^T \vec{\delta}^{[l](j)}]_i, \tag{2.49}$$

with

$$\vec{\delta}^{[l](j)} = \begin{bmatrix} \delta_{0^{[l]}}^{[l](j)} \\ \delta_{1^{[l]}}^{[l](j)} \\ \vdots \\ \delta_{N^{[l]}-1}^{[l](j)} \end{bmatrix}, \tag{2.50}$$

where $N^{[l]}$ indicates how many neurons a layer $[l]$ has. Note that the result of multiplying the transpose of the weight matrix by $\vec{\delta}^{[l](j)}$ is a vector and therefore (2.49) is the $i-$th element of this vector. Vectorizing (2.49), yields equation (215) from [2]

$$\vec{\theta}^{[l-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l-1](j)}} = (W^{[l]})^T \vec{\delta}^{[l](j)}, \tag{2.51}$$

which allows us to compute how a change in an activation layer $[l-1]$ affects the loss function. We see that $\vec{\theta}^{[l-1](j)}$ is essentially the weighted sum of the responsibility term of the subsequent layer. The question now is how do we compute the delta for a neuron in a layer prior to the output layer.

Similar to θ, we use the chain rule to write δ for an arbitrary layer $[l]$ prior to the output layer in terms of the pre-activation of the subsequent layer

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l+1]}}^{[l+1](j)}} \frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}} \text{ with } k^{[l+1]} \epsilon \{0, ..., N^{[l+1]-1}\}. \tag{2.52}$$

[12] A change in pre-activation leads to a change in the neuron's activation, which then propagates through the layers, and ultimately leads to a change in the loss function's output.

Here we recognize the definition of δ for the layer $[l+1]$. Thus,

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} \delta_{i^{[l+1]}}^{[l+1](j)} \frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}}. \tag{2.53}$$

Then, equations (2.9) and (2.10) allow us to write the term $z_{l^{[l+1]}}^{[l+1](j)}$ in (2.53) as

$$z_{k^{[l+1]}}^{[l+1](j)} = \sum_{i^{[l]}=0}^{N^{[l]}-1} w_{k^{[l+1]}i^{[l]}}^{[l+1]} \sigma_T(z_{k^{[l]}}^{[l](j)}) + b_{k^{[l+1]}}^{[l+1]}. \tag{2.54}$$

Since the pre-activations of different neurons in the same hidden layer are independent of each other, we write the derivative in (2.53) as

$$\frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}} = w_{k^{[l+1]}i^{[l]}}^{[l+1]} \sigma_T'(z_{i^{[l]}}^{[l](j)}). \tag{2.55}$$

Here $\sigma_T'(z_{i^{[l]}}^{[l](j)})$ is the derivative of $\sigma_T(z_{i^{[l]}}^{[l](j)})$ with respect to $z_{i^{[l]}}^{[l](j)}$. If we then substitute (2.55) into (2.53), we get

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \left(\sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} w_{k^{[l+1]}i^{[l]}}^{[l+1]} \delta_{i^{[l+1]}}^{[l+1](j)} \right) \sigma_T'(z_{i^{[l]}}^{[l](j)}), \tag{2.56}$$

which we can bring into vector form using the Hadamard product. Consequently,

$$\vec{\delta}^{[l](j)} = ((W^{[l+1]})^T \vec{\delta}^{[l+1](j)}) \odot \sigma_T'(\vec{z}^{[l](j)}), \tag{2.57}$$

which is the delta vector for layer $[l]$ as a function of the delta vector for the subsequent layer. In the last step, we use (2.51) and obtain equation (218) from [2]

$$\vec{\delta}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l](j)}} \odot \sigma_T'(\vec{z}^{[l](j)}). \tag{2.58}$$

This term measures the effect that a change in the pre-activation of a neuron in an arbitrary layer $[l]$ prior to the output layer, has on the loss function. Also, note that we must exclude the output layer for $\vec{\delta}^{[l](j)}$ and $\vec{\theta}^{[l-1](j)}$ because it uses a softmax and therefore the activations and pre-activations depend on each other. Which is not the case for the hidden layers $[l]$.

We now want to compute how a change in a bias of layer $[l]$ affects the loss function. Therefore, we use the chain rule and write $\delta_{i^{[l]}}^{[l](j)}$ in terms of the bias

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b_{k^{[l]}}^{[l](j)}} \frac{\partial b_{k^{[l]}}^{[l](j)}}{\partial z_{i^{[l]}}^{[l](j)}}. \tag{2.59}$$

Again, the detailed derivation of this term can be found in A.2. Subsequently, using (2.9) together with (2.10) for a neuron $k^{[l]}$, we can write

$$b^{[l]}_{k^{[l]}} = z^{[l](j)}_{k^{[l]}} - \sum_{p^{[l-1]}} w^{[l]}_{k^{[l]}p^{[l-1]}} a^{[l-1](j)}_{p^{[l-1]}}, \tag{2.60}$$

which leads to

$$\delta^{[l](j)}_{i^{[l]}} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b^{[l](j)}_{i^{[l]}}} \frac{\partial b^{[l](j)}_{i^{[l]}}}{\partial z^{[l](j)}_{i^{[l]}}} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b^{[l](j)}_{i^{[l]}}} \text{ for k = i.} \tag{2.61}$$

Finally, we vectorize everything and obtain equation (182) from [2]

$$\vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}. \tag{2.62}$$

So, by computing the delta vector for an arbitrary layer $[l]$ via (2.58), we obtain the gradient of the loss function with respect to the bias vector of an arbitrary layer $[l]$ prior to the output layer.

To compute how a change in a weight of layer $[l]$ changes the loss function, we again use the chain rule and write δ for layer $[l]$ in terms of the weights

$$\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial w^{[l](j)}_{i^{[l]},k^{[l-1]}}} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z^{[l](j)}_{i^{[l]}}} \frac{\partial z^{[l](j)}_{i^{[l]}}}{\partial w^{[l](j)}_{i^{[l]},k^{[l-1]}}} = \delta^{[l](j)}_{i^{[l]}} \frac{\partial z^{[l](j)}_{i^{[l]}}}{\partial w^{[l](j)}_{i^{[l]},k^{[l-1]}}}. \tag{2.63}$$

The derivative of the pre-activation with respect to the weights yields

$$\frac{\partial z^{[l](j)}_{i^{[l]}}}{\partial w^{[l](j)}_{i^{[l]},k^{[l-1]}}} = a^{[l-1](j)}_{k^{[l-1]}}, \tag{2.64}$$

considering that the pre-activation is defined as (2.9). Then, we can rewrite (2.63) using (2.64) as

$$\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial w^{[l](j)}_{i^{[l]},k^{[l-1]}}} = a^{[l-1](j)}_{k^{[l-1]}} \delta^{[l](j)}_{i^{[l]}}. \tag{2.65}$$

Expressing this equation in vector form results in the gradient of the loss function with respect to the weight matrix

$$\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)} (\vec{a}^{[l-1](j)})^T, \tag{2.66}$$

where $\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})$ is an $N^l \times N^{l-1}$ matrix, since $W^{[l]}$ is an $N^l \times N^{l-1}$ matrix. On the right hand side of (2.66), the vector $\vec{\delta}^{[l](j)}_{i^{[l]}}$ has N^l elements and the transposed activation vector has $N^{[l-1]}$ elements. Using this equation together with (2.58), (2.9), and (2.10), allows us to determine how a change in the weights of layer $[l]$ affects the loss function.

Equation 2.66 is equation (188) in [2] and the detailed derivation of it can be found in A.2. With these responsibility terms, we can calculate the gradient of the loss function. We do this by first computing the responsibility term of the output layer and then propagating it back to the second-to-last layer. Next, we compute the responsibility term of the second-to-last layer using the responsibility term of the output layer. Again, the resulting responsibility term is propagated back to the layer before it. By repeating this for all other layers, we calculate the gradient of the loss function. So, using equations (2.9), (2.10), (2.40), (2.61), (2.51), (2.58), (2.62), and (2.66), we define the backpropagation algorithm with a pseudo-code inspired by [2] and [14].

Backpropagation pseudo-code

for l **in** {0,1,...,L-1} **do:**

$\vec{a}^{[l](j)} = \sigma_T(W^{[l]}\vec{a}^{[l-1](j)} + \vec{b}^{l})$

end for

for i **in** {0,1,...,9} **do:**

$\delta_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{(j)} - y_{i^{[L-1]}}^{(j)}$

$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[L-1]}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} + \frac{1-y_{i^{[L-1]}}^{(j)}}{(1-\hat{y}_{i^{[L-1]}}^{(j)})}$

end for

for l **in** {L-2,...,0} **do:**

$\vec{\delta}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l](j)}} \odot \sigma_T'(\vec{z}^{[l](j)})$

$\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l](j)}} = (W^{[l]})^T \vec{\delta}^{[l](j)}$

end for

for l **in** {0,1,...,L-1} **do:**

$\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}(\vec{a}^{[l-1]})^T$

$\vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}$

end for

Intuitively, the backpropagation algorithm maximizes the activation of the neuron in the output layer that represents the number shown on the input image, while minimizing the activations of the other neurons in the output layer. The algorithm achieves this by adjusting the activations in the second-to-last layer accordingly. To adjust the activations in the second-to-last layer such that they produce the desired activations in the output layer, the algorithm adjusts the activations in the layer before. This process continues to the input layer.

Since the backpropagation algorithm yields the gradient of the loss function, but the gradient descent method requires the gradient of the cost function $J_{X,Y}(W, B)$ (2.28), we compute

the average of the loss function over all the training examples (batch). Consequently,

$$J_{X,Y}(\vec{P}) = \frac{1}{m}\sum_{j=1}^{m}\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}), \tag{2.67}$$

which allows us to rewrite the equations (2.62) and (2.66) into

$$\vec{\nabla}_{W^{[l]}} J_{X,Y}(\vec{P}) = \frac{1}{m}\sum_{j=1}^{m}\vec{\nabla}_{W^{[l]}}\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}), \tag{2.68}$$

and

$$\vec{\nabla}_{\vec{b}^{[l]}} J_{X,Y}(\vec{P}) = \frac{1}{m}\sum_{j=1}^{m}\vec{\nabla}_{\vec{b}^{[l]}}\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}). \tag{2.69}$$

Note that $\hat{y}$ depends on all weights and biases of the network (W, B), i.e., it depends on $\vec{P}$. We can now define the gradient descent algorithm by the following pseudo-code which is inspired by [2] and [14].

Batch Gradient Descent pseudo-code

Random initialization of parameters $\sim N(0, \sigma^2)$
'''
The model passes through all samples $(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})$
of the training data set n times
'''
for n **in** range(epochs) **do**
    # Backpropagation
    for j **in** {1,...,m} **do**
        for l **in** {0,...,L-1} **do**
            $\vec{a}^{[l](j)} = \sigma_T(W^{[l]}\vec{a}^{[l-1](j)} + \vec{b}^{l})$
        end for
        $\vec{\hat{y}}^{(j)} = \vec{a}^{[L-1](j)}$
        for i **in** {0,1,...,9} **do**:
            $\delta^{[L-1](j)}_{i^{[L-1]}} = \hat{y}^{(j)}_{i^{[L-1]}} - y^{(j)}_{i^{[L-1]}}$
            $\theta^{[L-1](j)}_{i^{[L-1]}} = \frac{\partial\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial\vec{a}^{[L-1]}} = -\frac{y^{(j)}_{i^{[L-1]}}}{\hat{y}^{(j)}_{i^{[L-1]}}} + \frac{1-y^{(j)}_{i^{[L-1]}}}{(1-\hat{y}^{(j)}_{i^{[L-1]}})}$
        end for
        for l **in** {L-2,L-3,...,0} **do**
            $\vec{\delta}^{[l](j)} = (W^{[l+1]})^T\vec{\delta}^{[l+1](j)} \odot \sigma_T'(\vec{z}^{[l](j)})$
            $\frac{\partial\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial\vec{a}^{[l](j)}} = (W^{[l]})^T\vec{\delta}^{[l](j)}$
        end for
        for l **in** {0,...,L-1} **do**
            $\vec{\nabla}_{W^{[l]}}\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}(\vec{a}^{[l-1]})^T$

$\vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}$

end for

end for

Update of parameters

for l **in** {0,...,L-1} **do**

$\vec{\nabla}_{W^{[l]}} J_{X,Y}(\vec{P}) = \frac{1}{m} \sum_{j=1}^{m} \vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})$

$\vec{\nabla}_{\vec{b}^{[l]}} J_{X,Y}(\vec{P}) = \frac{1}{m} \sum_{j=1}^{m} \vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})$

$W^{[l]} \leftarrow W^{[l]} - \alpha \vec{\nabla}_{W^{[l]}} J_{X,Y}(\vec{P})$

$\vec{b}^{[l]} \leftarrow \vec{b}^{[l]} - \alpha \vec{\nabla}_{\vec{b}^{[l]}} J_{X,Y}(\vec{P})$

end for

end for

So, the gradient descent algorithm essentially computes the gradients of the cost function using all the training samples (batch) at each iteration through the help of backpropagation. For this reason, we also refer to this algorithm as the Batch Gradient Descent (BGD) algorithm. The gradients are then used to update the weights and biases, and by repeating this process over n epochs, we ideally obtain an optimal set of parameters.

2.1.2.5 Stochastic Gradient Descent

This chapter uses ideas from [14], [12] [18], and [2].

The main problem with the Batch Gradient Descent algorithm (BGD) is that it considers the entire training data set when computing the gradient. Especially since Deep Learning requires large training data sets, BGD requires a lot of time and compute to bring the parameters to the global minimum of the loss landscape and thus to obtain an optimal set of parameters. Therefore, we resort to a method that requires less time and compute, the Stochastic Gradient Descent algorithm (SGD) [14]. This algorithm saves time and compute by not considering the entire training data set (X, Y), but only one randomly chosen sample $(x^{(j)}, y^{(j)})$ for computing the gradient of the cost function. But this is also the reason why the algorithm will not yield the true gradient $\nabla_{W,B} J_{X,Y}(W, B)$ as BGD does, rather it provides a rough estimate $\nabla_{W,B} \mathcal{L}(W, B)$ [14]. So, we have a trade-off between speed and accuracy.
As a suitable trade-off between speed and accuracy, we consider a predefined number s[13] of training samples, called a mini-batch χ, to compute the gradient of the cost function. This is also why the resulting algorithm is referred to as the Mini-Batch SGD algorithm. It brings the parameters to a region near the global minimum of the loss landscape. The Mini-Batch SGD algorithm randomly shuffles the samples within the training data set. This ensures that samples are randomly selected for the computation of the gradient. We will

[13] s is also referred to as the batch size.

see in a moment why this is useful. The random shuffling requires re-indexing the samples within the input matrix from section 2.1.2.2. This results in a new input matrix

$$X' = [\vec{x}^{(1')}, ..., \vec{x}^{(m')}], \tag{2.70}$$

with corresponding label matrix

$$Y' = [\vec{y}^{(1')}, ..., \vec{y}^{(m')}]. \tag{2.71}$$

We will use the index j' to indicate which of these shuffled and re-indexed samples we are considering. Recall, that m' is the number of samples within the training data set that remains the same after shuffling and re-indexing. By then dividing X into m/s mini-batches χ we get

$$\chi = [\vec{\chi}^{(1)}, ..., \vec{\chi}^{(m/s)}], \tag{2.72}$$

with corresponding label matrix

$$\xi = [\vec{\xi}^{(1)}, ..., \vec{\xi}^{(m/s)}], \tag{2.73}$$

where we will use the index g to indicate which of these mini-batches we are considering. We then compute the average of the loss function over all samples within a mini-batch $\vec{\chi}^{(g)}$. Thus, we obtain equations (233) and (234) from [2]

$$\vec{\nabla}_{W^{[l]}} J_{\vec{\chi}^{(g)},\vec{\xi}^{(g)}}(\vec{P}) \approx \frac{1}{s} \sum_{(\vec{x}^{(j')},(\vec{y}^{(j')}) \, \epsilon \, (\vec{\chi}^{(g)},\vec{\xi}^{(g)})} \vec{\nabla}_{W^{[l]}} \mathcal{L}((\vec{\hat{y}}^{(j')}, (\vec{y}^{(j')})), \tag{2.74}$$

and

$$\vec{\nabla}_{b^{[l]}} J_{\vec{\chi}^{(g)},\vec{\xi}^{(g)}}(\vec{P}) \approx \frac{1}{s} \sum_{(\vec{x}^{(j')},(\vec{y}^{(j')}) \, \epsilon \, (\vec{\chi}^{(g)},\vec{\xi}^{(g)})} \vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}((\vec{\hat{y}}^{(j')}, (\vec{y}^{(j')})). \tag{2.75}$$

With these equations, we define our Mini-Batch SGD algorithm using a pseudo-code that is inspired by [2] and [14].

```
-------------------------------------------------------------
# Mini-Batch Stochastic Gradient Descent pseudo-code

Random initialization of parameters ~ N(0, σ^2)
'''
The model passes through all samples (ŷ^(j), y^(j))
of the training data set n times
'''
for n in range(epochs) do
    Random shuffling of the training data set (X,Y)
    Dividing the training data set (X,Y) into m/s mini-batches,
    where one mini-batch χ^(g) consists of s samples
    χ_k = ((x^(1), y^(1)), ..., (x^(m/s), y^(m/s)))
```

```
    '''
    Scan through all mini-batches
    '''
    for g in {1,...,m/s} do
        for $\vec{x}^{j'}, \vec{y}^{j'}$ in $\vec{\chi}^{(g)}$, $\vec{\xi}^{(g)}$ do
            for l in {0,...,L-1} do
                $\vec{a}^{[l](j')} = \sigma_T(W^{[l]}\vec{a}^{[l-1](j')} + \vec{b}^{l})$
            end for
            $\vec{\hat{y}}^{(j)} = \vec{a}^{[L-1](j')}$
            $\delta^{[L-1](j')}_{i^{[L-1]}} = \hat{y}^{(j')}_{i^{[L-1]}} - y^{(j')}_{i^{[L-1]}}$
            $\theta^{[L-1](j')}_{i^{[L-1]}} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j')}, \vec{y}^{(j')})}{\partial \vec{a}^{[L-1]}} = -\frac{y^{(j')}_{i^{[L-1]}}}{\hat{y}^{(j')}_{i^{[L-1]}}} + \frac{1-y^{(j')}_{i^{[L-1]}}}{(1-\hat{y}^{(j')}_{i^{[L-1]}})}$
            for l in {L-2,L-3,...,0} do
                $\vec{\delta}^{[l](j')} = (W^{[l+1]})^T \vec{\delta}^{[l+1](j')} \odot \sigma'_T(\vec{z}^{[l](j')})$
                $\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j')}, \vec{y}^{(j')})}{\partial \vec{a}^{[l](j')}} = (W^{[l]})^T \vec{\delta}^{[l](j')}$
            end for
            for l in {0,...,L-1} do
                $\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j')}, \vec{y}^{(j')}) = \vec{\delta}^{[l](j')} (\vec{a}^{[l-1]})^T$
                $\vec{\nabla}_{b^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j')}, \vec{y}^{(j')}) = \vec{\delta}^{[l](j')}$
            end for
        end for
        for l in {0,...,L-1} do
            # Compute Gradient approximation
            $\vec{\nabla}_{W^{[l]}} J(\vec{P}) = \frac{1}{s} \sum_{(\vec{x}^{(j')}, (\vec{y}^{(j')}) \epsilon (\vec{\chi}^{(g)}, \vec{\xi}^{(g)})} \vec{\nabla}_{W^{[l]}} \mathcal{L}((\vec{\hat{y}}^{(j')}, (\vec{y}^{(j')}))$
            $\vec{\nabla}_{b^{[l]}} J(\vec{P}) = \frac{1}{s} \sum_{(\vec{x}^{(j')}, (\vec{y}^{(j')}) \epsilon (\vec{\chi}^{(g)}, \vec{\xi}^{(g)})} \vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}((\vec{\hat{y}}^{(j')}, (\vec{y}^{(j')}))$

            # Update Parameters
            $W^{[l]} \leftarrow W^{[l]} - \alpha \vec{\nabla}_{W^{[l]}} J(\vec{P})$
            $\vec{b}^{[l]} \leftarrow \vec{b}^{[l]} - \alpha \vec{\nabla}_{\vec{b}^{[l]}} J(\vec{P})$
    end for
end for
```

The added value of shuffling becomes clear when we consider that the loss landscape of our cost function $J_{X,Y}(\vec{P})$ has numerous local minima that can cause the algorithm to get stuck (Figure 9). Since BGD considers the entire training data set (X,Y) at each iteration, the loss landscape is the same at each iteration. This results in the BGD algorithm being unable to escape if it gets stuck in a local minimum. In contrast, Mini-Batch SGD selects a new mini-batch $(\chi^{(g)}, \vec{\xi}^{(g)})$ at each training iteration, such that the loss landscape changes at each iteration. Thus, if the algorithm gets stuck in a local minimum of the loss landscape given by $J_{\chi^{(g)}, \vec{\xi}^{(g)}}(\vec{P})$ with the training mini-batch $\chi^{(g)}$ in iteration g, then in the

next iteration the algorithm may no longer be trapped in this in this local minimum, as it considers a new mini-batch $(\chi^{(g+1)}, \vec{\xi}^{(g+1)})$ that changes the shape of the loss landscape given by $J_{\chi^{(g+1)},\vec{\xi}^{(g+1)}}(\vec{P})$. Thus, the more different loss landscapes the algorithm sees, the more likely it is to reach the global minimum.

To get a better intuition of the difference between BGD and Mini-Batch SGD, we have visualized the trajectory of both algorithms in an idealized[14] way (Figure 9 and figure 10). Compared to BGD, where the trajectory heads directly towards the global minimum (Figure 9), Mini-Batch SGD's downward trajectory oscillates (Figure 10) [12]. This is because BGD carefully computes each step in the downward direction by using all samples $(x^{(j)}, y^{(j)})$ in the training data set. The Mini-Batch SGD algorithm, on the other hand, only computes a rough estimate of the downward direction based on a small number of samples (mini-batch) [12]. Thus, a step in this direction is taken much faster than with BGD. This can also lead to the algorithm having to correct missteps which is the reason for the oscillating downward trajectory [12]. Nevertheless, the Mini-Batch SGD algorithm is able to land in a region close to the global minimum of the loss landscape. Furthermore, the mini-batch SGD requires less time per iteration and is computationally more efficient since it does not return the actual negative gradient of the cost function, which makes it catch up to BGD in terms of accuracy. According to [7], Mini-Batch SGD has proven to scale, perform and even generalize better than BGD, which is why we will resort to it in our implementation.

We can get mini-batch SGD to roll down the loss landscape even faster to the global minimum while reducing the probability that it will get stuck in a local minimum. To do this, we take a closer look at the learning rate α. This determines how big a step we take toward the negative gradient. Intuitively, it can be thought of as the size of a ball, with the ball itself representing our algorithm (Figure 10). The larger the learning rate, the larger the ball, the faster it can reach the global minimum of the loss landscape, and the less likely it is to get stuck in a local minimum [18]. However, if the ball is too large, it overshoots and does not reach the global minimum at all; the algorithm diverges [18]. This is also called overshooting. The difficult part is choosing the right learning rate. Choosing the right one is more of an art than a science, as it depends on several factors, e.g., batch size, optimization algorithm, etc. An additional method to speed up the learning process is momentum (see [24] and [18]). It dampens the oscillations of the Mini-Batch SGD algorithm and thus moves the algorithm more directly towards the minimum (Figure 10).

[14] Our example network has 13002 adjustable parameters, therefore we consider a 13002-dimensional parameter space, which we can no longer visualize spatially. For the sake of intuition, we therefore consider a simplified 3d loss landscape.

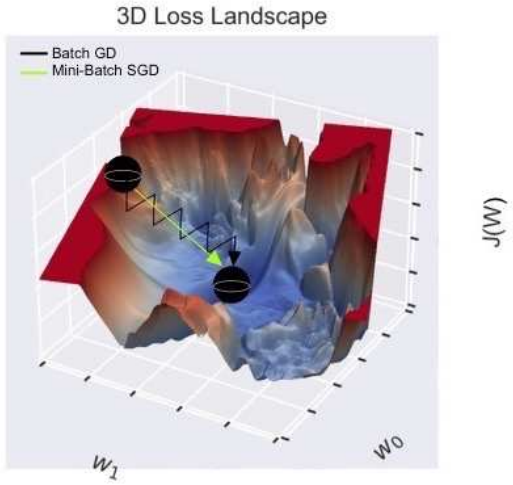

FIGURE 10: Idealized Stochastic Gradient Descent algorithm with and without momentum going from a random starting point on the loss landscape to its global minimum. Unlike the batch gradient descent algorithm, which considers the entire training set and therefore moves directly towards the global minimum (Figure 9), both Mini-Batch Gradient Descent Algorithms consider only a mini-batch of samples and therefore oscillate toward the global minimum. For the algorithm with momentum, the oscillations are attenuated so that it reaches the global minimum faster than the algorithm without momentum. This figure was taken from [11].

2.1.2.6 The Generalization Problem

This chapter uses ideas from [14].

The final goal is that our neural network, after being trained with the training data set (X,Y), can recognize images that were not in the data set. In other words, the network can recognize images that it has not seen before. This is also referred to as generalization.
To do this, we first train the network using the training data set (X,Y). Then, we give the network a set of images that the network has not seen before X^{Test}. This data set contains no labels and is also called the test data set. In chapter 2.1.2.1 we explained that the neural network classifies images by fitting a decision boundary, which is a function of all parameters (W,B), to our data points in a multidimensional feature space. This decision boundary can be a polynomial function of higher order and is key to generalization. However, unlike in 2.1.2.1, we now consider different decision boundaries not during training, but after training. We visualize them in a two-dimensional feature space where the data points also no longer belong to the training data set, but to the test data set. The red decision boundary in figure 11 belongs to a network that has not been trained well. It is not able to fit the data points in a way that allows for a clear distinction between the two classes. Therefore, the decision boundary is not able to describe the complexity of the test data set, and the network is not able to generalize to images beyond the training data. We refer to this case as underfitting. The green decision boundary in figure 11 belongs to a network that has

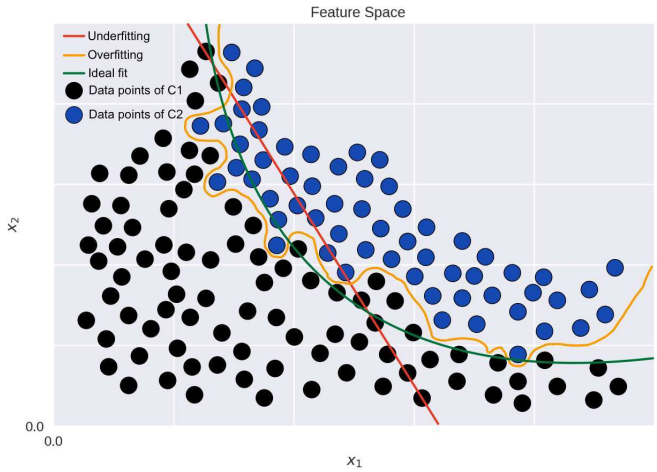

FIGURE 11: Different decision boundaries of a network trained with the training data set (X, Y). The decision boundaries are visualized in a 2d feature space, where the data points represent the test data set (X^{test}). The red decision boundary belongs to a network that has not been well-trained on (X, Y) and cannot properly describe the test data set. Consequently, it cannot generalize to images beyond the training data. The green decision boundary belongs to a network that has been well-trained on (X, Y) and can properly describe the test data set. Consequently, it can generalize. The orange decision boundary belongs to a network that has also been well-trained on (X, Y) and can perfectly describe the test data set. However, it cannot be generalized because its decision boundary is too volatile and therefore does not allow for a clear decision given a test image.

been trained with success, i.e., it has a low classification error. It fits well to the data and is therefore able to discriminate between classes. Consequently, the neural network can generalize. The orange decision boundary in figure 11 also belongs to a network that has been trained with success. It is a polynomial function of higher order and seems to fit the test data well since it considers every data point. It discriminates perfectly between the data points of the classes. The drawback is that the decision boundary is volatile, which means that the space of possible classification predictions is large [14]. Therefore, a clear decision for an unseen test sample X^{Test} is unlikely and the neural network is unable to generalize. We call this case overfitting. Simply put, overfitting means that the model "remembers" aspects of our training data set. This generally happens when we have many features and relatively few samples. Aside from having enough samples, we prevent overfitting when using the Mini-Batch SGD, by reshuffling the samples within the training data set at each epoch. This prevents the network from "memorizing" the order of the samples.

To conclude the chapter on learning: The neural network learns to classify MNIST numbers by iteratively minimizing a multidimensional cost function with respect to its weights and biases via a gradient descent algorithm. The resulting parameters are then used to approximate a decision boundary that can ideally distinguish between the different MNIST numbers, allowing the network to make a classification prediction for an unseen image.

2.1.3 Quantifying the flow of information

After having looked at the architecture of a deep feedforward network and how it learns, the following part of my thesis aims to shed more light on what happens inside a deep neural network by quantifying the flow of information through it. To do this, we look at information through an entropic lens by using ideas from [10], [23], and [6].

2.1.3.1 Entropy

To see information through an entropic lens, we first consider entropy, the well-familiar measure of disorder or uncertainty in thermodynamics.
The Boltzmann entropy states that the entropy of a macrostate is proportional to the logarithm of the number of available microstates for this macrostate, i.e., the volume in phase space occupied by all available microstates Γ. In other words, the entropy is proportional to the logarithm of the number of ways in which the particles of a thermodynamic system can be rearranged. Then, using the Boltzmann constant k_B as a constant of proportionality, we obtain

$$S_B(\Gamma) \coloneqq k_B \ln(\Gamma). \tag{2.76}$$

Note that the Boltzmann entropy assumes that every microstate is equally likely. If this is not the case, we obtain the more general expression for the entropy

$$S_G(p(x)) \coloneqq -k_B \sum_i p(x_i) \ln(p(x_i)), \tag{2.77}$$

where $p(x_i)$ is the probability of the system being in one of the microstates x_i. This entropy is referred to as Gibbs entropy.
Both Boltzmann entropy and Gibbs entropy essentially measure the uncertainty about the underlying microstate for a given macrostate.

By relating uncertainty to information, we enter the realm of information theory. To understand this relation we introduce the concept of "surprisal". The rarer an event is, the more surprised we are, and therefore we can interpret information as the knowledge we do not possess or as the degree of being surprised, i.e., surprisal. Thus, information or surprisal is inversely proportional to the probability of an event occurring. In this context, we interpret the probability as uncertainty. From this relationship between information and uncertainty, it follows that entropy is also a measure of information.
Claude Shannon's information theory [21] allows us to formalize this by considering one of a communication system's main components; the information source. It can be thought of as a system emitting sequences of signals. The probability that the information source emits signals $x = (x_1, x_2, ..., x_n)$, where x is a discrete random variable, is given by its underlying probability distribution $p(x) = (p_1, p_2, ..., p_n)$. In combination with the notion that the smaller the probability of a signal, the more surprised we are when the event occurs,

we can now quantify the concept of surprisal, i.e., how surprised we are when a signal occurs. Our first guess for the definition of surprisal is $1/p(x_i)$. However, an event can also be composed of two independent events that occur with probability $q(z_i)$ and $r(y_i)$, respectively. Intuitively, we then expect the surprisal of x_i to equal the surprisal of z_i plus the surprisal of y_i. But, $1/p(x_i) \neq 1/q(z_i) + 1/r(y_i)$, which is why the surprisal of a signal x_i is not correctly defined by $1/p(x_i)$. If instead we define the surprisal of a signal x_i as the logarithm of the inverse of the associated probability $p(x_i)$, additivity is satisfied since $-\ln(q(z_i)r(y_i)) = -(\ln(q(z_i)) + \ln(r(z_i)))$. Consequently, we define the surprisal of x_i as

$$I(x_i) \coloneqq \ln\left(\frac{1}{p(x_i)}\right) \text{ for } i \in \{1, ..., n\}. \tag{2.78}$$

By weighting the surprisals of different values x_i (2.78) with their probability of occurrence we get the expected surprisal of the information source

$$S_S(p(x)) \coloneqq -\sum_{i}^{N} p(x_i)\ln(p(x_i)), \tag{2.79}$$

which is always non-negative and maximal iff the signals are equally likely. It is zero iff the source always emits the same signal x_i, because then $S_S = -1\ln(1)$. Since we can think of the surprisal of x_i as an analogy for the information contribution of x_i, (2.79) is essentially a measure of how much information an information source produces. We see the similarity to (2.77), which is why this information measure is also called the Shannon entropy.

We will now look at Shannon's classical theory from the perspective of quantum mechanics. We, therefore, consider a quantum system Q with d degrees of freedom which is associated with a Hilbert space $\mathcal{H}$ of dimension d. An example of a quantum system modeled by a d-dimensional Hilbert space is the spin of an electron with a basis given by $\{|0\rangle, |1\rangle\}$. Here $|0\rangle$ and $|1\rangle$ are the eigenstates of the Pauli matrix σ_z. A pure state is described by a normalized vector in Hilbert space $|\phi\rangle \in \mathcal{H}$ with $\langle\phi|\phi\rangle = 1$, such that the density matrix takes the form $\hat{\rho} = |\phi\rangle\langle\phi|$ with $tr(\hat{\rho}) = 1$. This density matrix is a normalized hermitian operator. When a system is prepared in such a way that it is found in the state $|\phi_i\rangle$ with the probability p_i, which is also called a mixed state, then it is described by a hermitian normalized density matrix that takes the form $\hat{\rho} = \sum_i p_i|\phi_i\rangle\langle\phi_i|$ with $tr(\hat{\rho}) = 1$. This is a general expression to describe a quantum mechanical system. Intuitively, $\hat{\rho}$ represents the uncertainty about the preparation of the system [10].
The measure of this uncertainty, and thus for the information of a quantum system, which is described by $\hat{\rho}$, is the von Neumann entropy

$$S_N(\hat{\rho}) \coloneqq -tr(\hat{\rho}\ln(\hat{\rho})). \tag{2.80}$$

It is non-negative, zero iff the quantum state is pure, and maximal for a completely mixed state. Hence, it can also be regarded as a measure of the purity of a state. The von Neumann

entropy (2.80) has the same structure as (2.79), i.e., the expectation value of $\ln(p(x_i))$, only in the language of quantum mechanics. Therefore, it is considered to be the quantum mechanical counterpart of the Shannon entropy.

Since in quantum mechanics two different states are not necessarily completely distinguishable, we want to take a closer look at the concept of distinguishability. We, therefore, consider the state $\hat{\rho}$ of a quantum system D with associated Hibert space $\mathcal{H}$ and the state $\hat{q}$ of another quantum system P with associated Hilbert space $\mathcal{H}$. The measure of the distinguishability between two states $\hat{\rho}$ and $\hat{q}$ is the relative quantum entropy

$$D(\hat{\rho}||\hat{q}) \coloneqq tr(\hat{\rho}(\ln(\hat{\rho})) - \ln(\hat{q})), \tag{2.81}$$

which is ≥ 0 with equality iff $\hat{\rho} = \hat{q}$. This measure tells us how difficult it is to distinguish the state $\hat{\rho}$ from the state $\hat{q}$. More precisely, the quantum relative entropy provides a notion of distance on the set of quantum states, although it does not constitute a metric since it is not symmetric and does not satisfy the triangle inequality [6]. Nevertheless, it constitutes a pre-metric on the set of quantum states. We see the similarity to the von Neumann entropy and can even define (2.80) via the quantum relative entropy iff $\hat{\rho}$ and $\hat{q}$ commute

$$S_N(\hat{\rho}) \coloneqq -D(\hat{\rho}||\mathbb{1}) = -tr(\hat{\rho}\ln(\hat{\rho})). \tag{2.82}$$

To compare two probability distributions, we take inspiration from (2.81). Essentially, Essentially, we convert the relative quantum entropy into a classical analogue, which was first introduced by Kullback and Leibler. It tells us how difficult it is to distinguish the probability distribution $p(x)$ from the probability distribution $q(x)$. In other words, it is a measure of the difference in the information content of $q(x)$ relative to $p(x)$. To understand this intuitively, we consider a set of events x_i with probability distribution $p(x)$. In reality, however, we usually do not know the true distributions of the set of events and therefore must assume a probability distribution $q(x)$. For example, if we flip a coin n times and believe it to be fair, then we expect to get tails as often as heads. In reality, the coin is not fair because of its uneven weight distribution. Consequently, our expectation is incorrect and there will be a discrepancy between our expected probability distribution $q(x)$ and the actual probability distribution $p(x)$. We formalize this discrepancy by weighting the surprise $-\ln(q(x_i))$ by the actual probability $p(x_i)$ of a particular event x_i occurring. The expected surprise according to this false assumption is then $-\sum_i p(x_i)\ln(q(x_i))$. The actual expected surprise is given by the Shannon entropy $S(p(x)) = -\sum_i p(x_i)\ln(p(x_i))$. It is straightforward to show that $S(p(x)) \leq -\sum_i p(x_i)\ln(q(x_i))$ with equality iff $p(x_i) = q(x_i)$ for all x_i. Consequently, this discrepancy that resulted from our incorrect assumption is the difference between the two expected surprises and is called the classical relative entropy

$$D(p(x)||q(x)) \coloneqq \sum_i^n p(x_i)\ln\left(\frac{p(x_i)}{q(x_i)}\right), \tag{2.83}$$

which is non-negative and zero only iff the two distributions are identical. Since we can interpret the relative entropy as a measure of the distance between probability distributions according to [6], it is intuitively clear why (2.83) is not negative. However, [6] has shown that negativity can arise when both distributions are not normalized to the same measure. Additionally, negativity can arise when we do not have probability distributions, i.e., $\sum_i q(x_i) \neq 1$. Note that this measure of distance between $p(x)$ and $q(x)$ is not a metric since it does not satisfy the triangle inequality and is not symmetric $D(p(a)||p(b)) \neq D(p(b)||p(a))$ [6]. To intuitively understand this asymmetry, we consider a coin example. Suppose we have a coin that is fair and a coin that is completely unfair, e.g., we always get "heads". If we flip a coin and receive "tails", we assume we have the fair coin. If we flip the coin and receive "heads", it can be either of the two coins. In the case of the fair coin, if we flip it several times, we will eventually get "tails". In the case of the completely unfair coin, even after millions of "heads", we can never rule out it being the fair coin, since the outcome of "tails" is still statistically possible. So, how sure we are about which coin we are holding depends on which of the two coins we are holding and how different it is from the other.

2.1.3.2 Relative entropy of a multilayer perceptron

We have seen that the relative entropy is used in quantum information theory and classical information theory to provide a notion of distance on the set of quantum states or probability distributions. We will now leverage it to introduce a measure that can quantify the propagation of information through a deep hierarchical neural network. The following derivation has already been done by [6]. I performed it again myself to verify that I get the same final equation for the relative entropy. The step-by-step derivation can be found in A.3.

We assume that the network under consideration is a random neural network, which means that its weights and biases are independent and identically distributed (i.i.d). For a large layer width, this allows us to consider the network in the mean field limit [6]. Together with the mean field formalism of [16], which was developed for wide, hierarchical feedforward neural networks, we can then describe the pre-activation of each layer by a Gaussian characterized by its variance. This simplifies the theory of signal propagation through random networks to the extent that we can obtain an explicit expression for relative entropy as a function of depth, as we will see in the following.
In this sense, we first consider the normal distributions p and q, which describe the different layers of the neural network in the mean field limit

$$p(x) = \frac{1}{\sqrt{2\pi\sigma_p^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_p}{\sigma_p}\right)^2} \text{ and } q(x) = \frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2}, \tag{2.84}$$

Here x is a continuous random variable. Note that $p(x)$ and $q(x)$ potentially have different means and standard deviations. Using these Gaussian distributions leads to the continuous

form of the relative entropy

$$D(p||q) = \int dx\, p(x) \ln\left(\frac{p(x)}{q(x)}\right), \tag{2.85}$$

which can be rewritten into

$$D(p||q) = \int dx\, p(x) \ln\left(\frac{p(x)}{q(x)}\right) = \langle \ln(p(x))\rangle_p - \langle \ln(q(x))\rangle_p, \tag{2.86}$$

via the definition of the expectation value. The definition of the expectation value also allows us to rewrite the classical Shannon entropy into

$$S(p) = -\langle \ln(p(x))\rangle_p, \tag{2.87}$$

which we substitute into (2.86) so that we can write

$$D(p||q) = -S(p(x)) - \langle \ln(q(x))\rangle_p. \tag{2.88}$$

We simplify the term $-S(p(x))$ by plugging in the probability density functions $p(x)$ and $q(x)$ (see A.3). Hence,

$$-S(p(x)) = \langle \ln(p(x))\rangle_p = -\frac{1}{2}\ln(2\pi\sigma_p^2 \mathrm{e}). \tag{2.89}$$

We also rewrite the second term in (2.88) (see A.3). Consequently,

$$\langle \ln(q(x))\rangle_p = -\frac{1}{2\sigma_q^2}(\sigma_p^2 + (\mu_p - \mu_q)^2) - \frac{1}{2}\ln(2\pi\sigma_q^2). \tag{2.90}$$

Then substituting (2.89) and (2.90) into (2.88), we obtain the continuous relative entropy for Gaussian distributions

$$D(p||q) = -\frac{1}{2}\ln(2\pi \mathrm{e}\sigma_p^2) + \frac{1}{2\sigma_q^2}(\sigma_p^2 + (\mu_p - \mu_q)^2) + \frac{1}{2}\ln(2\pi\sigma_q^2), \tag{2.91}$$

where $D(p||q) = 0$ if $q = p$ since $\mu_p = \mu_q$.

We now introduce the multidimensional relative entropy and therefore consider the multivariate Gaussian probability distributions

$$p(\vec{x}) = \frac{1}{\sqrt{(2\pi)^N |\Sigma_p|}} \mathrm{e}^{-\frac{1}{2}(\vec{x}-\vec{\mu}_p)^T \Sigma_p^{-1} (\vec{x}-\vec{\mu}_p)} \text{ and } q(\vec{x}) = \frac{1}{\sqrt{(2\pi)^N |\Sigma_q|}} \mathrm{e}^{-\frac{1}{2}(\vec{x}-\vec{\mu}_q)^T \Sigma_q^{-1} (\vec{x}-\vec{\mu}_q)}, \tag{2.92}$$

with N-dimensional continuous random variable $\vec{x} = (x_1, ..., x_N)^T$, N-dimensional mean vector $\vec{\mu}_{p,q} = (\mu_1, ..., \mu_N)^T$, and $N \times N$ covariance matrix $\Sigma_{p,q}$. Since we consider a random network the covariance matrix is diagonal. Consequently, we obtain the following determinant

$$|\Sigma_p| = \sigma_{p,q}^N. \tag{2.93}$$

Substituting these probability density functions into the relative entropy (see A.3), we get

$$D(p||q) = \int d\vec{x}\, p(\vec{x}) \ln\left(\frac{p(\vec{x})}{q(\vec{x})}\right) = \langle \ln(p(\vec{x}))\rangle_p - \langle \ln(q(\vec{x}))\rangle_p. \tag{2.94}$$

Again, we express the first term on the right-hand side of the equation by the Shannon entropy

$$D(p||q) = -S(p(\vec{x})) - \langle \ln(q(\vec{x}))\rangle_p. \tag{2.95}$$

Applying the trace trick and thinking of a scalar as a 1×1 matrix A, we can say $x^T Ax = tr(x^T Ax) = tr(xx^T A) = tr(Axx^T)$. Consequently,

$$S(p(\vec{x})) = \frac{1}{2}\ln(|2\pi e \Sigma_p|). \tag{2.96}$$

Since the covariance matrix is diagonal, the second term on the right side of the equation (2.94) can be rewritten using the definition of variance (see A.3) into

$$\langle \ln(q(x))\rangle_p = -\frac{1}{2} tr(\Sigma_q^{-1}[\Sigma_p + (\mu_p - \mu_q)^2]) - \frac{1}{2}\ln(|2\pi\Sigma_q|). \tag{2.97}$$

Substituting (2.96) and (2.97) into (2.95) yields the relative entropy for the multivariate case

$$D(p(\vec{x})||q(\vec{x})) = -\frac{1}{2}\ln(|2\pi e\Sigma_p|) + \frac{1}{2} tr(\Sigma_q^{-1}[\Sigma_p + (\mu_p - \mu_q)^2]) + \frac{1}{2}\ln(|2\pi\Sigma_q|). \tag{2.98}$$

We now extend the relative entropy to a hierarchical network in which the reference distribution $p(z)$ describes the distribution of pre-activations in the first layer (first forward pass) and the target distribution $q(z)$ describes the distribution of pre-activations in an arbitrary subsequent layer.[15] Note that z is a multivariate continuous random variable because it consists of the pre-activations of all neurons within a specific layer, and each of these pre-activations is itself a continuous random variable. The relative entropy of a hierarchical network essentially calculates the relative entropy as a function of depth, where the reference distribution is fixed and the target distribution moves through all subsequent hidden layers. It is defined as

$$D(p(z^{[l]})||q(z^{[l+m]})) = \langle \ln(p(z^{[l]}))\rangle_p - \langle \ln(q(z^{[l+m]}))\rangle_p, \text{ where } m \in [0, N^{[l-1]} - 1]. \tag{2.99}$$

When $m = 0$ we recover $D(p(z^{[l]})||q(z^{[l+m]})) = 0$ since $p = q$, while at $m = N^{[l-1]} - 1$, we get the relative entropy between the first layer (first forward pass) and the last layer (last forward pass) for a network with $N^{[l-1]}$ layers. Considering only the case $m = 1$, we see that $z_{i^{[l+1]}}^{[l+1]}$ is a recursive function of $z_{i^{[l]}}^{[l]}$ given by equation (2.9). For clarity, we will drop the superscripts and subscripts. Thus,

$$z^{[l+1]} = w^{[l+1]}\sigma_T(z^{[l]}) + b^{[l+1]}, \tag{2.100}$$

[15] The reason for using the first forward pass and not the input image is explained in chapter 2.1.4.

where $w^{[l+1]}$ is a $N^{[l+1]} \times N^{[l]}$ matrix, $\sigma_T(z^{[l]})$ is a $N^{[l]}$-dimensional vector, and $b^{[l+1]}$ is a $N^{[l+1]}$-dimensional vector.
For $m = 1$ we get

$$D(p(z^{[l]})||q(z^{[l+1]})) = \langle \ln(p(z^{[l]}))\rangle_p - \langle \ln(q(z^{[l+1]}))\rangle_p \tag{2.101}$$

with

$$p(z^{[l]}) = \frac{1}{\sqrt{(2\pi)^{N^{[l]}}|\Sigma_p|}} \mathrm{e}^{-\frac{1}{2}(z^{[l]}-\mu_q^{[l]})^T \Sigma_p^{-1}(z^{[l]}-\mu_p^{[l]})}, \tag{2.102}$$

and

$$q(z^{[l+1]}) = \frac{1}{\sqrt{(2\pi)^{N^{[l+1]}}|\Sigma_q|}} \mathrm{e}^{-\frac{1}{2}(z^{[l+1]}-\mu_q^{[l+1]})^T \Sigma_q^{-1}(z^{[l+1]}-\mu_q^{[l+1]})}. \tag{2.103}$$

Proceeding analogously to the multidimensional relative entropy, we obtain

$$\langle \ln(p(z^{[l]}))\rangle_p = -\frac{1}{2}\ln(|2\pi \mathrm{e}\Sigma_p^{[l]}|), \tag{2.104}$$

and

$$\langle \ln(q(z^{[l+1]}))\rangle_p = -\frac{1}{2}\langle (z^{[l+1]} - \mu_q^{[l+1]})^T \Sigma_q^{-1}(z^{[l+1]} - \mu_q^{[l+1]})\rangle_p - \frac{1}{2}\ln(|2\pi\Sigma_q^{[l+1]}|). \tag{2.105}$$

The first term on the right-hand side of the equation (2.105) can again be simplified using the summation notation and the definition of the variance

$$\begin{aligned} &\langle (z^{[l+1]} - \mu_q^{[l+1]})^T (\Sigma_q^{[l+1]})^{-1}(z^{[l+1]} - \mu_q^{[l+1]})\rangle_p \\ &= \sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii} \langle (z_i^{[l+1]} - \mu_{q,i}^{[l+1]})^2\rangle_p \\ &= \sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii} (\langle (z_i^{[l+1]})^2\rangle_p - 2\langle z_i^{[l+1]}\rangle_p \mu_{q,i}^{[l+1]} + (\mu_{q,i}^{[l+1]})^2). \end{aligned} \tag{2.106}$$

We see that we have two recursive terms inside this equation. Therefore, we try to find a useful recursion relation

$$\begin{aligned} &\langle (z_i^{[l+1]})^2\rangle_p \\ =&\langle \sum_{j,k} w_{i,j}^{[l+1]} w_{i,k}^{[l+1]} \sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]})\rangle_p + 2\sum_j w_{i,j}^{[l+1]} \langle \sigma_T(z_j^{[l]})\rangle_p b_i^{[l+1]} + \langle (b_i^{[l+1]})^2\rangle_p \\ =&\sum_{j,k} w_{i,j}^{[l+1]} w_{i,k}^{[l+1]} \langle \sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]})\rangle_p + 2\sum_j w_{i,j}^{[l+1]} \langle \sigma_T(z_j^{[l]})\rangle_p b_i^{[l+1]} + (b_i^{[l+1]})^2, \end{aligned} \tag{2.107}$$

where we define

$$\langle \sigma_T(z_j^{[l]}) \rangle_p = \frac{1}{\sqrt{2\pi(\sigma_{p,j}^{[l]})^2}} \int dz_j^{[l]} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,j}^{[l]}}{\sigma_{p,j}^{[l]}}\right)^2} \sigma_T(z_j^{[l]}) := f_j^{[0]}$$

$$\langle \sigma_T(z_j^{[l]}) \sigma_T(z_k^{[l]}) \rangle_p = \frac{1}{2\pi\sigma_{p,j}^{[l]}\sigma_{p,k}^{[l]}} \int dz_j^{[l]} dz_k^{[l]} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,j}^{[l]}}{\sigma_{p,j}^{[l]}}\right)} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,k}^{[l]}}{\sigma_{p,k}^{[l]}}\right)} \sigma_T(z_j^{[l]}) \sigma_T(z_k^{[l]}) := f_{j,k}^{[0]}. \tag{2.108}$$

These two integrals have to be solved numerically. Using the recursion relation (2.107) together with the integrals (2.108), we now can rewrite (2.106) into

$$\begin{aligned} \sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii} [\sum_{j,k} w_{i,j}^{[l+1]} w_{i,k}^{[l+1]} f_{j,k}^0 + 2 \sum_j w_{i,j}^{[l+1]} (b_i^{[l+1]} - \mu_{q,i}^{[l+1]}) f_j^0 \\ + (b_i^{[l+1]} - \mu_{q,i}^{[l+1]})^2] := \langle Q^{[l+1]} \rangle_p, \end{aligned} \tag{2.109}$$

which we will call the Q-term. Substituting (2.109) into (2.105) and then (2.105) together with (2.104) into (2.101), we obtain

$$D(p(z^{[l]}) || q(z^{[l+1]})) = -\frac{1}{2} \ln(|2\pi \mathrm{e} \Sigma_p^{[l]}|) + \frac{1}{2} \ln(|2\pi \Sigma_q^{[l+1]}|) + \frac{1}{2} \langle Q^{[l+1]} \rangle_p. \tag{2.110}$$

For an arbitrary m, (2.109) becomes

$$\begin{aligned} \sum_{i=1}^{N^{l+m}} [(\Sigma_q^{[l+m]})^{-1}]_{ii} [\sum_{j,k} w_{i,j}^{[l+m]} w_{i,k}^{[l+m]} f_{j,k}^{[m-1]} + 2 \sum_j w_{i,j}^{[l+m]} (b_i^{[l+m]} - \mu_{q,i}^{[l+m]}) f_j^{[m-1]} \\ + (b_i^{[l+m]} - \mu_{q,i}^{[l+m]})^2] := \langle Q^{[l+m]} \rangle_p, \end{aligned} \tag{2.111}$$

with

$$\begin{aligned} f_j^{[m-1]} :=& \langle \sigma_T(z_j^{[l+m-1]}) \rangle_p \\ =& \frac{1}{\sqrt{|2\pi\Sigma_p^{[l]}|}} \int dz^{[l]} \mathrm{e}^{-\frac{1}{2}(z^{[l]}-\mu_p^{[l]})^T (\Sigma_p^{[l]})^{-1} (z^{[l]}-\mu_p^{[l]})} \sigma_T(z_j^{[l+m-1]}), \end{aligned} \tag{2.112}$$

and

$$\begin{aligned} f_{j,k}^{[m-1]} :=& \langle \sigma_T(z_j^{[l+m-1]}) \sigma_T(z_k^{[l+m-1]}) \rangle_p \\ =& \frac{1}{\sqrt{|2\pi\Sigma_p^{[l]}|}} \int dz^{[l]} \mathrm{e}^{-\frac{1}{2}(z^{[l]}-\mu_p^{[l]})^T (\Sigma_p^{[l]})^{-1} (z^{[l]}-\mu_p^{[l]})} \sigma_T(z_j^{[l+m-1]}) \sigma_T(z_k^{[l+m-1]}), \end{aligned} \tag{2.113}$$

where $dz^{[l]} := \prod_r dz_r^{[l]}$ such that we integrate over all vector components. We can further simplify by using

$$\sum_{i=1}^{N^{[l+m]}} [(\Sigma_q^{[l+m]})^{-1}]_{ii} = \frac{1}{\sigma_q^2}. \tag{2.114}$$

Hence, (2.111) can be rewritten into

$$\frac{1}{\sigma_q^2}\left(\sum_{j,k}^{N^{[m-1]}} f_{j,k}^{[m-1]} \sum_{i}^{N^{[m]}} w_{i,j}^{[m]} w_{i,k}^{[m]} + 2\sum_{j}^{N^{[m-1]}} f_j^{[m-1]} \sum_{i}^{N^{[m]}} w_{i,j}^{[m]}(b_i^{[m]} - \mu_{q,i}^{[m]}) + \sum_{i}^{N^{[m]}}(b_i^{[m]} - \mu_{q,i}^{[m]})^2\right) := \langle Q^{[l+m]}\rangle_p, \tag{2.115}$$

and subsequently into

$$\sigma_q^2\langle Q^{[l+m]}\rangle_p = \sum_{j,k}^{N^{[m-1]}} f_{j,k}^{[m-1]} \sum_{i}^{N^{[m]}} w_{i,j}^{[m]} w_{i,k}^{[m]} + 2\sum_{j}^{N^{[m-1]}} f_j^{[m-1]} \sum_{i}^{N^{[m]}} w_{i,j}^{[m]}(b_i^{[m]} - \mu_{q,i}^{[m]}) + \sum_{i}^{N^{[m]}}(b_i^{[m]} - \mu_{q,i}^{[m]})^2. \tag{2.116}$$

Substituting this Q-term for the one in (2.110), and considering all the m layers as possible target layers q, we get the final form of the relative entropy of a hierarchical network

$$D(p(z^{[l]})||q(z^{[l+m]})) = -\frac{1}{2}\ln(|2\pi e\Sigma_p^{[l]}|) + \frac{1}{2}\ln(|2\pi\Sigma_q^{[l+m]}|) + \frac{1}{2}\sigma_q^2\langle Q^{[l+m]}\rangle_p. \tag{2.117}$$

By implementing it into a program, we next want to understand how information flows through deeper and deeper layers of a neural network. The Python implementation of the relative entropy of a hierarchical network can be found in A.4.7.

2.1.4 Results

I now use the program for the relative entropy of a hierarchical network (see A.4.7) to quantify the propagation of information through a multilayer perceptron.[16]
I mainly want to investigate how a change in depth affects the propagation of information through a multilayer perceptron. Therefore, I implement an ensemble of networks with different depths (see A.4.3). The architecture of these neural networks is inspired by the random feedforward network in [6]. They consist of an input layer with 784 neurons, several hidden layers with a constant width of 784 neurons, a second-to-last layer with 400 neurons, and an output layer with 10 neurons. The large layer width ensures that the layers remain approximately Gaussian and that I can work in the mean field limit (see chapter 2.1.3.2). This allows me to describe each layer by a Gaussian, which is entirely characterized by its variance. This is indispensable for deriving the formula for the relative entropy of a hierarchical network (2.117). Indeed, if the layer distribution does not resemble a Gaussian, our approach for measuring the relative entropy through a deep neural network as a function of depth breaks down.[17] This is also the reason why the reference distribution p describes the first forward pass[18] and not the input image since the grayscale values of the input image represent a handwritten number and are therefore not necessarily Gaussian distributed.
My results indicate that for measuring the propagation of information through a network via the relative entropy, it is indispensable that the network is well-trained. The worse the network is trained, the worse the classification accuracy, and the less the layer distribution resembles a Gaussian.[19] Eventually, our measuring technique breaks down.
Therefore, I first have to make sure that my network is well-trained before measuring the relative entropy. According to [20], the necessary condition for the trainability of a network is that information can propagate all the way through it. The network has this property if its depth does not exceed the correlation length ξ. This is because correlation length controls the signal propagation through the network and therefore indicates how deep a neural network can be trained. In my case, it depends on the choice of weight variance σ_w^2. [16], [17], and [20] have demonstrated that a random feedforward neural network exhibits a phase transition, i.e., a transition from order ($\sigma_w^2 \lesssim \sigma_{wC}^2$) to chaos ($\sigma_w^2 \gtrsim \sigma_{wC}^2$) as a function of σ_w^2 with non-trivial critical points. The critical point σ_{wC}^2 is characterized by a divergent correlation length, which allows information about the input data to propagate entirely through a network of arbitrary depth. Therefore, even networks of large depth

[16]The use of graphics processing units (GPUs) has been key to training and investigating the flow of information through networks with great depth (see A.4.2).

[17]Recall that the relative entropy essentially measures the difference in information content between a reference distribution p, which represents the first forward pass of the network, and a target distribution q, which represents an arbitrary subsequent forward pass. The formula for the relative entropy cannot account for a change in layer width since it assumes that both p and q are normalized with respect to the same measure. [6] says that a decrease in the layer width causes the relative entropy to change unphysically, e.g., to become negative. Consequently, I neglect the last two layers during the measurement.

[18]A forward pass consists of the pre-activations of all neurons within a given layer.

[19]I will explain the reason for this in chapter 2.1.4.1.

can be trained if σ_w^2 is initialized near the critical point. Thus, the goal is to initialize my networks near the critical point. Therefore, I make use of the phase diagram in [6]. Even though it visualizes the phase structure of a multilayer perceptron with 30 layers, a fixed bias variance of $\sigma_b^2 = 0.05$, that was trained for 15 epochs, I use this phase diagram as a guide for initializing σ_w^2 near criticality. The critical point is approximately at $\sigma_{wC}^2 \approx 1.76$ and therefore I compute the relative entropy for a network with $\sigma_w^2 \epsilon \{1.5, 1.75, 2.0, 2.25, 2.5\}$. It turns out that figure 12 is a proper guide for initializing σ_w^2, since my network can be trained well for my choices of σ_w^2 (see chapter 2.1.4.2 and chapter 2.1.4.3).

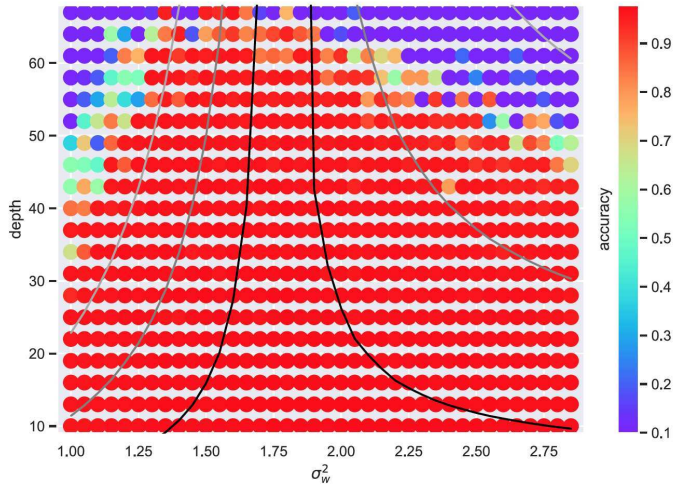

FIGURE 12: This figure is taken from [6]. It demonstrates the classification accuracy as a function of depth, weight variance, σ_w^2, and bias variance fixed at $\sigma_b^2 = 0.05$ from high accuracy (red) to low accuracy (colored). The critical point lies at $\sigma_w^2 \approx 1.76$. There the trainable depth is maximal which is why the network has a good classification accuracy even for large depths. It falls off the further I move into the ordered ($\sigma_w^2 \lesssim 1.76$) or chaotic phase ($\sigma_w^2 \gtrsim 1.76$).

2.1.4.1 The impact of training quality on the distribution of layers

Before investigating how depth affects the relative entropy, I investigate how training quality[20] affects the layer distribution. To do this, I consider a multilayer perceptron with a depth of 50 layers, a weight variance of $\sigma_w^2 = 2.5$, and a fixed bias variance of $\sigma_b^2 = 0.05$. I train this network for 15 epochs. To investigate the layer distribution I use the kernel density estimation (KDE). It approximates the probability density function of a layer by assigning a Gaussian kernel to each pre-activation and then superposing them.

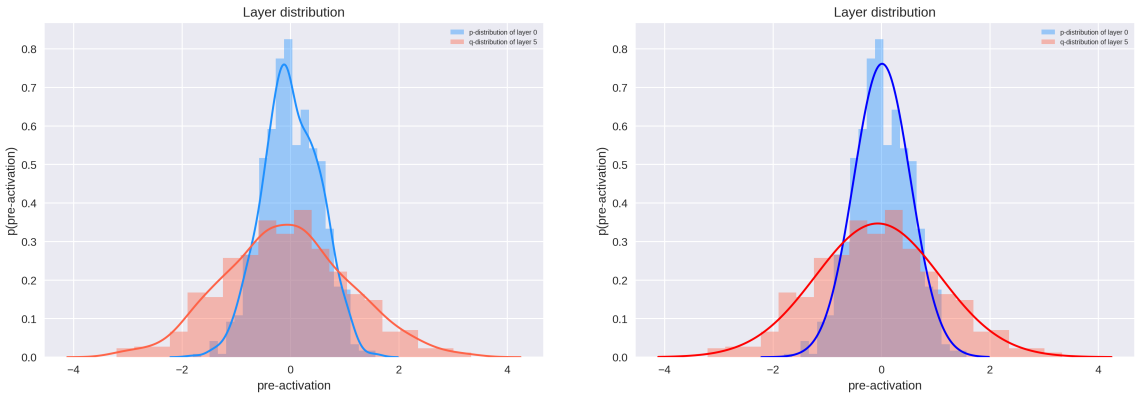

FIGURE 13: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 5th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 15 epochs.

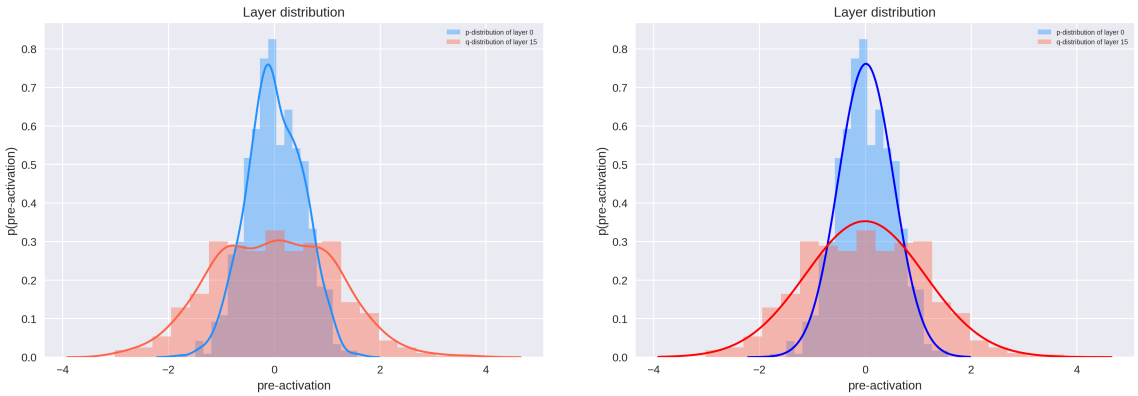

FIGURE 14: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 15th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 15 epochs.

[20] The higher the classification error, the lower the quality of the training.

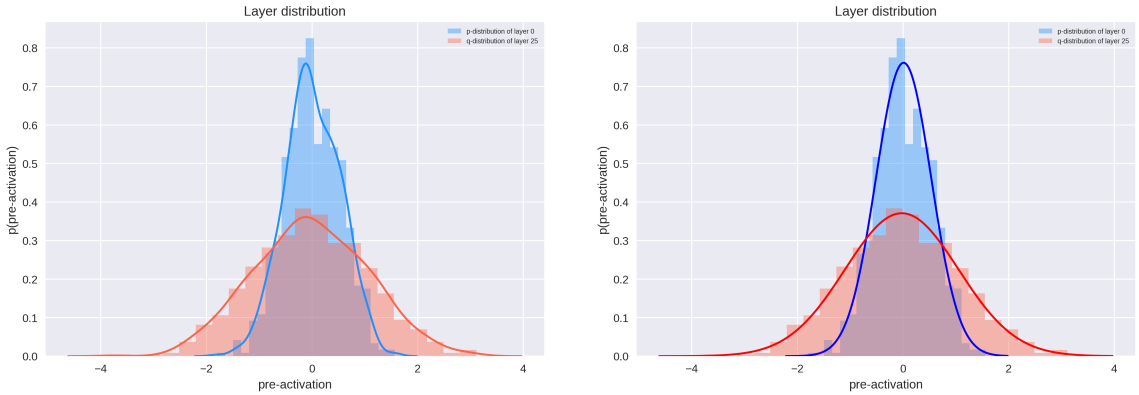

FIGURE 15: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 25th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 15 epochs.

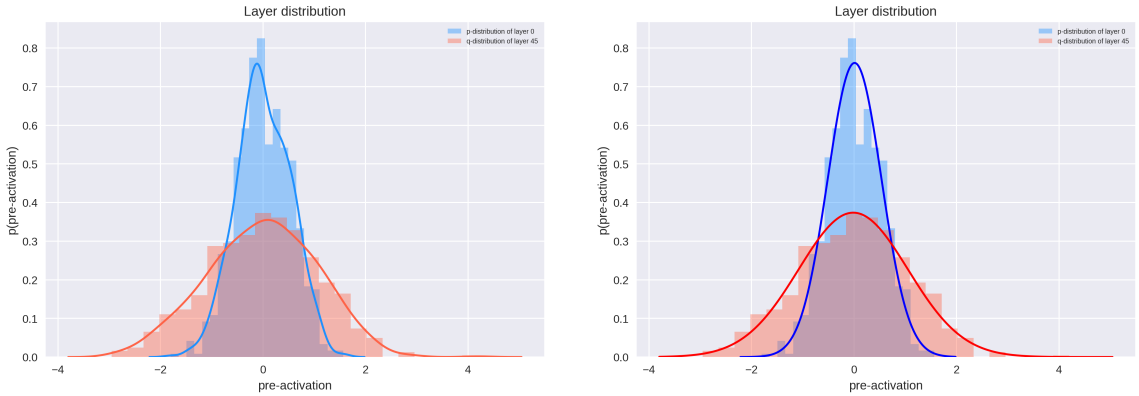

FIGURE 16: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 45th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 15 epochs.

After comparing the KDE fit of different layers with their corresponding normal distribution fit, I claim that the layer distributions can be considered approximately Gaussian in all cases. Consequently, the formula for the relative entropy (2.117) can measure the propagation of information through the network.
Now I want to see how the distribution of the layers changes when I change the training quality. Therefore, I look at the same network but train it for 50 epochs.

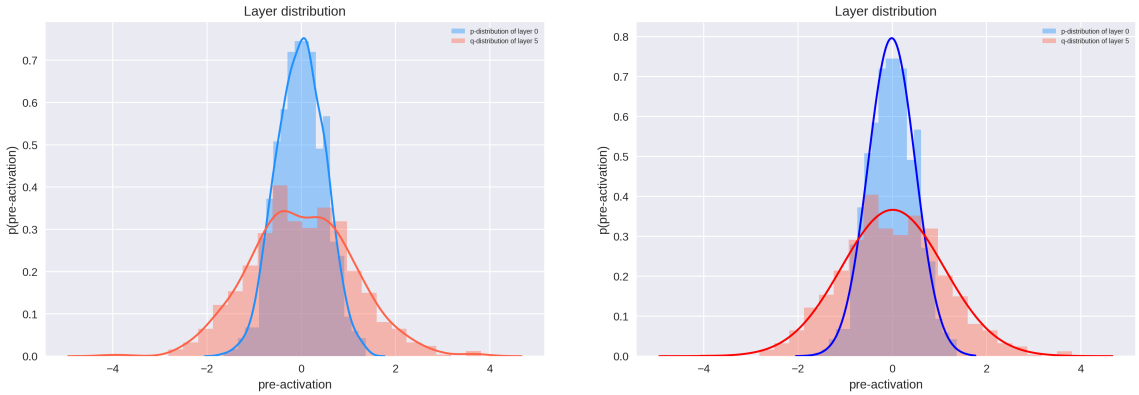

FIGURE 17: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 5th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 50 epochs.

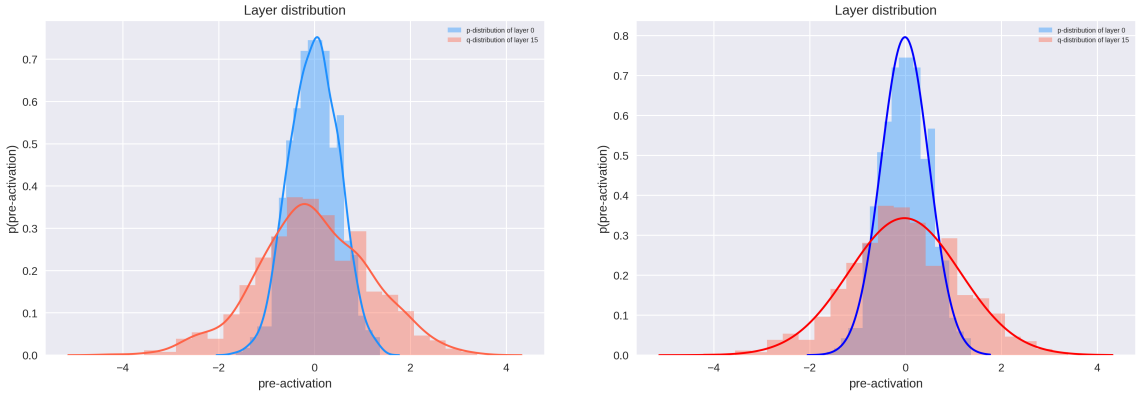

FIGURE 18: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 15th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 50 epochs.

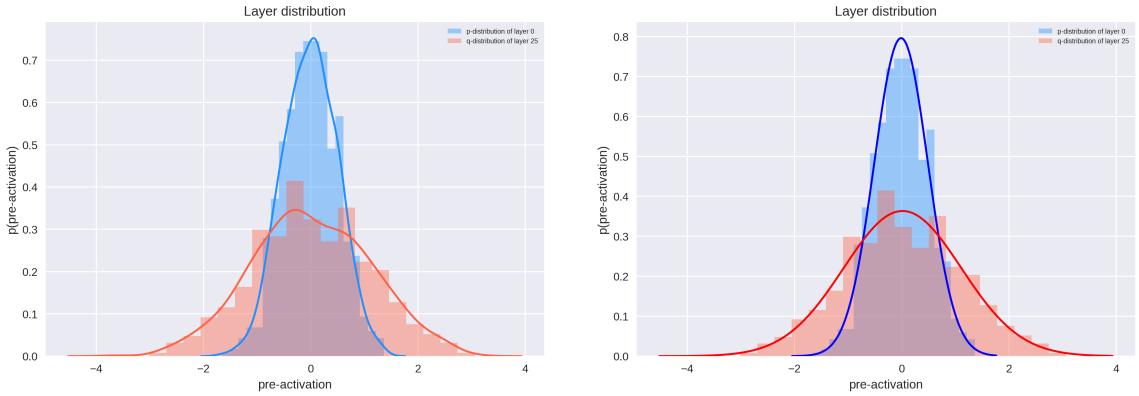

FIGURE 19: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 25th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 50 epochs.

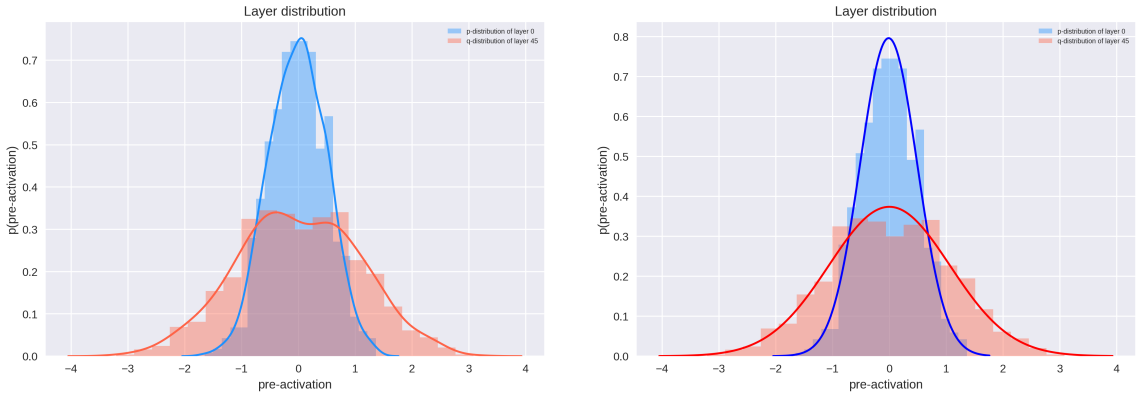

FIGURE 20: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 45th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 50 epochs.

Compared to the layer distributions for 15 epochs, e.g., figure 14 and figure 18, I notice that the layer distributions appear more Gaussian after 50 epochs. Thus, I claim that the more training the network undergoes, the more Gaussian the layers become. Chapter 2.1.4.4 shows that the worse the network is trained, the less the distribution of a layer resembles a Gaussian. For this reason, in my investigation of how depth affects the relative entropy as I make the neural network deeper and deeper, I increase the number of training epochs. By doing this, I want to ensure that the layers remain Gaussian, which is indispensable for using the formula for relative entropy.

2.1.4.2 The normalized relative Entropy and layer distribution of a 50-layer multilayer perceptron with respect to depth

To investigate how depth affects the relative entropy, I first consider a multilayer perceptron with a depth of 50 layers and a fixed bias variance of $\sigma_b^2 = 0.05$. I train this network for 15 epochs and observe a small classification error, so I assume that the network is well-trained and that the layers are Gaussian.[21] In figure 21, I visualize the relative entropy with respect to the network depth for the different σ_w^2.

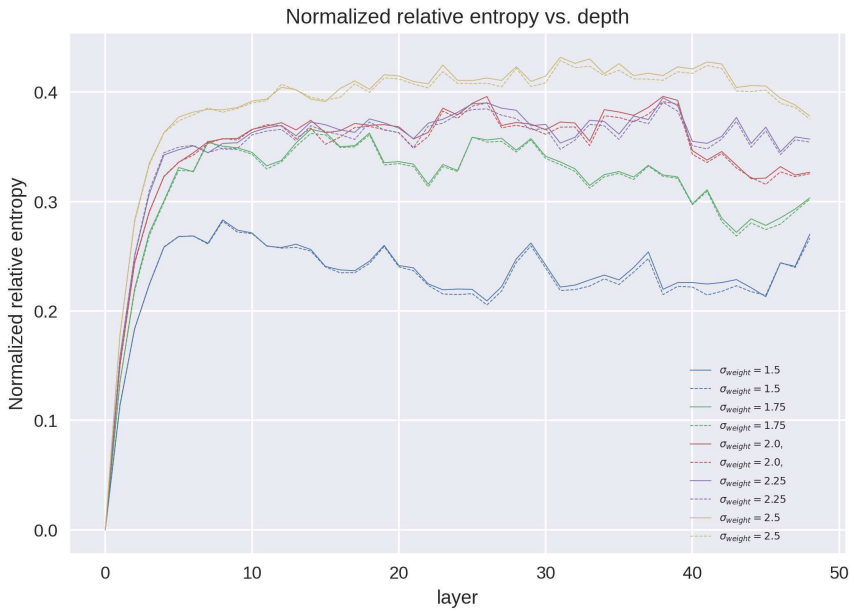

FIGURE 21: The normalized relative entropy before and after training for a multilayer perceptron with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, fixed depth of 50 layers, and a learning rate of 5e-5 after 15 epochs of training. The trajectory of the relative entropy increases steeply to an asymptote.

Figure 21 demonstrates that the relative entropy increases steeply and monotonically up to an asymptote that depends on σ_w^2. The larger the weight variance σ_w^2, the larger the value of the asymptote. According to [6], once the asymptote is reached, the information content, i.e. the quantity of information, ceases to evolve under subsequent layers. Furthermore, the shape of the relative entropy does not change significantly before and after training. I observe a decrease in the value of the asymptote post-training. This decrease is relatively small and seems to be approximately constant with respect to σ_w^2. Intuitively, the decrease of the relative entropy post-training vs. the relative entropy pre-training might be explained by a comparison with the renormalization group (RG). Both the neural network and the decimation RG involve a coarse-graining procedure in which irrelevant information is

[21] In 2.1.4.4, I observe that the worse the classification accuracy, the worse the network is trained, and the less a layer distribution resembles a Gaussian.

removed and only the information relevant to the classification problem is retained [6].[22] Therefore, as during RG, I expect information to be lost during Deep Learning. So, I expect the information content, i.e., the relative entropy, to decrease after training.[23]

2.1.4.3 The normalized relative entropy of a 100-layer multilayer perceptron with respect to depth

Next, I increase the depth of the network to 100 layers. Since increasing the depth makes training the network more difficult because more weights and biases need to be adjusted, I train the network for 50 epochs. By doing this, I want to ensure that the layers remain Gaussian (see 2.1.4.1). I observe a small classification error, so I assume that the network is well-trained and the layers are Gaussian. The resulting relative entropy for different σ_w^2 is visualized in (Figure 23).

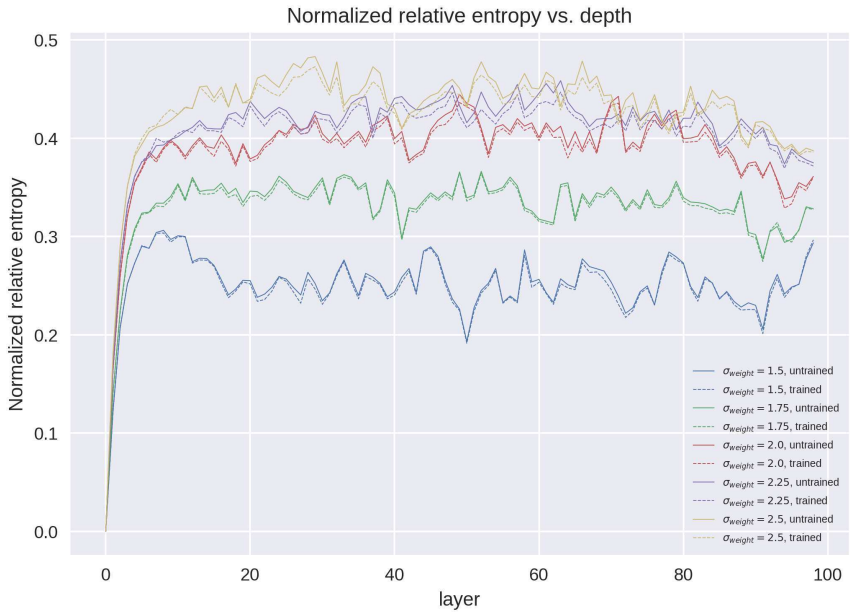

FIGURE 22: The normalized relative entropy before and after training for a multilayer perceptron with varying σ_w^2, fixed $\sigma_b^2 = 0.05$ and fixed depth of 100 layers after 50 epochs of training. The trajectory of the relative entropy increases steeply to an asymptote.

This network's relative entropy demonstrates qualitatively the same behavior as that of the one with 50 layers. As before, I observe a similar decrease in the value of the asymptote post-training.

[22] It is well known that information is lost during RG [6].
[23] I will discuss this further in chapter 2.3.

2.1.4.4 The normalized relative entropy of a 150-layer multilayer perceptron with respect to depth

The resulting relative entropy of different σ_w^2 for a network with a depth of 150 layers trained for 100 epochs is visualized in figure 23.

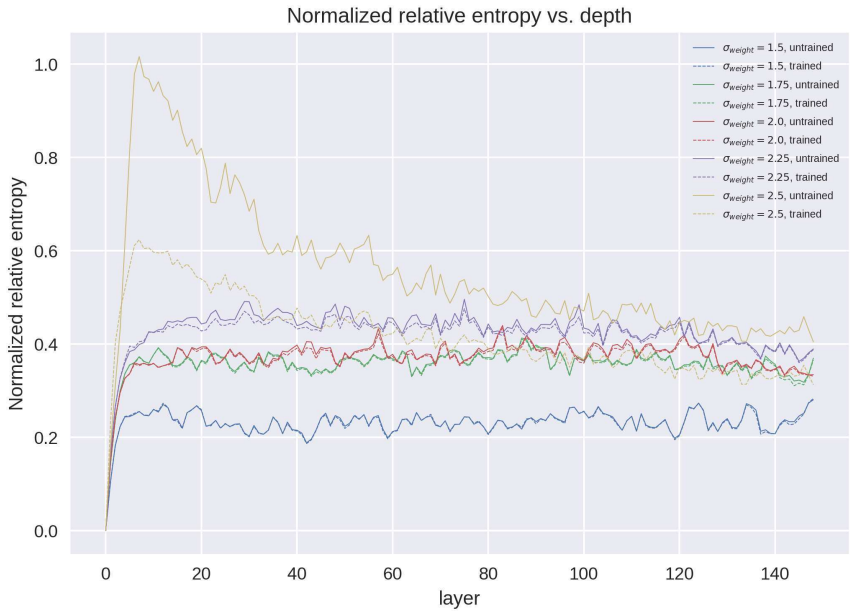

FIGURE 23: The normalized relative entropy before and after training for a multilayer perceptron with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, and fixed depth of 150 layers after 100 epochs of training. The trajectory of the relative entropy increases steeply to an asymptote. This is not the case for $\sigma_w^2 = 2.5$, since its relative entropy first jumps before it asymptotes

Again, the qualitative behavior is similar to the previous networks, except for $\sigma_w^2 = 2.5$. For $\sigma_w^2 = 2.5$, the relative entropy initially jumps before it asymptotes. Since the classification error is large for $\sigma_w^2 = 2.5$, I suspect that the network is no longer trainable, i.e., its depth has exceeded the correlation length ξ. This means that the information can no longer propagate through the entire network. To see how non-trainability affects the layer distribution, I directly analyze the different layers of this network.

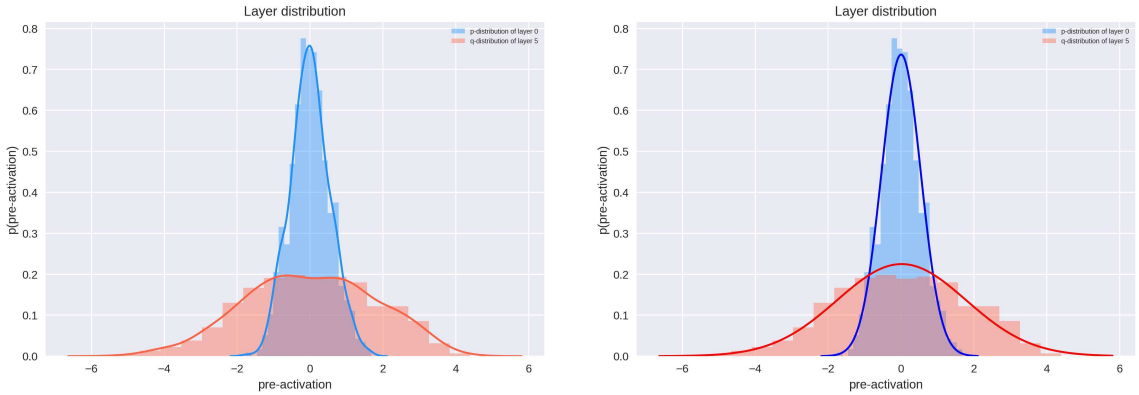

FIGURE 24: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 5th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

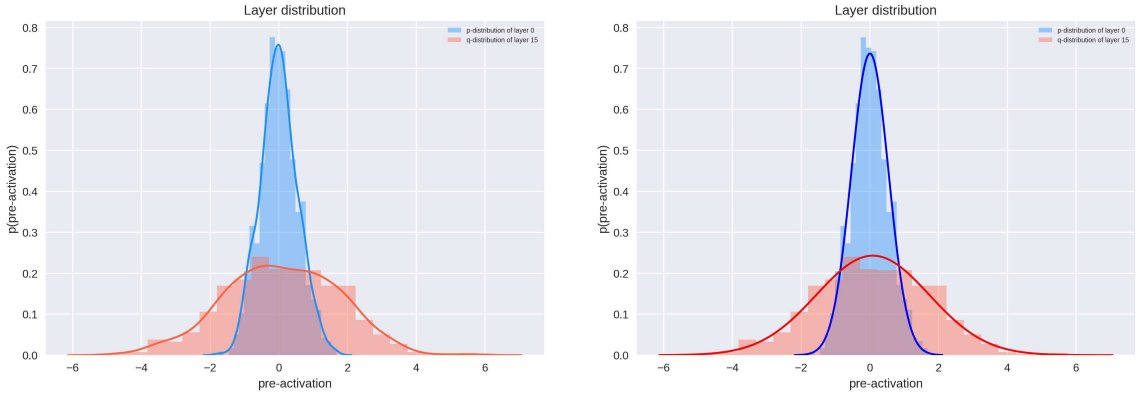

FIGURE 25: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 15th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

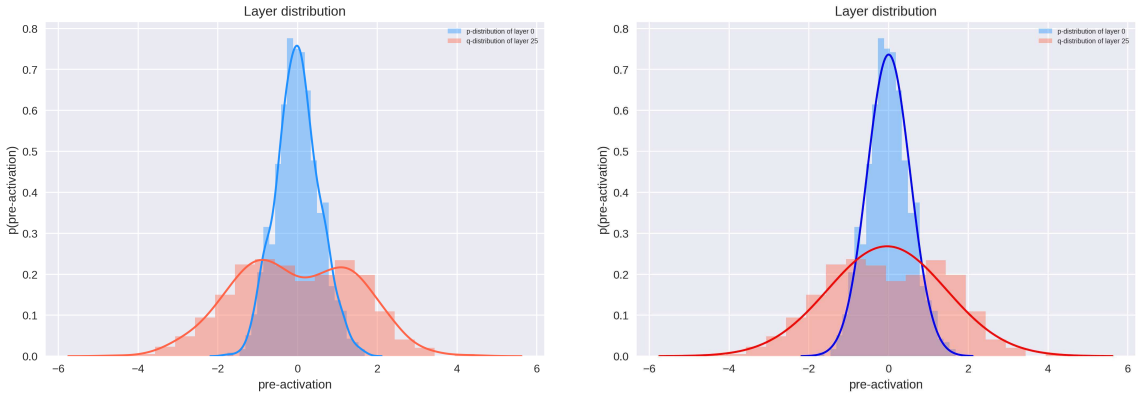

FIGURE 26: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 25th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

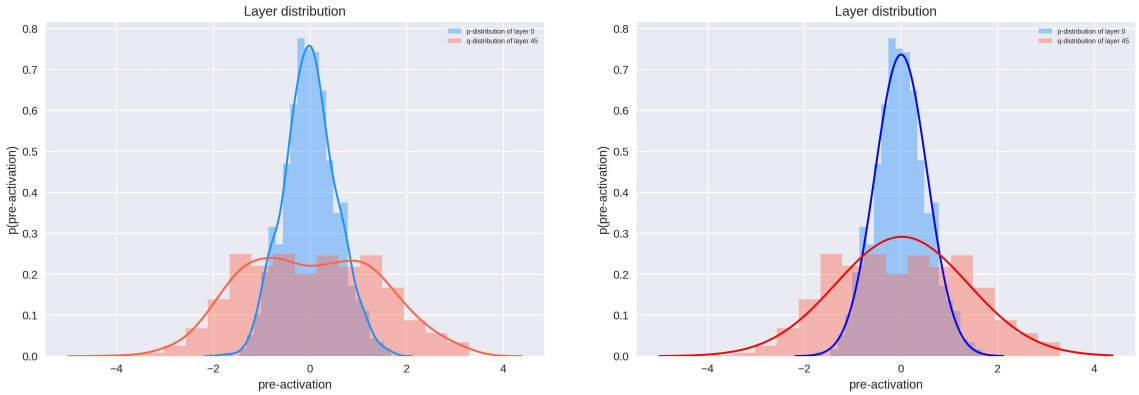

FIGURE 27: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 45th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

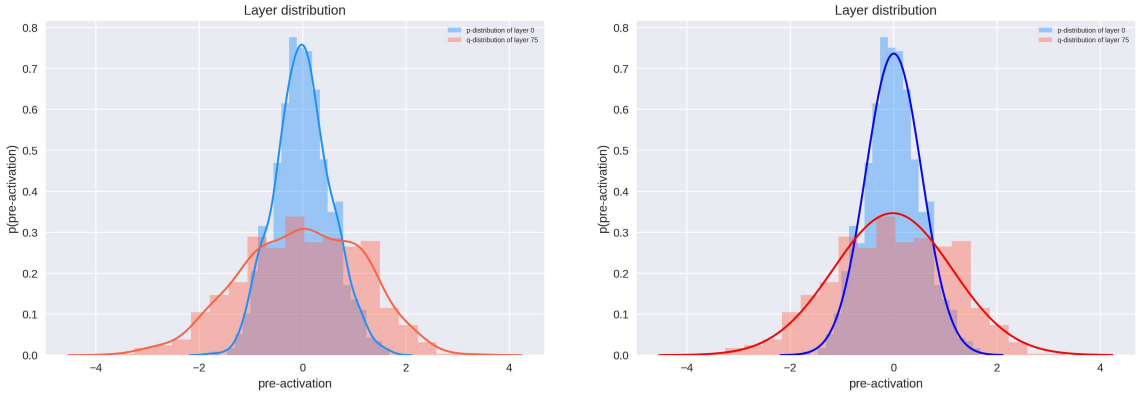

FIGURE 28: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 75th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

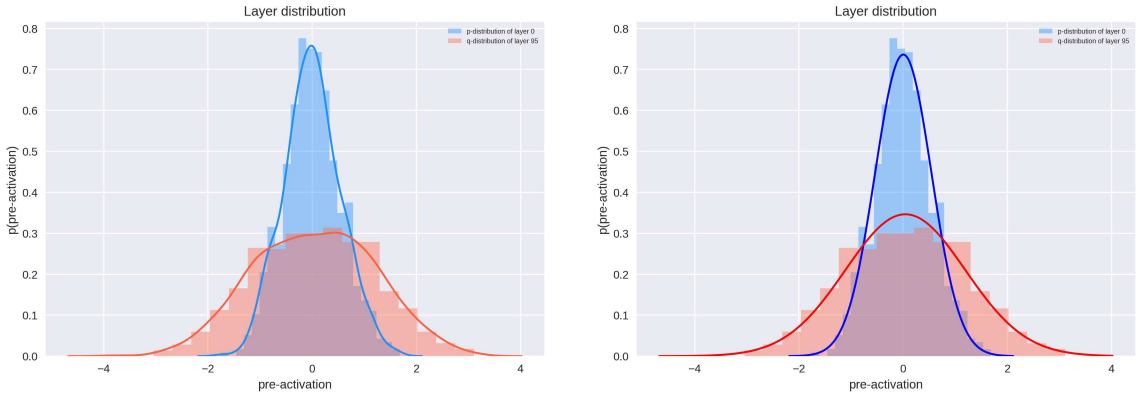

FIGURE 29: KDE fit (left) and normal distribution fit (right) for the layer distribution of the network's 95th layer. The network under consideration has a weight variance of $\sigma_w^2 = 2.5$ and was trained for 100 epochs.

Observing the distribution q of the different layers, I notice that in some cases, e.g., figure 26 and figure 27, the KDE fit looks like a superposition of multiple Gaussians.
Due to the behavior of the relative entropy and the large classification error I neglect the relative entropy for $\sigma_w^2 = 2.5$ when analyzing the propagation of information through a neural network.

2.1.4.5 Summary of results for a multilayer perceptron

In summary, the results indicate that untrainability goes hand in hand with the non-Gaussianity of the distributions of the layers. This untrainability manifests itself in a large classification error. The worse the network is trained, i.e., the larger the classification error, the less the layer distribution resembles a Gaussian. Eventually, our formula for relative entropy breaks down and I cannot interpret it physically.
The results also show that the further I move σ_w^2 away from the critical value, the faster the network becomes untrainable with increasing depth. The reason for this is that the network depth exceeds the correlation length ξ and thus information can no longer travel through the entire network.[24] So, the closer σ_w^2 is initialized to the critical value, the deeper I can train the neural network since ξ diverges at critical.
Moreover, these results demonstrate that the qualitative behavior of a physically meaningful relative entropy of a well-trained multilayer perceptron does not change with increasing depth; it continues to asymptote toward a value determined by σ_w^2.

[24]The depth to which a multilayer perceptron can be trained depends on its ability to pass information through the network, which is determined by the correlation length ξ.

2.2 Autoencoder

This chapter uses ideas from [14] and [3].

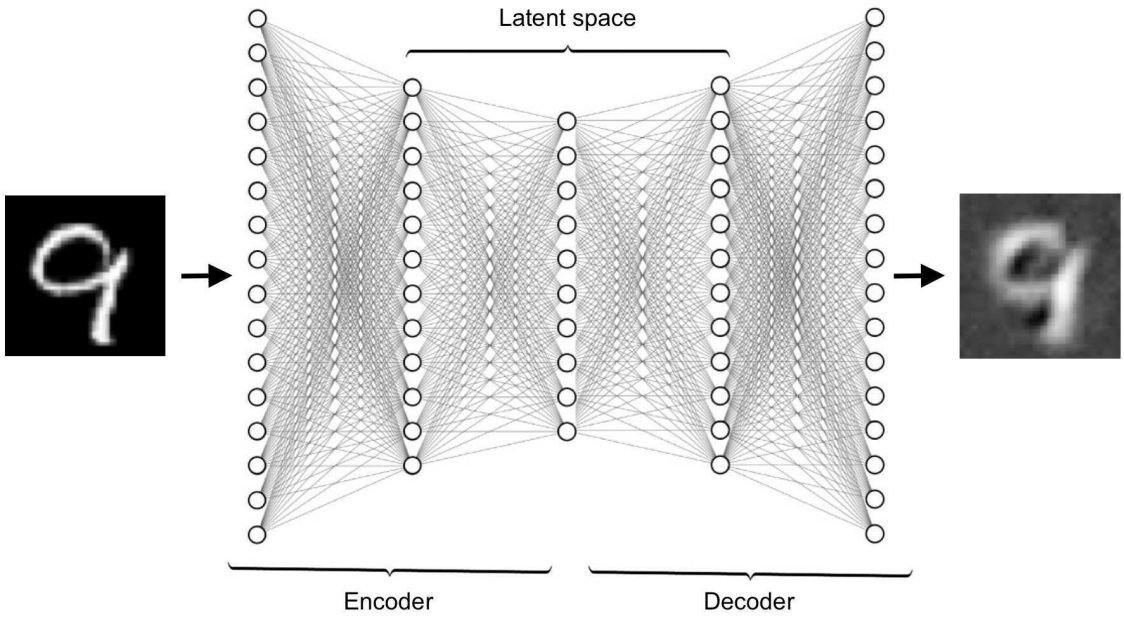

FIGURE 30: The architecture of an autoencoder that uses dimensionality reduction. The first half of the autoencoder constitutes the encoder and the second half constitutes the decoder. The hidden layers form the latent space and by reducing their dimension, the encoder projects the input images into a lower dimensional representation. The decoder then reconstructs the input images using this lower-dimensional representation.

2.2.1 Learning

In unsupervised learning, the neural network learns to exploit patterns in unlabeled data. It does this by modeling an internal representation of the data. A neural network can model an internal representation by imitating the input data. Unlike supervised learning, where we actively provide feedback to the network on its predictions by comparing them to the labels, in unsupervised learning we let the network figure out for itself whether it is capable of producing a good imitation. The unsupervised learning algorithm can do this by determining the discrepancy between the network-created imitation and the original data. This discrepancy indicates how well the internal representation can describe the complexity of the data. Subsequently, the discrepancy is used to correct the internal representation. This behavior is similar to the imitation behavior of a human child when it tries to learn something by itself.

2.2.2 Architecture

The autoencoder is a neural network used for unsupervised learning. It consists of two opposing multilayer perceptrons stacked on top of each other. Thus, it is a feedforward neural network with two symmetric parts, the encoder and the decoder (Figure 30). As the name implies, the encoder encodes input images, and the decoder decodes the encoded images, i.e., it reconstructs them. This is also why the input layer and the output layer have to be of the same width. The latent space, on the other hand, does not necessarily have to have the same dimension as the input and output layers.
In figure 30 the autoencoder uses a dimension reduction in the latent space of the encoder to project the input data into a lower-dimensional representation. This dimension reduction constrains the information that can traverse the network. We can intuitively understand dimension reduction by considering a three-dimensional space with random data points. In this case, the data has no structure, a data point cannot be described with less than three coordinates. However, if the data has structure, e.g. resembles a circle, then we can use polar coordinates to represent the data. Thus, we can use fewer than three coordinates to describe a data point without losing important information. Since our MNIST data has structure, the autoencoder can take advantage of it by fitting a nonlinear manifold to the data. This manifold depends on the input data and the parameters of the network (see chapter 2.1.2.1). It maps the data from a higher-dimensional input space to a lower-dimensional space. This manifold, which is the output of the encoder, is then used by the decoder for the reconstruction of the input images. If it is able to describe the complexity of the structure in the data in lower dimensions, the decoder yields a reconstruction that is close to the original image. In collaboration with the decoder, the manifold is forced to keep only the most important information of the input data.

Since our relative entropy (2.117) normalizes the target distribution q and the reference distribution p to the same measure, a change in the dimensionality of the layers distorts the measurement of the relative entropy [6]. Consequently, all layers of the autoencoder must have the same width. We refer to this type of autoencoder as a sparse autoencoder. Since we now no longer reduce the number of neurons, we must use a different constraint so that the autoencoder generates a lower-dimensional representation with only the most relevant information. For this, we take a closer look at the learning process and the cost function of the autoencoder.
As with the multilayer perceptron, we use a gradient descent algorithm to minimize the cost function's output, i.e., to minimize the reconstruction error. Since the sparse autoencoder wants to reconstruct the input image, the cost function is minimal when the output of the network is the input image. Without the dimensionality reduction, the sparse autoencoder will simply pass the input image from the input layer to the output layer. Therefore, it will not generate a lower-dimensional representation with the most important information of the input data. To prevent this, we introduce a sparsity constraint to our cost function. It forces

the autoencoder to selectively reduce the activity of certain neurons so that only the neurons that contain the most relevant information with respect to the reconstruction are considered [14]. As a result, the autoencoder learns a representation of the most important information in the input data. In other words, the autoencoder behaves like an information filter and we do not observe any overfitting of the data (see chapter 2.1.2.6). The sparsity constraint we resort to is the L2 regularization, which is the L2 norm of the weights. It is also referred to as the weight decay term because it is essentially a squared error penalty on the weights W. The cost function we use is the MSE cost function (2.13). But unlike the MSE cost function for a multilayer perceptron, the MSE cost function for an autoencoder does not compute the averaged differences between the predictions and the labels. It computes the differences between the original inputs and the reconstructions, averaged over the training samples. Consequently, we replace the labels in (2.13) with the feature vectors and add the weight decay term. This yields the cost function for the sparse autoencoder

$$J(W,B) = \frac{1}{m}\sum_{j=1}^{m}\frac{1}{N^{[L-1]}}\left\|\vec{x}^{(j)} - \hat{\vec{y}}^{(j)}\right\|^2 + \lambda\|W\|^2, \tag{2.118}$$

where λ is the weight decay parameter. This parameter controls the trade-off between minimizing the MSE cost function and the weight decay term. Minimizing the MSE cost function term in (2.118) essentially encourages our model to imitate the input data. Minimizing the weight decay term, on the other hand, discourages overfitting by driving the magnitude of the weights of certain neurons toward zero.
Using (2.118), the network can now compute the difference between the reconstructed images and the input images. This allows it to find out if its latent space has captured the most relevant information about the input data. If the reconstructed image is close to the original image, the network concludes that the latent space provides a representation of the most important information. If it is not, the difference between the reconstruction and the original input gets backpropagated so that the autoencoder can learn a better representation of the input data (see chapter 2.1.2.4).

All in all, given an unlabeled image of a handwritten number, the sparse autoencoder propagates the information through the latent space. In combination with (2.118), the network creates a lower-dimensional representation of the input image and uses it to reconstruct the image. If the representation does not retain the most relevant information about the input image, the reconstructed image deviates from the input image. This deviation is then minimized by backpropagation and gradient descent. In this way, the model learns to exploit the structure in the unlabeled data to find an efficient lower-dimensional representation that can reconstruct the input image.

2.2.3 Results MNIST

In this chapter, I use the relative entropy of a hierarchical network (2.117) to analyze the propagation of information through a sparse autoencoder. The autoencoder consists of an input layer with 784 neurons, 18 hidden layers with a constant width of 784 neurons, and an output layer with 784 neurons (see A.4.4). As with the multilayer perceptron, the large layer width ensures that the layer distribution is approximately Gaussian and that I can work in the mean field limit (see chapter 2.1.3.2). This allows me to describe each layer by a Gaussian and derive the formula for the relative entropy of a hierarchical network. The autoencoder essentially consists of two opposing multilayer perceptrons with constant layer width stacked on top of each other, which means I can use the relative entropy program (see A.4.7) without modifications.
Since the autoencoder reconstructs a handwritten number using the pixel values of the input image (Figure 31 and figure 32), and their distribution is not necessarily Gaussian, I neglect the last forward pass during the measurement.

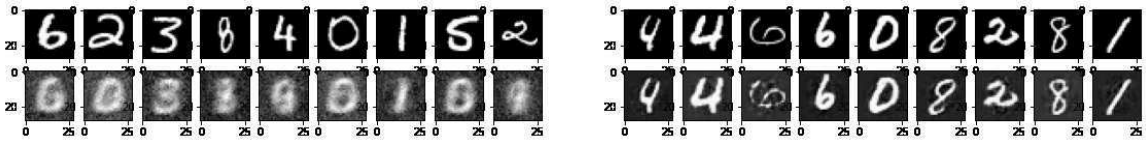

FIGURE 31: Reconstructed images of a network with a weight variance of $\sigma_w^2 = 1.5$ trained for 0 out of 1000 epochs (left) and 1000 out of 1000 epochs (right). The first row displays the input images and the second row the reconstructed images.

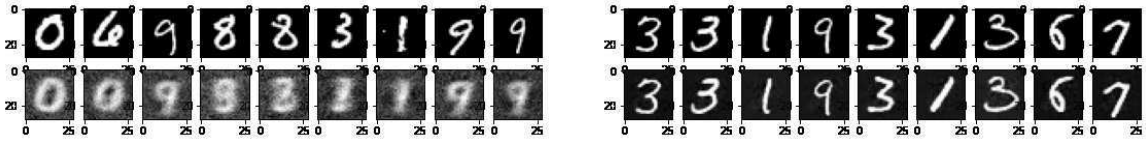

FIGURE 32: Reconstructed images of a network with a weight variance of $\sigma_w^2 = 1.75$ trained for 0 out of 1000 epochs (left) and 1000 out of 1000 epochs (right). The first row displays the input images and the second row the reconstructed images.

2.2.3.1 The normalized relative entropy of a sparse autoencoder trained on the MNIST data set

To analyze the propagation of information through a network using the relative entropy, I must first ensure that the network is well-trained so that the layers remain Gaussian after training. Only then does the relative entropy allow for a physical interpretation. According to [20], the correlation length diverges at an order-to-chaos transition regardless of many architectural decisions. Therefore, as with the multilayer perceptron, the closer the parameters of the network are initialized near critical, the better the network can be trained. To do this, I again use figure 12 as orientation and choose $\sigma_w^2 \epsilon \{1.5, 1.75\}$ as

weight variances. It turns out that this is an appropriate choice for an autoencoder that was trained for 1000 epochs in an unsupervised learning environment on unlabeled MNIST images. This is most likely because the sparse autoencoder consists of two multilayer perceptrons.
To determine how well the network is trained and how well the layers resemble a Gaussian, I cannot use a classification error as an indicator, as I did for the multilayer perceptron. I have to qualitatively determine whether the reconstruction process was good or bad using figure 31 and figure 32. Since this is inaccurate, I analyze the distribution of the layers.

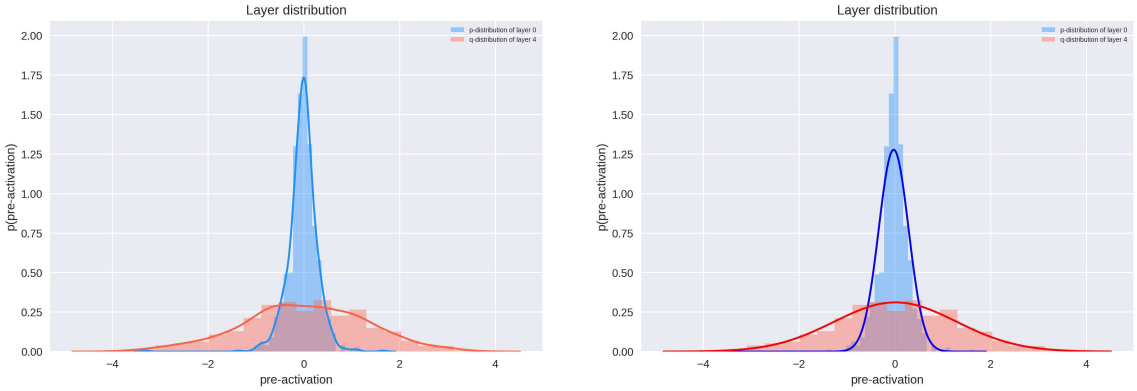

FIGURE 33: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 4th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

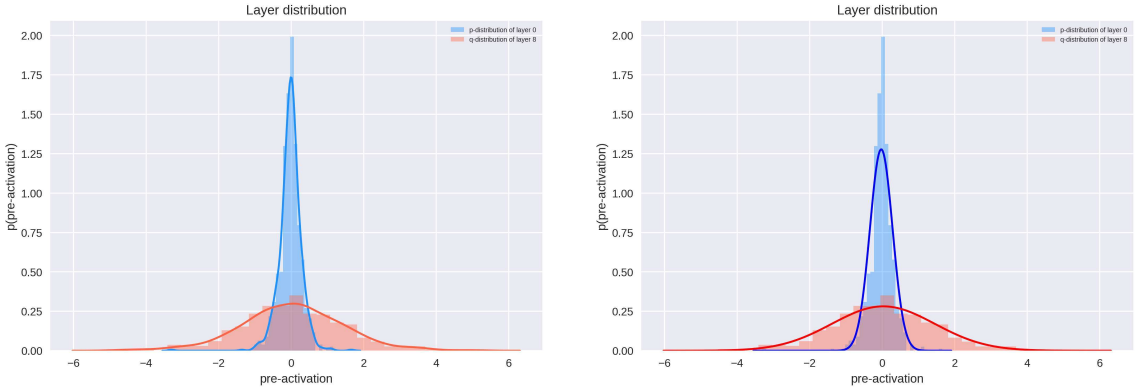

FIGURE 34: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 8th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

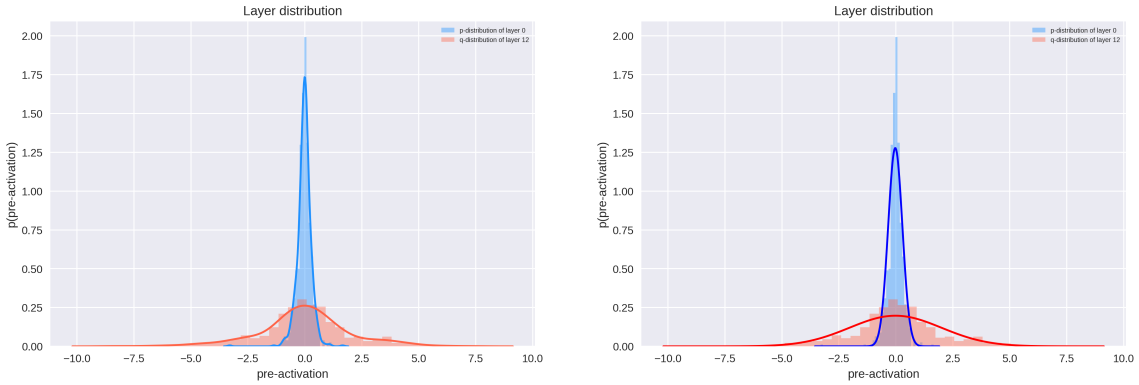

FIGURE 35: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 12th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

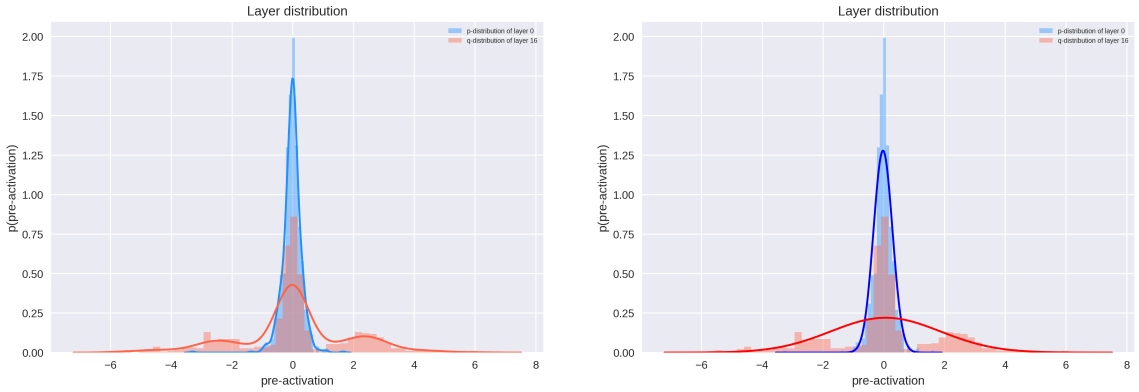

FIGURE 36: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 16th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

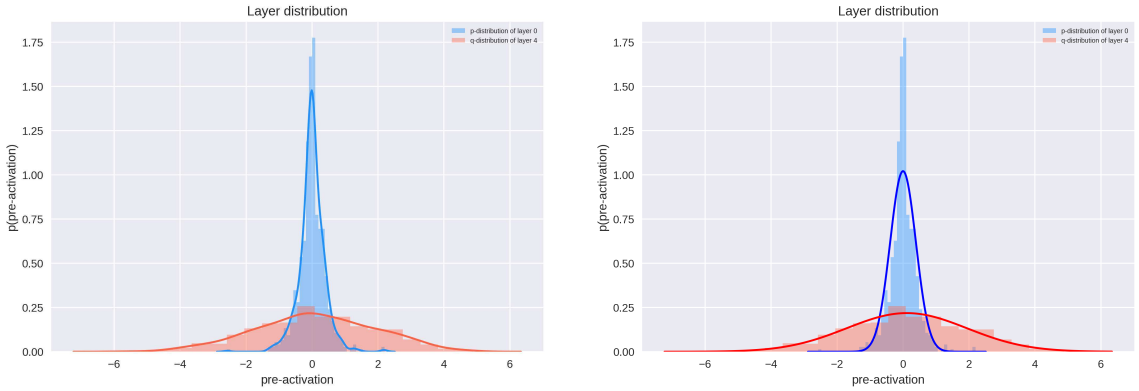

FIGURE 37: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 4th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

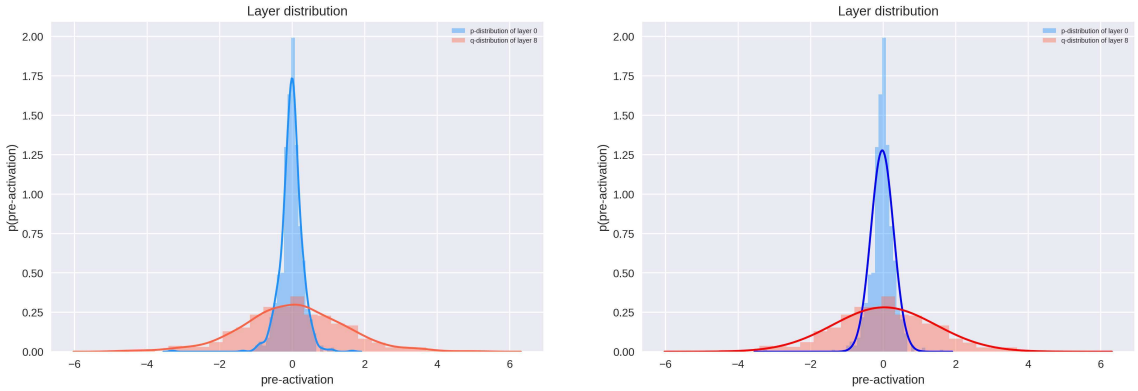

FIGURE 38: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 8th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

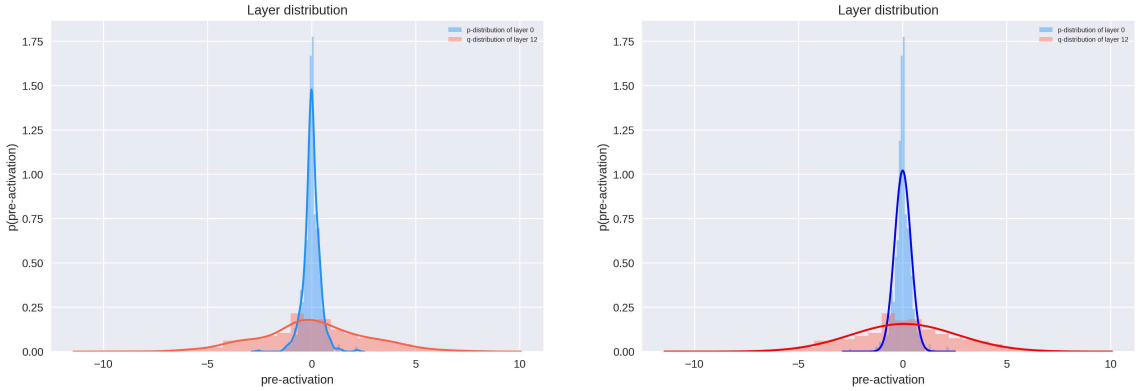

FIGURE 39: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 12th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

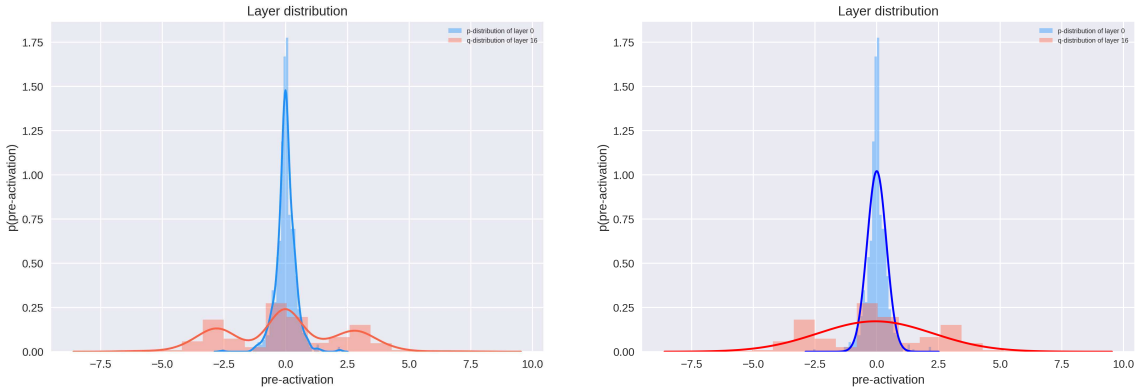

FIGURE 40: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 16th layer. The network was trained on handwritten numbers from the MNIST data set. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

Comparing the KDE fit with the normal distribution fit for $\sigma_w^2 = 1.5$ and $\sigma_w^2 = 1.75$ shows that, with exception of the 16th layer of a network with weight variance $\sigma_w^2 = 1.75$, all layer distributions approximately resemble a Gaussian. That 16th layer appears to be a superposition of multiple Gaussians.

I now measure the propagation of information via the relative entropy for $\sigma_w^2 \, \epsilon \, \{1.5, 1.75\}$ and visualize it in figure 41.

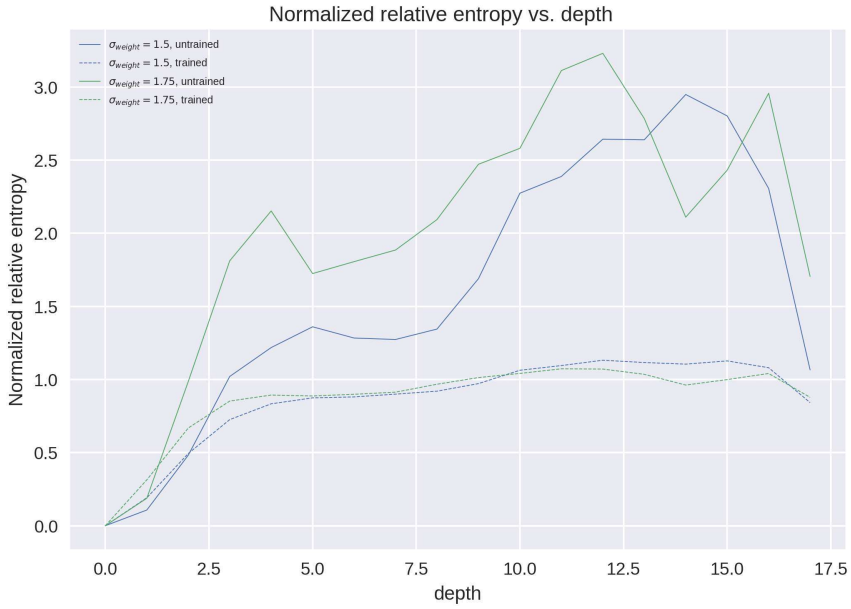

FIGURE 41: The normalized relative entropy before and after training for a sparse autoencoder with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, and fixed depth of 18 layers after 1000 epochs of training on the MNIST data set with a learning rate of 0.0003. For $\sigma_w^2 = 1.5$ the trajectory of the relative entropy increases steeply to a peak and then decreases. For $\sigma_w^2 = 1.75$ it demonstrates the same behavior but has two peaks. This could be because the 16th layer resembles a superposition of multiple Gaussians.

For $\sigma_w^2 = 1.5$ the relative entropy, i.e., the information content, increases up to a peak and then shrinks (Figure 41). For $\sigma_w^2 = 1.75$ it demonstrates the same behavior but has two peaks. The aforementioned 16th layer, which appears to be a superposition of multiple Gaussians, might explain the second peak in the trajectory of the relative entropy (Figure 41). That the relative entropy increases up to a peak and then shrinks corresponds to the behavior I intuitively expect for an autoencoder. This is because the network first decodes the input image into a lower-dimensional representation, where this representation should have a probability distribution that is very different from the distribution p. Therefore the relative entropy[25] is large. Subsequently, the autoencoder decodes the low-dimensional representa-

[25]The relative entropy measures how much the target distribution q differs from the reference distribution p [6].

tion so that its distribution more and more resembles the p distribution. Consequently, the closer the relative entropy gets to the output layer, the more it decreases. However, it does not become 0 because the layers are Gaussian and therefore the reconstructed images have Gaussian distributed greyscale values, unlike the input images (Figure 31 and figure 32).
According to my intuitive interpretation of the behavior of relative entropy and the fact that my sparse autoencoder uses weight decay as a sparsity constraint rather than a dimensionality reduction in latent space, i.e., a bottleneck, the transition between encoder and decoder appears to occur at the peak of the trajectory.
Moreover, the level of the relative entropy trajectory again increases with σ_w^2.
As with the multilayer perceptron, the relative entropy post-training lies below the relative entropy pre-training with a slight qualitative change in the shape of the trajectory. Namely, the post-training relative entropy is flatter compared to the pre-training relative entropy. Again, I explain this behavior by thinking of Deep Learning as a coarse-graining procedure that removes irrelevant information and leaves only the most important information with respect to the reconstruction process.

2.2.4 The Ising model

We now investigate how changing the input data set affects the relative entropy of the sparse autoencoder. To do this, we feed spin configurations of the 2d Ising model to the autoencoder for reconstruction. But before that, we take a look at the Ising model and how we can generate spin configurations.

Statistical physics allows us to derive global properties of a complex system by analyzing the behavior of its constituents. The complex system we want to focus on is a 2d ferromagnet whose constituents are dipoles that are interacting with each other. To study this interacting system, which is made of magnetic dipole moments of spins, we resort to the 2d Ising model. It has a ferromagnetic phase below a critical temperature T_c and a paramagnetic phase above T_c. These phases can be characterized by an order parameter, which in the case of the Ising model is the magnetization $\vec{M}$. It essentially describes the degree of symmetry. For instance, at $T > T_c$ the system is disordered and therefore the symmetry is high, i.e. a $\mathcal{Z}_2$ symmetry. At $T < T_c$ all spins are aligned in the same direction, therefore the system is globally ordered and the symmetry is low. That is because for $T \to T_c$ the correlation length diverges such that all degrees of freedom, i.e. spins, become strongly coupled. Thus, the system undergoes a phase transition at critical. Since the order parameter $\vec{M}$ changes continuously at critical this phase transition is a second order phase transition. In the following, an external magnetic field is neglected so that the direction of magnetization is chosen spontaneously at critical. Consequently, we observe spontaneous symmetry breaking.

We create the Ising model mathematically by defining a set of lattice sites that form a translationally invariant squared lattice of the size $L \times L$. A spin with two possible configurations, i.e. spin-up and spin-down $\sigma_i \, \epsilon \, \{-1, 1\}$, is assigned to each site i. In total, the squared lattice consists of N spins. We only consider the interactions between the nearest neighbors $<i,j>$ and for any two adjacent spins i, j there exists an interaction strength J_{ij}. We assume that this interaction strength is the same for all nearest neighbors and thus neglect the indices. The total energy of a 2d system without an external magnetic field can then be described by the following Hamiltonian

$$\mathcal{H} = -J \sum_{<i,j>} \sigma_i \sigma_j. \tag{2.119}$$

Since it only considers the nearest-neighbor interaction, it sums over all pairs of adjacent spins. Note that every pair is counted only once. We then rewrite (2.119) into

$$\mathcal{H} = -J \sum_i \sum_j \sigma_i \sigma_j, \tag{2.120}$$

where the first sum runs over all i spins of the lattice and the second sum runs over the j nearest neighbors of spin i. For a 2d lattice each spin will have four nearest neighbors.

The Hamiltonian of the Ising model essentially describes the energy resulting from the spin-spin interaction between a spin on the lattice site i and its nearest neighbors. Since we want a system that demonstrates ferromagnetic behavior below critical the coupling constant is $J > 0$. Consequently, configurations of adjacent spins that point in the same direction are more probable. We assume that the Ising model is a canonical system in thermal equilibrium with a heat reservoir, which is why the configuration probability of spins is given using the Boltzmann distribution

$$P(\sigma) = \frac{1}{Z} e^{\beta \mathcal{H}} \text{ with } \beta = \frac{1}{T k_B}. \tag{2.121}$$

Here T is the temperature, k_B is the Boltzmann constant, and Z is the canonical partition function which serves as a normalization constant to make (2.121) a probability measure. The canonical partition function is defined as

$$Z = \sum_{\sigma} e^{-\beta \mathcal{H}}, \tag{2.122}$$

where the sum goes over all the spin configurations σ. It tells us that the lower the temperature, the lower the probability of a spin flip. Physically, this is because we have fewer thermal fluctuations that can cause the spin to flip. Using (2.121) we can calculate observables of the system by summing over all states. As each spin can take two values, we have 2^N possible states. Therefore, a 10×10 lattice of spins would require a summation over 2^{100} states. Thus, summing over all states to compute observables is computationally inefficient, which is why we resort to a more efficient method, the Metropolis-Hasting algorithm.

2.2.4.1 Metropolis-Hasting algorithm

The Metropolis-Hasting algorithm is a Markov chain Monte Carlo method (MCMC). It allows us to obtain a sequence of samples from a probability distribution with probability density $p(x)$ without having to know $p(x)$ exactly, but only a function $f(x)$ proportional to it [8]. In other words, we do not need to know the normalization constant. The reason for this is that the MCMC algorithm only considers the ratio

$$\frac{p(\hat{x})}{p(x)} = \frac{f(\hat{x})}{f(x)}, \tag{2.123}$$

where x and $\hat{x}$ are sample points. These sample points are iteratively generated from $f(x)$, where the distribution of a future sample depends only on the current sample. Consequently, this sequence of samples forms a Markov chain.[26] The more samples we obtain, the more their distribution approaches $p(x)$.

[26] A Markov chain is a stochastic process that satisfies the Markov property. A stochastic process has the Markov property if it is memoryless.

From a stochastic point of view, the Ising model is a Markov random field, which means that the Markov property is extended to an interconnected network of elements. In the case of the Ising model, these elements are spins, which together form a spin configuration σ. This configuration follows the Boltzmann distribution (2.121)

$$p(\sigma) \propto \mathrm{e}^{-\beta \mathcal{H}(\sigma)}. \tag{2.124}$$

To obtain the probability for a specific spin configuration, we need to normalize by all possible spin configurations Z so that all probabilities sum to 1. This is computationally expensive. Fortunately, with our Metropolis-Hasting algorithm, we do not need to know Z. This is because it only considers the ratio of the probability density from which we draw samples

$$\frac{p(\hat{\sigma})}{p(\sigma)} = \frac{\mathrm{e}^{-\beta \mathcal{H}(\hat{\sigma})}}{\mathrm{e}^{-\beta \mathcal{H}(\sigma)}} = \mathrm{e}^{-\beta(\mathcal{H}(\hat{\sigma}) - \mathcal{H}(\sigma))}. \tag{2.125}$$

Here σ and $\hat{\sigma}$ are samples. Thus, it is sufficient to know the Boltzmann distribution to generate samples. This allows us to define the following implementation of the Metropolis-Hasting algorithm [15]:

1. Generate a random spin configuration σ with energy $\mathcal{H}(\sigma)$
2. Repeat:
   (a) Randomly choose a spin σ_i and flip it such that $\sigma_i \rightarrow \hat{\sigma}_i$. The energy of the new configuration $\hat{\sigma}$ is therefore $\mathcal{H}(\hat{\sigma})$
   (b) Compute the energy difference $dE = \mathcal{H}(\hat{\sigma}) - \mathcal{H}(\sigma)$ between a spin configuration and the same spin configuration only with a flipped spin at position i via
   $$dE = 2\sigma_i \sum_j \sigma_j. \tag{2.126}$$
   Here the sum runs over the four nearest neighbors of the spin at position i.
   (c) Accept the new configuration with the probability
   $$p(\sigma \rightarrow \hat{\sigma}) = \begin{cases} 1 & \text{if dE} < 0 \\ \exp(-\beta dE) & \text{if dE} \geq 0 \end{cases}, \tag{2.127}$$
   which results from (2.125). The first case states that the system always accepts the new configuration when it is energetically favored over the previous state. The second case states that even if the new configuration is not energetically favored, there exists a probability of $\exp(-\beta dE)$ that the spin still gets flipped due to thermal fluctuations.

So, this algorithm selects the new configuration $\hat{\sigma}$ relative to the old one with (2.127). It moves from one state to another with the probability $p(\sigma \rightarrow \hat{\sigma})$. The algorithm returns samples that follow the desired unknown distribution with probability density $p(\sigma)$. This allows us to simulate the behavior of the Ising model above and below T_c.

2.2.5 Results Ising

In this chapter, I use the relative entropy for a hierarchical network to analyze the propagation of information through a sparse autoencoder trained on spin configurations of the 2d Ising model. To train the network sufficiently well on images of 2d spin configurations, I reduce the number of hidden layers. The autoencoder now consists of an input layer with 784 neurons, 6 hidden layers with a constant width of 784 neurons, and an output layer with 784 neurons.

2.2.5.1 Simulation of spin configurations

To generate spin configurations of the 2d Ising model at different temperatures, I use the recently introduced Metropolis-Hasting algorithm (see chapter 2.2.4.1). The implementation of the algorithm can be found in A.4.8. It generates 20 spin configurations per 40 different temperatures, which are uniformly distributed between $0.1\, J/k_B$ and $5\, J/k_B$.[27] With these spin configurations, I train the autoencoder in an unsupervised learning environment so that it learns to reconstruct the spin configurations (Figure 42 and figure 43).

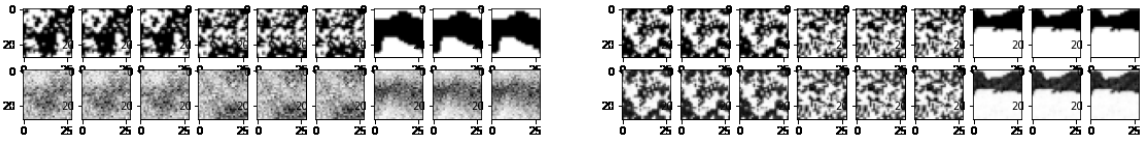

FIGURE 42: Reconstructed images of a network with a weight variance of $\sigma_w^2 = 1.5$ trained for 0 out of 1000 epochs (left) and 1000 out of 1000 epochs (right). The first row displays the input images and the second row the reconstructed images.

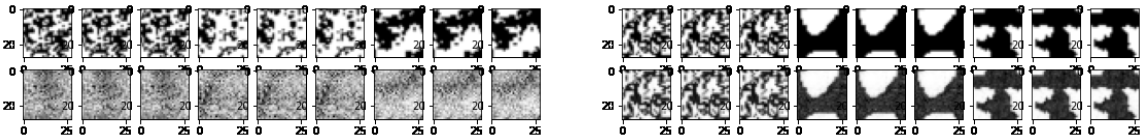

FIGURE 43: Reconstructed images of a network with a weight variance of $\sigma_w^2 = 1.75$ trained for 0 out of 1000 epochs (left) and 1000 out of 1000 epochs (right). The first row displays the input images and the second row the reconstructed images.

2.2.5.2 The normalized relative entropy for a sparse autoencoder trained on spin configurations of the 2d Ising model

For the same reasons as for the MNIST data set (see chapter 2.2.3), I neglect the last forward pass and use $\sigma_w^2 \, \epsilon \, \{1.5, 1.75\}$ to measure the information flow through the network via the relative entropy. Even though the sparse autoencoder seems to have reconstructed the 2d

[27] The critical temperature of the Ising model lies at $2.269\, J/k_B$ [25].

spin configurations well (Figure 42 and figure 43), I check whether the distributions of layers are Gaussian and therefore the relative entropy allows for physical interpretation.

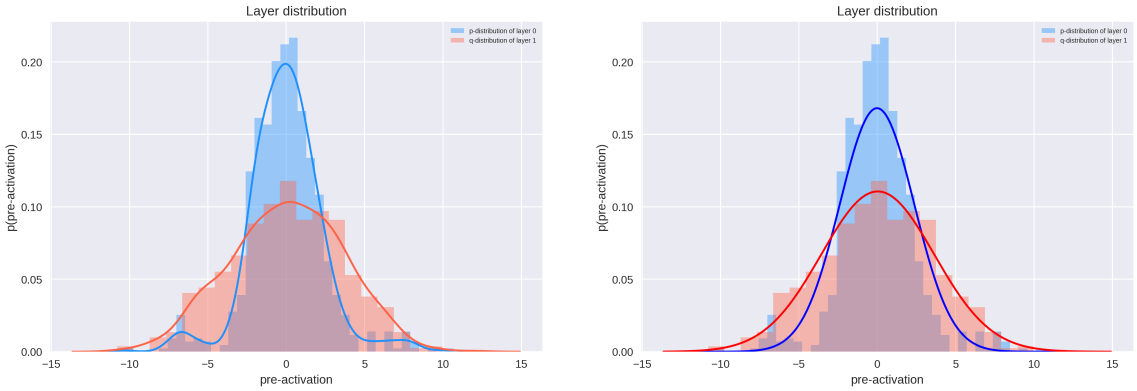

FIGURE 44: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 1st layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

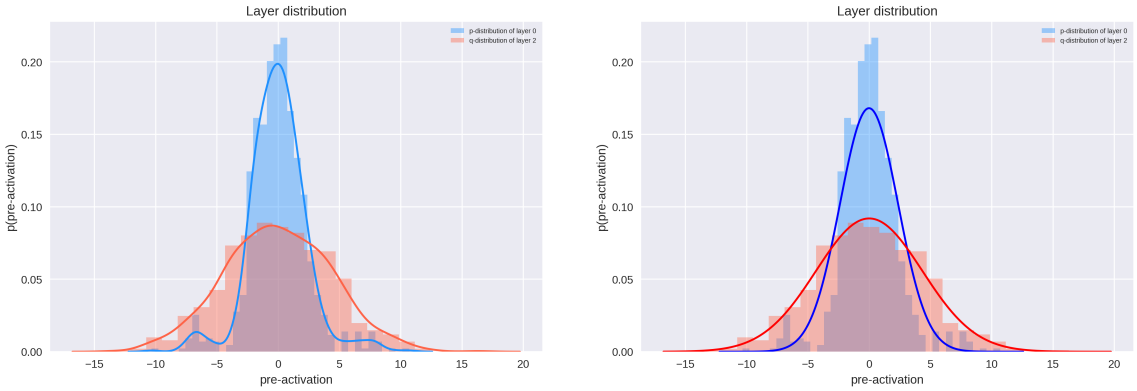

FIGURE 45: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 2nd layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

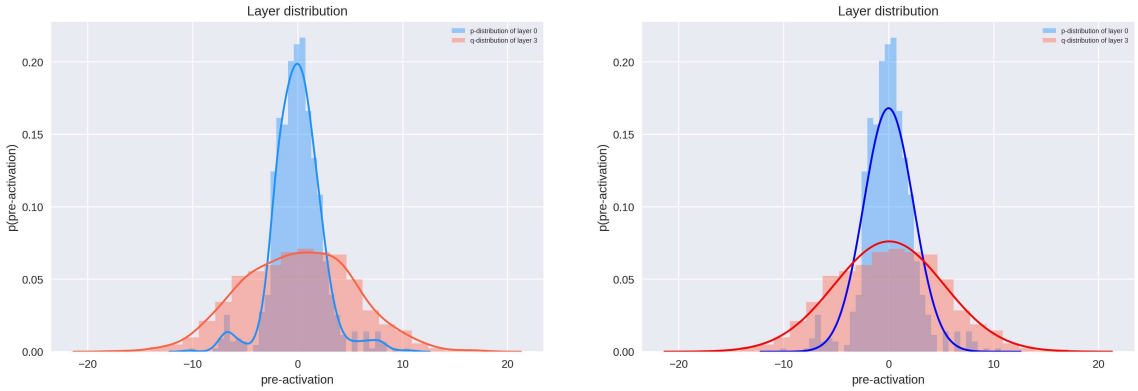

FIGURE 46: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 3rd layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

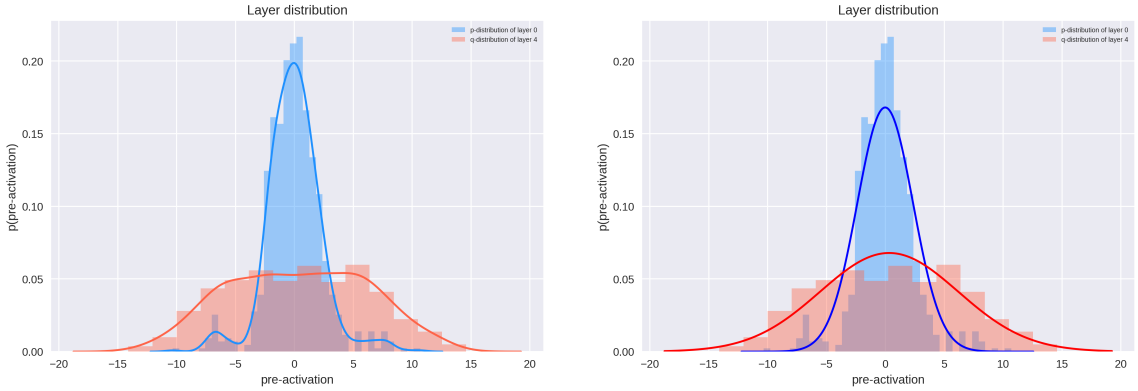

FIGURE 47: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 4th layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

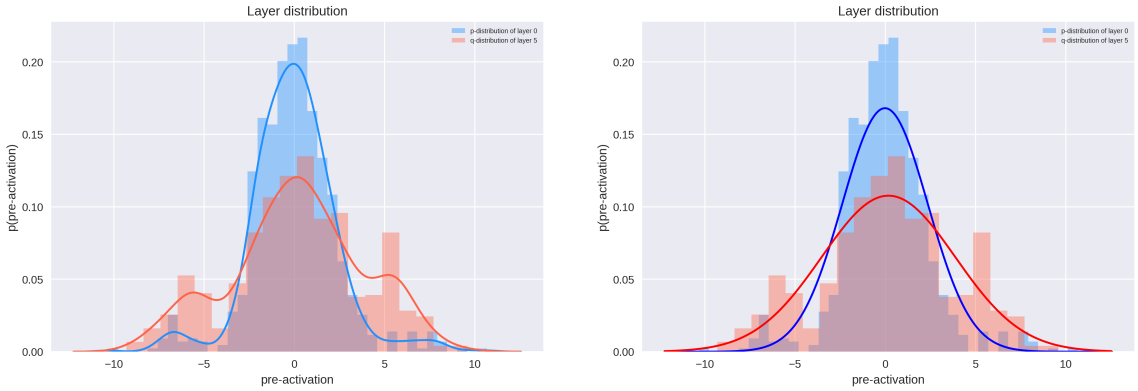

FIGURE 48: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 5th layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.5$ and was trained for 1000 epochs.

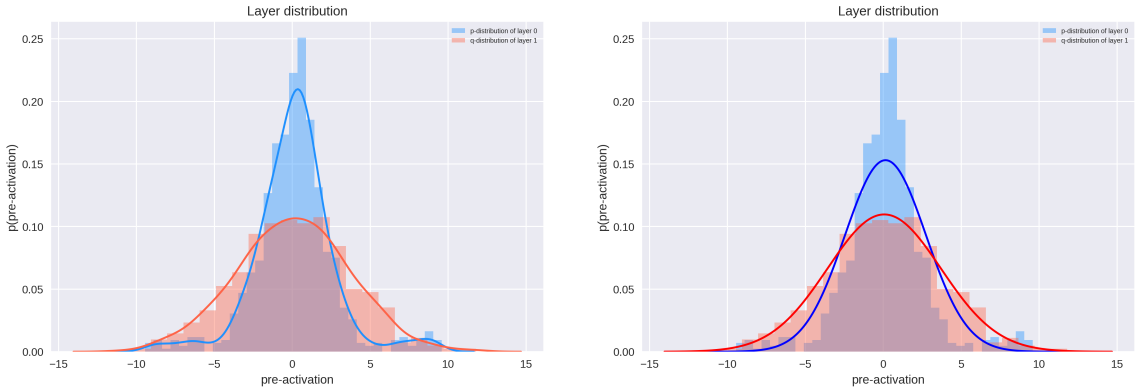

FIGURE 49: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 1st layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

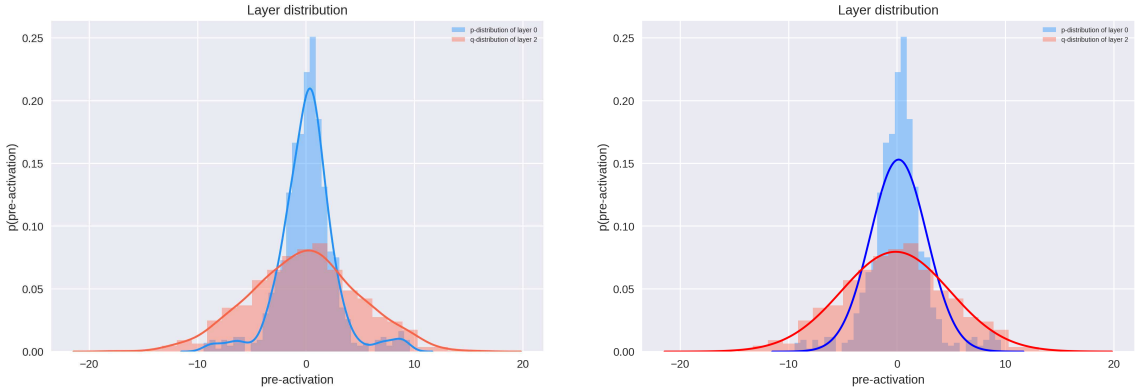

FIGURE 50: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 2nd layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

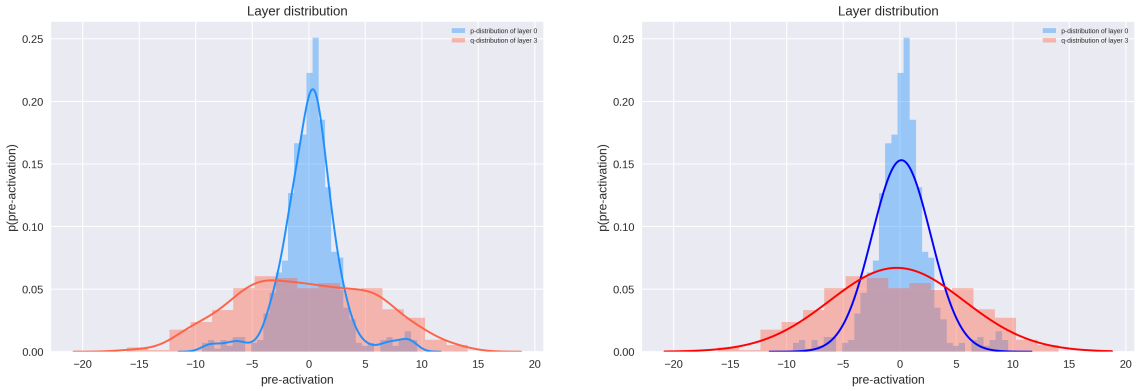

FIGURE 51: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 3rd layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

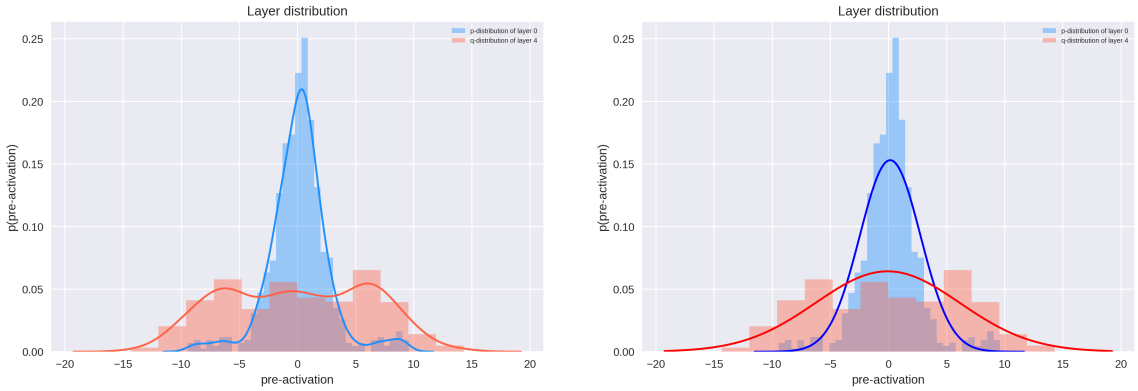

FIGURE 52: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 4th layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

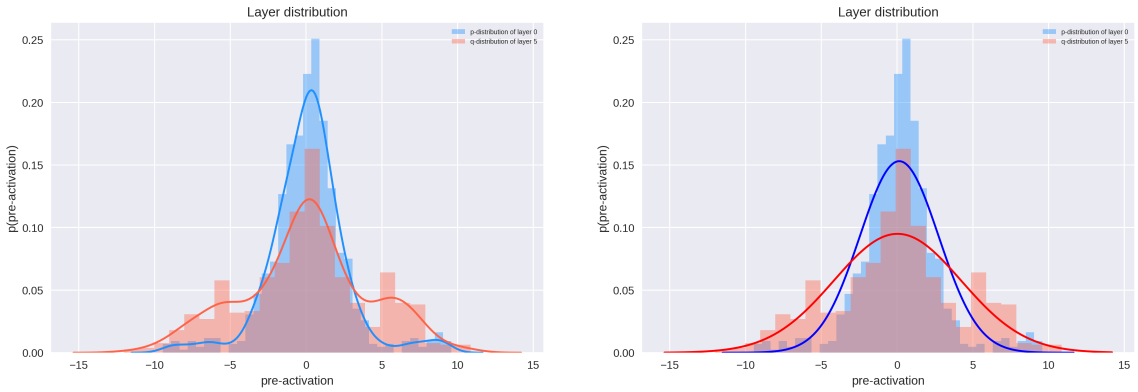

FIGURE 53: KDE fit (left) and normal distribution fit (right) for the layer distribution of the autoencoder's 5th layer. The network was trained on spin configurations generated by a 2d Ising model. It has a weight variance of $\sigma_w^2 = 1.75$ and was trained for 1000 epochs.

The distribution q in the encoding part of the autoencoder can be considered approximately Gaussian for both $\sigma_w^2 = 1.5$ and $\sigma_w^2 = 1.75$, while in the decoding part the distribution q resembles a superposition of multiple Gaussians.[28]

[28]Figure 54 indicates, where the encoding part ends and the decoding part begins.

I now measure the relative entropy for $\sigma_w^2 \, \epsilon \, \{1.5, 1.75\}$ and visualize it in figure 54.

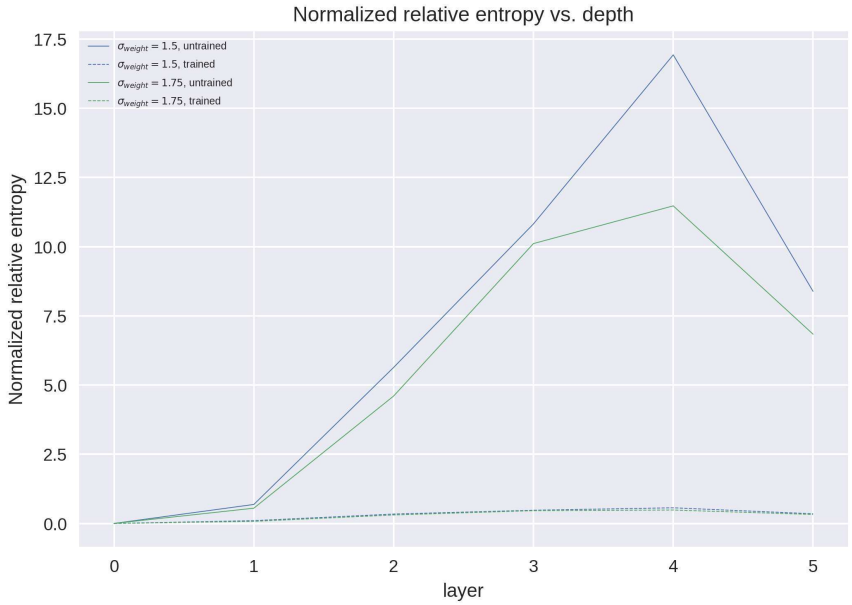

FIGURE 54: This plot demonstrates the relative entropy before and after training for a network with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, and fixed depth of 6 layers after 200 epochs of training on spin configurations of the Ising model with a learning rate of 0.0003. The trajectory of the relative entropy increases sharply up to a peak and then decreases

A qualitatively identical behavior to the autoencoder in chapter 2.2.3.1 can be observed. The relative entropy increases up to a peak and then decreases. For the spin configurations of the Ising model, this behavior is even more clear than for the MNIST data set. Again, I interpret the peak as the end of the encoding part and the beginning of the decoding part of the autoencoder. Also, the relative entropy post-training lies below the relative entropy pre-training. Compared to the MNIST case, the relative entropy post-training becomes even flatter. Again, I explain this behavior by considering Deep Learning as a coarse-graining procedure that removes irrelevant information and leaves only the most relevant information with respect to the reconstruction process.

2.2.5.3 Summary of results for a sparse autoencoder

All in all, both the autoencoder trained on MNIST images and the autoencoder trained on spin configurations demonstrate a relative entropy with qualitatively identical behavior. The relative entropy increases up to a peak and then decreases. In both cases, I suspect that this peak is the end of the autoencoder's encoding part and the beginning of its decoding part. Furthermore, in both cases, the relative entropy post-training is lower than the relative entropy pre-training. As for the multilayer perceptron, I explain this behavior by considering Deep Learning as a coarse-graining procedure that removes irrelevant information and leaves only the most relevant information with respect to the reconstruction process. For the autoencoder trained on spin configurations, the difference between the trajectory post-training and pre-training is larger. The main difference between the MNIST autoencoder and the Ising autoencoder is in the training process. In order to train the autoencoder sufficiently well on the spin configurations, I need to reduce the depth of the autoencoder to 6 layers.

2.3 Relative entropy pre-training vs. post-training

Contrary to chapters 2.1.4, 2.2.3, and 2.2.5, where the relative entropy post-training lies below the relative entropy pre-training, I observe that the relative entropy post-training can also lie above the relative entropy pre-training. This is the case for relatively shallow multilayer perceptrons and autoencoders (Figure 55 and figure 56). As the depth of the network increases, the post-training relative entropy eventually falls below the pre-training relative entropy. The depth at which this occurs seems to depend on the network architecture, the learning rate, and the data set. This means that my interpretation of the position of the relative entropies using the renormalization group's coarse-graining procedure is not generally true. Since this is counterintuitive to my intuition, it raises the question as to whether the position of the two relative entropies allows for a physical interpretation at all.

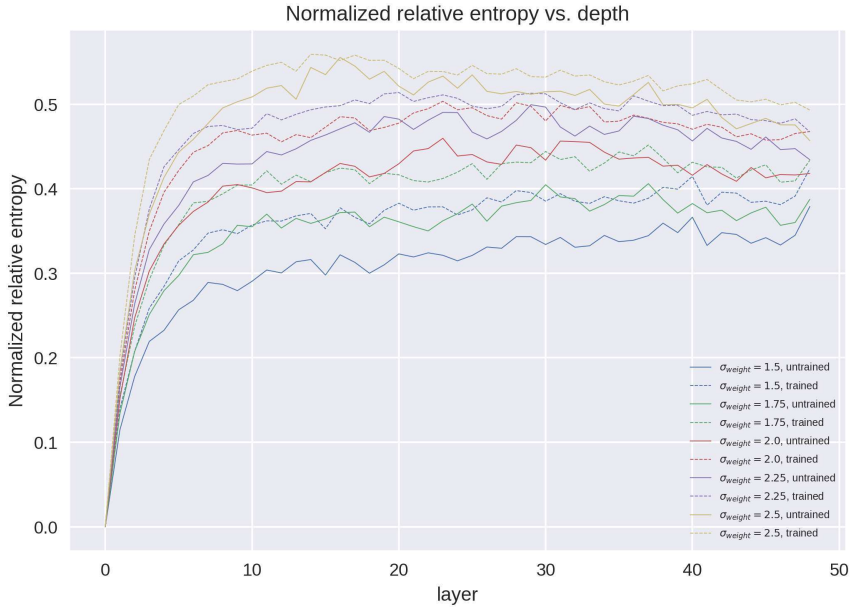

FIGURE 55: The normalized relative entropy pre-training and post-training for a 50-layer multilayer perceptron with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, and a learning rate of 0.005 after 50 epochs of training. The trajectory of the relative entropy increases steeply to an asymptote. The relative entropy post-training lies above the relative entropy pre-training.

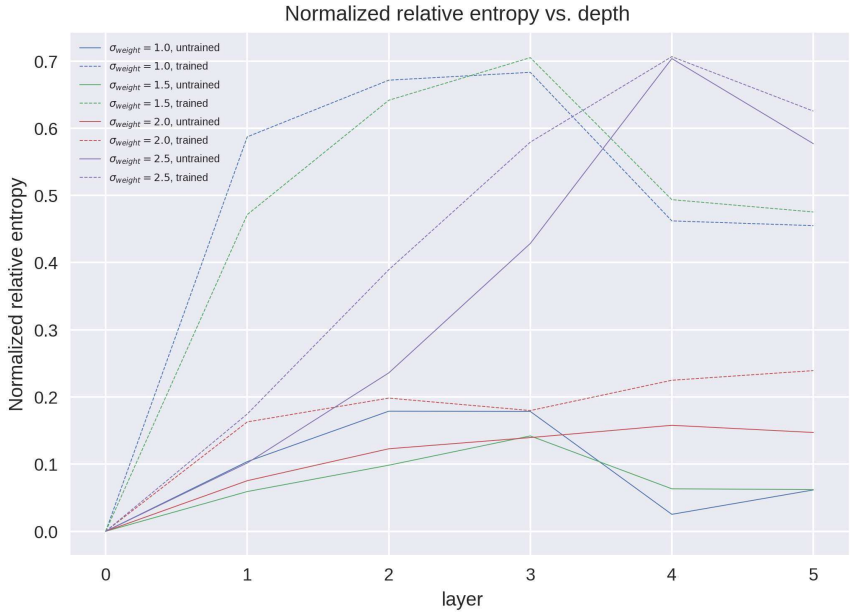

FIGURE 56: The normalized relative entropy pre-training and post-training for a sparse autoencoder with varying σ_w^2, fixed $\sigma_b^2 = 0.05$, and a depth of 6 layers after 200 epochs of training on the MNIST data set with a learning rate of 0.0003. The trajectory of the relative entropy increases steeply to a certain peak and then decreases. The relative entropy post-training lies above the relative entropy pre-training.

The first reason to argue against this is that the initialization sets the baseline curve and this initialization is random [6].
Moreover, I do not compare the target distribution q with the gray-scale values of the input image, but with the first forward pass (Figure 57). The problem here is that this forward pass consists of already processed information, which affects the relative entropy. A first step to get these two problems under control would be to disentangle the contributions of the network structure to the relative entropy from the contributions of the data to the relative entropy [6].

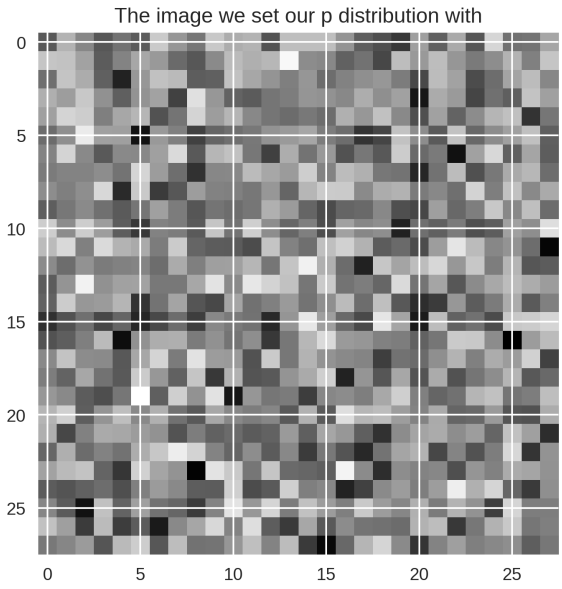

FIGURE 57: Visualization of the first forward passe's pre-activation values (distribution p). The greyscale intensity represents the neuron's strength and can be considered to be approximately Gaussian. The pre-activation values do not resemble the handwritten number of the input image.

Another reason that suggests that I cannot physically interpret the position of the relative entropies is that our formula takes the Gaussians that describe the different layers as an input. The problem with this is that each Gaussian is a coarse-grained state [6]. For instance, even flat networks have many thousands of neurons with parameters that allow realizing the same Gaussian with many different parameter configurations. Consequently, our method may be too coarse-grained to meaningfully quantify the information content of a layer.
All these aspects are also general criticism of our method.

3 Conclusion and Discussion

This bachelor thesis was a contribution to the growing intersection of Deep Learning and theoretical physics. Deep Learning models are used in increasingly important domains in science, as well as in industry. Since the behavior of these models has not yet been fundamentally understood, the goal of this thesis was to take a step toward fundamentally understanding Deep Learning using theoretical physics. The main contribution was the analysis of the propagation of information through a multilayer perceptron and a sparse autoencoder using the relative entropy.

Chapter 2.1.1 focused on deriving the architecture of two neural networks, the multilayer perceptron, and the sparse autoencoder. The derivation was inspired by a biological neural network, the human brain. The basic computational unit of the brain, the biological neuron, was translated into a mathematical model called a perceptron. Then, several of these perceptrons were arranged in layers and connected. This resulted in the mathematical architecture of these two neural networks.
The architecture of the multilayer perceptron alone was not capable of recognizing images. This had to be taught to the multilayer perceptron, which is also known as supervised learning. At the beginning of the learning process, the network was told what digit each input image from the MNIST database should represent. The cross-entropy cost function then provided the discrepancy between what an image should represent and what the network thought it represented as feedback. The multilayer perceptron used this feedback to iteratively minimize this discrepancy using gradient descent. This is how the network learned to recognize images (see chapter 2.2.1).
For the reconstruction of images, the multilayer perceptron and supervised learning were no longer suited. Instead, the sparse autoencoder and unsupervised learning were applied (see chapter 2.2). In unsupervised learning, the network was not told what digit the input image should represent. Only by using the input image and trying to imitate it, the network figured out by itself how to reconstruct it. The mean squared error cost function with weight decay provided the deviation between the input image and the reconstructed image as feedback. This feedback was subsequently used to minimize this discrepancy via the gradient descent algorithm. For the network to be able to recognize or reconstruct images that it had not seen before, the generalization problem was addressed at the end of chapter 2.2.1.

The main contribution of this work was to shed more light on what happens inside neural networks by using the relative entropy. The relative entropy analyzed how the information of the input images propagated through the multilayer perceptron and the sparse autoencoder. For the derivation of relative entropy in chapter 2.1.3, the concept of entropy was first discussed from the perspective of physics. The chapter then considered a neural network with a sufficient number of neurons per layer such that the layers could be assumed to be

approximately Gaussian. The application of mean field theory then allowed each layer to be described by a Gaussian distribution. As a result, the relative entropy for a hierarchical network could be derived, implemented in Python, and used to measure how the content of information changes through the layers of a neural network.

The results indicated that the networks had to be sufficiently well-trained before the relative entropy could be used. The reason for this was that the worse the network was trained, the less the distribution of a layer resembled a Gaussian distribution. Since the Gaussianity of the layers was a prerequisite for deriving the relative entropy, the measurement had no physical interpretation in scenarios where this was not the case. In addition, the results showed that the further σ_w^2 moved away from the critical point and the more the depth of the network was increased, the faster the network became untrainable, the faster the layers lost their Gaussianity, and the faster the relative entropy became physically uninterpretable. The reason for this was that the network could only be trained if the information from the input images could travel throughout the network. How far the information could travel was determined by the correlation length ξ. Since ξ diverges at the critical point and decreases away from it, σ_w^2 was initialized near the critical point.
These findings can be used to improve the study of relative entropy in other neural networks. To determine whether the layers of the multilayer perceptron were Gaussian, the classification error was used as an indicator. If this error was relatively large, the distribution of the layers was examined using kernel density estimation. For the autoencoder, the classification error could no longer be used as an indicator, so the layer distributions had to be examined for each relative entropy measurement.

Chapter 2.1.4 used the relative entropy to analyze how changing the number of neuron layers affected the propagation of information through a multilayer perceptron that was trained to classify images from the MNIST database. It turned out that the change in the number of layers had no effect on the qualitative behavior of the relative entropy and thus no effect on the information content. It increased up to an asymptote whose value depended on the choice of weight variance σ_w^2. According to [6], the increase up to an asymptote indicated that the information content did not change once the asymptote was reached. Consequently, it was tempting to say that this indicates that once the asymptote is reached, the additional layers are no longer necessary and can be removed to speed up the training process. But that the information content no longer changed did not mean that information was no longer processed [6]

Chapter 2.2 analyzed the propagation of information through a sparse autoencoder. The autoencoder was trained to reconstruct images from the MNIST database and spin configurations of a two-dimensional Ising model. A Metropolis-Hasting algorithm generated these spin configurations. Since the autoencoder could be viewed as two multilayer perceptrons

stacked on top of each other, the same Python program used for the multilayer perceptron was utilized for this analysis. The results showed that the relative entropy increased up to a peak and then decreased. Since the relative entropy measures how much the target distribution q deviates from the reference distribution p, this was the intuitively expected behavior. This was because the autoencoder first decoded the input image into a lower dimensional representation, where this representation had a probability distribution that was very different from the distribution p. Subsequently, the autoencoder decoded the low dimensional representation so that its distribution increasingly resembled the distribution p. This behavior was independent of whether we used MNIST images or spin configurations of a 2d Ising model as the input data set. Because of that and the structure of the autoencoder, I also assumed that the peak of the relative entropy marked the end of the encoder and the beginning of the decoder. The main difference between the autoencoder that reconstructed MNIST images and the one that reconstructed spin configurations was that the latter required a reduction in the number of layers to properly train it.

For both a multilayer perceptron with relatively few layers and an autoencoder with relatively few layers, the relative entropy pre-training was above the relative entropy post-training. As the depth of the network increased, the relative entropy post-training eventually fell below the relative entropy before training. In this case, I suspected that an intuitive look at the renormalization group might help to understand this behavior. This was because both the renormalization group and Deep Learning involve a coarse-graining procedure that removes irrelevant information and retains only the information relevant to the reconstruction problem and the classification problem. However, this interpretation was not applicable to networks with a small number of layers, making it not generally true. Since this contradicted my intuition, I wondered if it was even possible to meaningfully interpret this behavior using our method. I found several reasons that argued against it and questioned the method as a whole. Nevertheless, one way to better understand this behavior would be to examine the relative entropy post-training compared to the relative entropy pre-training for various supervised learning and unsupervised learning problems. Doing so could disentangle the contributions of the network structure to the relative entropy from the contributions of the data to the relative entropy.
Measuring the propagation of information through deep neural networks using the relative entropy seemed to have general weaknesses that made it difficult to explain the behavior of relative entropy pre-training vs. post-training and called the relative entropy as an accurate method for measuring the propagation of information into question. The first major criticism was that this method did not compare the target distribution q to the input image, but to the first forward pass. The problem with this was that it consisted of already processed information, which affected the accuracy and interpretability of the relative entropy. The second major criticism was that relative entropy took Gaussian distributions as input and one of these distributions could be realized by many different configurations of the many

thousands of neuron parameters. Consequently, this method may have been too coarse-grained to meaningfully measure the information content of the layers.

All in all, this thesis was able to shed more light on neural networks using the relative entropy. Although new insights were obtained on how information propagates in a deep neural network and how relative entropy can be used to measure the propagation of information, it could not be understood how information from an input image is processed in such a way that the network learns to recognize or reconstruct it. Therefore, the next step would be to investigate whether it is possible to show that Deep Learning removes irrelevant information with respect to the reconstruction or classification of images. Since we may need a measurement method that is fine-grained for this, new ideas are needed!

Zusammenfassung und Diskussion

Diese Bachelorarbeit war ein Beitrag zum wachsenden Zusammenspiel von Deep Learning und theoretischer Physik. Deep Learning-Modelle werden in immer wichtigeren Bereichen in der Wissenschaft, aber auch in der Industrie eingesetzt. Da das Verhalten dieser Modelle noch nicht grundlegend verstanden werden konnte, war es das Ziel dieser Arbeit, mithilfe der theoretischen Physik mehr fundamentales Verständnis über Deep Learning zu erlangen. Der Hauptbeitrag war die Untersuchung der Informationsausbreitung durch ein Multilayer Perceptron und einen Sparse Autoencoder mithilfe der Relativen Entropie.

Das Kapitel 2.1.1 beschäftigte sich mit der Herleitung der Architektur zweier neuronaler Netzwerke, dem Multilayer Perceptron und dem Sparse Autoencoder. Für die Herleitung wurde ein biologisches neuronales Netzwerk, das menschliche Gehirn als Inspiration verwendet. Die grundlegende Recheneinheit des Gehirns, das biologische Neuron, wurde in ein mathematisches Modell namens Perceptron übersetzt. Anschließend wurden mehrere dieser Perceptronen in Schichten angeordnet und verknüpft. So ergab sich die mathematische Architektur der neuronalen Netzwerke.
Die Architektur des Multilayer Perceptrons allein war jedoch nicht in der Lage, Bilder zu erkennen. Dies musste dem Multitlayer Perceptron beigebracht werden, was auch als Supervised Learning bezeichnet wird. Zu Beginn des Lernprozesses wurde dem Netzwerk mitgeteilt, welche Ziffer jedes Eingabebild der MNIST-Datenbank darstellen soll. Die Cross-Entropy Cost Function hat dem Netzwerk die Diskrepanz zwischen dem, was ein Bild darstellen sollte, und dem, was das Netzwerk glaubte zu sehen, als Feedback gegeben. Dieses Feedback nutze das Multilayer Perceptron, um die Diskrepanz iterativ mithilfe des Gradient Descent Optimierungsalgorithmus zu minimieren. Dadurch lernte das Netzwerk Bilder zu erkennen (siehe Kapitel 2.2.1).
Für die Rekonstruktion von Bildern wurde nicht mehr das Multilayer Perceptron und Supervised Learning verwendet, sondern der Sparse Autoencoder und Unsupervised Learning (siehe Kapitel 2.2). Bei Unsupervised Learning wurde dem Netzwerk nicht mitgeteilt, was das Eingabebild darstellen soll. Allein durch das Eingabebild fand das Netzwerk heraus, wie es dies zu rekonstruieren hat. Es tat dies, indem es versuchte das Bild zu imitieren. Die Mean Squared Error Cost Function with Weight Decay gab dem Autoencoder anschließend Feedback, ob dessen Imitation bzw. Rekonstruktion dem Eingabebild entsprach. Dieses Feedback wurde dann verwendet, um die Diskrepanz zwischen dem Originalbild und dem rekonstruierten Bild mithilfe von Gradient Descent zu minimieren. Damit das Netzwerk in der Lage war, Bilder zu erkennen oder zu rekonstruieren, die es noch nicht gesehen hatte, wurde das Problem der Generalisierung am Ende des Kapitels 2.2.1 behandelt.

Der Hauptbeitrag dieser Arbeit war es, mithilfe der Relativen Entropie mehr Licht in das Innere von neuronalen Netzwerken zu bringen. Die Relative Entropie analysierte, wie

sich die Informationen der Eingangsbilder durch das mehrschichtige Perceptron und den Sparse Autoencoder ausbreiteten. Für die Herleitung der Relativen Entropie in Kapitel 2.1.3 wurde zunächst das Konzept der Entropie aus der Perspektive der Physik behandelt. Anschließend betrachtete das Kapitel ein neuronales Netzwerk mit einer ausreichenden Anzahl von Neuronen pro Schicht, sodass die Schichten als annähernd gaußverteilt angenommen werden konnten. Dies erlaubte jede Schicht mithilfe der Molekularfeldnäherung durch eine Gaußsche Verteilung zu beschreiben. So konnte die Relative Entropie für ein hierarchisches Netzwerk hergeleitet werden, in Python implementiert werden und mit diesem Programm gemessen werden, wie sich der Gehalt an Information durch die Schichten eines neuronalen Netzwerks verändert.

Die Ergebnisse zeigten, dass die Netzwerke ausreichend gut trainiert sein mussten, bevor die relative Entropie verwendet werden konnte. Der Grund dafür war, dass die Verteilung einer Neuronenschicht umso weniger einer Gaußschen Verteilung ähnelte, je schlechter das Netzwerk trainiert wurde. Da die Gaußheit der Schichten eine Voraussetzung für die Herleitung der relativen Entropie war, war die Messung in Fällen, wo dies nicht der Fall war, physikalisch nicht zu interpretieren. Außerdem zeigten die Ergebnisse, dass je weiter sich σ_w^2 vom kritischen Punkt entfernte und je mehr die Tiefe des Netzwerkes erhöht wurde, desto schneller wurde das Netzwerk untrainierbar, desto schneller verloren die Schichten ihre Gaußheit und desto schneller wurde die relative Entropie physikalisch uninterpretierbar. Der Grund dafür war, dass das Netzwerk nur trainiert werden konnte, wenn die Informationen aus den Eingabebildern das gesamte Netzwerk durchdringen konnten. Wie weit die Informationen gelangen konnten, bestimmte die Korrelationslänge ξ. Da ξ am kritischen Punkt divergiert und davon entfernt abnimmt, wurde σ_w^2 in der Nähe des kritischen Punktes initialisiert.
Diese Erkenntnisse können verwendet werden, um die Untersuchung der Relativen Entropie bei anderen neuronalen Netzwerken zu verbessern.
Um herauszufinden, ob die Schichten des Mulltilayer Perceptrons gaußförmig waren, wurde der Klassifikationsfehler als Indikator verwendet. Wenn dieser relativ groß war, wurde die Verteilung der Schichten mithilfe der Kernel Density Estimation untersucht. Für den Autoencoder konnte der Klassifikationsfehler nicht mehr als Indikator verwendet werden, sodass die Schichtverteilungen für jede Messung der Relativen Entropie untersucht werden mussten.

Das Kapitel 2.1.4 untersuchte mithilfe der relativen Entropie, wie sich die Änderung der Anzahl der Neuronenschichten auf die Ausbreitung von Informationen durch ein Multilayer Perceptron, welches für die Klassifizierung von Bildern aus der MNIST-Datenbank trainiert wurde, auswirkte. Es stelle sich heraus, dass die Anzahl der Schichten keinen Einfluss auf das qualitative Verhalten der Relativen Entropie und somit auf den Informationsgehalt hatte. Diese stieg bis zu einer Asymptote an, deren Wert von der Wahl der Gewichtsvarianz σ_w^2

abhing. Laut [6] deutet der Anstieg bis zu einer Asymptote darauf hin, dass sich der Informationsgehalt nicht mehr ändert, sobald die Asymptote erreicht wurde. Folglich war die Versuchung groß zu sagen, dass dies darauf hindeutet, dass nach Erreichen der Asymptote die zusätzlichen Schichten nicht mehr notwendig sind und entfernt werden können, um den Trainings-prozess erheblich zu beschleunigen. Aber dass sich der Informationsgehalt nicht mehr ändert, bedeutete nicht, dass Information nicht mehr verarbeitet wurde [6]

Kapitel 2.2 untersuchte die Propagation von Informationen durch einen Sparse Autoencoder. Der Autoencoder wurde trainiert, um Bilder aus der MNIST-Datenbank sowie Spin-Konfigurationen eines zweidimensionalen Ising Modells zu rekonstruieren. Ein Metropolis-Hasting Algorithmus erzeugte diese Spinkonfigurationen. Da der Autoencoder als zwei übereinandergestapelte Multilayer Perceptrons angesehen werden konnte, wurde das gleiche Python-Programm wie für das Multilayer Perceptron verwendet. Die Ergebnisse zeigten, dass die Relative Entropie bis zu einem Peak anstieg und dann abnahm. Da die relative Entropie misst, wie sehr die Zielverteilung q von der Referenzverteilung p abweicht, war dies das intuitiv zu erwartende Verhalten. Das lag daran, dass der Autoencoder das Eingangsbild zunächst in eine niedrigere dimensionale Repräsentation decodierte und diese Repräsentation eine Wahrscheinlichkeitsverteilung hat, die sich stark von der Verteilung p unterschied. Anschließend decodierte der Autoencoder die niedrig dimensionale Repräsentation, sodass ihre Verteilung mehr und mehr der Verteilung p ähnelte. Dieses Verhalten war unabhängig davon, ob wir die MNIST-Bilder oder die Spin-Konfigurationen eines 2d-Ising-Modells als Eingabedatensatz verwendeten. Deswegen und wegen der Struktur des Autoencoders habe ich auch angenommen, dass die Spitze der relativen Entropie das Ende des Encoders und den Anfang des Decoders markiert. Der Hauptunterschied zwischen dem Autoencoder, der MNIST-Bilder rekonstruierte, und dem, der Spin-Konfigurationen rekonstruierte, bestand darin, dass bei Letzterem die Anzahl der Schichten reduziert werden musste, um ihn angemessen zu trainieren.

Sowohl für ein mehrschichtige Perzeptron mit relativ wenigen Schichten als auch für einen Autoencoder mit relativ wenigen Schichten lag die Relative Entropie vor dem Training über der Relativen Entropie nach dem Training. Mit zunehmender Tiefe des Netzwerkes fiel die Relative Entropie nach dem Training schließlich unter die Relative Entropie vor dem Training. In diesem Fall habe ich vermutet, dass ein intuitiver Blick auf die Renormalisierungsgruppe helfen kann, dieses Verhalten zu verstehen. Dies lag daran, dass sowohl die Renormalisierungsgruppe als auch Deep Learning eine Coarse-Graining Procedure beinhalten, bei der irrelevante Informationen entfernt werden und nur die für das Rekonstruktionsproblem oder das Klassifizierungsproblem relevanten Informationen erhalten bleiben. Diese Interpretation war jedoch nicht auf Netzwerke mit einer geringen Anzahl von Schichten anwendbar, wodurch diese nicht allgemeingültig war. Da dies meiner Intuition widersprach, fragte ich mich anschließend, ob es überhaupt möglich ist, dieses

Verhalten mit unserer Methode sinnvoll zu interpretieren. Ich fand mehrere Gründe, die dagegen sprachen und die Methode generell in Frage stellten. Nichtsdestotrotz bestünde eine Möglichkeit, dieses Verhalten besser zu verstehen, darin, die relative Entropie nach dem Training im Vergleich zur relativen Entropie vor dem Training für verschiedene Supervised Learning und Unsupervised Learning Probleme zu untersuchen. Auf diese Weise könnten die Beiträge der Netzwerkstruktur zur relativen Entropie von den Beiträgen der Daten zur relativen Entropie getrennt werden.
Die Messung der Informationsausbreitung durch tiefe neuronale Netzwerke mithilfe der relativen Entropie schien jedoch generell Schwächen zu haben, die sowohl die Erklärung des Verhaltens der relativen Entropie vor dem Training gegenüber der nach dem Training erschwerten als auch die Realtive Entropie als genaue Methode zur Messung der Informationsausbreitung in Frage stellten. Der erste Hauptkritikpunkt war, dass diese Methode die Zielverteilung q nicht mit dem Eingangsbild verglich, sondern mit dem ersten Forward Pass. Das Problem dabei war, dass dieser aus bereits verarbeiteter Information bestand, was die Genauigkeit und Interpretierbarkeit der Messung der relativen Entropie beeinträchtigte. Der zweite Hauptkritikpunkt war, dass die Relative Entropie Gaußsche Verteilungen als Input entgegennahm und eine dieser Verteilungen durch viele verschiedene Konfigurationen der vielen Tausend Neuronenparameter realisiert werden konnte. Aus diesem Grund könnte diese Methode zu coarse-grained gewesen sein, um den Informationsgehalt der Schichten sinnvoll zu messen.

Alles in allem konnte diese Thesis mithilfe der relativen Entropie mehr Licht in ein neuronales Netzwerk bringen. Obwohl neue Erkenntnisse darüber gewonnen wurden, wie sich Information in einem tiefen neuronalen Netzwerke ausbreitet und wie die relative Entropie zur Messung der Informationsausbreitung verwendet werden kann, konnte nicht verstanden werden, wie Information aus einem Eingangsbild so verarbeitet wird, dass das Netzwerk lernt, dieses zu erkennen oder zu rekonstruieren. Deswegen wäre der nächste Schritt, zu untersuchen, ob es möglich ist zu zeigen, dass Deep Learning irrelevante Informationen im Hinblick auf die Rekonstruktion oder Klassifizierung entfernt. Da wir dafür womöglich eine Messmethode benötigen, die fine-grained ist, braucht es neue Ideen!

References

[1] Ergün Akgün and Metin Demir. Modeling course achievements of elementary education teacher candidates with artificial neural networks. https://www.researchgate.net/publication/326417061_Modeling_Course_Achievements_of_Elementary_Education_Teacher_Candidates_with_Artificial_Neural_Networks, 2018.

[2] Bagheri. An introduction to deep feedforward neural networks. https://towardsdatascience.com/an-introduction-to-deep-feedforward-neural-networks-1af281e306cd, 2020.

[3] Yasaman Bahri, Jonathan Kadmon, Jeffrey Pennington, Sam S. Schoenholz, Jascha Sohl-Dickstein, and Surya Ganguli. Statistical mechanics of deep learning. https://doi.org/10.1146/annurev-conmatphys-031119-050745, 2020.

[4] Ami Citri and Robert Malenka. Synaptic plasticity: Multiple forms, functions, and mechanisms. https://doi.org/10.1038/sj.npp.1301559, 2008.

[5] Johanna Erdmenger. Lecture notes: Statistical mechanics and thermodynamics, 2020.

[6] Johanna Erdmenger, Kevin T. Grosvenor, and Ro Jefferson. Towards quantifying information flows: relative entropy in deep neural networks and the renormalization group. https://arxiv.org/abs/2107.06898.

[7] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. https://arxiv.org/abs/1509.01240, 2015.

[8] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. https://www.jstor.org/stable/2334940, 1970.

[9] Egan Talmage Hemmings, Hugh. Pharmacology and physiology for anesthesia e-book: Foundations and clinical application. ISBN9781455737932, 2012.

[10] Felix Leditzky. Relative entropies and their use in quantum information theory. https://arxiv.org/abs/1611.08802, 2016.

[11] Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets. http://arxiv.org/abs/1712.09913, 2017.

[12] Andrew Ng. Optimization algorithms: Mini-batch gradient descent. https://cs230.stanford.edu/files/C2M2.pdf, 2017.

[13] Andrew Ng Stanford University. cs229-notes2. https://cs229.stanford.edu/lectures-spring2022/main_notes.pdf, 2022.

[14] Andrew Ng Stanford University. Lecture notes on sparse autoencoders. https://cs229.stanford.edu/lectures-spring2022/main_notes.pdf, 2022.

[15] Laurin Pannullo. The 2d ising model. https://itp.uni-frankfurt.de/~mwagner/teaching/C_WS19/projects/Ising_proj.pdf, 2020.

[16] Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. https://arxiv.org/abs/1606.05340, 2016.

[17] Daniel A. Roberts, Sho Yaida, and Boris Hanin. The principles of deep learning theory. https://arxiv.org/abs/2106.10165, 2021.

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms. https://arxiv.org/abs/1609.04747, 2017.

[19] Peter Sadowski University of California Irvine. Notes on backpropagation. https://www.ics.uci.edu/~pjsadows/notes.pdf, 2021.

[20] Samuel S. Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. https://arxiv.org/abs/1611.01232, 2016.

[21] C. E. Shannon. A mathematical theory of communication, 1948.

[22] Volodymyr Turchenko, Eric Chalmers, and Artur Luczak. A deep convolutional auto-encoder with pooling - unpooling layers in caffe. https://arxiv.org/pdf/1701.04949.pdf, 2017.

[23] V. Vedral. The role of relative entropy in quantum information theory. https://doi.org/10.1103%2Frevmodphys.74.197, 2002.

[24] Bao Wang, Hedi Xia, Tan Nguyen, and Stanley Osher. How does momentum benefit deep neural networks architecture design? https://arxiv.org/abs/2110.07034, 2021.

[25] Lilian Witthauer and Manuel Dieterle. The phase transition of the 2d-ising model. https://www.researchgate.net/publication/265359289_The_Phase_Transition_of_the_2D-Ising_Model, 2022.

A Appendix

A.1 Linear activation function

To demonstrate that linear activation functions result in a network that behaves like a single-layer network, we consider a multilayer perceptron consisting entirely of layers with linear activation functions. We first concentrate on the layer $[L-2]$ which can be described by

$$a_{i^{[L-2]}}^{[L-2]} = g(w_{i^{[L-2]},k^{[L-3]}}^{[L-2]} a_{k^{[L-3]}}^{[L-3]} + b_{i^{[L-2]}}^{[L-2]}), \tag{A.1}$$

where g is a linear function with $z_{i^{[L-2]}}^{[L-2]}$ as its input. Since g is linear, we can write (2.10) as

$$a_{i^{[L-2]}}^{[L-2]} = c z_{i^{[L-2]}}^{[L-2]}, \tag{A.2}$$

where c is an arbitrary constant. We then express $a_{i^{[L-1]}}^{[L-1]}$ in terms of $a_{i^{[L-2]}}^{[L-2]}$. Consequently,

$$\begin{aligned} a_{i^{[L-1]}}^{[L-1]} &= c(w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} a_{k^{[L-2]}}^{[L-2]} + b_{i^{[L-1]}}^{[L-1]}) \\ &= c(w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} c(w_{i^{[L-2]},k^{[L-3]}}^{[L-2]} a_{k^{[L-3]}}^{[L-3]} + b_{i^{[L-2]}}^{[L-2]}) + b_{i^{[L-1]}}^{[L-1]}) \\ &= c^2(w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} w_{i^{[L-2]},k^{[L-3]}}^{[L-2]} a_{k^{[L-3]}}^{[L-3]} + (w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} b_{i^{[L-2]}}^{[L-2]} + b_{i^{[L-1]}}^{[L-1]})). \end{aligned} \tag{A.3}$$

We define the activation of the last layer as

$$\hat{y}_{i^{[L-1]}}^{(j)} := a^{[L-1]} = c^2(w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} w_{i^{[L-2]},k^{[L-3]}}^{[L-2]} a_{k^{[L-3]}}^{[L-3]} + (w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} b_{i^{[L-2]}}^{[L-2]} + b_{i^{[L-1]}}^{[L-1]})), \tag{A.4}$$

where $\hat{y}_{i^{[L-1]}}^{(j)}$ is the $i^{[L-1]}$-th element of the output vector $\hat{\vec{y}}^{(j)}$. This is the network's prediction given the feature vector $\vec{x}^{(j)}$ which consists of $x_{i^{[0]}}^{(j)} := a_{i^{[0]}}^{[0]}$. We see that we can merge layers $[L-2]$ and $[L-1]$ into one layer while keeping the activation linear. This is possible since a constant to the n-th power remains a constant. The weight matrix of this new layer is $w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} w_{i^{[L-2]},k^{[L-3]}}^{[L-2]}$ and its bias vector is $w_{i^{[L-1]},k^{[L-2]}}^{[L-1]} b_{i^{[L-2]}}^{[L-2]} + b_{i^{[L-1]}}^{[L-1]}$. Consequently, a neural network consisting only of linear activations can be merged into one linear layer. Hence, the whole network behaves like a single-layer neural network.

A.2 Derivation of the responsibility terms θ and δ

A.2.1 Responsibility term θ for the output layer

The responsibility term θ measures the effect that a change in the activation of the output layer neuron has on the loss function. This term is the partial derivative of the cross-entropy loss function $\mathcal{L}$ with respect to the activation of the $i-$th neuron in the output layer

$$\theta_{i^{[L-1]}}^{[L-1](j)} := \frac{\partial \mathcal{L}(\hat{\vec{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}}. \tag{A.5}$$

The use of a softmax (2.15) in the output layer means that the activations of individual neurons are interdependent. This together with the derivative of the logarithm leads us to

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)} \frac{\frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial a_{i^{[L-1]}}^{[L-1](j)}}}{\hat{y}_{k^{[L-1]}}^{(j)}} \text{ with } k \in \{0, ..., N^{[L-1]} - 1\}, \tag{A.6}$$

Since we are considering the output layer, we can say that $a_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{[L-1](j)}$, which allows us to rewrite (A.6) as

$$\begin{aligned}
\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} &= -\frac{y_{i^{[L-1]}}^{(j)} \partial \hat{y}_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)} \partial \hat{y}_{i^{[L-1]}}^{(j)}} - \sum_{k^{[L-1]} \neq i^{[L-1]}} \frac{y_{k^{[L-1]}}^{(j)}}{\hat{y}_{k^{[L-1]}}^{(j)}} \frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial \hat{y}_{i^{[L-1]}}^{(j)}} \\
&= -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} - \sum_{k^{[L-1]} \neq i^{[L-1]}} \frac{y_{k^{[L-1]}}^{(j)}}{\hat{y}_{k^{[L-1]}}^{(j)}} \frac{\frac{d\hat{y}_{i^{[L-1]}}^{(j)}}{dz_{i^{[L-1]}}^{[L-1]}}}{\frac{d\hat{y}_{k^{[L-1]}}^{(j)}}{dz_{i^{[L-1]}}^{[L-1]}}}.
\end{aligned} \tag{A.7}$$

For the derivative of the softmax $\hat{y}_{k^{[L-1]}}^{(j)}$ with respect to $z_{i^{[L-1]}}^{[L-1]}$ in (A.7), we need to distinguish between two cases.
For $k = i$, all $z_{l^{[L-1]}}^{[L-1]}$ with $l \neq i$ are a constant and therefore their derivative vanishes. Thus,

$$\begin{aligned}
\frac{d\hat{y}_{k^{[L-1]}}}{dz_{i^{[L-1]}}^{[L-1]}} &= \frac{e^{z_{k^{[L-1]}}^{[L-1]}} (\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}} - e^{z_{k^{[L-1]}}^{[L-1]}})}{(\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}})^2} \\
&= \frac{e^{z_{k^{[L-1]}}^{[L-1]}}}{\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}}} \left(1 - \frac{e^{z_{k^{[L-1]}}^{[L-1]}}}{\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}}}\right) \\
&= \hat{y}_{k^{[L-1]}}(1 - \hat{y}_{k^{[L-1]}}) \\
&= \hat{y}_{i^{[L-1]}}(1 - \hat{y}_{i^{[L-1]}}).
\end{aligned} \tag{A.8}$$

For $k \neq i$ we get

$$\begin{aligned}
\frac{d\hat{y}_{k^{[L-1]}}}{dz_{i^{[L-1]}}^{[L-1]}} &= -\frac{e^{z_{k^{[L-1]}}^{[L-1]}} e^{z_{i^{[L-1]}}^{[L-1]}}}{(\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}})^2} \\
&= -\frac{e^{z_{k^{[L-1]}}^{[L-1]}} e^{z_{i^{[L-1]}}^{[L-1]}}}{\sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}} \sum_{l^{[L-1]}}^{N^{[L-1]}} e^{z_{l^{[L-1]}}^{[L-1]}}} \\
&= -\hat{y}_{k^{[L-1]}} \hat{y}_{i^{[L-1]}}.
\end{aligned} \tag{A.9}$$

The equations (A.8) and (A.9) can be summarized into

$$\frac{d\hat{y}_{k^{[L-1]}}}{dz_{i^{[L-1]}}^{[L-1]}} = \begin{cases} \hat{y}_{i^{[L-1]}}(1 - \hat{y}_{i^{[L-1]}}), & \text{if } k = i \\ -\hat{y}_{k^{[L-1]}} \hat{y}_{i^{[L-1]}}, & \text{if } k \neq i \end{cases}, \tag{A.10}$$

and used to simplify (A.7) into

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} - \sum_{k^{[L-1]} \neq i^{[L-1]}} \frac{y_{k^{[L-1]}}^{(j)}}{\hat{y}_{k^{[L-1]}}^{(j)}} \frac{-\hat{y}_{k^{[L-1]}}^{(j)} \hat{y}_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}(1 - \hat{y}_{i^{[L-1]}}^{(j)})}$$
$$= -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} + \frac{1}{(1 - \hat{y}_{i^{[L-1]}}^{(j)})} \sum_{k^{[L-1]} \neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)}. \tag{A.11}$$

Recalling that $y_{k^{[L-1]}}^{(j)}$ is an one-hot encoded label vector's element, we rewrite the sum in (A.11) as

$$\sum_{k^{[L-1]} \neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)} = 1 - y_{i^{[L-1]}}^{(j)}. \tag{A.12}$$

This is possible because only one element is 1 while the others are 0. Substituting this into (A.11) yields the final expression for the responsibility term θ of the output layer

$$\theta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[L-1]}}^{[L-1](j)}} = -\frac{y_{i^{[L-1]}}^{(j)}}{\hat{y}_{i^{[L-1]}}^{(j)}} + \frac{1 - y_{i^{[L-1]}}^{(j)}}{(1 - \hat{y}_{i^{[L-1]}}^{(j)})}. \tag{A.13}$$

This equation is equation (222) from [2].

A.2.2 Responsibility term δ for the output layer

The responsibility term δ measures the effect that a change in the pre-activation of the $i-$th output layer neuron has on the loss function. It is the partial derivative of the loss function with respect to the pre-activation of the $i-$th output layer neuron

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[L-1]}}^{[L-1](j)}}. \tag{A.14}$$

Since a change in pre-activation changes the activation of a neuron and knowing that a change in activation of a neuron in the output layer affects the loss function (A.13) we use the chain rule to write δ in terms of (A.5)

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[L-1]}}^{[L-1](j)}} = \sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \hat{y}_{k^{[L-1]}}^{(j)}} \frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial z_{i^{[L-1]}}^{[L-1](j)}} \text{ with } k \in \{0, ..., N^{[L-1]} - 1\}. \tag{A.15}$$

Again, $a_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{[L-1](j)}$ holds. Then using the cross-entropy loss function in combination with the softmax allows us to rewrite (A.15) as

$$\delta_{i^{[L-1]}}^{[L-1](j)} = -\sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)} \frac{\frac{\partial \hat{y}_{k^{[L-1]}}^{(j)}}{\partial z_{i^{[L-1]}}^{[L-1](j)}}}{\hat{y}_{k^{[L-1]}}^{(j)}}. \tag{A.16}$$

With (A.10) we then get

$$\begin{aligned}
\delta_{i^{[L-1]}}^{[L-1](j)} &= -y_{i^{[L-1]}}^{(j)} \frac{\hat{y}_{i^{[L-1]}}^{(j)}(1-\hat{y}_{i^{[L-1]}}^{(j)})}{\hat{y}_{i^{[L-1]}}^{(j)}} + \sum_{k^{[L-1]}\neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)} \frac{\hat{y}_{k^{[L-1]}}^{(j)}\hat{y}_{i^{[L-1]}}^{(j)}}{\hat{y}_{k^{[L-1]}}^{(j)}} \\
&= -y_{i^{[L-1]}}^{(j)}(1-\hat{y}_{i^{[L-1]}}^{(j)}) + \sum_{k^{[L-1]}\neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)}\hat{y}_{i^{[L-1]}}^{(j)} \\
&= -y_{i^{[L-1]}}^{(j)} - y_{i^{[L-1]}}^{(j)}\hat{y}_{i^{[L-1]}}^{(j)} + \sum_{k^{[L-1]}\neq i^{[L-1]}} y_{k^{[L-1]}}^{(j)}\hat{y}_{i^{[L-1]}}^{(j)} \\
&= -y_{i^{[L-1]}}^{(j)} + \hat{y}_{i^{[L-1]}}^{(j)} \sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)}.
\end{aligned} \tag{A.17}$$

Since $y_{k^{[L-1]}}^{(j)}$ is one of the one-hot encoded label vector's elements, we simplify the sum in (A.17) into

$$\sum_{k^{[L-1]}=0}^{N^{[L-1]}-1} y_{k^{[L-1]}}^{(j)} = 1, \tag{A.18}$$

which allows us to rewrite (A.17) into the final expression for the responsibility term δ of the output layer

$$\delta_{i^{[L-1]}}^{[L-1](j)} = \hat{y}_{i^{[L-1]}}^{(j)} - y_{i^{[L-1]}}^{(j)}. \tag{A.19}$$

This equation is equation (210) from [2].

A.2.3 Responsibility term θ for a hidden layer

The responsibility term θ for the $i-$th neuron in the $l-$th layer for a single training sample (j)

$$\theta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[l]}}^{[l](j)}}, \tag{A.20}$$

measures the effect that a change in activation of an arbitrary neuron in a hidden layer has on the loss function. To derive this term we consider θ for a neuron in layer $[l-1]$. By using the chain rule, we write (A.20) in terms of the pre-activation of layer $[l]$. Thus,

$$\theta_{i^{[l-1]}}^{[l-1](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}} \frac{\partial z_{k^{[l]}}^{[l](j)}}{\partial a_{i^{[l-1]}}^{[l-1](j)}}. \tag{A.21}$$

According to (2.9) and (2.10), we can write $z_{k^{[l]}}^{[l](j)}$ in terms of the activation of layer $[l-1]$. Thus, the derivative with respect to $a_{i^{[l-1]}}^{[l-1](j)}$ becomes

$$\frac{\partial z_{k^{[l]}}^{[l](j)}}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = \frac{\partial (\sum_{i^{[l-1]}=0}^{N^{[l-1]}-1} w_{k^{[l]}i^{[l-1]}}^{[l]} a_{i^{[l-1]}}^{[l-1](j)} + b_{k^{[l]}}^{[l]})}{\partial a_{i^{[l-1]}}^{[l-1](j)}} = w_{k^{[l]}i^{[l-1]}}^{[l]}, \tag{A.22}$$

which we substitute into (A.21). Hence

$$\theta_{i^{[l-1]}}^{[l-1](j)} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}} w_{k^{[l]}i^{[l-1]}}^{[l]} = \sum_{k^{[l]}=0}^{N^{[l]}-1} [(W^{[l]})^T]_{ik} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l]}}^{[l](j)}}, \tag{A.23}$$

where $[(W^{[l]})^T]_{ik}$ is the element ik of the transposed weight matrix of layer $[l]$. We see that we can substitute the partial derivative in (A.23) by the definition of δ (A.14). We then write (A.23) as

$$\theta_{i^{[l-1]}}^{[l-1](j)} = [(W^{[l]})^T \vec{\delta}^{[l](j)}]_i, \tag{A.24}$$

with

$$\vec{\delta}^{[l](j)} = \begin{bmatrix} \delta_{0^{[l]}}^{[l](j)} \\ \delta_{1^{[l]}}^{[l](j)} \\ \vdots \\ \delta_{N^{[l]}-1}^{[l](j)} \end{bmatrix}, \tag{A.25}$$

Note that the result of multiplying the transpose of the weight matrix by $\vec{\delta}^{[l](j)}$ is a vector and therefore (A.24) is the $i-$th element of this vector. Vectorizing (A.24), yields the final expression for the responsibility term θ for a hidden layer

$$\vec{\theta}^{[l-1](j)} = \frac{\mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l-1](j)}} = (W^{[l]})^T \vec{\delta}^{[l](j)}. \tag{A.26}$$

This equation is equation (215) from [2].

A.2.4 Responsibility term δ for a hidden layer

The δ for an arbitrary layer $[l]$ prior to the output layer is written in terms of the pre-activation of the subsequent by using the chain rule

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{k^{[l+1]}}^{[l+1](j)}} \frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}} \text{ with } k^{[l+1]} \epsilon \{0, ..., N^{[l+1]-1}\}. \tag{A.27}$$

Here we recognize the definition of the δ for the layer $[l+1]$. Thus,

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} \delta_{i^{[l+1]}}^{[l+1](j)} \frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}}. \tag{A.28}$$

The equations (2.9) and (2.10), allow us to write the term $z_{l^{[l+1]}}^{[l+1](j)}$ in (A.28) as

$$z_{k^{[l+1]}}^{[l+1](j)} = \sum_{i^{[l]}=0}^{N^{[l]}-1} w_{k^{[l+1]}i^{[l]}}^{[l+1]} \sigma_T(z_{k^{[l]}}^{[l](j)}) + b_{k^{[l+1]}}^{[l+1]}, \tag{A.29}$$

and since the pre-activations of different neurons in the same hidden layer are independent of each other, we write the derivative in (A.28) as

$$\frac{\partial z_{k^{[l+1]}}^{[l+1](j)}}{\partial z_{i^{[l]}}^{[l](j)}} = w_{k^{[l+1]}i^{[l]}}^{[l+1]} \sigma_T'(z_{i^{[l]}}^{[l](j)}). \tag{A.30}$$

Here $\sigma_T'(z_{i^{[l]}}^{[l](j)})$ is the derivative of $\sigma_T(z_{i^{[l]}}^{[l](j)})$ with respect to $z_{i^{[l]}}^{[l](j)}$. If we then substitute (A.30) into (A.28), we get

$$\begin{aligned} \delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} &= \sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} w_{k^{[l+1]}i^{[l]}}^{[l+1]} \delta_{i^{[l+1]}}^{[l+1](j)} \sigma_T'(z_{i^{[l]}}^{[l](j)}) \\ &= \left(\sum_{k^{[l+1]}=0}^{N^{[l+1]}-1} w_{k^{[l+1]}i^{[l]}}^{[l+1]} \delta_{i^{[l+1]}}^{[l+1](j)} \right) \sigma_T'(z_{i^{[l]}}^{[l](j)}), \end{aligned} \tag{A.31}$$

which we can bring into vector form using the Hadamard product. Consequently,

$$\vec{\delta}^{[l](j)} = ((W^{[l+1]})^T \vec{\delta}^{[l+1](j)}) \odot \sigma_T'(\vec{z}^{[l](j)}), \tag{A.32}$$

which is the delta vector for layer $[l]$ as a function of the delta vector of the subsequent layer. In the last step, we use (A.26) and obtain the final expression for δ of an arbitrary hidden layer

$$\vec{\delta}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial \vec{a}^{[l](j)}} \odot \sigma_T'(\vec{z}^{[l](j)}). \tag{A.33}$$

This equation is equation (218) from [2].

A.2.5 Responsibility term δ for biases

To compute how a change in a bias of layer $[l]$ affects the loss function, we use the chain rule and write $\delta_{i^{[l]}}^{[l](j)}$ in terms of the bias

$$\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} = \sum_{k^{[l]}=0}^{N^{[l]}-1} \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b_{k^{[l]}}^{[l](j)}} \frac{\partial b_{k^{[l]}}^{[l](j)}}{\partial z_{i^{[l]}}^{[l](j)}}. \tag{A.34}$$

Then, using (2.9) together with (2.10) for a neuron $k^{[l]}$ allows us to write

$$b_{k^{[l]}}^{[l]} = z_{k^{[l]}}^{[l](j)} - \sum_{p^{[l-1]}} w_{k^{[l]}p^{[l-1]}}^{[l]} a_{p^{[l-1]}}^{[l-1](j)}. \tag{A.35}$$

The partial derivative of the bias with respect to the pre-activation in (A.34) can be written as

$$\frac{\partial b_{k^{[l]}}^{[l]}}{\partial z_{i^{[l]}}^{(j)}} = 0 \text{ for k} \neq \text{i}, \tag{A.36}$$

and

$$\frac{\partial b_{i^{[l]}}^{[l]}}{\partial z_{i^{[l]}}^{(j)}} = 1 \text{ for k = i.} \tag{A.37}$$

With (A.37) we can now rewrite (A.34) into

$$\begin{aligned}\delta_{i^{[l]}}^{[l](j)} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b_{i^{[l]}}^{[l](j)}} \frac{\partial b_{i^{[l]}}^{[l](j)}}{\partial z_{i^{[l]}}^{[l](j)}} &= \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b_{i^{[l]}}^{[l](j)}} \frac{\partial b_{i^{[l]}}^{[l](j)}}{\partial z_{i^{[l]}}^{[l](j)}} \\ &= \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial b_{i^{[l]}}^{[l](j)}} \text{ for k = i.}\end{aligned} \tag{A.38}$$

Finally, we vectorize everything and get equation (182) from [2]

$$\vec{\nabla}_{\vec{b}^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)}. \tag{A.39}$$

So, by computing the delta vector for an arbitrary layer $[l]$ via (A.33), we obtain the gradient of the loss function with respect to the bias vector of an arbitrary layer $[l]$ prior to the output layer.

A.2.6 Responsibility term δ for weights

To compute how a change in a weight of layer $[l]$ changes the loss function, we again use the chain rule and write δ for layer $[l]$ in terms of the weights. Hence,

$$\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial w_{i^{[l]},k^{[l-1]}}^{[l](j)}} = \frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial z_{i^{[l]}}^{[l](j)}} \frac{\partial z_{i^{[l]}}^{[l](j)}}{\partial w_{i^{[l]},k^{[l-1]}}^{[l](j)}} = \delta_{i^{[l]}}^{[l](j)} \frac{\partial z_{i^{[l]}}^{[l](j)}}{\partial w_{i^{[l]},k^{[l-1]}}^{[l](j)}}. \tag{A.40}$$

The derivative of the pre-activation with respect to the weights yields

$$\frac{\partial z_{i^{[l]}}^{[l](j)}}{\partial w_{i^{[l]},k^{[l-1]}}^{[l](j)}} = a_{k^{[l-1]}}^{[l-1](j)}, \tag{A.41}$$

considering that the pre-activation is defined as (2.9). Then, we can rewrite (A.40) using (A.41) as

$$\frac{\partial \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})}{\partial w_{i^{[l]},k^{[l-1]}}^{[l](j)}} = a_{k^{[l-1]}}^{[l-1](j)} \delta_{i^{[l]}}^{[l](j)}. \tag{A.42}$$

Expressing this equation in vector form results in the gradient of the loss function with respect to the weight matrix

$$\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)}) = \vec{\delta}^{[l](j)} (\vec{a}^{[l-1](j)})^T, \tag{A.43}$$

where $\vec{\nabla}_{W^{[l]}} \mathcal{L}(\vec{\hat{y}}^{(j)}, \vec{y}^{(j)})$ is an $N^l \times N^{l-1}$ matrix, since $W^{[l]}$ is an $N^l \times N^{l-1}$ matrix. On the right hand side of (A.43), the vector $\vec{\delta}_{i^{[l]}}^{[l](j)}$ has N^l elements and the transposed activation vector has $N^{[l-1]}$ elements. This expression is the responsibility term δ for weights of an arbitrary layer prior to the output layer and can be found in [2] as equation (188).

A.3 Derivation of the relative entropy for a multilayer perceptron

A.3.1 Relative entropy for Gaussian distributions

We consider the normal distributions p and q, which describe the different layers of the neural network in the mean field limit

$$p(x) = \frac{1}{\sqrt{2\pi\sigma_p^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_p}{\sigma_p}\right)^2} \text{ and } q(x) = \frac{1}{\sqrt{2\pi\sigma_q^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2}. \tag{A.44}$$

Here x is a continuous random variable. The distributions $p(x)$ and $q(x)$ potentially have different means and standard deviations. Using these Gaussians leads us to the continuous form of the relative entropy

$$D(p||q) = \int dx\, p(x) \ln\left(\frac{p(x)}{q(x)}\right), \tag{A.45}$$

which can be rewritten into

$$D(p||q) = \int dx\, p(x) \ln\left(\frac{p(x)}{q(x)}\right) = \langle \ln(p(x)) \rangle_p - \langle \ln(q(x)) \rangle_p, \tag{A.46}$$

via the definition of the expectation value

$$\langle x \rangle_f = \int dx\, x f(x). \tag{A.47}$$

Here $f(x)$ is the probability density function, which in our case is $p(x)$ or $q(x)$. The definition of the expectation value also allows us to rewrite the classical Shannon entropy into

$$S(p) = -\langle \ln(p(x)) \rangle_p, \tag{A.48}$$

which we substitute into (A.46) such that we can write

$$D(p||q) = -S(p(x)) - \langle \ln(q(x)) \rangle_p. \tag{A.49}$$

We simplify the term $-S(p(x))$ by plugging in the probability density functions $p(x)$ and $q(x)$. Hence,

$$\begin{aligned} -S(p(x)) &= \langle \ln(p(x)) \rangle_p \\ &= \langle -\frac{1}{2}\left(\frac{x-\mu_p}{\sigma_p}\right)^2 - \ln\left(\sqrt{2\pi\sigma_p^2}\right) \rangle_p \\ &= -\frac{1}{2\sigma_p^2}\langle (x-\mu_p)^2 \rangle_p - \frac{1}{2}\langle \ln(2\pi\sigma_p^2) \rangle_p \\ &= -\frac{1}{2\sigma_p^2}\langle (x-\mu_p)^2 \rangle_p - \frac{1}{2}\ln(2\pi\sigma_p^2), \end{aligned} \tag{A.50}$$

where the term $\langle (x-\mu_p)^2 \rangle_p$ can be simplified by the definition of the variance

$$\sigma_p^2 = \langle x^2 \rangle_p - \langle x \rangle_p^2 \Leftrightarrow \langle x^2 \rangle_p = \sigma_p^2 + \langle x \rangle_p^2. \tag{A.51}$$

Thus,

$$\begin{aligned} \langle (x-\mu_p)^2 \rangle_p &= \langle x^2 - 2\mu_p x + \mu_p^2 \rangle_p \\ &= \langle x^2 \rangle_p - 2\mu_p \langle x \rangle_p + \mu_p^2 \\ &= \sigma_p^2 + \mu_p^2 - 2\mu_p^2 + \mu_p^2 \\ &= \sigma_p^2. \end{aligned} \tag{A.52}$$

Substituting this term into (A.50), we get

$$\begin{aligned} -S(p(x)) = \langle \ln(p(x)) \rangle_p &= -\frac{1}{2} - \frac{1}{2}\ln(2\pi\sigma_p^2) \\ &= -\frac{1}{2}(1 + \ln(2\pi\sigma_p^2)) \\ &= -\frac{1}{2}(\ln(e) + \ln(2\pi\sigma_p^2)) \\ &= -\frac{1}{2}\ln(2\pi\sigma_p^2 \mathrm{e}). \end{aligned} \tag{A.53}$$

Analogously, we rewrite the second term in (A.49). Consequently,

$$\begin{aligned} \langle \ln(q(x)) \rangle_p &= \langle -\frac{1}{2}\left(\frac{x-\mu_q}{\sigma_q}\right)^2 - \ln\left(\sqrt{2\pi\sigma_q^2}\right) \rangle_p \\ &= -\frac{1}{2\sigma_q^2}\langle (x-\mu_q)^2 \rangle_p - \frac{1}{2}\langle \ln(2\pi\sigma_q^2) \rangle_p \\ &= -\frac{1}{2\sigma_q^2}\langle (x-\mu_q)^2 \rangle_p - \frac{1}{2}\ln(2\pi\sigma_q^2), \end{aligned} \tag{A.54}$$

where the term $\langle (x-\mu_q)^2 \rangle_p$ can be simplified into

$$\begin{aligned} \langle (x-\mu_q)^2 \rangle_p &= \langle x^2 - 2\mu_q x + \mu_q^2 \rangle_p \\ &= \langle x^2 \rangle_p - 2\mu_q \langle x \rangle_p + \mu_q^2 \\ &= \langle x^2 \rangle_p - 2\mu_q \mu_p + \mu_q^2 \\ &= \sigma_p^2 + \langle x \rangle_p^2 - 2\mu_q \mu_p + \mu_q^2 \\ &= \sigma_p^2 + (\mu_p - \mu_q)^2. \end{aligned} \tag{A.55}$$

Substituting this into (A.54) gives us

$$\langle \ln(q(x)) \rangle_p = -\frac{1}{2\sigma_q^2}(\sigma_p^2 + (\mu_p - \mu_q)^2) - \frac{1}{2}\ln(2\pi\sigma_q^2). \tag{A.56}$$

Then substituting (A.50) and (A.56) into (A.49), we obtain the relative entropy for Gaussian distributions

$$D(p||q) = -\frac{1}{2}\ln(2\pi e\sigma_p^2) + \frac{1}{2\sigma_q^2}(\sigma_p^2 + (\mu_p - \mu_q)^2) + \frac{1}{2}\ln(2\pi\sigma_q^2). \tag{A.57}$$

Here $D(p||q) = 0$ if $q = p$

$$D(p||q) = -\frac{1}{2}\ln(2\pi e\sigma_p^2) + \frac{1}{2}\ln(2\pi\sigma_q^2) + \frac{1}{2} = 0, \tag{A.58}$$

since $\mu_p = \mu_q$.

A.3.2 Relative entropy for multivariate Gaussians

For the multidimensional relative entropy, we consider the multivariate Gaussians

$$p(\vec{x}) = \frac{1}{\sqrt{(2\pi)^N|\Sigma_p|}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu}_p)^T\Sigma_p^{-1}(\vec{x}-\vec{\mu}_p)} \text{ and } q(\vec{x}) = \frac{1}{\sqrt{(2\pi)^N|\Sigma_q|}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu}_q)^T\Sigma_q^{-1}(\vec{x}-\vec{\mu}_q)}, \tag{A.59}$$

with N-dimensional continuous random variable $\vec{x} = (x_1, ..., x_N)^T$, N-dimensional mean vector $\vec{\mu}_{p,q} = (\mu_1, ..., \mu_N)^T$, and $N \times N$ covariance matrix $\Sigma_{p,q}$. Since we consider a random network the covariance matrix is diagonal

$$\Sigma_{p,q} = \begin{pmatrix} \sigma_{p,q,1}^2 & & \\ & \ddots & \\ & & \sigma_{p,q,N}^2 \end{pmatrix}. \tag{A.60}$$

Consequently, we obtain the following determinant

$$|\Sigma_p| = \sigma_{p,q}^N. \tag{A.61}$$

We substitute these probability density functions into the relative entropy and get

$$D(p||q) = \int d\vec{x}\, p(\vec{x}) \ln\left(\frac{p(\vec{x})}{q(\vec{x})}\right) = \langle \ln(p(\vec{x}))\rangle_p - \langle \ln(q(\vec{x}))\rangle_p. \tag{A.62}$$

Again, we express the first term on the right-hand side of the equation through the Shannon entropy

$$D(p||q) = -S(p(\vec{x})) - \langle \ln(q(\vec{x}))\rangle_p. \tag{A.63}$$

Applying the trace trick and thinking of a scalar as a 1×1 matrix A, we can say $x^T Ax = tr(x^T Ax) = tr(xx^T A) = tr(Axx^T)$. Consequently,

$$S(p(\vec{x})) = \frac{1}{2}\ln(|2\pi e\Sigma_p|). \tag{A.64}$$

Since the covariance matrix is diagonal, the second term on the right side of the equation (A.62) can be rewritten into

$$\langle \ln(q(x)) \rangle_p = -\frac{1}{2} tr(\Sigma_q^{-1}[\Sigma_p + (\mu_p - \mu_q)^2]) - \frac{1}{2}\ln(|2\pi\Sigma_q|), \tag{A.65}$$

using the definition of the variance together with the summation notation for the exponent of the probability density function

$$\begin{aligned}
\langle (\vec{x} - \vec{\mu}_q)^T \Sigma_q^{-1} (\vec{x} - \vec{\mu}_q) \rangle_p &= \sum_{i=1}^{N} [\Sigma_q^{-1}]_{ii} \langle (z_i - \mu_{q,i})^2 \rangle_p \\
&= \sum_{i=1}^{N} [\Sigma_q^{-1}]_{ii} ([\Sigma_p]_{ii} + \mu_{p,i} - 2\mu_{p,i}\mu_{q,i} + \mu_{q,i}^2) \\
&= \sum_{i=1}^{N} [\Sigma_q^{-1}]_{ii} ([\Sigma_p]_{ii} + (\mu_{p,i} - \mu_{q,i})^2) \\
&= tr(\Sigma_q^{-1}[\Sigma_p + (\mu_p - \mu_q)^2]).
\end{aligned} \tag{A.66}$$

Substituting (A.64) and (A.65) into (A.63) yields the relative entropy for the multivariate case

$$D(p(\vec{x})||q(\vec{x})) = -\frac{1}{2}\ln(|2\pi e\Sigma_p|) + \frac{1}{2} tr(\Sigma_q^{-1}[\Sigma_p + (\mu_p - \mu_q)^2]) + \frac{1}{2}\ln(|2\pi\Sigma_q|). \tag{A.67}$$

A.3.3 Relative Entropy for a hierarchical network

The relative entropy of a hierarchical network is defined as

$$D(p(z^{[l]})||q(z^{[l+m]})) = \langle \ln(p(z^{[l]})) \rangle_p - \langle \ln(q(z^{[l+m]})) \rangle_p, \text{ where } m \in [0, N^{[l-1]} - 1]. \tag{A.68}$$

Considering only the case $m = 1$, we see that $z_{i^{[l+1]}}^{[l+1]}$ is a recursive function of $z_{i^{[l]}}^{[l]}$ given by equation (2.9). For clarity, we will now drop the superscripts and subscripts. Thus, we get

$$z^{[l+1]} = w^{[l+1]} \sigma_T(z^{[l]}) + b^{[l+1]}, \tag{A.69}$$

where $w^{[l+1]}$ is a $N^{[l+1]} \times N^{[l]}$ matrix, $\sigma_T(z^{[l]})$ is a $N^{[l]}$-dimensional vector, and $b^{[l+1]}$ is a $N^{[l+1]}$-dimensional vector. For $m = 1$ we get

$$D(p(z^{[l]})||q(z^{[l+1]})) = \langle \ln(p(z^{[l]})) \rangle_p - \langle \ln(q(z^{[l+1]})) \rangle_p, \tag{A.70}$$

with

$$p(z^{[l]}) = \frac{1}{\sqrt{(2\pi)^{N^{[l]}}|\Sigma_p|}} e^{-\frac{1}{2}(z^{[l]} - \mu_q^{[l]})^T \Sigma_p^{-1} (z^{[l]} - \mu_p^{[l]})}, \tag{A.71}$$

and

$$q(z^{[l+1]}) = \frac{1}{\sqrt{(2\pi)^{N^{[l+1]}}|\Sigma_q|}} e^{-\frac{1}{2}(z^{[l+1]} - \mu_q^{[l+1]})^T \Sigma_q^{-1} (z^{[l+1]} - \mu_q^{[l+1]})}. \tag{A.72}$$

Proceeding analogously to the multidimensional relative entropy, we obtain

$$\langle \ln(p(z^{[l]}))\rangle_p = -\frac{1}{2}\ln(|2\pi \mathrm{e}\Sigma_p^{[l]}|), \tag{A.73}$$

and

$$\langle \ln(q(z^{[l+1]}))\rangle_p = -\frac{1}{2}\langle (z^{[l+1]}-\mu_q^{[l+1]})^T\Sigma_q^{-1}(z^{[l+1]}-\mu_q^{[l+1]})\rangle_p - \frac{1}{2}\ln(|2\pi\Sigma_q^{[l+1]}|). \tag{A.74}$$

The first term on the right-hand side of the equation A.74 can again be simplified using the summation notation

$$\begin{aligned}
&\langle (z^{[l+1]}-\mu_q^{[l+1]})^T(\Sigma_q^{[l+1]})^{-1}(z^{[l+1]}-\mu_q^{[l+1]})\rangle_p \\
&= \sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii}\langle (z_i^{[l+1]}-\mu_{q,i}^{[l+1]})^2\rangle_p \\
&= \sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii}(\langle (z_i^{[l+1]})^2\rangle_p - 2\langle z_i^{[l+1]}\rangle_p \mu_{q,i}^{[l+1]}(\mu_{q,i}^{[l+1]})^2).
\end{aligned} \tag{A.75}$$

We see that we have two recursive terms inside this equation. Therefore, we try to find a useful recursion relation

$$\begin{aligned}
&\langle (z_i^{[l+1]})^2\rangle_p \\
=&\langle \sum_{j,k} w_{i,j}^{[l+1]}w_{i,k}^{[l+1]}\sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]})\rangle_p + 2\sum_j w_{i,j}^{[l+1]}\langle \sigma_T(z_j^{[l]})\rangle_p b_i^{[l+1]} + \langle (b_i^{[l+1]})^2\rangle_p \\
=&\sum_{j,k} w_{i,j}^{[l+1]}w_{i,k}^{[l+1]}\langle \sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]})\rangle_p + 2\sum_j w_{i,j}^{[l+1]}\langle \sigma_T(z_j^{[l]})\rangle_p b_i^{[l+1]} + (b_i^{[l+1]})^2,
\end{aligned} \tag{A.76}$$

where we define

$$\langle \sigma_T(z_j^{[l]})\rangle_p = \frac{1}{\sqrt{2\pi(\sigma_{p,j}^{[l]})^2}}\int dz_j^{[l]} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,j}^{[l]}}{\sigma_{p,j}^{[l]}}\right)^2}\sigma_T(z_j^{[l]}) =: f_j^{[0]}$$

$$\langle \sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]})\rangle_p = \frac{1}{2\pi\sigma_{p,j}^{[l]}\sigma_{p,k}^{[l]}}\int dz_j^{[l]}dz_k^{[l]} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,j}^{[l]}}{\sigma_{p,j}^{[l]}}\right)}\mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[l]}-\mu_{p,k}^{[l]}}{\sigma_{p,k}^{[l]}}\right)}\sigma_T(z_j^{[l]})\sigma_T(z_k^{[l]}) =: f_{j,k}^{[0]}. \tag{A.77}$$

These two integrals have to be solved numerically. Using the recursion relation (A.76) together with the integrals (A.77), we now can rewrite (A.75) into

$$\begin{aligned}
&\sum_{i=1}^{N^{[l+1]}} [(\Sigma_q^{[l+1]})^{-1}]_{ii}[\sum_{j,k} w_{i,j}^{[l+1]}w_{i,k}^{[l+1]}f_{j,k}^0 + 2\sum_j w_{i,j}^{[l+1]}(b_i^{[l+1]}-\mu_{q,i}^{[l+1]})f_j^0 + (b_i^{[l+1]}-\mu_{q,i}^{[l+1]})^2] \\
&\qquad := \langle Q^{[l+1]}\rangle_p,
\end{aligned} \tag{A.78}$$

which we will call the Q-term. Substituting (A.78) into (A.74) and then (A.74) together with (A.73) into (A.70), we obtain

$$D(p(z^{[l]})||q(z^{[l+1]})) = -\frac{1}{2}\ln(|2\pi e\Sigma_p^{[l]}|) + \frac{1}{2}\ln(|2\pi\Sigma_q^{[l+1]}|) + \frac{1}{2}\langle Q^{[l+1]}\rangle_p. \tag{A.79}$$

For an arbitrary m, (A.78) becomes

$$\begin{aligned}\sum_{i=1}^{N^{l+m}}[(\Sigma_q^{[l+m]})^{-1}]_{ii}[\sum_{j,k} w_{i,j}^{[l+m]}w_{i,k}^{[l+m]}f_{j,k}^{[m-1]} + 2\sum_j w_{i,j}^{[l+m]}(b_i^{[l+m]} - \mu_{q,i}^{[l+m]})f_j^{[m-1]} \\ +(b_i^{[l+m]} - \mu_{q,i}^{[l+m]})^2] := \langle Q^{[l+m]}\rangle_p,\end{aligned} \tag{A.80}$$

with

$$\begin{aligned}f_j^{[m-1]} :=& \langle\sigma_T(z_j^{[l+m-1]})\rangle_p \\ =& \frac{1}{\sqrt{|2\pi\Sigma_p^{[l]}|}}\int dz^{[l]}\mathrm{e}^{-\frac{1}{2}(z^{[l]}-\mu_p^{[l]})^T(\Sigma_p^{[l]})^{-1}(z^{[l]}-\mu_p^{[l]})}\sigma_T(z_j^{[l+m-1]}),\end{aligned} \tag{A.81}$$

and

$$\begin{aligned}f_{j,k}^{[m-1]} :=& \langle\sigma_T(z_j^{[l+m-1]})\sigma_T(z_k^{[l+m-1]})\rangle_p \\ =& \frac{1}{\sqrt{|2\pi\Sigma_p^{[l]}|}}\int dz^{[l]}\mathrm{e}^{-\frac{1}{2}(z^{[l]}-\mu_p^{[l]})^T(\Sigma_p^{[l]})^{-1}(z^{[l]}-\mu_p^{[l]})}\sigma_T(z_j^{[l+m-1]})\sigma_T(z_k^{[l+m-1]}),\end{aligned} \tag{A.82}$$

where $dz^{[l]} := \prod_r dz_r^{[l]}$ such that we integrate over all vector components. We can further simplify (A.80) by using

$$\sum_{i=1}^{N^{[l+m]}}[(\Sigma_q^{[l+m]})^{-1}]_{ii} = \frac{1}{\sigma_q^2}. \tag{A.83}$$

Hence, (A.80) can be rewritten into

$$\begin{aligned}\frac{1}{\sigma_q^2}\left(\sum_{j,k}^{N^{[m-1]}} f_{j,k}^{[m-1]}\sum_i^{N^{[m]}} w_{i,j}^{[m]}w_{i,k}^{[m]} + 2\sum_j^{N^{[m-1]}} f_j^{[m-1]}\sum_i^{N^{[m]}} w_{i,j}^{[m]}(b_i^{[m]} - \mu_{q,i}^{[m]}) + \sum_i^{N^{[m]}}(b_i^{[m]} - \mu_{q,i}^{[m]})^2\right) \\ := \langle Q^{[l+m]}\rangle_p,\end{aligned} \tag{A.84}$$

and subsequently into

$$\begin{aligned}&\sigma_q^2\langle Q^{[l+m]}\rangle_p \\ &= \sum_{j,k}^{N^{[m-1]}} f_{j,k}^{[m-1]}\sum_i^{N^{[m]}} w_{i,j}^{[m]}w_{i,k}^{[m]} + 2\sum_j^{N^{[m-1]}} f_j^{[m-1]}\sum_i^{N^{[m]}} w_{i,j}^{[m]}(b_i^{[m]} - \mu_{q,i}^{[m]}) + \sum_i^{N^{[m]}}(b_i^{[m]} - \mu_{q,i}^{[m]})^2.\end{aligned} \tag{A.85}$$

Substituting this Q-term for the one in (A.79), and considering all m layers as possible target layers q, we get the final form of the relative entropy of a hierarchical network

$$D(p(z^{[l]})||q(z^{[l+m]})) = -\frac{1}{2}\ln(|2\pi e\Sigma_p^{[l]}|) + \frac{1}{2}\ln(|2\pi\Sigma_q^{[l+m]}|) + \frac{1}{2}\sigma_q^2\langle Q^{[l+m]}\rangle_p. \tag{A.86}$$

A.4 Python Implementation

The following code can be used for both a multilayer perceptron and a sparse autoencoder trained with the MNSIT data set.

A.4.1 Libraries, Data Loader, and GPU access for MNIST

```
# _____Import of machine learning and artificial intelligence libraries_____
import torch
import torchvision
# The torch.nn module provides us with the building blocks for creating and training neural networks
import torch.nn as nn
# The DataLoader class provides us with an iterable over a given data set such that we can iterate over the data set
from torch.utils.data import DataLoader
# The torchvision.transforms module gives us a set of composable image transformations
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# torch.device allows us to specify the device type responsible for loading a tensor into memory and responsible for storing our tensor during runtime
# We will use a GPU, which makes a significant speed difference when using large tensors
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# _____Settings_____
# We specify the size of the input images, which in the case of the MNSIT image is 28×28.
input_size = 28 * 28
# We specify between how many classes the neural network has to discriminate in order to solve the classification problem (see chapter 2.1.2)
# The MNIST data set has 10 mutually exclusive classes
num_classes = 10
# We specify the depth of the neural network
hidden_layers = 100
# We define the learning parameters
learning_rate = 0.00005
batch_size = 128
num_epochs = 10
momentumSGD = 0.8

# _____Load the MNIST data set_____
# The torchvision.datasets module allows us to use the pre-classified data set of handwritten digits called MNIST
# We transform the images to tensors
# We create two dataloaders
#The first is used to train the network, the second to test the neural network
#Through the latter, we will see if the network is able to generalize (see chapter 2.1.2.6)
train_data = torchvision.datasets.MNIST('MNIST', train=True, download=True, transform=torchvision.transforms.ToTensor())
test_data = torchvision.datasets.MNIST('MNIST', train=False, transform=torchvision.transforms.ToTensor())

# By using Shuffle=True, a random generator permutates the indices of all the image samples so that the images are shuffled in each epoch during the training
# Thus, we have 128 images arranged in random order in each epoch
# In chapter 2.1.2.6, we discussed how this will enhance the network's ability to generalize
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)
```

A.4.2 Libraries, Data Loader, and GPU access for Ising

```
# _____Import of machine learning and artificial intelligence libraries_____
import torch
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder
import os
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# _____Settings_____
input_size = 28 * 28
output_size = 400
hidden_layers = 6
learning_rate = 0.0000003
Learning_rateSGD = 0.001
batch_size = 6
num_epochs = 200
momentumSGD = 0.8

# _____Load spin configurations _____
# Accessing the training data set and the test data set of the spin configurations
test_dataset_path = '/content/IsingPlot/Test'
train_dataset_path = '/content/IsingPlot/Train'
```

```
26
27  # The images are resized and converted to a tensor
28  # We create one dataloader with training data and one with test data using the spin configuration data sets
29  transforms = transforms.Compose([transforms.Resize((28, 28)), transforms.ToTensor()])
30  train_dataset = datasets.ImageFolder(root=train_dataset_path, transform=transforms)
31  test_dataset = datasets.ImageFolder(root=test_dataset_path, transform=transforms)
32
33  train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
34  test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

A.4.3 Architecture and training of a multilayer perceptron

The following code was created in collaboration with Yanick Thurn.

```
1   ____________________________________________Multilayer perceptron architecture_____________________________________________
2   """
3   The class, NN, defines the blueprint of the multilayer perceptron
4   """
5   class NN:
6       """
7       This constructor initializes the instance attributes when an object of this class is created.
8       """
9       def __init__(self, N, variances, lr=learning_rate):
10          # Dimension that each layer of the network has
11          self.N = N
12          # Weight variance σ_w^2
13          self.variances = variances
14          # Learning rate
15          self.lr = lr
16
17          # The list, layers, stores the generated layers
18          layers = []
19          # Generation of Gaussian linear layers with weights and biases that are initialized according to a Gaussian
20          for i in range(len(N) - 1):
21              # We create Gaussian linear layers and append them to the layers list using the layer dimensions that were
                    stored in N
22              # These layers will have the pre-activations
23              layers.append(torch.nn.Linear(N[i], N[i + 1]))
24              # Using layers[-1].weight and layers[-1].bias together with torch.nn.Parameter we can access and modify the
                    parameters of each layer
25              # We initialize them according to a Gaussian and take for the biases the bias variance σ_b^2 as variance and
                    for the weights, the weight variance σ_w^2 scaled with the layer width as variance [6]
26              layers[-1].weight = torch.nn.Parameter(
27                  torch.normal(mean=torch.zeros((N[i + 1], N[i])),
28                               std=np.sqrt(variances[0] / N[i])))
29              layers[-1].bias = torch.nn.Parameter(
30                  torch.normal(mean=torch.zeros((N[i + 1])),
31                               std=np.sqrt(variances[1])))
32              # We run the pre-activations through activation functions by stacking a tanh layer on top of each linear
                    layer
33              layers.append(torch.nn.Tanh())
34
35          # We create the deep neural network nn.model by stacking the generated layers in the layers list in the order in
                which they were created (see chapter 2.1.1)
36          # We do this by using a sequential container called nn.Sequential
37          self.model = torch.nn.Sequential(
38              *layers)
39          # We pass the neural network to the GPU
40          self.model.to(device)
41          # For updating the neural network parameters self.model.parameters(), we use the Mini-Batch SGD algorithm (see
                chapter 2.1.2.4) with the given learning rate and momentum
42          self.optimizer = torch.optim.SGD(self.model.parameters(), lr=learning_rate, momentum=momentumSGD)
43
44  # ____________________________________________________Training loop_____________________________________________________
45      # We use the cross-entropy cost function as cost function (see chapter 2.1.2.4)
46      criterion = nn.CrossEntropyLoss()
47      '''
48      With this cost function, we train the network for a number of num_epochs using the test data set. As explained in
            chapter 2.1.2.4, we use the Mini-Batch SGD algorithm to minimize the discrepancy between the predictions of the
            network and the labels by updating the parameters of the network
49      '''
50      def train(self, train_loader, test_loader, num_epochs):
51          # Stores the discrepancies between the predictions of the network and the labels
52          losses = []
53          running_loss = 0.0
54          running_correct = 0
55          n_total_steps = len(train_loader)
56          # We train the network for a number of num_epochs (see Mini-Batch Stochastic Gradient Descent pseudo-code in
                chapter 2.1.2.4)
57          for epoch in range(num_epochs):
58              # We train the neural network with the training data set
59              self.model.train()
60              # We iterate through the images and the labels of these images in the train_loader (see Mini-Batch
                    Stochastic Gradient Descent pseudo-code in chapter 2.1.2.4))
61              for i, (images, labels) in enumerate(train_loader):
62                  # Images are converted to a vector (tensor) and passed to the GPU
63                  images = images.view(-1, input_size).to(device)
```

```
                labels = labels.to(device)

                # ________Forward pass_________
                # The information from the input images is forward propagated through the layers of the network so that
                      the network can make a prediction/output with the given parameter settings
                outputs = self.model(images)
                # Computation of the deviations between the predictions of the network for the given input images and
                      the labels of those images
                loss = self.criterion(outputs, labels)

                # ________Backward pass_________
                # We delete the values of the gradient, because we do not need them for the backward pass
                 self.optimizer.zero_grad()
                # Backpropagation of the deviations between the predictions of the network for the given input images
                      and the labels of those input images, so that the parameters can be updated by the Mini-Batch SGD
                      algorithm
                loss.backward()

                # Updating the parameters
                self.optimizer.step()

                # We print how well the neural network can classify images from the training data set
                      losses.append(loss.item())
                if (i + 1) % 100 == 0:
                    print(
                        f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1} of {n_total_steps}], Loss: {loss.item():.4f}')

# ____________________________________________________Evalutation_____________________________________________________
    '''
    The eval(self,test_loader) function tests whether the neural network is able to classify images from the test data set
          . This tests if the network is able to correctly classify unseen images and therefore if the network is able
         to generalize
    '''
    def eval(self, test_loader):
        # We use the test data set
        self.model.eval()
        # In the test phase we do not need to compute the gradients
        with torch.no_grad():
            n_correct = 0
            n_samples = 0
            # We iterate through the images and the labels of these images in the test_loader
            for images, labels in test_loader:
                # Images are converted to a vector (tensor) and passed to the GPU
                images = images.view(-1, input_size).to(device)
                labels = labels.to(device)
                 # ________Forward pass_________
                # The information from the input images is forward propagated through the layers of the network so that
                      the network can make a prediction with the post-training parameters settings
                outputs = self.model(images)

                # We track the number of samples and the number of correct classifications by checking whether the
                      prediction of the network matches the label
                _, predicted = torch.max(outputs.data, 1)
                n_samples += labels.size(0)
                n_correct += (predicted == labels).sum().item()
            # We print out how well the neural network can classify images from the test data set
            acc = 100.0 * n_correct / n_samples
            print(f'{acc} %')

'''
The function get_wide_network(input_dim,output_dim,layers,variances) defines how many layers the deep neural network has
      and also the dimension of each layer
'''
def get_wide_network(input_dim, output_dim, layers, variances):
    if layers < 3:
        raise Exception("Too few layers; minimum allowed depth is 3")
    # The first layer and the hidden layers have the dimension of the input image except for the last hidden layer,
         which has a width of 400 neurons
    # The output layer has as many neurons as the MNIST data set has classes, namely 10
    widths = [input_dim] * layers + [400, output_dim]
    return NN(widths, variances)
```

A.4.4 Architecture and training of a sparse autoencoder

```
# ____________________________________________Sparse autoencoder architecture_____________________________________________
class Autoencoder:
    def __init__(self, N_autoencoder, variances, lr=learning_rate):
        self.N_autoencoder = N_autoencoder
        self.variances = variances
        self.lr = lr

        layers_autoencoder = []
        for i in range(len(N_autoencoder) - 1):
            layers_autoencoder.append(torch.nn.Linear(N_autoencoder[i], N_autoencoder[i + 1]))
            layers_autoencoder[-1].weight = torch.nn.Parameter(
                torch.normal(mean=torch.zeros((N_autoencoder[i + 1], N_autoencoder[i])),
                             std=np.sqrt(variances[0] / N_autoencoder[i])))
            layers_autoencoder[-1].bias = torch.nn.Parameter(
```

```
                torch.normal(mean=torch.zeros((N_autoencoder[i + 1])), std=np.sqrt(variances[1])))
            layers_autoencoder.append(torch.nn.Tanh())

        self.model = torch.nn.Sequential(
            *layers_autoencoder)
        self.model.to(device)
        # Instead of using the Mini-Batch SGD algorithm to update the parameters of the autoencoder, we use the ADAM
            optimization algorithm. The reason for this is that it allows the network to be better trained. The ADAM
            optimization algorithm is essentially a Mini-Batch SGD algorithm that uses an adaptive learning rate. This
            means that the learning rate changes with respect to the loss landscape (see chapter 2.1.2.4)
        # To determine the deviation between the input images and the reconstructed images, we use an MSE cost function
            with weight decay (2.118)
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr, weight_decay=1e-8)
        self.criterion = nn.MSELoss()

# ____________________________________________________Training loop_____________________________________________________
    def train(self, train_loader, num_epochs):
        # We store the input images and the reconstructed images in the results list
        results = []
        losses = []
        n_total_steps = len(train_loader)
        for epoch in range(num_epochs):
            self.model.train()
            # We iterate through the images in the train_loader
            for i, (images, _) in enumerate(train_loader):
                images = images.reshape(-1, input_size).to(device)

                #_____ Forward pass____
                # The autoencoder uses the input image and reconstructs it by processing its information in each layer
                recon = self.model(images)
                # The MSE cost function with weight decay determines the deviation of the input images from the
                    reconstructed images
                loss = self.criterion(recon, images)

                # ______Backward pass_____
                self.optimizer.zero_grad()
                # The deviation between the input images and the reconstructed images is backpropagated and the
                    parameters are updated with the ADAM optimization algorithm
                loss.backward()
                self.optimizer.step()

                # The deviations determined by the MSE cost function with weight decay are printed
                # The input images and the reconstructed images are stored in the list results
                losses.append(loss.item())
                if (i + 1) % 100 == 0:
                    print(
                        f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1} of {n_total_steps}], Loss: {loss.item():.4f}')
                    results.append((epoch, images, recon))


        # Visualization of the input image and the reconstructed image to qualitatively determine if the network was
            able to reconstruct the input image
        for k in range(0, num_epochs, 4):
            plt.figure(figsize=(9, 2))
            plt.gray()
            imgs = results[k][1].cpu().detach().numpy()
            recon = results[k][2].cpu().detach().numpy()

            for i, item in enumerate(imgs):
                if i >= 9: break
                plt.subplot(2, 9, i + 1)
                item = item.reshape(-1, 28, 28)
                plt.imshow(item[0])

            for i, item in enumerate(recon):
                if i >= 9: break
                plt.subplot(2, 9, 9 + i + 1)
                item = item.reshape(-1, 28, 28)
                item = item.reshape(-1, 28, 28)
                plt.imshow(item[0])

# ____________________________________________________Evalutation_____________________________________________________
    def eval(self, test_loader):
        self.model.eval()
        with torch.no_grad():
            # We iterate through the images in the test_loader
            for (images,_) in test_loader:
                images = images.view(-1, input_size).to(device)
                # We reconstruct an image from the test data set
                # In other words, we reconstruct an unseen image to determine if the network is able to generalize
                recon = self.model(images)
'''
We define the dimension of each layer within the sparse autoencoder. Since it is sparse, each layer has the same
    dimension, namely the dimension of the input images
'''
def plain_network(layers, variances):
    layers_autoencoder = [784] * 2 * layers
    return Autoencoder(layers_autoencoder, variances)
```

Note, that I only commented on the lines of code that are different from A.4.3.

A.4.5 Register

The following code was created by Yanick Thurn and slightly modified by me. It can be used for a multilayer perceptron as well as for a sparse autoencoder.

```
# Import libraries
from typing import Tuple, Any
import pickle

'''
The class HookRegister creates a register for storing network properties and parameters. The register is a list with
    further sub-lists.
'''
class HookRegister:
    '''
    This constructur initalizes the register
    '''
    def __init__(self):
        self.register = [[]]
        self.layers = []
        # Storing the forward_hooks
        self.conns = []
        self.i = 0

    """
    With this function the instances can be called
    """
    def __call__(self, name: str, forward: Tuple[torch.tensor], backward: Tuple[
        torch.tensor]):
        self.register[self.i].append(forward[0].detach())

    """
    This function sets the state of the register to either True or False. If True, the hooks are created and the pre-
        activations are stored. If False, the hooks are removed.
    """
    def __getitem__(self, item: bool):

        if item:
            # Iterating over all the layers we have stored in the layers list via the hook(self,layer:torch.nn.Module)
                function
            for layer in self.layers:
                # We register a forward hook to each layer
                # We make use of the fact that the layer can be called due to __call__
                    (self,name:str,forward:Tuple[torch.tensor],backward:Tuple[torch.tensor]) and add it to the list
                    register
                self.conns.append(layer.register_forward_hook(self))

        else:
            for conn in self.conns:
                # We remove the forward hook from the layers
                conn.remove()


    '''
    This function stores the layers of the given neural network together with all its parameters in the list layers
    '''
    def hook(self, layer: torch.nn.Module):
        self.layers.append(layer)

    '''
    This function allows us to store the register as a pickle file
    '''
    def save(self, file: str):
        with open(file, "wb") as doc:
            pickle.dump(self.register, doc)
        self.clear()

    '''
    This function clears the memory of the register
    '''
    def clear(self):
        self.register = []

    '''
    This function adds all weight variances of a given neural network to the register and stores them in a sub-list of
        the list register
    '''
    def add(self, addition: Any):
        self.register[self.i].append(addition)

    '''
    This function adds a new sub-list to the register and points to it
    '''
    def step(self):
        # Adds a new sub-list to the register
        self.register.append([])
        # Points to the new sub-list
        self.i += 1

    """
```

```
    This function stores the weights and biases of a given neural network and passes them to the function
        add(self, addition : Any)
    """
    def add_parameters(self, network: torch.nn.Sequential):
        # Storing the weights and biases of a given neural network in the weights and biass lists
        weights = []
        biass = []
        for name, para in network.named_parameters():
            if name[-6:] == "weight":
                weights.append(para.detach())
            elif name[-4:] == "bias":
                biass.append(para.detach())
            else:
                raise Exception(f"Classification of parameter {name} failed")

        self.add(weights)
        self.add(biass)

'''
 This function takes a neural network and stores its pre-activations and parameters in a register
'''
def hook_network(network: torch.nn.Sequential) -> HookRegister:
    # Creating an instance of the HookRegister class, namely a register
    register = HookRegister()
    for layer in network:
        if isinstance(layer, torch.nn.Tanh):
            # Calling the hook function and giving it the layers of the network together with the corresponding
                parameters
            # This function then adds data hooks to the layers
            register.hook(layer)
    return register
```

A.4.6 Executer

The following Code can be used for both a multilayer perceptron and a sparse autoencoder.

```
'''
For both the multilayer perceptron and the sparse autoencoder this function executes the previously implemented
    functions and gives them the necessary arguments. Furthermore, this function creates a register with the class
    HookRegister (see A.4.5) to store the properties and parameters of the network before and after training.
'''
def train_networks(*varss):
    for i, vars in enumerate(varss):
        # _______Executing the code for creating a deep neural network_______
        # Multilayer perceptron
        network = get_wide_network(input_size, 10, hidden_layers, vars)
        # Sparse autoencoder
        network = plain_network(hidden_layers, vars)

        # We create a register and feed it the neural network so that hooks can be added
        register = hook_network(network.model)

        # We give the register the weight variances and store them inside a sub-list of the register
        # --> register = [[weight variances]]
        register.add(vars)
        # We create a new sub-list and point to it
        # --> register = [[weight variances], []]
        register.step()

        # We add the weights and biases of the untrained neural network to the newly created sub-list
        # --> register = [[weight variances], [weights, biases]]
        register.add_parameters(network.model)
        # We create a new sub-list and point to it
        # --> register = [[weight variances], [weights_pre, biases_pre], []]
        register.step()

        # We activate the register so that we create a forward hook and store the  layers inside the register
        # --> cons = [forward_hook_pre_training(tanh)] and register = [[weight variances], [weights_pre, biases_pre], [
            forward_hook_pre_training(tanh)] ]
        register[True]
        print(f"For var_w=({vars[0]}) the accuracy of the network with no training is")
        # We test the network with only one batch from the dataloader
        network.eval([next(iter(test_loader))])
        # We create a new sub-list and point to it
        # --> register = [[weight variances], [weights_pre, biases_pre], [forward_hook_pre_training(tanh)], []]
        register.step()

        # For training, we remove the forward hook and therefore do not add layers to the register
        register[False]
        # We train the network with the given training settings
        network.train(train_loader, test_loader, num_epochs)
        print(f"The the accuracy of the network after training where var_w = ({vars[0]}) is")

        # The weights and biases after training are added to the newly created sub-list
        # --> register = [[weight variances], [weights_pre, biases_pre],[forward_hook_pre_training(tanh)], [weights_post
            , biases_post]]
        register.add_parameters(network.model)
        # We create a new sub-list and point to it
```

```
        # register = [[weight variances], [weights_pre, biases_pre], [forward_hook_pre_training(tanh)] ,[weights_post, biases_post], []]
        register.step()

        # We add the layers after training to the register
        # --> cons = [forward_hook_pre_training(tanh), forward_hook_post_training(tanh)] and register = [[weight variances], [weights_pre, biases_pre], [forward_hook_pre_training(tanh)] ,[weights_post, biases_post], [forward_hook_post_training(tanh)]]
        register[True]
        # Using one batch from the dataloader, we test whether the network is able to classify unseen images and thus whether it is able to generalize
        network.eval([next(iter(test_loader))])
        # We save the register as a pickle file
        register.save(f"var={vars[0]}, {vars[1]})_depth=40_data.pkl")

# We create different samples of weight variances
vars_weight = np.linspace(1.0, 2.5, 7, True).tolist()
# We fix the bias and pass the weights and biases to the following function
# By executing the following line we start the program for the architecture, training and evaluation of the deep neural network
train_networks(*[(var_weight, 0.05) for var_weight in vars_weight])
```

A.4.7 Relative entropy and analysis of the layer distribution

For the implementation of the relative entropy, only the term $\langle Q^{[l+m]} \rangle_p$ is problematic [6], since it consists of the recursive functions $f_{j,k}^{[0]}$ and $f_j^{[0]}$. Due to the high dimensionality of this problem, we solve it numerically using the Monte Carlo method.
This method allows us to compute a multidimensional integral over a subregion χ of $\mathbb{R}^d$, where the continuous random variable $\vec{x}$ is high-dimensional, and therefore lives in a space that is too large for an exact calculation

$$\int_{\chi} dx\, p(\vec{x}) f(\vec{x}) = \mathbb{E}_p[f(\vec{x})]. \tag{A.87}$$

Here p is the probability density and f is some function. The idea now is to sample n_{MC} x_i, $i \in \{1, ..., n_{MC}\}$ from p, substitute these x_i into f and compute the average. For the limit of $n_{MC} \to \infty$, the strong law of large numbers allows us to say

$$\mathbb{E}_p[f(\vec{x})] \approx \frac{1}{n_{MC}} \sum_{i}^{n_{MC}} f(\vec{x}_i)\ ,\ \vec{x}_i \sim p. \tag{A.88}$$

Next, we enhance the Monte Carlo method with importance sampling. To do this, we introduce a new probability density function q that must be > 0 if $p(\vec{x}) f(\vec{x}) \neq 0$. We thus obtain an expected value of the function $[\frac{p(\vec{x})}{q(\vec{x})} f(\vec{x})]$ with respect to q

$$\mathbb{E}_p[f(\vec{x})] = \int_{\chi} dx\, p(\vec{x}) f(\vec{x}) \frac{q(\vec{x})}{q(\vec{x})} = \int_{\chi} dx\, q(\vec{x}) [\frac{p(\vec{x})}{q(\vec{x})} f(\vec{x})] = \mathbb{E}_q[\frac{p(\vec{x})}{q(\vec{x})} f(\vec{x})]. \tag{A.89}$$

Subsequently, the application of the Monte Carlo method leads to

$$\mathbb{E}_q \left[\frac{p(\vec{x})}{q(\vec{x})} f(\vec{x}) \right] \approx \frac{1}{n_{MC}} \sum_{i}^{n_{MC}} \frac{p(\vec{x})}{q(\vec{x})} f(\vec{x})\ ,\ \vec{x}_i \sim q, \tag{A.90}$$

where n_{MC} x_i are sampled from the distribution q. In simple terms, with importance sampling, we weight each sample x_i by an amount proportional to how often it occurs in the

ensemble. The magic of this method for our case is that we can simply choose q in (A.90) to be a Gaussian.
We now apply it to $f_{j,k}^{[0]}$ and $f_j^{[0]}$ (see (2.108)) and obtain

$$f_j^{[0]} = \frac{1}{\sqrt{2\pi(\sigma_{p,j}^{[0]})^2}} \int dz_j^{[0]} \, \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[0]}-\mu_{p,j}^{[0]}}{\sigma_{p,j}^{[0]}}\right)^2} \sigma_T(z_j^{[0]}) = \frac{1}{n_{MC}} \sum_{s=1}^{n_{MC}} \sigma_{T,s}(z_j^{[0]}), \tag{A.91}$$

and

$$\begin{aligned} f_{j,k}^{[0]} &= \frac{1}{2\pi\sigma_{p,j}^{[0]}\sigma_{p,k}^{[0]}} \int dz_j^{[0]} dz_k^{[l]} \, \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[0]}-\mu_{p,j}^{[0]}}{\sigma_{p,j}^{[0]}}\right)} \mathrm{e}^{-\frac{1}{2}\left(\frac{z_j^{[0]}-\mu_{p,k}^{[0]}}{\sigma_{p,k}^{[0]}}\right)} \sigma_T(z_j^{[0]})\sigma_T(z_k^{[0]}) \\ &= \frac{1}{n_{MC}} \sum_{s=1}^{n_{MC}} \sigma_{T,s}(z_j^{[0]})\sigma_{T,s}(z_k^{[0]}). \end{aligned} \tag{A.92}$$

To implement the relative entropy of a hierarchical network, we set the reference layer to $l = 0$ and the target layer to an arbitrary subsequent layer $m > l$. We then use the algorithm introduced by [6]:

1. At $m = 1$, we draw n_{MC} samples from the input layer's pre-activations vector $z^{[0]}$, where each neuron's pre-activation $z_i^{[0]} \sim \mathbb{N}(\mu^{[0]}, (\sigma^{[0]})^2)$. After applying the activation function σ_T, we obtain n_{MC} samples of the activations vector $\sigma_{T,s}(z^{[0]})$, where $s \in \{1, ..., n_{MC}\}$ labels the sample.
2. For $m = 2$, we multiply each element of the activation vector $\sigma_{T,s}(z^{[0]})$ by the corresponding weight, add the corresponding bias, and apply the activation function to obtain n_{MC} $\sigma_{T,s}(z^{[1]})$.
3. For $m \geq 3$, we repeat the previous step and obtain n_{MC} $\sigma_{T,s}(z^{[m-1]})$.
4. Next, we use n_{MC} samples of each layer's activation vector to evaluate $f_{j,k}^{[m-1]}$ and $f_j^{[m-1]}$ via Monte Carlo integration with importance sampling. Since the integrals are evaluated with respect to the reference layer, we no longer need to perform sampling when $m > 1$. We simply construct each $\sigma_T(z_i^{[m]})$ from the previous layer $\sigma_T(z_i^{[m-1]})$ and insert them all into the Monte Carlo integrator to obtain $f_{j,k}^{[m-1]}$ and $f_j^{[m-1]}$.
5. Finally, we substitute $f_{j,k}^{[m-1]}$ and $f_j^{[m-1]}$ into $\langle Q^{[l+m]} \rangle_p$ and compute the relative entropy.

In the following Python code, I have implemented this algorithm. With the resulting program, we want to quantify the propagation of information through a deep neural network using the relative entropy for a hierarchical neural network (2.117).